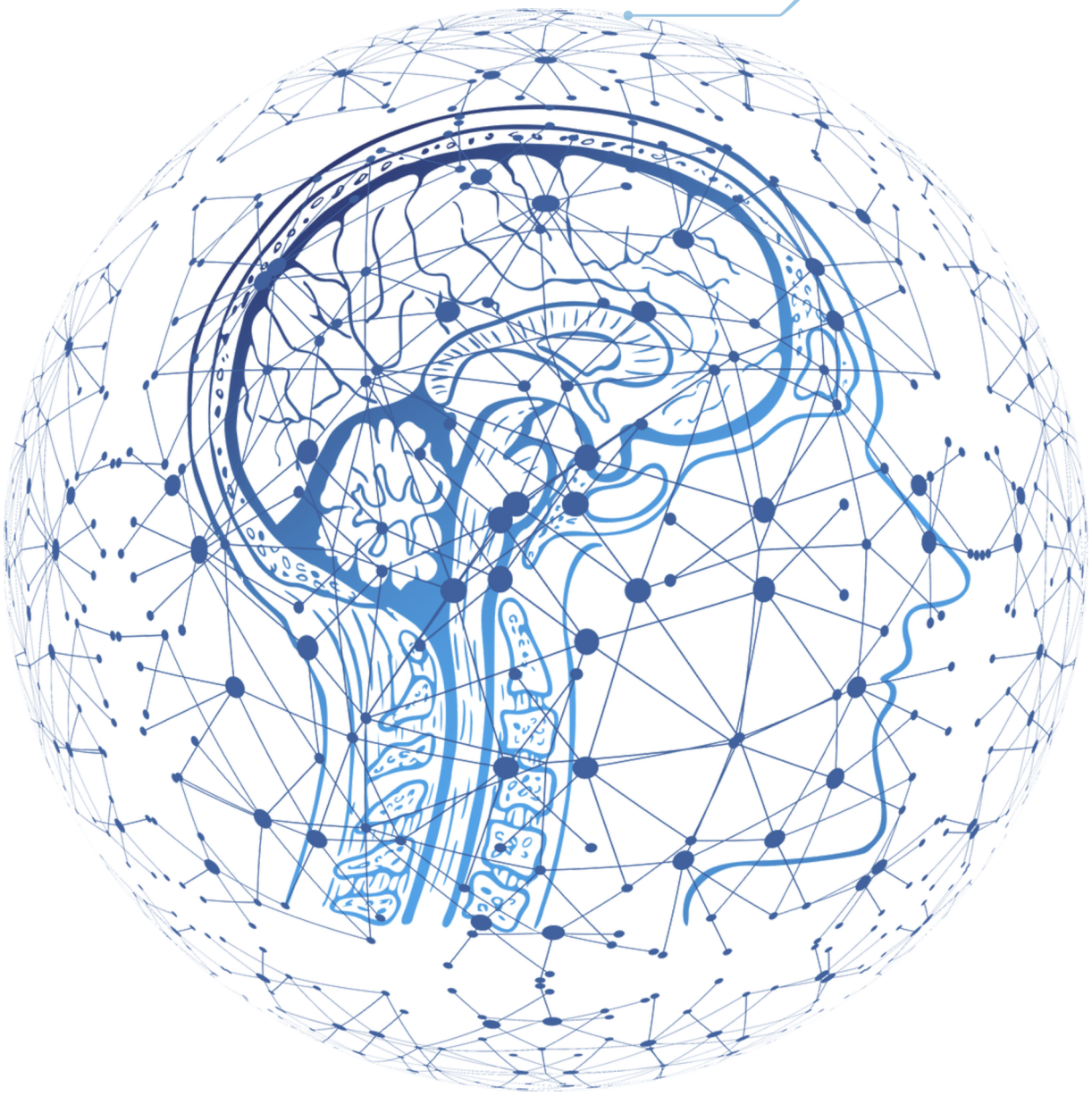
DEEP LEARNING

NỀN TẢNG LÝ THUYẾT CÁC MÔ HÌNH HỌC SÂU



Ian Goodfellow - Yoshua Bengio - Aaron Courville

Nguyễn Đình Quý và Nguyễn Khánh Chi dịch

MỤC LỤC

THUẬT NGỮ

Phần này cung cấp tài liệu tham khảo ngắn gọn mô tả các ký hiệu được sử dụng trong cuốn sách này. Nếu bắt gặp những khái niệm chưa quen thuộc, bạn có thể tìm thấy những giải thích chi tiết cùng các thuật toán liên quan trong các chương 2–4.

Số và mảng

a	Giá trị vô hướng (số nguyên hoặc số thực)
$\boldsymbol{a}$	Vector
$\boldsymbol{A}$	Ma trận
A	Tensor
$\boldsymbol{I_n}$	Ma trận đơn vị có n hàng và n cột
$\boldsymbol{I}$	Ma trận đơn vị với số chiều ngầm định theo ngữ cảnh
$\mathbf{e}^{(i)}$	Vector cơ sở tiêu chuẩn $[0, \dots, 0, 1, 0, \dots, 0]$ với số 1 tại vị trí i
$\text{diag}(\mathbf{a})$	Ma trận vuông trong đó các phần tử của a nằm trên đường chéo chính
a	Biến ngẫu nhiên vô hướng
$\mathbf{a}$	Biến ngẫu nhiên dạng vector
$\mathbf{A}$	Biến ngẫu nhiên dạng ma trận

Tập hợp và đồ thị

$\mathbb{A}$	Tập hợp
$\mathbb{R}$	Tập hợp số thực
$\{0, 1\}$	Tập hợp chứa 0 và 1
$\{0, 1, \dots, n\}$	Tập hợp các số nguyên từ 0 đến n
$[a, b]$	Khoảng số thực bao gồm cả a và b
$(a, b]$	Khoảng số thực bao gồm b nhưng không bao gồm a
$\mathbb{A} \setminus \mathbb{B}$	Phép trừ tập hợp, là tập chứa các phần tử của $\mathbb{A}$ nhưng không thuộc $\mathbb{B}$
$\mathcal{G}$	Đồ thị
$Pa_{\mathcal{G}}(x_i)$	Các nút cha của x_i trong $\mathcal{G}$

Đánh chỉ mục

a_i	Phần tử thứ i của vector $\mathbf{a}$, với chỉ số bắt đầu từ 1
a_{-i}	Tất cả các phần tử của vector $\mathbf{a}$ ngoại trừ phần tử thứ i
$A_{i,j}$	Phần tử hàng i, cột j của ma trận $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Hàng thứ i của ma trận $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Cột thứ i của ma trận $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Phần tử (i, j, k) của tensor 3 chiều $\mathbf{A}$
$\mathbf{A}_{:,:,i}$	Mặt cắt 2 chiều tại kênh i của tensor 3 chiều
a_i	Phần tử thứ i của vector ngẫu nhiên $\mathbf{a}$

Các phép toán trong Đại số Tuyến tính

$\boldsymbol{A}^\top$	Chuyển vị của ma trận $\boldsymbol{A}$
$\boldsymbol{A}^+$	Ma trận giả nghịch đảo Moore-Penrose của $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Phép nhân từng phần tử (Hadamard) của $\boldsymbol{A}$ và $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Định thức của ma trận $\boldsymbol{A}$

Giải tích

$\frac{dy}{dx}$	Đạo hàm của y theo x
$\frac{\partial y}{\partial x}$	Đạo hàm riêng của y theo x
$\nabla_x y$	Gradient của y theo vector x
$\nabla_{\mathbf{X}} y$	Đạo hàm ma trận của y theo ma trận $\mathbf{X}$
$\nabla_{\mathsf{X}} y$	Tensor chứa các đạo hàm của y theo $\mathbf{X}$
$\frac{\partial f}{\partial x}$	Ma trận Jacobian $\mathbf{J} \in \mathbb{R}^{m \times n}$ của $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla_x^2 f(x)$ hoặc $\mathbf{H}(f)(x)$	Ma trận Hessian của f tại điểm đầu vào x
$\int f(x)dx$	Tích phân xác định trên toàn bộ miền của x
$\int_{\mathbb{S}} f(x)dx$	Tích phân xác định của x trên tập $\mathbb{S}$

Lý thuyết Xác suất và Thông tin

$a \perp b$	Các biến ngẫu nhiên a và b là độc lập
$a \perp b \mid c$	Chúng độc lập có điều kiện với c
$P(a)$	Phân phối xác suất trên biến rời rạc
$p(a)$	Phân phối xác suất trên biến liên tục hoặc chưa xác định loại
$a \sim P$	Biến ngẫu nhiên a có phân phối P
$\mathbb{E}_{\mathbf{x}\sim P}[f(x)]$ hoặc $\mathbb{E}f(x)$	Kỳ vọng của $f(x)$ theo $P(\mathbf{x})$
$\mathrm{Var}(f(x))$	Phương sai của $f(x)$ dưới $P(\mathbf{x})$
$\mathrm{Cov}(f(x), g(x))$	Hiệp phương sai của $f(x)$ và $g(x)$ dưới $P(\mathbf{x})$
$H(\mathbf{x})$	Entropy Shannon của biến ngẫu nhiên $\mathbf{x}$
$D_{\mathrm{KL}}(P \parallel Q)$	Độ đo phân kỳ Kullback-Leibler giữa P và Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Phân phối Gaussian trên $\boldsymbol{x}$ với trung bình $\boldsymbol{\mu}$ và hiệp phương sai $\boldsymbol{\Sigma}$

Hàm số

$f : \mathbb{A} \to \mathbb{B}$	Hàm số f với miền xác định $\mathbb{A}$ và miền giá trị $\mathbb{B}$.
$f \circ g$	Thành phần của hai hàm f và g.
$f(\boldsymbol{x}; \boldsymbol{\theta})$	Một hàm của $\boldsymbol{x}$ được tham số hóa bởi $\boldsymbol{\theta}$.
$\log \boldsymbol{x}$	Logarit tự nhiên của x.
$\sigma(x)$	Hàm sigmoid logistic, $\frac{1}{1+\exp(-x)}$.
$\zeta(x)$	Hàm Softplus, $\log(1 + \exp(x))$.
$\|\boldsymbol{x}\|_p$	Chuẩn L^p của $\boldsymbol{x}$.
$\|\boldsymbol{x}\|$	Chuẩn L^2 của $\boldsymbol{x}$.
x^+	Phần dương của x, tức là $\max(0, x)$.
$\mathbf{1}_{\text{điều kiện}}$	Nhận giá trị 1 nếu điều kiện đúng, ngược lại bằng 0.

Tập dữ liệu và Phân phối

p_{data}	Phân phối sinh dữ liệu
$\hat{p}_{\text{data}}$	Phân phối thực nghiệm được xác định bởi tập huấn luyện
$\mathbb{X}$	Tập hợp các mẫu huấn luyện
$\boldsymbol{x}^{(i)}$	Mẫu dữ liệu thứ i (đầu vào) từ một tập dữ liệu
$y^{(i)}$ hoặc $\boldsymbol{y}^{(i)}$	Nhãn hoặc giá trị mục tiêu tương ứng với $\boldsymbol{x}^{(i)}$ trong học có giám sát
$\mathbf{X}$	Ma trận $m \times n$ với các giá trị $\boldsymbol{x}^{(i)}$ trong hàng $\mathbf{X}_{i,:}$, biểu thị toàn bộ dữ liệu đầu vào của hàng i

CHƯƠNG 1. GIỚI THIỆU

Từ rất lâu con người đã mơ ước tạo ra những cỗ máy có khả năng suy nghĩ. Khát vọng này được xuất phát ngay từ thời Hy Lạp cổ đại. Những nhân vật thần thoại như Pygmalion, Daedalus và Hephaestus được xem như những nhà phát minh huyền thoại, còn Galatea, Talos và Pandora lại là hình ảnh của sự sống nhân tạo (Ovid và Martin, 2004; Sparkes, 1996; Tandy, 1997).

Khi khái niệm về máy tính có thể lập trình lần đầu xuất hiện, nhiều người đã đặt câu hỏi liệu những cỗ máy này có thể trở nên thông minh hay không — điều này xảy ra cả trăm năm trước khi chiếc máy tính đầu tiên ra đời (Lovelace, 1842). Ngày nay, trí tuệ nhân tạo (AI) là một lĩnh vực phát triển mạnh mẽ với nhiều ứng dụng thực tiễn và hướng nghiên cứu sôi động. Chúng ta kỳ vọng phần mềm thông minh có thể thay thế con người làm các công việc lặp đi lặp lại, hiểu được hình ảnh, lời nói, chẩn đoán bệnh và thậm chí hỗ trợ nghiên cứu khoa học.

Trong những ngày đầu phát triển, trí tuệ nhân tạo đã nhanh chóng giải quyết những vấn đề tưởng chừng rất khó đối với con người, nhưng lại dễ xử lý với máy tính — đó là những bài toán có thể diễn tả bằng tập hợp các quy tắc toán học rõ ràng. Tuy nhiên, thách thức thực sự lại nằm ở chỗ ngược lại: làm sao để máy tính thực hiện những việc mà con người làm rất tự nhiên, như nhận diện giọng nói hoặc khuôn mặt — những việc mà chúng ta làm gần như theo bản năng nhưng lại khó mô tả một cách chính xác bằng công thức.

Cuốn sách này nói về cách giải quyết những bài toán trực giác như vậy. Giải pháp nằm ở việc để cho máy học từ dữ liệu, tự xây dựng hiểu biết về thế giới thông qua một hệ thống các khái niệm phân cấp — trong đó mỗi khái niệm phức tạp được xây dựng từ những khái niệm đơn giản hơn. Việc học từ dữ liệu giúp máy tính không cần con người phải lập trình sẵn toàn bộ kiến thức. Cấu trúc phân cấp này cho phép máy tính dần dần học được các khái niệm phức tạp bằng cách ghép nối những phần đơn giản hơn lại. Nếu ta biểu diễn sự kết nối giữa các khái niệm này bằng đồ thị, thì đồ thị đó sẽ có nhiều lớp, và do đó phương pháp này được gọi là học sâu (deep learning).

Những thành công ban đầu của AI chủ yếu xuất hiện trong các môi trường có tính quy chuẩn và đơn giản. Chẳng hạn, hệ thống chơi cờ của IBM tên là Deep Blue đã đánh bại nhà vô địch thế giới Garry Kasparov vào năm 1997 (Hsu, 2002). Trò chơi cờ vua, dù phức tạp với con người, nhưng lại rất dễ mô tả bằng các quy tắc hình thức: bàn cờ có 64 ô, 32 quân cờ với các nước đi rõ ràng. Dù thắng được Kasparov là một thành tựu lớn, nhưng điều đó không đòi hỏi AI phải hiểu thế giới theo cách mà con người vẫn sống và cảm nhận.

Điều mỉa mai là những công việc mang tính hình thức, toán học — vốn khó khăn với con người — lại dễ dàng với máy tính. Trong khi đó, những kỹ năng đời thường như nhận diện đồ vật hay giọng nói — vốn rất đơn giản với chúng ta — lại là thử thách lớn đối với AI. Để trở nên thông minh, máy tính cần có lượng kiến thức khổng lồ về thế giới. Phần lớn

kiến thức này mang tính chủ quan và trực giác nên rất khó diễn đạt bằng ngôn ngữ hình thức. Và một trong những thách thức lớn nhất của AI là: làm sao để đưa loại kiến thức đó vào máy tính.

Một số dự án AI đã từng cố gắng lập trình sẵn kiến thức vào máy thông qua các ngôn ngữ hình thức. Máy tính khi đó sẽ suy luận dựa trên các quy tắc logic. Phương pháp này được gọi là cách tiếp cận cơ sở tri thức (knowledge base). Tuy nhiên, các dự án kiểu này hiếm khi thành công. Một ví dụ nổi tiếng là hệ thống Cyc (Lenat và Guha, 1989), một hệ thống suy diễn sử dụng cơ sở dữ liệu viết bằng ngôn ngữ CycL. Các câu lệnh được nhập vào bởi con người, nhưng quy trình này lại rất phức tạp và khó mở rộng. Cyc thậm chí từng không hiểu được một câu chuyện đơn giản về Fred đang cạo râu vào buổi sáng (Linde, 1992). Do Fred cầm một chiếc dao cạo điện, Cyc kết luận rằng "FredWhileShaving" có các bộ phận điện, và vì thế nghi ngờ rằng Fred lúc đó có còn là con người không.

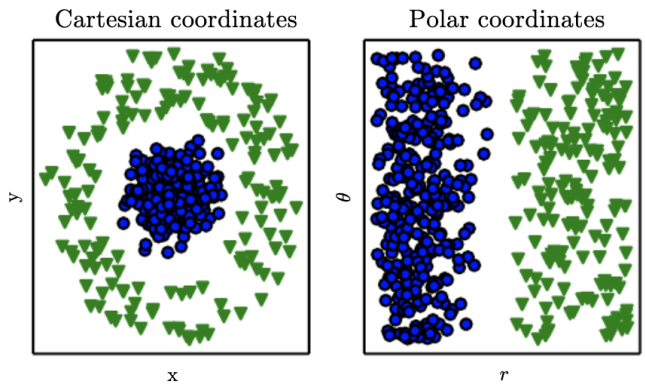
Những hạn chế của cách tiếp cận dựa vào tri thức lập trình sẵn cho thấy: AI cần khả năng tự học, tự rút ra quy luật từ dữ liệu gốc. Đây chính là nền tảng của lĩnh vực gọi là học máy (machine learning). Nhờ học máy, máy tính có thể xử lý các bài toán liên quan đến hiểu biết thế giới và đưa ra những quyết định có tính chủ quan. Chẳng hạn, một thuật toán đơn giản như *hồi quy logistic* có thể hỗ trợ bác sĩ quyết định có nên chỉ định mổ lấy thai hay không (Mor-Yosef et al., 1990). Hay một thuật toán đơn giản khác là *naive Bayes* có thể phân loại email rác và email hợp lệ.

Hiệu quả của các thuật toán học máy phụ thuộc rất nhiều vào cách biểu diễn dữ liệu (representation). Ví dụ, trong bài toán mổ lấy thai, hệ thống không tự quan sát bệnh nhân mà chỉ nhận đầu vào từ bác sĩ, như việc bệnh nhân có sẹo tử cung hay không. Mỗi thông tin như vậy được gọi là một đặc trưng (feature). Thuật toán học cách các đặc trưng này liên quan đến kết quả đầu ra. Tuy nhiên, nó không có khả năng quyết định đặc trưng nào nên được sử dụng. Nếu thay vì báo cáo của bác sĩ, thuật toán được cung cấp ảnh MRI, thì nó sẽ không thể đưa ra quyết định vì từng điểm ảnh riêng lẻ không mang nhiều ý nghĩa.

Việc phụ thuộc vào cách biểu diễn không chỉ giới hạn trong AI mà còn phổ biến trong khoa học máy tính và đời sống thường ngày. Chẳng hạn, việc tìm kiếm trong một cơ sở dữ liệu sẽ nhanh hơn nhiều nếu dữ liệu được sắp xếp thông minh. Tương tự, chúng ta làm toán với chữ số Ả Rập rất nhanh, nhưng lại gặp khó khăn với chữ số La Mã. Không có gì bất ngờ khi cách biểu diễn dữ liệu ảnh hưởng rất lớn đến hiệu quả của các thuật toán học máy. Xem ví dụ trực quan trong hình 1.2.

Nhiều bài toán AI có thể giải quyết bằng cách thiết kế ra tập đặc trưng phù hợp, rồi sử dụng một thuật toán học máy đơn giản. Ví dụ, trong bài toán nhận diện người nói, một đặc trưng hữu ích là ước lượng kích thước ống thanh quản — yếu tố này giúp phân biệt giữa giọng nam, nữ và trẻ em.

Tuy nhiên, trong nhiều bài toán khác, rất khó để xác định đặc trưng nào nên dùng. Ví dụ, nếu muốn viết chương trình nhận diện xe hơi trong ảnh, ta biết rằng xe có bánh xe, vậy có thể dùng sự xuất hiện của bánh xe làm đặc trưng. Nhưng vấn đề là rất khó mô tả



Hình 1.1: Ví dụ về các cách biểu diễn khác nhau: Giả sử chúng ta muốn phân tách hai loại dữ liệu bằng một đường thẳng trên biểu đồ phân tán. Ở biểu đồ bên trái, dữ liệu được biểu diễn theo hệ toạ độ Descartes (tọa độ vuông góc), và việc phân tách trở nên rất khó khăn — không thể kẻ được đường thẳng nào phù hợp. Nhưng ở biểu đồ bên phải, dữ liệu được chuyển sang hệ toạ độ cực, và lúc này ta có thể dễ dàng phân tách hai nhóm chỉ bằng một đường thẳng đứng. (Hình minh hoạ hợp tác cùng David Warde-Farley.)

hình ảnh của bánh xe bằng giá trị điểm ảnh, vì nó có thể bị bóng đổ, ánh sáng phản chiếu, vật thể khác che khuất, v.v.

Một cách để giải quyết khó khăn trong việc thiết kế đặc trưng là dùng học máy không chỉ để học hàm ánh xạ từ đặc trưng đến đầu ra, mà còn để học cả cách biểu diễn dữ liệu — tức là học đặc trưng một cách tự động. Phương pháp này gọi là học biểu diễn (representation learning). Khi để máy học ra cách biểu diễn phù hợp, kết quả thường vượt trội hơn nhiều so với đặc trưng do con người thiết kế thủ công. Đồng thời, việc này còn giúp hệ thống AI dễ dàng thích nghi với các tác vụ mới mà không cần nhiều can thiệp từ con người.

Một thuật toán học biểu diễn có thể khám phá ra tập đặc trưng tốt cho một nhiệm vụ đơn giản chỉ trong vài phút, hoặc trong vài giờ tới vài tháng nếu là nhiệm vụ phức tạp. Trong khi đó, việc thiết kế đặc trưng thủ công cho một tác vụ phức tạp có thể mất hàng chục năm và cần sự đóng góp của cả một cộng đồng nghiên cứu.

Ví dụ điển hình nhất cho thuật toán học biểu diễn là autoencoder. Một autoencoder bao gồm hai thành phần: encoder — biến đổi đầu vào thành một biểu diễn mới, và decoder — biến biểu diễn đó trở lại định dạng ban đầu. Autoencoder được huấn luyện sao cho giữ lại càng nhiều thông tin càng tốt sau khi dữ liệu đi qua encoder và sau đó là decoder. Đồng thời, mô hình cũng cố gắng điều chỉnh biểu diễn sao cho có các thuộc tính mong muốn — tuỳ thuộc vào loại autoencoder mà những thuộc tính này có thể khác nhau.

Khi thiết kế đặc trưng, hoặc xây dựng thuật toán học đặc trưng, mục tiêu thường là tách biệt các yếu tố biến thiên (factors of variation) gây ra sự đa dạng trong dữ liệu quan sát được. Ở đây, "yếu tố" đơn giản là các nguồn ảnh hưởng khác nhau — không nhất thiết phải kết hợp với nhau theo phép nhân. Những yếu tố này thường không quan sát được trực tiếp. Chúng có thể là các vật thể hoặc lực tác động trong thế giới vật lý, hoặc là các khái

niệm trừu tượng mà con người dùng để giải thích dữ liệu. Nói cách khác, chúng là các khái niệm giúp ta hiểu được sự phong phú trong dữ liệu.

Ví dụ, khi phân tích một đoạn ghi âm, các yếu tố biến thiên có thể bao gồm: tuổi tác của người nói, giới tính, giọng địa phương, và nội dung họ đang nói. Với một bức ảnh chụp xe hơi, các yếu tố bao gồm vị trí xe trong ảnh, màu sắc, góc nhìn, và độ sáng của ánh sáng mặt trời.

Một thách thức lớn trong các ứng dụng AI thực tế là: rất nhiều yếu tố biến thiên này ảnh hưởng đến toàn bộ dữ liệu mà ta quan sát được. Ví dụ, ảnh của một chiếc xe đỏ có thể trông gần như đen nếu chụp vào ban đêm. Hình dạng chiếc xe trong ảnh cũng thay đổi theo góc nhìn. Hầu hết các ứng dụng yêu cầu ta phải "tách rời" các yếu tố biến thiên và loại bỏ những yếu tố không liên quan.

Tuy nhiên, việc rút trích ra các đặc trưng trừu tượng từ dữ liệu thô là vô cùng khó. Nhiều yếu tố, như giọng địa phương của người nói, chỉ có thể nhận diện khi ta có mức hiểu biết gần như con người. Nếu việc học biểu diễn cũng khó như giải bài toán gốc, thì học biểu diễn tưởng như chẳng giúp ích gì nhiều.

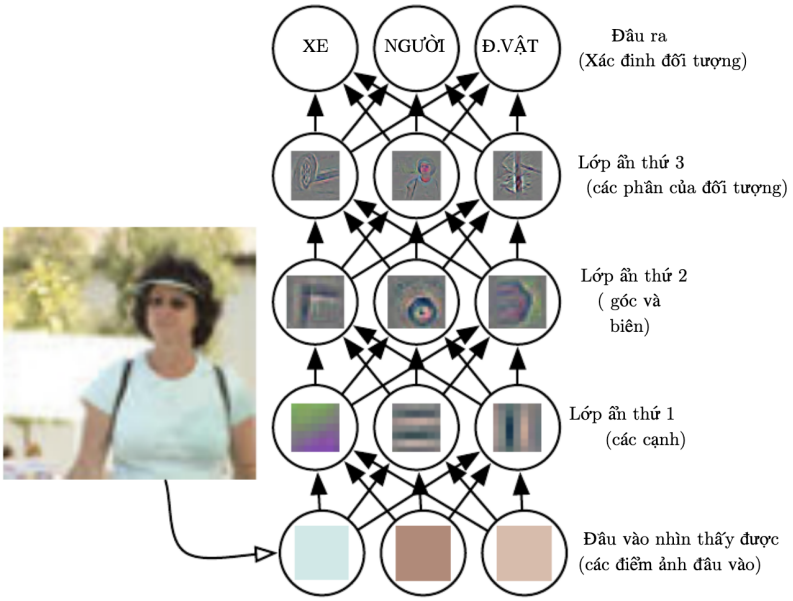
Học sâu (deep learning) chính là giải pháp cho vấn đề cốt lõi này. Nó làm được điều đó bằng cách xây dựng biểu diễn phức tạp thông qua nhiều lớp biểu diễn đơn giản hơn. Học sâu cho phép máy tính tạo ra khái niệm phức tạp từ các khái niệm đơn giản hơn. Hình **??** minh họa cách một hệ thống học sâu có thể biểu diễn khái niệm "hình ảnh một người" bằng cách kết hợp các khái niệm đơn giản như đường viền và góc cạnh, vốn được xây dựng từ các cạnh nhỏ hơn.

Mô hình tiêu biểu nhất của học sâu là mạng nơ-ron truyền thẳng nhiều lớp, hay còn gọi là multilayer perceptron (MLP). Một MLP chỉ đơn giản là một hàm toán học ánh xạ từ tập giá trị đầu vào sang đầu ra, bằng cách ghép nhiều hàm đơn giản lại với nhau. Mỗi hàm đơn giản có thể được xem như tạo ra một lớp biểu diễn mới từ đầu vào ban đầu.

Nhìn từ góc độ biểu diễn, học sâu cho phép máy tính học cách biểu diễn dữ liệu phù hợp. Nhưng từ một góc độ khác, ta có thể xem học sâu như cho phép máy tính học được một chương trình máy tính nhiều bước. Mỗi lớp trong mạng có thể được hiểu như là trạng thái của bộ nhớ sau khi thực thi một bước chương trình. Càng nhiều lớp, chương trình càng thực hiện được nhiều bước liên tiếp. Điều này rất mạnh mẽ, vì các bước sau có thể dựa trên kết quả của các bước trước.

Ở góc nhìn này, không phải tất cả thông tin trong mỗi lớp biểu diễn đều dùng để mô tả yếu tố biến thiên của dữ liệu. Một phần thông tin có thể đóng vai trò như trạng thái nội bộ của chương trình — giống như một biến đếm hoặc con trỏ trong một chương trình máy tính truyền thống. Nó không chứa nội dung của đầu vào, nhưng lại giúp mô hình tổ chức quá trình xử lý dữ liệu một cách hiệu quả.

Có hai cách phổ biến để đo độ sâu của một mô hình. Cách đầu tiên dựa trên số lượng các bước xử lý liên tiếp cần thực hiện để tính toán đầu ra của mô hình từ đầu vào. Ta có



Hình 1.2: Minh hoạ về một mô hình học sâu. Máy tính rất khó để hiểu được ý nghĩa của dữ liệu đầu vào cảm biến thô — ví dụ như hình ảnh được biểu diễn bằng tập hợp các giá trị điểm ảnh. Hàm ánh xạ từ điểm ảnh đến nhận diện vật thể là một hàm rất phức tạp, và nếu cố gắng học trực tiếp hàm đó thì gần như là bất khả thi. Học sâu giải quyết vấn đề này bằng cách chia nhỏ quá trình ánh xạ phức tạp đó thành một chuỗi các phép biến đổi đơn giản lồng nhau, mỗi phép biến đổi tương ứng với một lớp (layer) khác nhau trong mô hình. Dữ liệu đầu vào được đưa vào lớp quan sát (visible layer) — gọi như vậy vì các biến ở lớp này là những gì ta có thể quan sát được. Sau đó, một chuỗi các lớp ẩn (hidden layers) sẽ lần lượt trích xuất ra các đặc trưng ngày càng trừu tượng từ hình ảnh gốc. Chúng được gọi là "ẩn" vì giá trị của chúng không có sẵn trong dữ liệu, mà phải do mô hình tự học ra, dựa trên cách giải thích tốt nhất cho dữ liệu quan sát được. Các hình ảnh minh hoạ trong mô hình thể hiện kiểu đặc trưng mà mỗi đơn vị ẩn học được. Từ các điểm ảnh ban đầu, lớp ẩn thứ nhất có thể phát hiện ra các cạnh bằng cách so sánh độ sáng giữa các điểm ảnh lân cận. Từ các cạnh này, lớp ẩn thứ hai có thể dễ dàng xác định các góc và đường viền dài — những cấu trúc được hình thành từ tập hợp các cạnh. Tiếp theo, lớp ẩn thứ ba dùng thông tin về góc và đường viền để nhận diện từng phần cụ thể của vật thể. Cuối cùng, sự kết hợp các phần này cho phép mô hình nhận diện vật thể có trong hình ảnh.

thể hình dung điều này như chiều dài của đường đi dài nhất trong một sơ đồ khối (flow chart) mô tả quá trình tính toán đầu ra từ đầu vào. Giống như hai chương trình máy tính có thể thực hiện cùng một tác vụ nhưng có độ dài khác nhau tuỳ vào ngôn ngữ lập trình, một hàm toán học cũng có thể được biểu diễn bằng sơ đồ khối có độ sâu khác nhau, tuỳ thuộc vào việc ta cho phép sử dụng những hàm nào làm thành phần cơ bản. Hình **??** minh họa việc lựa chọn tập hợp các bước tính toán cơ bản sẽ ảnh hưởng đến cách đo độ sâu như thế nào.

Một cách tiếp cận khác — thường dùng trong các mô hình xác suất sâu (deep probabilistic models) — không đo độ sâu của sơ đồ tính toán, mà đo độ sâu của sơ đồ mô tả mối quan hệ giữa các khái niệm. Theo cách hiểu này, một biểu diễn phức tạp có thể được tính toán qua nhiều bước hơn sơ đồ khái niệm gốc, vì mô hình có thể liên tục cập nhật hiểu biết về các khái niệm đơn giản khi biết thêm thông tin về các khái niệm phức tạp hơn.

Chẳng hạn, khi hệ thống AI phân tích ảnh khuôn mặt và chỉ thấy một mắt do phần còn

lại bị bóng tối che khuất, nó ban đầu có thể kết luận chỉ có một mắt. Nhưng sau khi phát hiện ra đó là khuôn mặt, hệ thống có thể suy luận rằng khả năng cao còn có một mắt nữa, dù không quan sát thấy rõ. Trong trường hợp này, sơ đồ khái niệm chỉ có hai tầng — tầng "mắt" và tầng "khuôn mặt". Tuy nhiên, sơ đồ tính toán thực tế có thể sâu hơn nhiều nếu hệ thống thực hiện việc cập nhật thông tin lặp lại nhiều lần giữa các khái niệm.

Vì không luôn rõ ràng rằng nên đo độ sâu theo sơ đồ tính toán hay sơ đồ khái niệm, và vì mỗi người lại chọn tập hợp các hàm cơ bản khác nhau để xây dựng mô hình, nên không tồn tại một giá trị tuyệt đối nào cho "độ sâu" của một kiến trúc. Điều này giống như việc không có một độ dài duy nhất đúng cho một chương trình máy tính. Đồng thời, cũng chưa có sự đồng thuận rõ ràng về việc "bao nhiêu tầng thì được gọi là deep". Tuy vậy, nhìn chung, học sâu có thể được hiểu là nghiên cứu các mô hình có mức độ kết hợp nhiều tầng biểu diễn — hoặc là các hàm học được, hoặc là các khái niệm học được — nhiều hơn so với học máy truyền thống.

Tóm lại, học sâu — chủ đề chính của cuốn sách này — là một cách tiếp cận thuộc lĩnh vực trí tuệ nhân tạo. Cụ thể hơn, nó là một nhánh của học máy (machine learning), một kỹ thuật cho phép hệ thống máy tính cải thiện dần qua kinh nghiệm và dữ liệu. Chúng tôi cho rằng học máy là con đường khả thi duy nhất để xây dựng hệ thống AI có thể hoạt động trong các môi trường phức tạp của thế giới thực. Và trong số đó, học sâu là một phương pháp mạnh mẽ nhờ khả năng biểu diễn thế giới thông qua một hệ thống phân cấp các khái niệm — trong đó mỗi khái niệm trừu tượng hơn được định nghĩa dựa trên những khái niệm đơn giản hơn. Hình 1.3 minh họa mối quan hệ giữa các nhánh AI khác nhau, còn Hình 1.4 cho thấy sơ đồ tổng quát về cách các hệ thống này hoạt động.

1.1 Ai nên đọc cuốn sách này?

Cuốn sách này phù hợp với nhiều đối tượng, nhưng chúng tôi đặc biệt hướng đến hai nhóm người đọc:

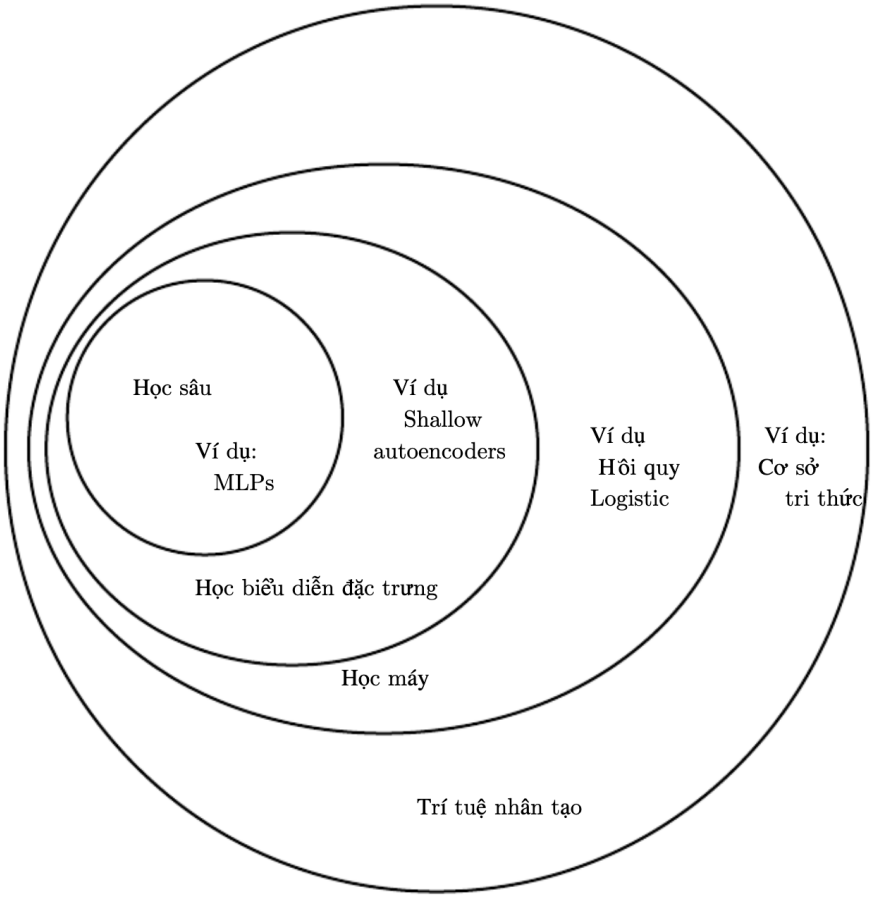
- Sinh viên đại học và sau đại học đang học về học máy, đặc biệt là những người mới bắt đầu sự nghiệp nghiên cứu trong học sâu và trí tuệ nhân tạo.
- Kỹ sư phần mềm không có nền tảng về học máy hay thống kê, nhưng muốn nhanh chóng tiếp cận và ứng dụng học sâu vào sản phẩm hoặc nền tảng của mình.

Học sâu đã chứng minh được hiệu quả trong nhiều lĩnh vực phần mềm như: thị giác máy tính, xử lý tiếng nói và âm thanh, xử lý ngôn ngữ tự nhiên, robot, tin sinh học và hóa học, trò chơi điện tử, công cụ tìm kiếm, quảng cáo trực tuyến và tài chính.

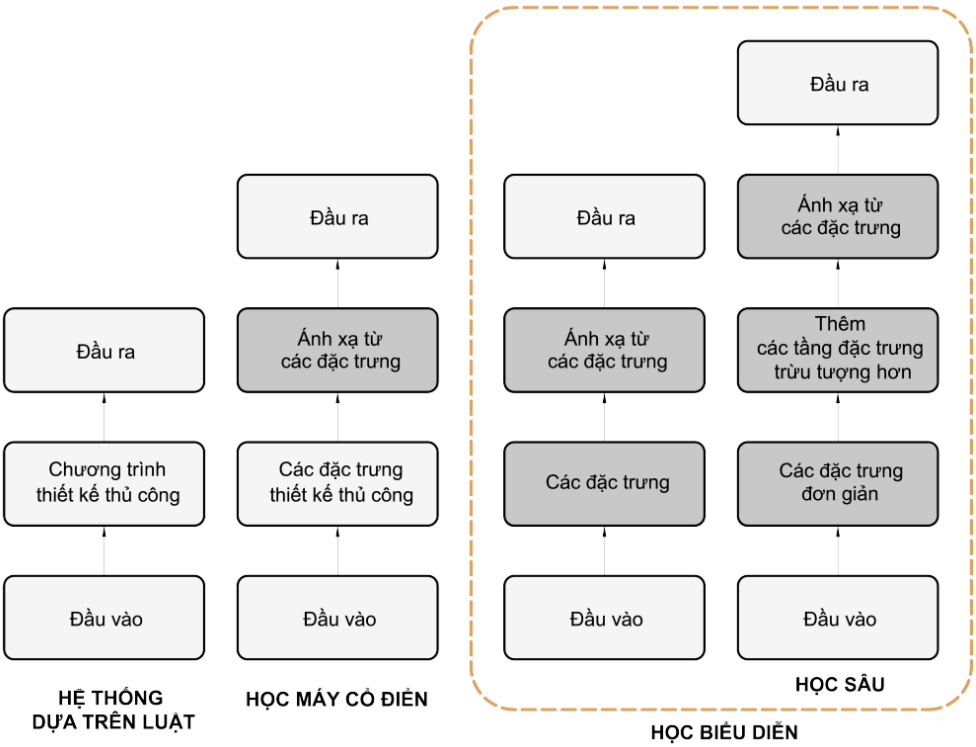
Cuốn sách được chia làm hai phần lớn để phù hợp với các nhóm người đọc khác nhau:

- Phần I: Giới thiệu các công cụ toán học cơ bản và các khái niệm trong học máy.
- Phần II: Trình bày các thuật toán học sâu đã được chứng minh và áp dụng thành công.

Người đọc có thể bỏ qua các phần không phù hợp với sở thích hoặc nền tảng của mình.



Hình 1.3: Sơ đồ Venn minh họa mối quan hệ giữa các lĩnh vực trong trí tuệ nhân tạo. Học sâu (deep learning) là một nhánh con của học biểu diễn (representation learning), và học biểu diễn lại nằm trong phạm vi của học máy (machine learning). Học máy là một trong nhiều phương pháp được sử dụng trong trí tuệ nhân tạo (AI), nhưng không phải là tất cả. Mỗi vùng trong sơ đồ có kèm một ví dụ cụ thể về công nghệ AI tương ứng: ví dụ, hệ chuyên gia thuộc về AI nhưng không dựa vào học máy; nhận diện chữ viết tay dùng học máy; trích xuất đặc trưng bằng tay là ví dụ của representation learning không học sâu; và nhận diện giọng nói tự động bằng mạng nơ-ron sâu là một ứng dụng của học sâu.



Hình 1.4: Các sơ đồ dòng cho thấy cách các thành phần khác nhau của một hệ thống AI liên kết với nhau trong các lĩnh vực AI khác nhau. Các ô có nền xám biểu thị các thành phần có khả năng học từ dữ liệu.

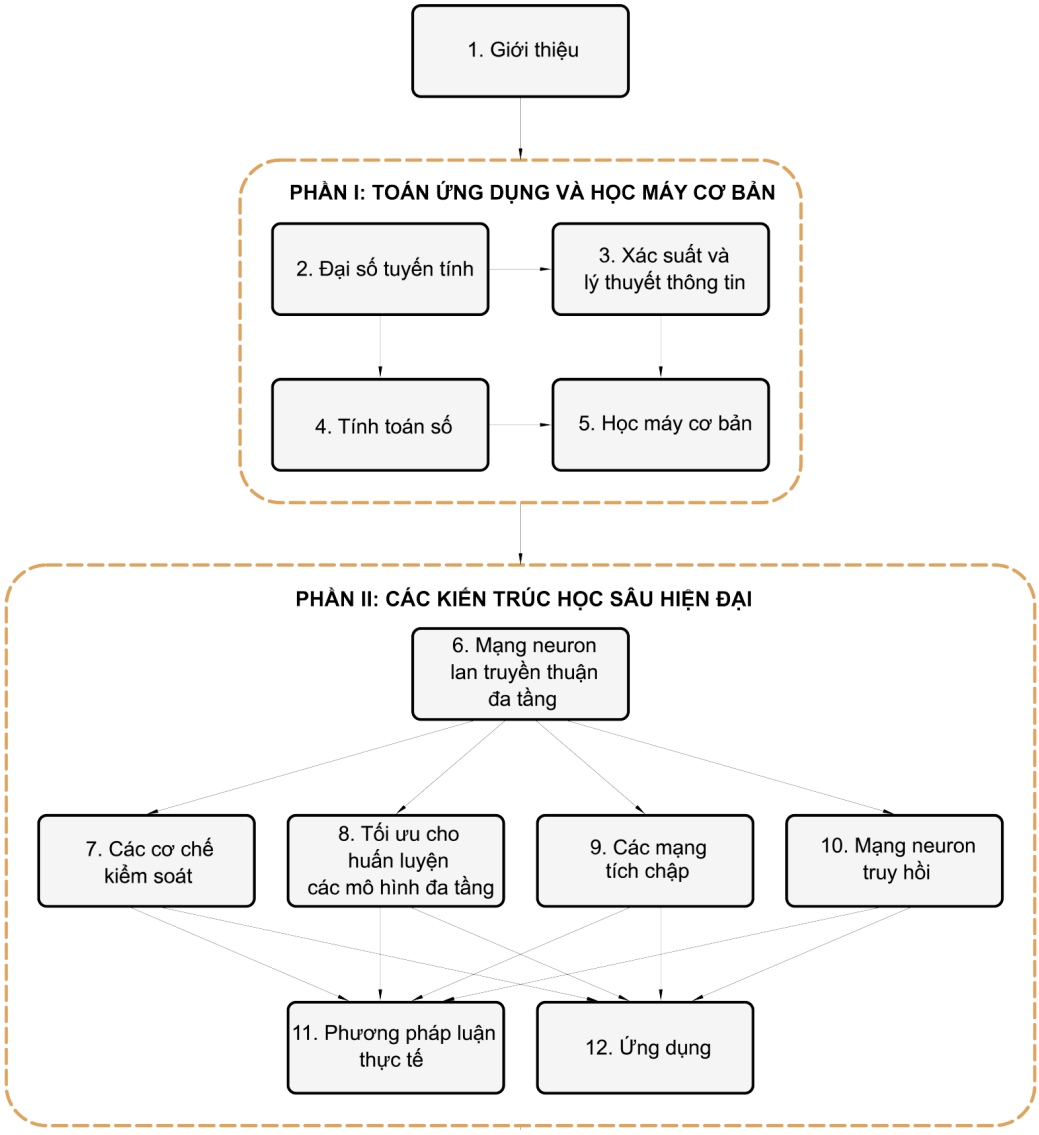
Ví dụ, những ai đã quen thuộc với đại số tuyến tính, xác suất và các khái niệm cơ bản về học máy có thể bỏ qua phần I. Còn những ai chỉ cần triển khai hệ thống học sâu mà không đi sâu vào lý thuyết có thể đọc đến phần II. Hình 1.6 cung cấp sơ đồ luồng chương trình đọc sách, giúp bạn lựa chọn chương nào phù hợp với mình.

Chúng tôi giả định người đọc đều có nền tảng về khoa học máy tính, bao gồm: biết lập trình, hiểu cơ bản về hiệu suất tính toán, lý thuyết độ phức tạp, giải tích cơ bản và một số thuật ngữ của lý thuyết đồ thị.

1.2 Các xu hướng lịch sử trong học sâu

Để hiểu rõ hơn về học sâu, chúng ta nên nhìn qua bối cảnh lịch sử của lĩnh vực này. Thay vì trình bày một dòng thời gian chi tiết, phần này sẽ tập trung vào một số xu hướng chính:

- Học sâu có một lịch sử lâu dài và phong phú, nhưng nó đã từng được biết đến dưới nhiều cái tên khác nhau, phản ánh các quan điểm triết lý khác nhau qua từng thời kỳ. Đã có lúc học sâu trở nên rất phổ biến, rồi sau đó ít được quan tâm, trước khi quay trở lại mạnh mẽ trong những năm gần đây.
- Học sâu ngày càng hữu ích hơn khi khối lượng dữ liệu huấn luyện tăng lên. Các mô hình học sâu thường cần một lượng lớn dữ liệu để thể hiện được sức mạnh của mình, và trong thời đại dữ liệu ngày nay, chúng đã phát huy tối đa tiềm năng.



Hình 1.5: Cấu trúc nội dung các phần của cuốn sách này

- Kích thước các mô hình học sâu liên tục tăng theo thời gian, nhờ vào sự cải thiện nhanh chóng của hạ tầng tính toán, bao gồm cả phần cứng (GPU, TPU,...) và phần mềm (framework như TensorFlow, PyTorch,...).
- Học sâu đã dần giải quyết được những bài toán ngày càng phức tạp với độ chính xác ngày càng cao. Từ nhận diện chữ viết tay, học sâu đã tiến đến xử lý ngôn ngữ tự nhiên, thị giác máy tính, và cả những tác vụ mang tính sáng tạo như tổng hợp giọng nói, vẽ tranh, làm thơ.

1.2.1 Những tên gọi khác nhau và sự thay đổi thăng trầm của mạng nơ-ron

Nhiều người khi lần đầu tiếp cận học sâu có thể ngạc nhiên khi thấy phần "lịch sử"trong một cuốn sách về một công nghệ tưởng như mới mẻ. Thực ra, học sâu đã xuất hiện từ những năm 1940. Nó chỉ có vẻ mới vì từng trải qua một giai đoạn dài không được chú ý và đã được gọi bằng nhiều tên khác nhau trước khi cái tên “deep learning” trở nên phổ biến như hiện nay.

Lịch sử phát triển của học sâu rất phong phú và đã trải qua ba làn sóng chính:

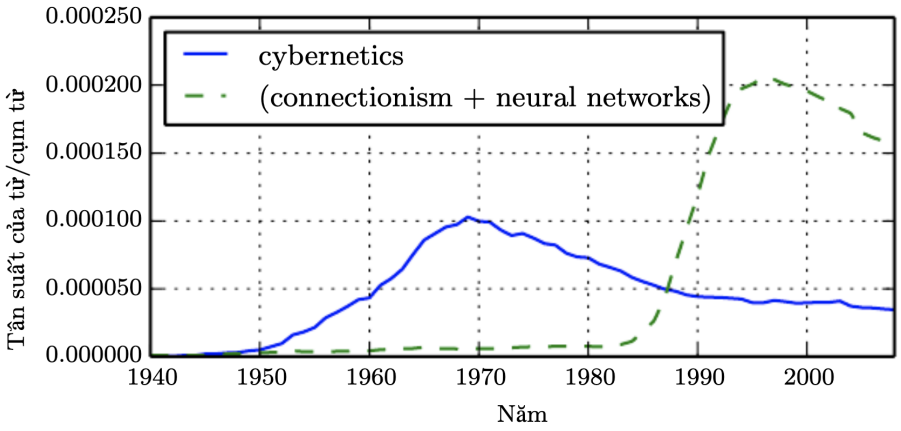
- Làn sóng thứ nhất diễn ra vào khoảng từ thập niên 1940 đến 1960, được biết đến với cái tên *cybernetics*. Đây là thời kỳ các nhà nghiên cứu xây dựng lý thuyết về học tập sinh học (McCulloch và Pitts, 1943; Hebb, 1949) và các mô hình đầu tiên như *perceptron* (Rosenblatt, 1958), cho phép huấn luyện một nơ-ron đơn lẻ.
- Làn sóng thứ hai bắt đầu từ khoảng năm 1980 đến giữa thập niên 1990, với tên gọi *connectionism*, khi thuật toán *back-propagation* (Rumelhart et al., 1986a) được sử dụng để huấn luyện các mạng có một hoặc hai tầng ẩn.
- Làn sóng hiện tại, hay học sâu (deep learning), bắt đầu khoảng năm 2006 (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007a) và hiện đang là trung tâm của nhiều tiến bộ lớn trong trí tuệ nhân tạo.

Hình 1.7 minh họa tần suất xuất hiện của các thuật ngữ “cybernetics”, “connectionism” và “neural networks” trong sách, giúp chúng ta thấy rõ ba làn sóng này.

Ban đầu, nhiều thuật toán học được phát triển nhằm mô phỏng cách não bộ học tập. Vì vậy, học sâu từng được gọi là *artificial neural networks* (mạng nơ-ron nhân tạo), mang theo quan điểm rằng đây là các hệ thống kỹ thuật lấy cảm hứng từ não bộ sinh học.

Tuy nhiên, dù đôi khi được dùng để nghiên cứu cách hoạt động của não (Hinton và Shallice, 1991), các mạng nơ-ron trong học máy không phải là mô hình sinh học chính xác. Chúng được thúc đẩy bởi hai ý tưởng chính:

- Thứ nhất, não bộ là bằng chứng sống cho thấy hành vi thông minh là có thể, và một cách hợp lý để xây dựng trí thông minh nhân tạo là tái tạo lại các nguyên lý tính toán của não.
- Thứ hai, việc hiểu rõ nguyên lý hoạt động của trí thông minh con người là một mục tiêu khoa học quan trọng, do đó mô hình học máy giúp làm sáng tỏ điều này có giá



Hình 1.6: Tần suất xuất hiện của các cụm từ "cybernetics" và "connectionism" hoặc "neural networks" trong Google Books, minh họa ba làn sóng lớn trong nghiên cứu mạng nơ-ron. Làn sóng thứ ba còn quá mới để được ghi nhận đầy đủ.

trị riêng, ngoài việc ứng dụng vào kỹ thuật.

Tuy nhiên, thuật ngữ "deep learning" hiện đại không chỉ giới hạn trong quan điểm sinh học. Nó nhấn mạnh vào nguyên lý tổng quát hơn: học thông qua việc kết hợp nhiều tầng biểu diễn — các khái niệm trừu tượng hơn được xây dựng từ các khái niệm đơn giản hơn.

Mô hình đơn giản đầu tiên là các mô hình tuyến tính, ví dụ như:

$$f(x, w) = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n,$$

được phát triển trong giai đoạn *cybernetics*. Mô hình McCulloch-Pitts (1943) là một ví dụ sớm mô phỏng hoạt động của nơ-ron. Đến thập niên 1950, perceptron (Rosenblatt, 1958, 1962) ra đời, là mô hình đầu tiên có khả năng học trọng số từ dữ liệu. Một mô hình tương tự là ADALINE (Widrow và Hoff, 1960) có thể dự đoán số thực, và huấn luyện bằng một phiên bản đơn giản của *stochastic gradient descent* — nền tảng của các thuật toán huấn luyện hiện đại.

Mặc dù đơn giản và hiệu quả, các mô hình tuyến tính này có nhiều hạn chế. Một ví dụ nổi tiếng là chúng không thể học hàm XOR. Điều này dẫn đến sự chỉ trích mạnh mẽ (Minsky và Papert, 1969), làm suy giảm niềm tin vào mạng nơ-ron trong nhiều năm.

Ngày nay, khoa học thần kinh vẫn là nguồn cảm hứng quan trọng, nhưng không còn là định hướng chính. Lý do là vì chúng ta chưa đủ khả năng để quan sát và hiểu được hoạt động chi tiết của hàng nghìn nơ-ron sinh học cùng lúc. Do đó, những hiểu biết về não bộ vẫn còn rất hạn chế (Olshausen và Field, 2005).

Tuy nhiên, các thí nghiệm như việc loài chồn có thể "nhìn" bằng vùng não xử lý âm thanh nếu được nối lại khác thường (Von Melchner et al., 2000) cho thấy có thể tồn tại một thuật toán chung cho nhiều tác vụ não bộ. Điều này khiến cộng đồng học máy ngày

càng thống nhất hơn, thay vì phân mảnh như trước.

Cuối cùng, một số kiến trúc nổi tiếng như *neocognitron* (Fukushima, 1980) và các mạng tích chập hiện đại (LeCun et al., 1998b) vẫn chịu ảnh hưởng từ cấu trúc của hệ thống thị giác sinh học. Ngày nay, phần lớn các mạng nơ-ron sử dụng đơn vị gọi là *rectified linear unit* (ReLU), vốn là phiên bản đơn giản hóa của các mô hình sinh học phức tạp hơn.

Dù khoa học thần kinh mang lại nhiều cảm hứng, việc mô phỏng trung thực chức năng của nơ-ron sinh học chưa mang lại lợi ích rõ ràng cho hiệu suất mô hình học máy. Tuy vậy, tinh thần học hỏi từ não bộ vẫn tiếp tục ảnh hưởng đến thiết kế kiến trúc mạng, ngay cả khi chưa thể hướng dẫn cụ thể về thuật toán học được sử dụng.

Trong giai đoạn những năm 1990, nhiều tiến bộ đã được thực hiện trong việc mô hình hóa chuỗi với mạng nơ-ron. Hochreiter (1991) và Bengio et al. (1994) đã xác định được một số khó khăn toán học cơ bản trong việc mô hình hóa các chuỗi dài (được trình bày ở Mục 10.7). Hochreiter và Schmidhuber (1997) đã đề xuất mô hình long short-term memory (LSTM) để khắc phục những khó khăn này. Hiện nay, LSTM được sử dụng rộng rãi trong nhiều bài toán mô hình chuỗi, đặc biệt là trong xử lý ngôn ngữ tự nhiên tại Google.

Làn sóng thứ hai của nghiên cứu mạng nơ-ron kéo dài đến giữa những năm 1990. Khi đó, các công ty khởi nghiệp dựa trên mạng nơ-ron và các công nghệ AI khác bắt đầu đưa ra các tuyên bố quá mức nhằm thu hút đầu tư. Khi kết quả không đạt được như kỳ vọng, các nhà đầu tư tỏ ra thất vọng. Đồng thời, các lĩnh vực học máy khác như kernel machines (Boser et al., 1992; Cortes và Vapnik, 1995; Schölkopf et al., 1999) và mô hình đồ thị (Jordan, 1998) cũng đạt được kết quả tốt trong nhiều bài toán quan trọng, khiến mạng nơ-ron bị giảm sút mức độ phổ biến cho đến tận năm 2007.

Tuy nhiên, trong giai đoạn này, mạng nơ-ron vẫn tiếp tục đạt được một số kết quả ấn tượng (LeCun et al., 1998b; Bengio et al., 2001). Viện nghiên cứu nâng cao Canada (CIFAR) đã giúp duy trì nghiên cứu mạng nơ-ron thông qua sáng kiến *Neural Computation and Adaptive Perception (NCAP)*. Sáng kiến này đã kết nối các nhóm nghiên cứu tại Đại học Toronto (Geoffrey Hinton), Đại học Montreal (Yoshua Bengio) và Đại học New York (Yann LeCun). CIFAR NCAP là một chương trình đa ngành, kết hợp cả các nhà khoa học thần kinh và chuyên gia về thị giác người và máy.

Vào thời điểm đó, mạng nơ-ron sâu thường bị xem là rất khó huấn luyện. Ngày nay chúng ta biết rằng các thuật toán từ những năm 1980 vẫn hoạt động tốt, nhưng điều này chưa rõ ràng vào khoảng năm 2006. Có lẽ nguyên nhân là do các thuật toán đó tiêu tốn tài nguyên tính toán quá lớn so với phần cứng khi đó, dẫn đến rất ít thử nghiệm thực tế.

Làn sóng thứ ba của nghiên cứu mạng nơ-ron bắt đầu với một đột phá vào năm 2006. Geoffrey Hinton cho thấy rằng một loại mạng gọi là deep belief network có thể được huấn luyện hiệu quả bằng chiến lược greedy layer-wise pretraining (Hinton et al., 2006), sẽ được trình bày chi tiết ở Mục 15.1. Các nhóm nghiên cứu khác trong CIFAR nhanh chóng chứng minh rằng chiến lược này cũng có thể áp dụng cho nhiều loại mạng sâu khác

(Bengio et al., 2007; Ranzato et al., 2007a), giúp cải thiện đáng kể khả năng tổng quát trên tập kiểm tra.

Làn sóng này đã phổ biến thuật ngữ "deep learning" để nhấn mạnh rằng các nhà nghiên cứu giờ đây có thể huấn luyện các mạng nơ-ron sâu hơn so với trước đây, đồng thời cũng tập trung vào tầm quan trọng lý thuyết của độ sâu (Bengio và LeCun, 2007; Delalleau và Bengio, 2011; Pascanu et al., 2014a; Montufar et al., 2014). Ở thời điểm đó, mạng nơ-ron sâu đã vượt qua các hệ thống AI cạnh tranh khác, kể cả các mô hình học máy hiện đại và các hệ thống chức năng thiết kế thủ công.

Làn sóng thứ ba của mạng nơ-ron vẫn đang tiếp diễn cho đến thời điểm cuốn sách này được viết. Tuy nhiên, trọng tâm nghiên cứu trong cộng đồng deep learning đã thay đổi đáng kể. Nếu như ban đầu cộng đồng quan tâm đến các kỹ thuật học không giám sát mới và khả năng tổng quát từ các tập dữ liệu nhỏ, thì hiện nay mối quan tâm chính đã chuyển sang các thuật toán học có giám sát cũ và khả năng tận dụng tập dữ liệu được gán nhãn lớn.

Kể từ sau năm 2012, deep learning đã nhanh chóng trở thành công nghệ cốt lõi trong nhiều hệ thống AI hiện đại. Sự kiện nổi bật đánh dấu bước ngoặt này là chiến thắng thuyết phục của mạng nơ-ron sâu trong cuộc thi nhận dạng hình ảnh ImageNet Large Scale Visual Recognition Challenge (ILSVRC) năm 2012, khi mô hình AlexNet (Krizhevsky et al., 2012) vượt trội so với các phương pháp truyền thống.

Kể từ đó, nhiều lĩnh vực đã chứng kiến sự thay đổi mạnh mẽ nhờ vào deep learning:

- Thị giác máy tính (Computer Vision): Các kiến trúc như ResNet, EfficientNet và Vision Transformer đã đẩy mạnh độ chính xác và hiệu quả trong các bài toán nhận diện ảnh, phân đoạn đối tượng, theo dõi vật thể và sinh ảnh.
- Xử lý ngôn ngữ tự nhiên (NLP): Các mô hình ngôn ngữ lớn như BERT (Devlin et al., 2018), GPT (Radford et al., 2018, 2019, 2020), và các biến thể mới hơn như ChatGPT, Gemini, Claude, LLaMA đã thiết lập nền tảng cho một loạt ứng dụng như dịch máy, tóm tắt văn bản, sinh văn bản, đối thoại và tìm kiếm thông minh.
- Trí tuệ tổng hợp đa nhiệm (Foundation Models): Thay vì huấn luyện một mô hình riêng cho từng nhiệm vụ, các mô hình nền tảng (foundation models) được huấn luyện trên tập dữ liệu cực lớn và sau đó được tinh chỉnh (fine-tuned) hoặc dùng trực tiếp (zero-shot, few-shot) để giải quyết nhiều tác vụ khác nhau.
- Tự động hóa và robot: AI đang được tích hợp vào robot tự hành, drone, cánh tay robot và hệ thống điều khiển công nghiệp để học từ môi trường, thích nghi với tình huống và tối ưu hiệu suất theo thời gian thực.
- Y học và khoa học sự sống: Deep learning giúp cải thiện chẩn đoán hình ảnh y tế, thiết kế thuốc mới, phân tích gene và dự đoán cấu trúc protein (ví dụ như AlphaFold).
- Sáng tạo nội dung (Generative AI): Các mô hình như DALL·E, Stable Diffusion, MidJourney cho phép sinh ảnh từ văn bản; các mô hình tổng hợp giọng nói và âm

nhạc cũng đạt chất lượng cao đáng kể, mở ra kỷ nguyên sáng tạo nội dung dựa trên AI.

Trong những năm tới, chúng ta có thể kỳ vọng AI sẽ tiếp tục phát triển theo các hướng sau:

- AI mô-đun và có khả năng tương tác: Mô hình AI sẽ không chỉ là một hệ thống đơn lẻ, mà là tập hợp các thành phần có thể tương tác và học lẫn nhau (multi-agent systems, tool-using AI).
- AI tích hợp kiến thức và suy luận: Kết hợp deep learning với symbolic reasoning hoặc graph-based reasoning để đạt được khả năng suy luận có cấu trúc, giải thích được.
- AI dân chủ hóa: Với sự phát triển của các mô hình mã nguồn mở như LLaMA, Mistral, hay các hệ sinh thái phần cứng tiết kiệm năng lượng, AI sẽ ngày càng trở nên dễ tiếp cận hơn đối với cộng đồng nghiên cứu và doanh nghiệp nhỏ.
- AI gắn với đạo đức và xã hội: Các hệ thống AI sẽ được thiết kế không chỉ để tối ưu hóa hiệu suất, mà còn để tuân thủ các tiêu chuẩn đạo đức, luật pháp và giá trị con người.

Với bối cảnh hiện nay, deep learning không còn là một công nghệ mới mẻ, mà đang từng bước trở thành một công cụ cơ bản, không thể thiếu trong nhiều lĩnh vực của đời sống, khoa học và công nghiệp hiện đại.

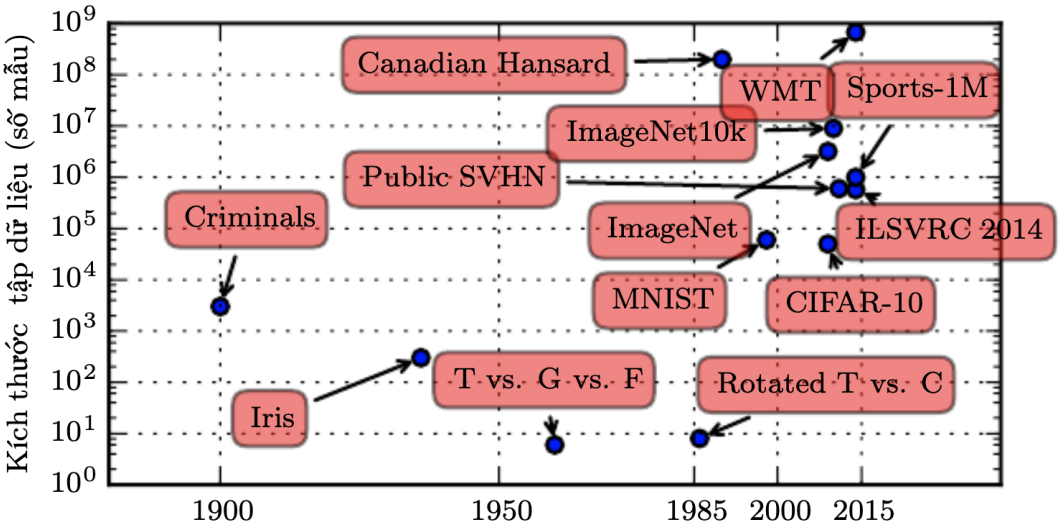
1.2.2 Tăng kích thước tập dữ liệu

Có thể bạn sẽ thắc mắc: Tại sao học sâu chỉ mới gần đây được công nhận là một công nghệ quan trọng, trong khi các thí nghiệm đầu tiên với mạng nơ-ron nhân tạo đã xuất hiện từ những năm 1950? Thực tế, học sâu đã được sử dụng thành công trong các ứng dụng thương mại từ thập niên 1990, nhưng thời đó nó vẫn bị xem là một lĩnh vực mang tính nghệ thuật nhiều hơn là công nghệ — điều gì đó mà chỉ những chuyên gia mới có thể sử dụng hiệu quả.

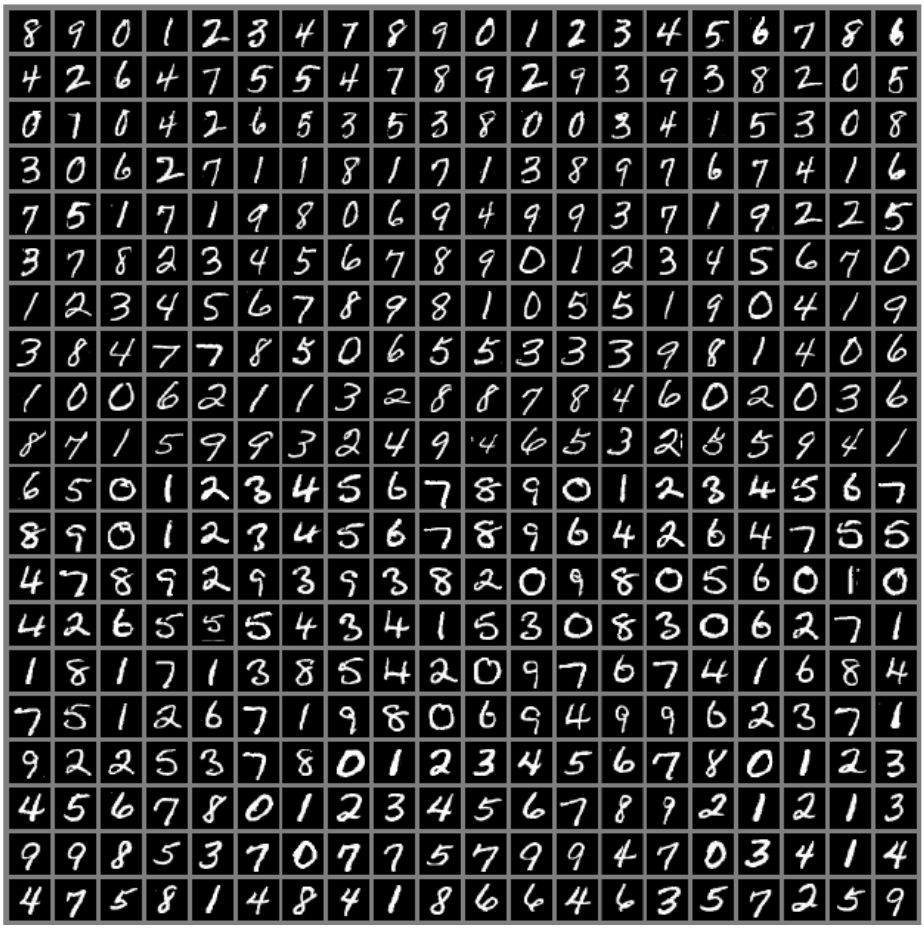
Đúng là cần một số kỹ năng nhất định để có thể đạt được hiệu quả cao với các thuật toán học sâu. Tuy nhiên, may mắn thay, lượng kỹ năng cần thiết sẽ giảm dần khi chúng ta có nhiều dữ liệu huấn luyện hơn. Những thuật toán hiện nay đạt đến hoặc vượt qua mức độ chính xác của con người trong các tác vụ phức tạp thực chất không khác mấy so với những thuật toán từng rất chật vật với các bài toán đồ chơi vào những năm 1980. Điểm khác biệt chính là các mô hình hiện tại đã có nhiều thay đổi giúp việc huấn luyện các kiến trúc sâu trở nên dễ dàng hơn.

Điểm phát triển quan trọng nhất là: ngày nay chúng ta có thể cung cấp đủ tài nguyên cho các thuật toán học để chúng hoạt động hiệu quả. Hình 1.7 cho thấy kích thước của các tập dữ liệu chuẩn dùng để đánh giá đã tăng lên đáng kể theo thời gian.

Xu hướng này được thúc đẩy bởi quá trình số hoá ngày càng sâu rộng trong xã hội. Khi



Hình 1.7: Sự gia tăng kích thước tập dữ liệu theo thời gian. Vào đầu những năm 1900, các nhà thống kê làm việc với các tập dữ liệu có quy mô chỉ vài trăm đến vài nghìn điểm dữ liệu được thu thập thủ công (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). Từ những năm 1950 đến 1980, các nhà tiên phong trong học máy lấy cảm hứng từ sinh học thường sử dụng những tập dữ liệu nhân tạo nhỏ, như ảnh bitmap độ phân giải thấp của các chữ cái. Những tập này được thiết kế để giảm thiểu chi phí tính toán và dùng để chứng minh rằng mạng nơ-ron có thể học được các loại hàm cụ thể (Widrow và Hoff, 1960; Rumelhart et al., 1986b). Trong thập niên 1980 và 1990, học máy dần mang tính thống kê hơn và bắt đầu tận dụng các tập dữ liệu lớn hơn, với quy mô hàng chục nghìn ví dụ — chẳng hạn như tập MNIST (minh hoạ ở Hình 1.8), gồm các ảnh quét số viết tay (LeCun et al., 1998b). Bước sang những năm 2000, các tập dữ liệu tinh vi hơn với quy mô tương tự tiếp tục xuất hiện, ví dụ như CIFAR-10 (Krizhevsky và Hinton, 2009). Tuy nhiên, đến cuối thập niên đó và suốt nửa đầu những năm 2010, các tập dữ liệu lớn hơn rất nhiều — từ hàng trăm nghìn đến hàng chục triệu ví dụ — đã hoàn toàn thay đổi tiềm năng của học sâu. Một số ví dụ tiêu biểu gồm tập Street View House Numbers công khai (Netzer et al., 2011), các phiên bản khác nhau của ImageNet (Deng et al., 2009, 2010a; Russakovsky et al., 2014a), và Sports-1M (Karpathy et al., 2014). Phía trên cùng của biểu đồ là các tập dữ liệu dịch máy như tập của IBM dựa trên biên bản Hansard của quốc hội Canada (Brown et al., 1990) và tập tiếng Anh–Pháp WMT 2014 (Schwenk, 2014), vốn thường vượt xa về quy mô so với các loại tập dữ liệu khác.



Hình 1.8: Một số ví dụ ảnh từ tập dữ liệu MNIST. "NIST" là viết tắt của National Institute of Standards and Technology — cơ quan đã thu thập dữ liệu này ban đầu. Chữ "M" là viết tắt của "modified" (đã được điều chỉnh), vì dữ liệu đã qua xử lý trước để dễ dùng hơn với các thuật toán học máy. MNIST bao gồm các ảnh quét số viết tay và nhãn tương ứng mô tả chữ số từ 0 đến 9 trong từng ảnh. Bài toán phân loại đơn giản này là một trong những bài kiểm tra phổ biến và kinh điển nhất trong nghiên cứu học sâu. Dù ngày nay đã khá dễ để giải quyết bằng các phương pháp hiện đại, tập dữ liệu này vẫn rất được ưa chuộng. Geoffrey Hinton từng ví MNIST như là "con ruồi giấm của học máy" — vì nó giúp các nhà nghiên cứu kiểm nghiệm thuật toán trong điều kiện phòng thí nghiệm kiểm soát, giống như cách mà các nhà sinh học thường nghiên cứu ruồi giấm.

ngày càng nhiều hoạt động của chúng ta diễn ra trên máy tính, ngày càng nhiều hành vi được ghi lại. Và khi các máy tính ngày càng kết nối với nhau, việc tập trung và xử lý các dữ liệu này thành tập dữ liệu huấn luyện phù hợp với học máy cũng trở nên dễ dàng hơn.

Kỷ nguyên của "Dữ liệu lớn (Big Data)" đã làm cho học máy dễ dàng hơn, vì gánh nặng của việc ước lượng thống kê — nghĩa là làm sao để mô hình tổng quát hóa tốt từ một lượng dữ liệu nhỏ — đã được giảm đi đáng kể.

Tính đến năm 2016, một quy tắc kinh nghiệm là: một thuật toán học sâu có giám sát (supervised deep learning) thường cho hiệu năng chấp nhận được khi mỗi lớp (category) có khoảng 5.000 ví dụ đã được gán nhãn, và có thể đạt đến hoặc vượt qua hiệu suất của con người khi được huấn luyện trên tập dữ liệu có ít nhất 10 triệu ví dụ được gán nhãn.

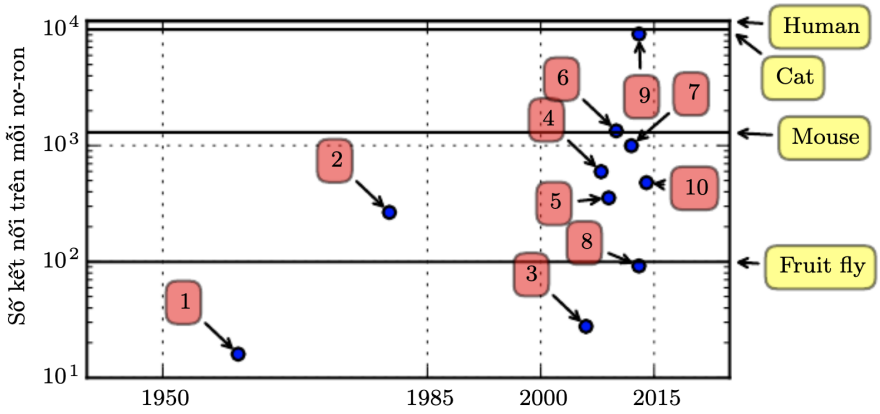
Tất nhiên, việc xây dựng các hệ thống hoạt động tốt trên các tập dữ liệu nhỏ hơn vẫn là một hướng nghiên cứu quan trọng. Trong đó, một câu hỏi lớn là: làm thế nào để tận dụng được lượng dữ liệu chưa gán nhãn rất lớn thông qua học không giám sát (unsupervised) hoặc bán giám sát (semi-supervised).

1.2.3 Tăng kích thước mô hình

Một lý do quan trọng khác khiến mạng nơ-ron ngày nay trở nên cực kỳ thành công — sau một thời gian dài ít được chú ý kể từ những năm 1980 — chính là vì hiện tại chúng ta đã có đủ tài nguyên tính toán để vận hành các mô hình có quy mô lớn hơn nhiều.

Một trong những ý tưởng cốt lõi của trường phái connectionism (tạm dịch: học kết nối) là: trí thông minh của động vật không đến từ từng nơ-ron riêng lẻ, mà đến từ việc một số lượng lớn nơ-ron cùng hoạt động với nhau. Một nơ-ron đơn lẻ, hay thậm chí một cụm nhỏ các nơ-ron, không đủ để tạo ra hành vi thông minh.

Trong sinh học, các nơ-ron không hề kết nối với mật độ cao một cách đặc biệt. Như minh họa trong Hình 1.9, số lượng kết nối trên mỗi nơ-ron trong các mô hình học máy từ lâu đã nằm trong cùng một bậc độ lớn (order of magnitude) với não bộ của các loài động vật có vú.



Hình 1.9: Số kết nối trên mỗi nơ-ron theo thời gian. Dữ liệu về kích thước mạng nơ-ron sinh học được lấy từ Wikipedia (2015).

Một số mô hình học máy được thể hiện trong đồ thị 1.9 bao gồm:

1. Adaptive linear element (Widrow và Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. Mạng tích chập sử dụng GPU (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov và Hinton, 2009a)
5. Mạng tích chập không giám sát (Jarrett et al., 2009)
6. MLP dùng GPU (Ciresan et al., 2010)
7. Autoencoder phân tán (Le et al., 2012)

8. Mạng tích chập dùng nhiều GPU (Krizhevsky et al., 2012)
9. Mạng tích chập không giám sát trên hệ HPC thương mại (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

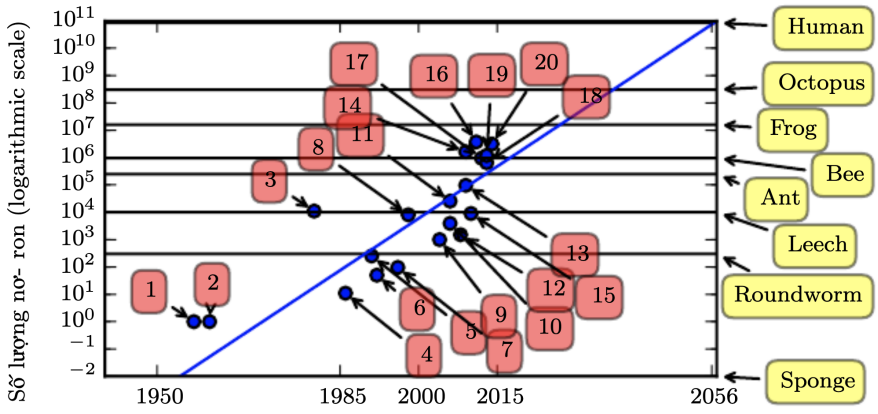
Tuy nhiên, nếu xét về tổng số nơ-ron thì mạng nơ-ron nhân tạo vẫn còn rất nhỏ — cho đến tận gần đây, như minh họa trong Hình 1.10. Kể từ khi khái niệm "nút ẩn" (hidden unit) được giới thiệu, kích thước của mạng nơ-ron nhân tạo đã tăng gấp đôi khoảng mỗi 2.4 năm. Sự tăng trưởng này chủ yếu nhờ vào: vi xử lý ngày càng nhanh hơn, bộ nhớ máy tính lớn hơn, và các tập dữ liệu huấn luyện lớn hơn.

Mạng càng lớn thì khả năng đạt được độ chính xác cao trong các bài toán phức tạp càng cao. Xu hướng này dự kiến sẽ còn tiếp diễn trong nhiều thập kỷ tới. Trừ khi xuất hiện một công nghệ mới mang tính cách mạng giúp tăng tốc vượt bậc, thì phải đến khoảng những năm 2050 mạng nơ-ron nhân tạo mới đạt đến số lượng nơ-ron tương đương với não người.

Ngoài ra, cần lưu ý rằng mỗi nơ-ron sinh học có thể thực hiện các phép toán phức tạp hơn nhiều so với nơ-ron nhân tạo hiện nay. Vì vậy, quy mô thực sự của mạng thần kinh sinh học có thể còn lớn hơn nhiều so với những gì đồ thị hiện tại thể hiện.

Nhìn lại, việc các mạng nơ-ron nhỏ hơn cả hệ thần kinh của một con đỉa không thể giải các bài toán AI phức tạp là điều hoàn toàn dễ hiểu. Ngay cả các mô hình ngày nay — vốn đã được xem là "lớn" về mặt tính toán — cũng vẫn còn nhỏ hơn hệ thần kinh của một số loài động vật có xương sống đơn giản như ếch.

Việc tăng quy mô mô hình qua thời gian, nhờ vào sự phát triển của CPU nhanh hơn, GPU đa năng (sẽ được nói đến trong Mục 12.1.2), tốc độ mạng cao hơn, cùng với hạ tầng phần mềm phân tán tốt hơn — chính là một trong những xu hướng quan trọng nhất trong lịch sử của học sâu. Xu hướng này được kỳ vọng sẽ tiếp tục duy trì trong nhiều năm tới.



Hình 1.10: Sự gia tăng kích thước mạng nơ-ron theo thời gian. Kể từ khi các nút ẩn được giới thiệu, mạng nơ-ron nhân tạo đã tăng gấp đôi quy mô khoảng mỗi 2.4 năm. Dữ liệu mạng nơ-ron sinh học lấy từ Wikipedia (2015).

Các mô hình tiêu biểu theo thời gian trong Hình 1.10 bao gồm:

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow và Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Mạng lan truyền ngược đầu tiên (Rumelhart et al., 1986b)
5. Mạng nơ-ron hồi tiếp cho nhận diện giọng nói (Robinson và Fallside, 1991)
6. Multilayer perceptron cho nhận diện giọng nói (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Echo state network (Jaeger và Haas, 2004)
10. Deep belief network (Hinton et al., 2006)
11. Mạng tích chập dùng GPU (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov và Hinton, 2009a)
13. Deep belief network dùng GPU (Raina et al., 2009)
14. Mạng tích chập không giám sát (Jarrett et al., 2009)
15. MLP dùng GPU (Ciresan et al., 2010)
16. Mạng OMP-1 (Coates và Ng, 2011)
17. Autoencoder phân tán (Le et al., 2012)
18. Mạng tích chập dùng nhiều GPU (Krizhevsky et al., 2012)
19. Mạng tích chập không giám sát trên HPC thương mại (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)

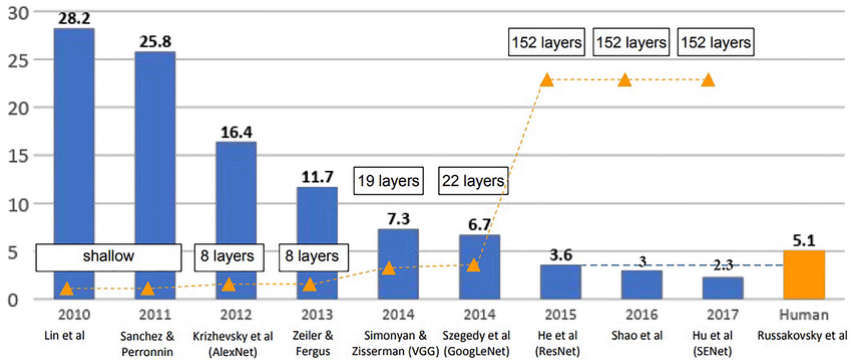
1.2.4 Tăng độ chính xác, độ phức tạp và tác động thực tiễn

Từ những năm 1980 đến nay, học sâu liên tục cải thiện khả năng nhận dạng và dự đoán chính xác. Bên cạnh đó, học sâu cũng được áp dụng ngày càng rộng rãi vào nhiều lĩnh vực khác nhau.

Các mô hình học sâu ban đầu được dùng để nhận diện các vật thể đơn lẻ trong những bức ảnh rất nhỏ, được cắt sát vào đối tượng (Rumelhart et al., 1986a). Theo thời gian, mạng nơ-ron dần xử lý được ảnh có kích thước lớn hơn. Ngày nay, các mạng nhận diện đối tượng hiện đại có thể xử lý ảnh chụp chất lượng cao, giàu thông tin, và không yêu cầu phải cắt ảnh quá sát đối tượng (Krizhevsky et al., 2012).

Không chỉ vậy, các mạng ban đầu chỉ phân biệt được một hoặc hai loại đối tượng, hoặc thậm chí chỉ phân biệt sự có mặt hay vắng mặt của một đối tượng cụ thể. Trong khi đó, các mô hình hiện đại thường nhận diện được ít nhất 1.000 loại đối tượng khác nhau.

Một trong những cuộc thi nhận diện hình ảnh lớn nhất là ImageNet Large Scale Visual



Hình 1.11: Tỷ lệ lỗi giảm dần theo thời gian. Kể từ khi các mạng nơ-ron sâu đạt đến quy mô đủ lớn để cạnh tranh trong cuộc thi ImageNet Large Scale Visual Recognition Challenge (ILSVRC), chúng liên tục giành chiến thắng mỗi năm và kéo theo đó là tỷ lệ lỗi ngày càng giảm.

Recognition Challenge (ILSVRC) được tổ chức hàng năm. Một cột mốc đáng nhớ trong sự phát triển nhanh chóng của học sâu là khi một mạng nơ-ron tích chập (convolutional neural network) lần đầu tiên giành chiến thắng trong cuộc thi này và vượt xa các phương pháp khác. Cụ thể, nó đã hạ tỷ lệ lỗi top-5 từ 26.1% xuống còn 15.3% (Krizhevsky et al., 2012). Điều này có nghĩa là mô hình sẽ đưa ra danh sách 5 nhãn dự đoán cho mỗi ảnh, và nhãn đúng nằm trong danh sách đó với xác suất lên đến 84.7%.

Kể từ đó, các mô hình tích chập sâu liên tục thắng thế trong các cuộc thi này, và cho đến thời điểm viết cuốn sách, tỷ lệ lỗi top-5 đã giảm xuống còn 3.6%, như minh họa trong Hình 1.11.

Học sâu cũng tạo ra bước tiến vượt bậc trong nhận diện giọng nói. Sau nhiều cải tiến trong suốt thập niên 1990, tỷ lệ lỗi trong nhận dạng tiếng nói gần như không thay đổi kể từ năm 2000. Nhưng sự xuất hiện của học sâu (Dahl et al., 2010; Deng et al., 2010b; Seide et al., 2011; Hinton et al., 2012a) đã khiến tỷ lệ lỗi giảm mạnh — có trường hợp giảm còn một nửa. Lịch sử chi tiết sẽ được trình bày thêm trong Mục 12.3.

Các mạng học sâu cũng đã gặt hái thành công lớn trong các bài toán như phát hiện người đi bộ, phân vùng ảnh (image segmentation) (Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013), và thậm chí còn vượt qua con người trong phân loại biển báo giao thông (Ciresan et al., 2012).

Khi độ chính xác và quy mô mô hình tăng lên, độ phức tạp của các tác vụ mà học sâu có thể giải quyết cũng tăng theo. Goodfellow et al. (2014d) chỉ ra rằng mạng nơ-ron có thể học cách tạo ra một chuỗi ký tự từ ảnh đầu vào — tức là giải bài toán nhận dạng chuỗi thay vì chỉ nhận diện một đối tượng đơn lẻ. Trước đây, người ta tin rằng cần phải gán nhãn từng phần tử trong chuỗi thì mới học được bài toán như vậy (Gülçehre và Bengio, 2013).

Ngày nay, các mạng nơ-ron hồi tiếp như mô hình LSTM không chỉ xử lý chuỗi đầu vào cố định, mà còn học mối quan hệ giữa chuỗi đầu vào và chuỗi đầu ra. Kiểu học này — gọi là học chuỗi sang chuỗi (sequence-to-sequence learning) — đang trên đà tạo ra cuộc cách mạng trong lĩnh vực dịch máy (Sutskever et al., 2014; Bahdanau et al., 2015).

Xu hướng tăng độ phức tạp này đã được đẩy đến giới hạn cao hơn nữa với sự ra đời của Neural Turing Machines (NTM) (Graves et al., 2014). Những mô hình này học cách đọc và ghi vào bộ nhớ một cách linh hoạt, cho phép chúng học các chương trình đơn giản từ ví dụ. Ví dụ, NTM có thể học cách sắp xếp một danh sách các số chỉ từ các cặp ví dụ gồm danh sách lộn xộn và danh sách đã sắp xếp. Dù còn rất sơ khai, công nghệ "tự lập trình" này về mặt lý thuyết có thể được ứng dụng cho gần như bất kỳ tác vụ nào trong tương lai.

Một thành tựu nổi bật khác của học sâu là sự mở rộng sang lĩnh vực học tăng cường (reinforcement learning). Trong học tăng cường, một tác nhân (agent) phải tự học cách thực hiện một nhiệm vụ thông qua quá trình thử và sai, mà không có sự hướng dẫn trực tiếp từ con người.

DeepMind đã chứng minh rằng một hệ thống học tăng cường kết hợp với học sâu có thể học cách chơi các trò chơi Atari, đạt hiệu năng ngang tầm con người ở nhiều trò khác nhau (Mnih et al., 2015). Học sâu cũng giúp cải thiện đáng kể hiệu quả của học tăng cường trong điều khiển robot (Finn et al., 2015).

Nhiều ứng dụng của học sâu hiện nay đã trở thành nền tảng sinh lợi lớn. Các công ty công nghệ hàng đầu như Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA và NEC đều đang ứng dụng học sâu vào sản phẩm và dịch vụ của mình.

Sự phát triển của học sâu cũng gắn liền với những tiến bộ trong hạ tầng phần mềm. Các thư viện như `Theano` (Bergstra et al., 2010; Bastien et al., 2012), `PyLearn2` (Goodfellow et al., 2013c), `Torch` (Collobert et al., 2011b), `DistBelief` (Dean et al., 2012), `Caffe` (Jia, 2013), `MXNet` (Chen et al., 2015) và `TensorFlow` (Abadi et al., 2015) đều đóng vai trò quan trọng trong các dự án nghiên cứu và ứng dụng thương mại.

Học sâu cũng đang tạo ảnh hưởng sâu rộng đến nhiều lĩnh vực khoa học khác. Các mạng nơ-ron tích chập hiện đại được sử dụng trong nhận diện đối tượng đang cung cấp mô hình xử lý thị giác mà các nhà thần kinh học có thể nghiên cứu (DiCarlo, 2013). Ngoài ra, học sâu còn là công cụ hữu ích trong việc xử lý lượng lớn dữ liệu khoa học và đưa ra dự đoán có giá trị. Một số ứng dụng nổi bật gồm:

- Dự đoán sự tương tác giữa các phân tử để hỗ trợ thiết kế thuốc mới trong ngành dược (Dahl et al., 2014);
- Tìm kiếm hạt hạ nguyên tử trong vật lý (Baldi et al., 2014);
- Phân tích ảnh hiển vi để xây dựng bản đồ 3D não người (Knowles-Barley et al., 2014).

Chúng tôi tin rằng trong tương lai, học sâu sẽ tiếp tục xuất hiện trong nhiều lĩnh vực khoa học hơn nữa.

Tóm lại, học sâu là một nhánh của học máy, được phát triển từ sự kết hợp giữa kiến thức về não bộ con người, thống kê và toán ứng dụng. Trong những năm gần đây, học sâu đã phát triển mạnh mẽ về độ phổ biến và hiệu quả, phần lớn nhờ vào sự tiến bộ của phần cứng, sự sẵn có của các tập dữ liệu lớn, và các kỹ thuật huấn luyện mạng sâu. Phía trước

là một chặng đường đầy thử thách nhưng cũng rất nhiều cơ hội để tiếp tục cải tiến học sâu và đưa nó đến những lĩnh vực hoàn toàn mới.

CHƯƠNG 2. ĐẠI SỐ TUYẾN TÍNH

Đại số tuyến tính là một nhánh quan trọng của toán học, được ứng dụng rộng rãi trong khoa học và kỹ thuật. Tuy nhiên, do đại số tuyến tính thuộc về toán học liên tục (continuous mathematics) thay vì toán học rời rạc (discrete mathematics), nhiều nhà khoa học máy tính có ít kinh nghiệm với lĩnh vực này. Việc hiểu rõ đại số tuyến tính là rất quan trọng để có thể nắm bắt và làm việc với nhiều thuật toán học máy, đặc biệt là các thuật toán học sâu. Vì vậy, trước khi đi vào phần giới thiệu về học sâu, chúng tôi sẽ trình bày một số kiến thức nền tảng quan trọng về đại số tuyến tính.

Nếu bạn đã quen thuộc với đại số tuyến tính, bạn có thể bỏ qua chương này. Nếu bạn đã từng học qua các khái niệm này nhưng cần một tài liệu tham khảo để xem lại các công thức quan trọng, chúng tôi khuyến nghị cuốn *The Matrix Cookbook* (Petersen and Pedersen, 2006). Nếu bạn chưa từng tiếp xúc với đại số tuyến tính, chương này sẽ cung cấp đủ kiến thức để bạn đọc cuốn sách này, nhưng chúng tôi khuyến nghị bạn nên tham khảo thêm các tài liệu chuyên sâu về đại số tuyến tính như của *Shilov (1977)*. Chương này sẽ lược bỏ các nội dung nâng cao của đại số tuyến tính, chỉ tập trung vào những kiến thức cốt lõi cần thiết để hiểu và áp dụng trong học sâu.

2.1 Số vô hướng, Véc-tơ, Ma trận và Tensor

Đại số tuyến tính gồm các khái niệm chính sau:

- Số vô hướng (Scalar): Một số vô hướng chỉ là một số đơn lẻ, khác với hầu hết các đối tượng khác trong đại số tuyến tính thường bao gồm nhiều số. Các số vô hướng thường được ký hiệu bằng chữ in nghiêng. Thông thường, chúng ta đặt tên cho số vô hướng bằng chữ cái thường, chẳng hạn như s hoặc n. Khi giới thiệu một số vô hướng, ta cần chỉ rõ nó thuộc loại số nào.
  Ví dụ, "Một chiếc xe di chuyển với vận tốc $v = 60$ km/h". Ở đây, v là một số vô hướng đại diện cho tốc độ.

- Véc-tơ (Vector): Một véc-tơ là một mảng số có thứ tự. Mỗi phần tử trong véc-tơ có thể được xác định bởi chỉ số của nó. Các véc-tơ thường được ký hiệu bằng chữ in đậm, ví dụ như $\boldsymbol{x}$. Các phần tử của véc-tơ được ký hiệu bằng chữ in nghiêng với chỉ số con, ví dụ phần tử đầu tiên của $\boldsymbol{x}$ là x_1, phần tử thứ hai là x_2, v.v.

  Nếu mỗi phần tử của véc-tơ là một số thực và véc-tơ có n phần tử, ta nói rằng véc-tơ đó nằm trong không gian $\mathbb{R}^n$. Khi cần viết cụ thể các phần tử của véc-tơ, ta có thể dùng biểu diễn cột:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \tag{2.1}$$

Véc-tơ có thể được hiểu là một điểm trong không gian, trong đó mỗi phần tử tương ứng với một tọa độ theo một trục nhất định.

Đôi khi, chúng ta cần trích xuất một tập hợp phần tử từ một véc-tơ. Trong trường hợp này, ta có thể định nghĩa một tập hợp chứa các chỉ số của các phần tử cần lấy và sử dụng tập hợp đó làm chỉ số con.

Ví dụ, để lấy các phần tử x_1, x_3, x_6 của véc-tơ $\boldsymbol{x}$, ta định nghĩa tập chỉ số:

$$S = \{1, 3, 6\}$$

và viết $\boldsymbol{x}_S$ để biểu diễn véc-tơ con chứa các phần tử tương ứng của $\boldsymbol{x}$.

Ngoài ra, ta cũng có thể sử dụng ký hiệu "$-$"để lấy phần bù của một tập hợp. Chẳng hạn, $\boldsymbol{x}_{-1}$ là véc-tơ chứa tất cả các phần tử của $\boldsymbol{x}$ ngoại trừ x_1. Tương tự, $\boldsymbol{x}_{-S}$ là véc-tơ chứa tất cả các phần tử của $\boldsymbol{x}$ ngoại trừ x_1, x_3, x_6.

- Ma trận (Matrix). Một ma trận là một mảng hai chiều chứa các số, trong đó mỗi phần tử được xác định bởi hai chỉ số thay vì một như véc-tơ. Chúng ta thường ký hiệu ma trận bằng chữ cái in đậm, chẳng hạn như $\mathbf{A}$. Nếu một ma trận số thực có chiều cao m và chiều rộng n, ta nói rằng $\mathbf{A} \in \mathbb{R}^{m \times n}$.

  Các phần tử của ma trận được ký hiệu bằng chữ in nghiêng nhưng không in đậm, với các chỉ số phân tách bằng dấu phẩy. Ví dụ, $A_{1,1}$ là phần tử ở góc trên bên trái của ma trận, trong khi $A_{m,n}$ là phần tử ở góc dưới bên phải.

  Ta có thể chọn toàn bộ các phần tử có tọa độ hàng i bằng cách sử dụng ký hiệu dấu hai chấm " : " ở tọa độ cột. Ví dụ, $A_{i,:}$ biểu diễn hàng thứ i của $\mathbf{A}$. Tương tự, ta có thể lấy toàn bộ cột thứ j bằng cách viết $A_{:,j}$. Khi cần hiển thị toàn bộ các phần tử của ma trận, ta có thể viết chúng dưới dạng một mảng ma trận được bao trong dấu ngoặc vuông:

  $$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}. \tag{2.2}$$

  Đôi khi, ta cần làm việc với các biểu thức ma trận phức tạp hơn thay vì chỉ một chữ cái. Trong những trường hợp này, ta có thể sử dụng chỉ số dưới sau biểu thức mà không làm thay đổi kiểu chữ. Ví dụ, $f(\mathbf{A})_{i,j}$ biểu diễn phần tử (i, j) của ma trận thu được sau khi áp dụng hàm f lên $\mathbf{A}$.

- Tensor. Trong một số trường hợp, ta cần làm việc với mảng có nhiều hơn hai chiều. Một mảng số được sắp xếp trên một lưới đều với số chiều bất kỳ được gọi là một *tensor*. Ta ký hiệu tensor bằng chữ in đậm, ví dụ $\mathbf{A}$. Mỗi phần tử của tensor được xác định bởi nhiều chỉ số. Nếu $\mathbf{A}$ là một tensor ba chiều, phần tử của nó tại tọa độ (i, j, k) được ký hiệu là $\mathsf{A}_{i,j,k}$.

Một phép toán quan trọng trên ma trận là phép chuyển vị (transpose). Chuyển vị của

một ma trận chính là hình ảnh phản chiếu của ma trận qua một đường chéo, được gọi là đường chéo chính, chạy từ góc trên bên trái xuống dưới bên phải. Xem ví dụ 1.5 để có cái nhìn trực quan về phép toán này. Chúng ta ký hiệu chuyển vị của ma trận $\mathbf{A}$ là $\mathbf{A}^T$, và định nghĩa như sau:

$$(\mathbf{A}^T)_{i,j} = \mathbf{A}_{j,i}. \tag{2.3}$$

Vectors có thể được xem như ma trận chỉ có một cột. Do đó, chuyển vị của một vector là một ma trận chỉ có một hàng. Chẳng hạn, ta có thể viết một vector dưới dạng ma trận hàng và dùng phép chuyển vị để biến nó thành ma trận cột:

$$\mathbf{x} = [x_1, x_2, x_3]^T. \tag{2.4}$$

Một số ví dụ về phép chuyển vị:

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \Rightarrow \mathbf{A}^T = \begin{bmatrix} A_{1,1} & A_{2,1} \\ A_{1,2} & A_{2,2} \end{bmatrix}. \tag{2.5}$$

Một số vô hướng có thể được xem như một ma trận chỉ chứa một phần tử duy nhất. Từ đây, ta có thể thấy rằng một số vô hướng chính là chuyển vị của chính nó:

$$a = a^\top. \tag{2.6}$$

Ta có thể cộng hai ma trận với nhau miễn là chúng có cùng kích thước, bằng cách cộng từng phần tử tương ứng:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \quad \text{hay} \quad C_{i,j} = A_{i,j} + B_{i,j}. \tag{2.7}$$

Ngoài ra, ta cũng có thể cộng một số vô hướng vào một ma trận hoặc nhân một ma trận với một số vô hướng bằng cách thực hiện phép toán này trên từng phần tử của ma trận:

$$\mathbf{D} = a \cdot \mathbf{B} + c \quad \text{hay} \quad D_{i,j} = a \cdot B_{i,j} + c. \tag{2.8}$$

Trong bối cảnh học sâu, ta thường sử dụng một số quy ước ít phổ biến hơn. Chẳng hạn, ta có thể cộng một ma trận với một vectơ, tạo thành một ma trận mới:

$$\mathbf{C} = \mathbf{A} + \mathbf{b}, \quad \text{hay} \quad C_{i,j} = A_{i,j} + b_j. \tag{2.9}$$

Nói cách khác, vectơ $\mathbf{b}$ được cộng vào từng hàng của ma trận. Ký hiệu này giúp ta tránh phải định nghĩa một ma trận có $\mathbf{b}$ được sao chép vào từng hàng trước khi thực hiện phép

cộng. Việc sao chép tự động này của $\mathbf{b}$ đến nhiều vị trí được gọi là broadcasting.

2.2 Nhân Ma Trận và Véc-tơ

Một trong những phép toán quan trọng nhất liên quan đến ma trận là phép nhân hai ma trận. Phép nhân ma trận của hai ma trận $\mathbf{A}$ và $\mathbf{B}$ tạo ra một ma trận thứ ba $\mathbf{C}$. Để phép nhân này có nghĩa thì số cột của $\mathbf{A}$ phải bằng số hàng của $\mathbf{B}$. Nếu $\mathbf{A}$ có kích thước $m \times n$ và $\mathbf{B}$ có kích thước $n \times p$, thì $\mathbf{C}$ sẽ có kích thước $m \times p$. Chúng ta có thể viết phép nhân ma trận đơn giản bằng cách đặt hai hoặc nhiều ma trận cạnh nhau, chẳng hạn:

$$\mathbf{C} = \mathbf{AB}. \tag{2.10}$$

Phép toán nhân này được định nghĩa như sau:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}. \tag{2.11}$$

Lưu ý rằng tích thông thường của hai ma trận không đơn giản chỉ là phép nhân từng phần tử tương ứng. Một phép toán như vậy tồn tại và được gọi là tích từng phần tử (element-wise product) hoặc tích Hadamard, và được ký hiệu là $\mathbf{A} \odot \mathbf{B}$.

Tích vô hướng (dot product) giữa hai véc-tơ $\boldsymbol{x}$ và $\boldsymbol{y}$ có cùng số chiều chính là tích ma trận $\boldsymbol{x}^T\boldsymbol{y}$. Chúng ta có thể hiểu tích ma trận $\mathbf{C} = \mathbf{AB}$ như một cách tính tích vô hướng giữa hàng thứ i của $\mathbf{A}$ và cột thứ j của $\mathbf{B}$.

Phép nhân ma trận có nhiều tính chất hữu ích giúp việc phân tích toán học trở nên thuận tiện hơn. Ví dụ, phép nhân ma trận có tính chất phân phối:

$$A(B + C) = AB + AC. \tag{2.12}$$

Nó cũng có tính chất kết hợp:

$$A(BC) = (AB)C. \tag{2.13}$$

Tuy nhiên, phép nhân ma trận không có tính giao hoán, tức là thông thường $AB \neq BA$, không giống như phép nhân số vô hướng. Ngược lại, tích vô hướng giữa hai véc-tơ lại có tính giao hoán:

$$\boldsymbol{x}^\top \boldsymbol{y} = \boldsymbol{y}^\top \boldsymbol{x}. \tag{2.14}$$

Ngoài ra, phép chuyển vị của tích hai ma trận có dạng đơn giản:

$$(AB)^\top = B^\top A^\top. \tag{2.15}$$

Dựa vào điều này, ta có thể chứngbfTích vô hướng (dot product) giữa hai véc-tơ $\boldsymbol{x}$ và $\boldsymbol{y}$ có cùng số chiều chính là tích ma trận $\boldsymbol{x}^T\boldsymbol{y}$. Chúng ta có thể hiểu tích ma trận $\mathbf{C} = \mathbf{AB}$ như một cách tính tích vô hướng giữa hàng thứ i của $\mathbf{A}$ và cột thứ j của $\mathbf{B}$.

Phép nhân ma trận có nhiều tính chất hữu ích giúp việc phân tích toán học trở nên thuận tiện hơn. Ví dụ, phép nhân ma trận có tính chất phân phối:

$$A(B + C) = AB + AC. \tag{2.16}$$

Nó cũng có tính chất kết hợp:

$$A(BC) = (AB)C. \tag{2.17}$$

Tuy nhiên, phép nhân ma trận không có tính giao hoán, tức là thông thường $AB \neq BA$, không giống như phép nhân số vô hướng. Ngược lại, tích vô hướng giữa hai véc-tơ lại có tính giao hoán:

$$\boldsymbol{x}^\top\boldsymbol{y} = \boldsymbol{y}^\top\boldsymbol{x}. \tag{2.18}$$

Ngoài ra, phép chuyển vị của tích hai ma trận có dạng đơn giản:

$$(AB)^\top = B^\top A^\top. \tag{2.19}$$

Dựa vào điều này, ta có thể chứng rằng tích vô hướng của hai véc-tơ là một đại lượng vô hướng, và do đó nó bằng với chuyển vị của chính nó:

$$\boldsymbol{x}^\top\boldsymbol{y} = (\boldsymbol{x}^\top\boldsymbol{y})^\top = \boldsymbol{y}^\top\boldsymbol{x}. \tag{2.20}$$

Vì quyển sách này không tập trung vào đại số tuyến tính, chúng tôi không trình bày toàn bộ các tính chất của phép nhân ma trận mà chỉ đề cập một số tính chất quan trọng. Người đọc nên biết rằng còn nhiều tính chất khác của ma trận.

Bây giờ, chúng ta đã có đủ kiến thức về đại số tuyến tính để viết một hệ phương trình tuyến tính:

$$A\boldsymbol{x} = \mathbf{b} \tag{2.21}$$

trong đó:

- $A \in \mathbb{R}^{m\times n}$ là một ma trận đã biết.
- $\mathbf{b} \in \mathbb{R}^m$ là một véc-tơ đã biết.
- $\boldsymbol{x} \in \mathbb{R}^n$ là một véc-tơ chứa các biến chưa biết mà ta cần tìm.

Mỗi phần tử x_i trong véc-tơ $\boldsymbol{x}$ là một biến ẩn. Mỗi hàng của ma trận A và mỗi phần tử của $\mathbf{b}$ tạo thành một ràng buộc khác nhau. Ta có thể viết lại hệ phương trình trên dưới dạng:

$$A_1\boldsymbol{x} = b_1 \tag{2.22}$$

$$A_2\boldsymbol{x} = b_2 \tag{2.23}$$

$$\dots$$

$$A_m\boldsymbol{x} = b_m \tag{2.24}$$

hoặc dưới dạng biểu thức cụ thể hơn:

$$A_{1,1}x_1 + A_{1,2}x_2 + \cdots + A_{1,n}x_n = b_1. \tag{2.25}$$

$$A_{2,1}x_1 + A_{2,2}x_2 + \cdots + A_{2,n}x_n = b_2 \tag{2.26}$$

$$\dots$$

$$A_{m,1}x_1 + A_{m,2}x_2 + \cdots + A_{m,n}x_n = b_m. \tag{2.27}$$

2.3 Ma trận Đơn vị và Ma trận Nghịch đảo

Đại số tuyến tính cung cấp một phương pháp mạnh mẽ gọi là nghịch đảo ma trận, giúp chúng ta giải phương trình 1.21 một cách chính xác cho nhiều giá trị của A.

Để mô tả nghịch đảo ma trận, trước tiên ta cần định nghĩa khái niệm về ma trận đơn vị. Ma trận đơn vị là ma trận không làm thay đổi bất kỳ véc-tơ nào khi nhân với nó. Ta ký hiệu ma trận đơn vị kích thước $n \times n$ là $\boldsymbol{I}_n$. Về mặt toán học, nó thỏa mãn điều kiện:

$$forallx \in \mathbb{R}^n, \quad \boldsymbol{I}_n x = x. \tag{2.28}$$

Cấu trúc của ma trận đơn vị rất đơn giản: tất cả các phần tử trên đường chéo chính đều bằng 1, trong khi các phần tử còn lại đều bằng 0. Ví dụ:

$$\boldsymbol{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.29}$$

Ma trận nghịch đảo của A được ký hiệu là A^{-1}, và nó được định nghĩa là ma trận thỏa mãn điều kiện:

$$A^{-1}A = \boldsymbol{I}_n. \tag{2.30}$$

Dựa vào tính chất này, ta có thể giải phương trình 1.21 theo các bước sau:

$$Ax = b, \tag{2.31}$$

$$A^{-1}Ax = A^{-1}b, \tag{2.32}$$

$$\boldsymbol{I}_n x = A^{-1}b. \tag{2.33}$$

Suy ra nghiệm của hệ phương trình:

$$x = A^{-1}b. \tag{2.34}$$

Tuy nhiên, quá trình này phụ thuộc vào việc tìm được A^{-1}. Chúng ta sẽ thảo luận điều kiện tồn tại của A^{-1} trong phần tiếp theo.

Khi A^{-1} tồn tại, có nhiều thuật toán có thể tính nó một cách tường minh. Trên lý thuyết, ma trận nghịch đảo có thể được dùng để giải phương trình cho nhiều giá trị khác nhau của b. Tuy nhiên, trên thực tế, A^{-1} chủ yếu có giá trị trong lý thuyết và không nên được sử dụng trực tiếp trong hầu hết các ứng dụng phần mềm, vì việc tính toán nó thường không ổn định về mặt số học. Do đó, trong thực tế, các thuật toán giải hệ phương trình thường không tính trực tiếp A^{-1} mà thay vào đó sử dụng các phương pháp khác để tìm nghiệm gần đúng của x.

2.4 Sự phụ thuộc tuyến tính và Không gian sinh

Để ma trận A^{-1} tồn tại, phương trình 1.21 phải có đúng một nghiệm duy nhất với mọi giá trị của $\mathbf{b}$. Hệ phương trình cũng có thể không có nghiệm hoặc có vô số nghiệm đối với một số giá trị của $\mathbf{b}$. Tuy nhiên, nếu một hệ phương trình có nhiều hơn một nghiệm nhưng không vô hạn nghiệm với một giá trị cụ thể của $\boldsymbol{b}$, thì nếu $\boldsymbol{x}$ và $\boldsymbol{y}$ đều là nghiệm, ta có:

$$\boldsymbol{z} = \alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y} \tag{2.35}$$

cũng là một nghiệm với mọi giá trị thực α.

Để phân tích số lượng nghiệm của phương trình, hãy xem các cột của ma trận A như các hướng di chuyển từ gốc tọa độ (điểm được biểu diễn bởi véc-tơ toàn số 0). Khi đó, ta cần xác định có bao nhiêu cách để đến điểm $\boldsymbol{b}$.

Với quan điểm này, mỗi phần tử của véc-tơ $\boldsymbol{x}$ xác định mức độ di chuyển theo từng hướng, trong đó x_i chỉ định mức độ di chuyển theo hướng của cột i:

$$A\boldsymbol{x} = \sum_i x_i A_{:,i}. \tag{2.36}$$

Nói chung, phép toán này được gọi là tổ hợp tuyến tính (*linear combination*). Một tổ hợp tuyến tính của một tập hợp các véc-tơ $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ được xác định bằng cách nhân

từng véc-tơ $\boldsymbol{v}^{(i)}$ với một hệ số vô hướng tương ứng và cộng lại:

$$\sum_i c_i \boldsymbol{v}^{(i)}. \tag{2.37}$$

Không gian sinh (*span*) của một tập hợp véc-tơ là tập hợp tất cả các điểm có thể đạt được bằng tổ hợp tuyến tính của các véc-tơ ban đầu.

Xác định xem phương trình $\boldsymbol{Ax} = \boldsymbol{b}$ có nghiệm hay không thực chất là kiểm tra xem véc-tơ $\boldsymbol{b}$ có thuộc không gian cột của ma trận $\boldsymbol{A}$ hay không. Không gian này còn được gọi là tập giá trị (range) của $\boldsymbol{A}$.

Để hệ phương trình $\boldsymbol{Ax} = \boldsymbol{b}$ có nghiệm với mọi giá trị của $\boldsymbol{b}$ trong $\mathbb{R}^m$, ta cần yêu cầu không gian cột của $\boldsymbol{A}$ phải bao phủ toàn bộ không gian $\mathbb{R}^m$. Nếu có một điểm nào đó của $\mathbb{R}^m$ không thuộc không gian cột, thì điểm đó sẽ là một giá trị $\boldsymbol{b}$ mà hệ phương trình không có nghiệm. Điều này có nghĩa là $\boldsymbol{A}$ phải có ít nhất m cột, tức là $n \geq m$. Nếu không, không gian cột sẽ có số chiều nhỏ hơn m và không thể bao phủ toàn bộ $\mathbb{R}^m$.

Ví dụ, xét một ma trận 3×2, véc-tơ $\mathbf{b}$ thuộc không gian 3D, trong khi véc-tơ ẩn số $\mathbf{x}$ thuộc không gian 2D. Khi đó, thay đổi giá trị của $\mathbf{x}$ chỉ giúp ta sinh ra một mặt phẳng 2D trong không gian 3D. Hệ phương trình chỉ có nghiệm khi $\mathbf{b}$ nằm trên mặt phẳng đó.

Giả sử ta có hệ phương trình tuyến tính:

$$A\mathbf{x} = \mathbf{b}$$

với ma trận A kích thước 3×2:

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix}$$

Véc-tơ ẩn số $\mathbf{x}$ thuộc $\mathbb{R}^2$:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

và véc-tơ $\mathbf{b}$ thuộc $\mathbb{R}^3$:

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Do A có 2 cột, không gian nghiệm của phương trình được sinh bởi hai véc-tơ cột của

A. Khi đó, hệ phương trình có thể được viết lại dưới dạng:

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Điều này có nghĩa là tập hợp tất cả các giá trị có thể có của $\mathbf{b}$ tạo thành một mặt phẳng 2D trong không gian 3D. Hệ phương trình chỉ có nghiệm khi $\mathbf{b}$ nằm trên mặt phẳng này.

Điều kiện $n \geq m$ là điều kiện cần để mỗi giá trị của $\boldsymbol{b}$ đều có nghiệm, nhưng không phải là điều kiện đủ. Một số cột trong $\boldsymbol{A}$ có thể dư thừa, làm giảm số chiều của không gian cột.

Hiện tượng dư thừa này được gọi là sự phụ thuộc tuyến tính (*linear dependence*). Một tập hợp véc-tơ được gọi là độc lập tuyến tính (*linearly independent*) nếu không có véc-tơ nào trong tập hợp có thể biểu diễn thành tổ hợp tuyến tính của các véc-tơ còn lại. Nếu ta thêm một véc-tơ mới nhưng nó chỉ là tổ hợp của các véc-tơ đã có, thì véc-tơ mới không làm thay đổi không gian sinh của tập hợp.

Điều này có nghĩa là ma trận cần phải là ma trận vuông, tức là yêu cầu $m = n$ và tất cả các cột của ma trận phải độc lập tuyến tính. Một ma trận vuông có các cột phụ thuộc tuyến tính được gọi là ma trận suy biến.

Nếu A không phải là ma trận vuông hoặc là ma trận vuông nhưng suy biến, ta vẫn có thể giải hệ phương trình, nhưng không thể sử dụng phương pháp nghịch đảo ma trận để tìm nghiệm.

Nếu ma trận A không phải là ma trận vuông hoặc là một ma trận suy biến, hệ phương trình tuyến tính vẫn có thể có nghiệm, nhưng ta không thể tìm nghiệm bằng phương pháp nghịch đảo ma trận. Thay vào đó, ta phải sử dụng các phương pháp khác như phương pháp khử Gauss hoặc phân rã ma trận để tìm nghiệm.

Cho đến nay, ta đã thảo luận về nghịch đảo của ma trận khi nhân ở bên trái. Tuy nhiên, cũng có thể định nghĩa một nghịch đảo khi nhân ở bên phải:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{2.38}$$

Đối với ma trận vuông, nghịch đảo trái và nghịch đảo phải là như nhau.

2.5 Chuẩn của véc-tơ

Trong nhiều trường hợp, ta cần đo kích thước của một véc-tơ. Trong học máy, ta thường sử dụng một hàm gọi là chuẩn (norm) để đo độ lớn của véc-tơ. Chuẩn L^p được định nghĩa bởi:

$$\|\boldsymbol{x}\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}} \tag{2.39}$$

với $p \in \mathbb{R}, p \geq 1$.

Các chuẩn, bao gồm cả L^p, là các hàm ánh xạ véc-tơ vào các giá trị không âm. Ở mức độ trực quan, chuẩn của một véc-tơ đo khoảng cách từ gốc tọa độ đến điểm $\mathbf{x}$. Một cách chặt chẽ hơn, một hàm được gọi là chuẩn nếu thỏa mãn ba tính chất sau:

- $f(\boldsymbol{x}) = 0 \Leftrightarrow \boldsymbol{x} = 0$
- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$ (*bất đẳng thức tam giác*)
- $forall\alpha \in \mathbb{R}, f(\alpha\boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

Chuẩn L^2, với $p = 2$, được gọi là chuẩn Euclid (Euclidean norm), đơn giản chính là khoảng cách Euclid từ gốc tọa độ đến điểm $\boldsymbol{x}$. Vì chuẩn L^2 được sử dụng rất phổ biến trong học máy, nó thường được ký hiệu đơn giản là $\|\boldsymbol{x}\|$ mà không cần ghi chỉ số 2. Ngoài ra, một cách khác để đo kích thước của một véc-tơ là sử dụng chuẩn bình phương L^2, có thể tính bằng:

$$\boldsymbol{x}^T\boldsymbol{x}. \tag{2.40}$$

Chuẩn bình phương L^2 thường thuận tiện hơn trong tính toán so với chuẩn L^2 do tính chất đạo hàm. Cụ thể, đạo hàm của chuẩn bình phương L^2 theo từng phần tử của $\boldsymbol{x}$ chỉ phụ thuộc vào chính phần tử đó, trong khi đạo hàm của chuẩn L^2 phụ thuộc vào toàn bộ véc-tơ. Trong một số trường hợp, chuẩn bình phương L^2 có thể gây ra bất lợi do nó tăng nhanh gần gốc tọa độ.

Trong nhiều ứng dụng học máy, ta cần phân biệt giữa các phần tử có giá trị bằng 0 và các phần tử có giá trị rất nhỏ nhưng khác 0. Khi đó, ta sử dụng một chuẩn có tốc độ tăng đều trên toàn bộ miền giá trị, đó là chuẩn L^1. Chuẩn này có dạng đơn giản hơn:

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i|. \tag{2.41}$$

Chuẩn L^1 thường được sử dụng trong học máy khi sự khác biệt giữa các phần tử bằng không và các phần tử khác không rất quan trọng. Mỗi khi một phần tử của $\boldsymbol{x}$ dịch chuyển khỏi 0 một lượng ϵ, chuẩn L^1 sẽ tăng thêm ϵ.

Chúng ta đôi khi đo độ lớn của một véc-tơ bằng cách đếm số phần tử khác không của nó. Một số tác giả gọi đây là "chuẩn L^0", nhưng thuật ngữ này không chính xác. Số phần tử khác không trong một véc-tơ không phải là một chuẩn, vì nhân véc-tơ với một số vô hướng α không làm thay đổi số phần tử khác không. Do đó, chuẩn L^1 thường được sử dụng như một phương án thay thế để đo lường số lượng phần tử khác không.

Một loại chuẩn khác thường xuất hiện trong học máy là chuẩn L^∞, hay còn gọi là chuẩn vô hạn (max norm). Chuẩn này đơn giản là giá trị tuyệt đối lớn nhất trong véc-tơ:

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i|. \tag{2.42}$$

Trong một số trường hợp, chúng ta cũng cần đo kích thước của một ma trận. Trong lĩnh vực học sâu, một cách phổ biến để làm điều này là sử dụng chuẩn Frobenius:

$$\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}, \tag{2.43}$$

chuẩn này tương tự với chuẩn L^2 của một véc-tơ.

Tích vô hướng của hai véc-tơ có thể được viết lại dưới dạng các chuẩn:

$$\boldsymbol{x}^T\boldsymbol{y} = \|\boldsymbol{x}\|_2\|\boldsymbol{y}\|_2 \cos\theta, \tag{2.44}$$

trong đó θ là góc giữa hai véc-tơ $\boldsymbol{x}$ và $\boldsymbol{y}$.

2.6 Các ma trận và véc-tơ đặc biệt

Một số loại ma trận và véc-tơ đặc biệt rất hữu ích trong tính toán.

Ma trận đường chéo (diagonal matrix) chủ yếu chứa các phần tử bằng 0 và chỉ có các phần tử khác 0 nằm trên đường chéo chính. Nói một cách chính xác, một ma trận D được gọi là đường chéo nếu và chỉ nếu $D_{ij} = 0$ với mọi $i \neq j$. Một ví dụ quen thuộc về ma trận đường chéo là ma trận đơn vị, trong đó tất cả các phần tử trên đường chéo chính đều bằng 1. Chúng ta ký hiệu $\mathrm{diag}(\boldsymbol{v})$ để biểu diễn một ma trận đường chéo có các phần tử trên đường chéo chính được lấy từ véc-tơ $\boldsymbol{v}$.

Nhân một véc-tơ với một ma trận đường chéo rất hiệu quả về mặt tính toán, vì ta chỉ cần nhân từng phần tử của véc-tơ với phần tử tương ứng trên đường chéo. Trong trường hợp $\mathrm{diag}(\boldsymbol{v})\boldsymbol{x} = \boldsymbol{v} \odot \mathbf{x}$ (trong đó $\odot$ là phép nhân từng phần tử).

Đảo ma trận đường chéo cũng rất dễ thực hiện, nếu tất cả các phần tử trên đường chéo khác 0. Khi đó, $\mathrm{diag}(\mathbf{v})^{-1} = \mathrm{diag}\left(\left[\frac{1}{v_1}, \ldots, \frac{1}{v_n}\right]^\top\right)$. Vì vậy, trong nhiều thuật toán, ta thường giới hạn bài toán chỉ xét các ma trận đường chéo để giảm chi phí tính toán.

Không phải tất cả các ma trận đường chéo đều là ma trận vuông. Ta có thể xây dựng một ma trận đường chéo hình chữ nhật. Tuy nhiên, trong nhiều trường hợp, chúng không có ma trận nghịch đảo, nhưng vẫn có thể thực hiện một số phép toán hữu ích. Ví dụ, với một ma trận đường chéo không vuông D, ta có thể nhân với một véc-tơ $\boldsymbol{x}$ bằng cách nhân từng phần tử tương ứng, hoặc có thể thêm hoặc loại bỏ các phần tử của kết quả tùy theo kích thước của D.

Ma trận đối xứng (symmetric matrix) là ma trận thỏa mãn điều kiện:

$$A = A^T. \tag{2.45}$$

Ma trận đối xứng thường xuất hiện khi các phần tử trong ma trận đại diện cho một hàm khoảng cách giữa hai đối tượng không phụ thuộc vào thứ tự. Ví dụ, nếu A là ma trận khoảng cách giữa các điểm, với A_{ij} là khoảng cách từ điểm i đến điểm j, thì $A_{ij} = A_{ji}$ do khoảng cách là một đại lượng đối xứng.

Véc-tơ đơn vị (unit vector) là véc-tơ có chuẩn bằng 1:

$$\|\boldsymbol{x}\|_2 = 1. \tag{2.46}$$

Hai véc-tơ $\boldsymbol{x}$ và $\boldsymbol{y}$ được gọi là trực giao (orthogonal) nếu tích vô hướng của chúng bằng 0:

$$\boldsymbol{x}^T\boldsymbol{y} = 0. \tag{2.47}$$

Nếu cả hai véc-tơ đều khác 0, điều này có nghĩa là chúng tạo với nhau một góc 90 độ. Trong không gian $\mathbb{R}^n$, một tập hợp các véc-tơ có thể trực giao với nhau nhưng không nhất thiết phải có chuẩn đơn vị. Nếu chúng vừa trực giao vừa có chuẩn bằng 1, ta gọi chúng là trực giao chuẩn (orthonormal).

Ma trận trực giao (orthogonal matrix) là ma trận vuông có các hàng và các cột đều trực giao với nhau và đều có chuẩn đơn vị:

$$A^TA = AA^T = I. \tag{2.48}$$

Điều này dẫn đến hệ quả:

$$A^{-1} = A^T. \tag{2.49}$$

Vì vậy, các ma trận trực giao rất quan trọng, vì ma trận nghịch đảo của chúng có thể được tính toán một cách dễ dàng bằng cách chỉ cần chuyển vị. Tuy nhiên, cần chú ý rằng một ma trận có hàng trực giao không nhất thiết phải có cột trực giao, và ngược lại. Khi đó, ma trận không phải là trực giao hoàn toàn mà chỉ có tính chất trực giao một phần (semi-orthogonal).

2.7 Phân rã giá trị riêng

Nhiều đối tượng toán học có thể được hiểu rõ hơn bằng cách chia nhỏ chúng thành các thành phần nhỏ hơn hoặc tìm ra một số tính chất chung không phụ thuộc vào cách chúng được biểu diễn.

Ví dụ, các số nguyên có thể được phân tích thành thừa số nguyên tố. Cách biểu diễn số 12 có thể khác nhau tùy theo cơ số (thập phân hoặc nhị phân), nhưng luôn đúng rằng $12 = 2 \times 2 \times 3$. Từ cách biểu diễn này, ta có thể suy ra nhiều tính chất hữu ích, chẳng hạn

như 12 không chia hết cho 5 nhưng bất kỳ bội số nào của 12 cũng sẽ chia hết cho 3.

Tương tự, chúng ta có thể tìm hiểu về bản chất của ma trận bằng cách phân rã chúng. Một trong những phương pháp phân rã quan trọng nhất là phân rã giá trị riêng (eigendecomposition), trong đó ta phân tách ma trận thành các véc-tơ riêng và giá trị riêng.

Một véc-tơ riêng của ma trận vuông $\mathbf{A}$ là một véc-tơ khác không $\boldsymbol{v}$ sao cho phép nhân với $\boldsymbol{A}$ chỉ làm thay đổi độ lớn của $\boldsymbol{v}$:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}. \tag{2.50}$$

Hệ số λ được gọi là giá trị riêng tương ứng với véc-tơ riêng này. Ta cũng có thể định nghĩa véc-tơ riêng trái, tức là véc-tơ hàng $\boldsymbol{v}^\top$ sao cho:

$$\boldsymbol{v}^\top \mathbf{A} = \lambda \boldsymbol{v}^\top. \tag{2.51}$$

Tuy nhiên, trong hầu hết các trường hợp, ta chỉ quan tâm đến véc-tơ riêng phải.

Nếu $\boldsymbol{v}$ là một véc-tơ riêng của $\mathbf{A}$, thì bất kỳ véc-tơ nào có dạng $s\boldsymbol{v}$ với $s \neq 0$ cũng là véc-tơ riêng của $\mathbf{A}$ với cùng giá trị riêng λ. Do đó, ta thường chỉ xét các véc-tơ riêng đơn vị (đã chuẩn hóa).

Giả sử ma trận $\mathbf{A}$ có n véc-tơ riêng độc lập tuyến tính $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ với các giá trị riêng tương ứng $\{\lambda_1, \ldots, \lambda_n\}$. Khi đó, ta có thể tập hợp các véc-tơ riêng thành một ma trận $\mathbf{V}$ với mỗi cột là một véc-tơ riêng:

$$\mathbf{V} = [\boldsymbol{v}^{(1)}, \ldots, \mathbf{v}^{(n)}]. \tag{2.52}$$

Tương tự, ta có thể tập hợp các giá trị riêng thành một véc-tơ $\lambda = [\lambda_1, \ldots, \lambda_n]$ hoặc thành một ma trận đường chéo:

$$\mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_n). \tag{2.53}$$

Với các định nghĩa trên, ta có thể biểu diễn ma trận $\mathbf{A}$ dưới dạng:

$$\mathbf{A} = \mathbf{V} \operatorname{diag}(\lambda) \mathbf{V}^{-1}. \tag{2.54}$$

2.8 Phân rã giá trị suy biến

Trong mục 1.7, chúng ta đã tìm hiểu cách phân rã một ma trận thành các trị riêng và vector riêng. Phân rã giá trị suy biến (Singular Value Decomposition - SVD) cung cấp một cách khác để phân tích ma trận thành các vector suy biến và giá trị suy biến. SVD giúp khám phá những thông tin tương tự như phân rã trị riêng, nhưng có phạm vi áp dụng rộng hơn. Mọi ma trận thực đều có phân rã giá trị suy biến, trong khi điều đó không đúng với phân rã trị riêng.

Chẳng hạn, nếu một ma trận không vuông, việc phân rã trị riêng không được xác định, và ta phải sử dụng phân rã giá trị suy biến thay thế.

Hãy nhớ lại rằng phân rã trị riêng liên quan đến việc phân tích một ma trận $\mathbf{A}$ để tìm một ma trận $\mathbf{V}$ chứa các vector riêng và một vector $\boldsymbol{\lambda}$ chứa các trị riêng sao cho:

$$\mathbf{A} = \mathbf{V}\,\mathrm{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1}. \tag{2.55}$$

Phân rã giá trị suy biến có dạng tương tự, nhưng lần này ta viết $\mathbf{A}$ dưới dạng tích của ba ma trận:

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\top}. \tag{2.56}$$

Giả sử rằng A là một ma trận kích thước $m \times n$. Khi đó, ta có:

- U là ma trận kích thước $m \times m$,
- D là ma trận kích thước $m \times n$,
- V là ma trận kích thước $n \times n$.

Các ma trận này có những tính chất đặc biệt:

- U và V là ma trận trực giao.
- D là ma trận đường chéo (nhưng không nhất thiết phải là ma trận vuông).

Các phần tử trên đường chéo của D được gọi là giá trị suy biến (singular values) của ma trận A. Các cột của U được gọi là vector suy biến trái (left-singular vectors), trong khi các cột của V được gọi là vector suy biến phải (right-singular vectors).

SVD có thể được hiểu thông qua phân rã trị riêng (eigendecomposition) của ma trận A:

- Vector suy biến trái của A là các vector riêng của $AA^{\top}$.
- Vector suy biến phải của A là các vector riêng của $A^{\top}A$.
- Các giá trị suy biến không bằng 0 của A chính là căn bậc hai của các trị riêng của $A^{\top}A$ (hoặc $AA^{\top}$).

Một ứng dụng quan trọng của SVD là giúp ta có thể xấp xỉ nghịch đảo của một ma trận không vuông.

2.9 Giả nghịch đảo Moore-Penrose

Nghịch đảo ma trận thông thường không được xác định cho các ma trận không vuông. Giả sử ta muốn tìm một ma trận nghịch đảo bên trái B của ma trận A để giải phương trình tuyến tính:

$$Ax = y \tag{2.57}$$

Nhân cả hai vế với B từ bên trái, ta có:

$$x = By. \tag{2.58}$$

Tuy nhiên, tùy vào cấu trúc của bài toán, có thể không thiết lập được một ánh xạ duy nhất từ A sang B:

- Nếu A có nhiều hàng hơn cột ($m > n$), phương trình có thể không có nghiệm.
- Nếu A có nhiều cột hơn hàng ($m < n$), phương trình có thể có vô số nghiệm.

Giả nghịch đảo Moore-Penrose giúp ta xử lý những trường hợp này. Nghịch đảo giả của A, ký hiệu A^+, được xác định theo công thức:

$$A^+ = \lim_{\alpha \to 0} (A^\top A + \alpha I)^{-1} A^\top. \tag{2.59}$$

Trong thực tế, các thuật toán tính nghịch đảo giả thường không sử dụng công thức này mà thay vào đó dựa trên phân rã giá trị suy biến (SVD):

$$A^+ = V D^+ U^\top, \tag{2.60}$$

trong đó:

- U, D, V là các ma trận từ phân rã SVD của A,
- D^+ là nghịch đảo giả của ma trận đường chéo D, thu được bằng cách lấy nghịch đảo của các phần tử khác không trên đường chéo rồi chuyển vị.

Nếu A có nhiều hàng hơn cột ($m > n$), lời giải $x = A^+ y$ có chuẩn Euclid nhỏ nhất, tức là trong tất cả các nghiệm, đây là nghiệm có độ dài ngắn nhất.

Nếu A có nhiều cột hơn hàng ($m < n$), phương trình có thể không có nghiệm chính xác. Khi đó, nghiệm $x = A^+ y$ sẽ là nghiệm xấp xỉ tốt nhất theo chuẩn Euclid $\|\mathbf{A}x - \mathbf{y}\|_2$.

2.10 Toán tử Vết (Trace Operator)

Toán tử vết của một ma trận được định nghĩa là tổng của tất cả các phần tử nằm trên đường chéo chính:

$$\mathrm{Tr}(\mathbf{A}) = \sum_i A_{i,i}. \tag{2.61}$$

Toán tử vết rất hữu ích trong nhiều ứng dụng khác nhau. Một số phép toán khó biểu diễn mà không dùng ký hiệu tổng có thể được viết gọn gàng hơn bằng cách sử dụng tích ma trận và toán tử vết. Ví dụ, toán tử vết cung cấp một cách viết khác cho chuẩn Frobenius của một ma trận:

$$\|\mathbf{A}\|_F = \sqrt{\mathrm{Tr}(\mathbf{A}\mathbf{A}^\top)}. \tag{2.62}$$

Việc biểu diễn một biểu thức bằng toán tử vết giúp ta dễ dàng biến đổi và khai thác nhiều tính chất hữu ích. Chẳng hạn, toán tử vết không thay đổi khi lấy chuyển vị:

$$\mathrm{Tr}(\mathbf{A}) = \mathrm{Tr}(\mathbf{A}^\top). \tag{2.63}$$

Toán tử vết của một tích ma trận vuông cũng không thay đổi khi ta hoán vị vòng các ma trận (nếu tích đó được định nghĩa):

$$\mathrm{Tr}(\mathbf{ABC}) = \mathrm{Tr}(\mathbf{CAB}) = \mathrm{Tr}(\mathbf{BCA}). \tag{2.64}$$

Tính chất này có thể mở rộng cho một tích nhiều ma trận:

$$\mathrm{Tr}\left(\prod_{i=1}^{n} \mathbf{F}^{(i)}\right) = \mathrm{Tr}\left(\mathbf{F}^{(n)} \prod_{i=1}^{n-1} \mathbf{F}^{(i)}\right). \tag{2.65}$$

Ngay cả khi phép hoán vị này làm thay đổi hình dạng của tích ma trận, tính chất hoán vị vẫn đúng. Ví dụ, với hai ma trận $\mathbf{A} \in \mathbb{R}^{m\times n}$ và $\mathbf{B} \in \mathbb{R}^{n\times m}$, ta có:

$$\mathrm{Tr}(\mathbf{AB}) = \mathrm{Tr}(\mathbf{BA}). \tag{2.66}$$

Mặc dù $\mathbf{AB} \in \mathbb{R}^{m\times m}$ nhưng $\mathbf{BA} \in \mathbb{R}^{n\times n}$, toán tử vết vẫn giữ nguyên giá trị.

Một tính chất quan trọng khác là một số vô hướng cũng có thể được xem như vết của chính nó:

$$a = \mathrm{Tr}(a). \tag{2.67}$$

2.11 Định thức (Determinant)

Định thức của một ma trận vuông, ký hiệu là $\det(\mathbf{A})$, là một hàm ánh xạ ma trận sang các số thực. Định thức bằng tích của tất cả các giá trị riêng (eigenvalues) của ma trận.

Giá trị tuyệt đối của định thức có thể được xem như một phép đo mức độ mà phép nhân với ma trận làm giãn hoặc co không gian. Nếu định thức bằng 0, không gian bị co lại hoàn toàn theo ít nhất một chiều, dẫn đến việc mất toàn bộ thể tích. Nếu định thức bằng 1, phép biến đổi bảo toàn thể tích.

2.12 Ví dụ: Phân tích thành phần chính (Principal Components Analysis)

Một thuật toán học máy đơn giản, phân tích thành phần chính (PCA), có thể được suy luận chỉ bằng kiến thức cơ bản của đại số tuyến tính.

Giả sử ta có một tập hợp gồm m điểm $\{x^{(1)}, \ldots, x^{(m)}\} \in \mathbb{R}^n$ và muốn nén dữ liệu theo cách mất mát (lossy compression). Nén mất mát có nghĩa là lưu trữ các điểm theo cách tốn ít bộ nhớ hơn nhưng có thể làm mất một phần thông tin. Mục tiêu là giảm thiểu mức mất mát này.

Một cách để mã hóa các điểm này là biểu diễn chúng trong một không gian có số chiều thấp hơn. Với mỗi điểm $x^{(i)} \in \mathbb{R}^n$, ta tìm một mã tương ứng $c^{(i)} \in \mathbb{R}^l$. Nếu l nhỏ hơn n, việc lưu trữ mã sẽ tốn ít bộ nhớ hơn so với lưu trữ dữ liệu gốc. Chúng ta cần một hàm mã hóa tạo ra mã từ một đầu vào, $F(x) = c$, và một hàm giải mã để tái tạo lại đầu vào từ mã, $x \approx g(x)$.

PCA được xác định bởi lựa chọn của hàm giải mã. Để đơn giản hóa bài toán, ta sử dụng phép nhân ma trận để ánh xạ mã trở lại không gian gốc: $g(c) = Dc$, trong đó $D \in \mathbb{R}^{n \times l}$ là ma trận thực hiện giải mã.

Tính toán mã tối ưu cho bộ giải mã này có thể là một vấn đề khó. Để đảm bảo bài toán có nghiệm duy nhất, PCA đặt ràng buộc lên các cột của D sao cho chúng trực giao với nhau.

Trong PCA, ta tìm mã tối ưu c^* bằng cách tối thiểu hóa khoảng cách giữa điểm đầu vào và tái tạo của nó, sử dụng chuẩn L^2:

$$c^* = \arg\min_c \|x - g(c)\|_2 \tag{2.68}$$

Ta có thể sử dụng chuẩn bình phương L^2 thay vì chuẩn L^2 vì cả hai đều được cực tiểu hóa bởi cùng một giá trị của c. Khi đó:

$$c^* = \arg\min_c \|x - g(c)\|_2^2 \tag{2.69}$$

Hàm mục tiêu cần cực tiểu hóa trở thành:

$$(x - g(c))^\top (x - g(c)) \tag{2.70}$$

Bằng cách sử dụng chuẩn L^2 (1.21), ta có phương trình sau:

$$= x^\top x - x^\top g(c) - g(c)^\top x + g(c)^\top g(c). \tag{2.71}$$

Áp dụng tính chất phân phối:

$$= x^\top x - 2x^\top g(c) + g(c)^\top g(c). \tag{2.72}$$

Vì $g(c)^\top x$ là vô hướng nên bằng với chuyển vị của chính nó. Ta có thể bỏ qua hạng tử

$x^\top x$ vì nó không phụ thuộc vào c, do đó:

$$c^* = \arg\min_c -2x^\top g(c) + g(c)^\top g(c). \tag{2.73}$$

Do $g(c) = Dc$, ta thay vào phương trình trên:

$$c^* = \arg\min_c -2x^\top Dc + c^\top D^\top Dc. \tag{2.74}$$

$$= \arg\min_c -2x^\top Dc + c^\top Ic. \tag{2.75}$$

Do tính chất trực giao và chuẩn đơn vị của D:

$$= \arg\min_c -2x^\top Dc + c^\top c. \tag{2.76}$$

Chúng ta có thể giải bài toán tối ưu này bằng cách sử dụng phép tính đạo hàm trên không gian vector. Nếu chưa quen với phương pháp này, hãy tham khảo mục 4.3.

$$\nabla_c(-2x^\top \mathbf{D}c + c^\top c) = 0. \tag{2.77}$$

$$-2\mathbf{D}^\top x + 2c = 0. \tag{2.78}$$

$$c = \mathbf{D}^\top x. \tag{2.79}$$

Do đó, thuật toán này hoạt động hiệu quả vì ta có thể mã hóa x chỉ bằng một phép nhân ma trận-vector đơn giản. Công thức ánh xạ dữ liệu về không gian có số chiều thấp hơn như sau:

$$f(x) = \mathbf{D}^\top x. \tag{2.80}$$

Tiếp theo, bằng một phép nhân ma trận khác, ta có thể định nghĩa phép tái cấu trúc dữ liệu trong PCA:

$$r(x) = g(f(x)) = \mathbf{D}\mathbf{D}^\top x. \tag{2.81}$$

Chúng ta cần chọn ma trận mã hóa $\mathbf{D}$. Để làm được điều này, ta xem xét việc tối thiểu hóa khoảng cách L^2 giữa đầu vào và tái cấu trúc. Vì sử dụng cùng một ma trận $\mathbf{D}$ để giải mã tất cả điểm dữ liệu, ta không thể chỉ xét riêng từng điểm mà cần tối thiểu hóa chuẩn Frobenius của ma trận lỗi trên toàn bộ tập dữ liệu:

$$\mathbf{D}^* = \arg\min_{\mathbf{D}} \sqrt{\sum_{i,j} \left(x_j^{(i)} - r(x^{(i)})_j \right)^2} \quad \text{subject to} \quad \mathbf{D}^\top \mathbf{D} = \mathbf{I}_l. \tag{2.82}$$

Để tìm thuật toán cho $\mathbf{D}^*$, trước tiên chúng ta xem xét trường hợp $l = 1$. Khi đó, $\mathbf{D}$ chỉ là một vector duy nhất, ký hiệu là $\mathbf{d}$. Thay thế phương trình 1.81 vào phương trình 1.82 và đơn giản hóa $\mathbf{D}$ thành $\mathbf{d}$, bài toán trở thành:

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \sum_i \|\mathbf{x}^{(i)} - \mathbf{d}\mathbf{d}^T\mathbf{x}^{(i)}\|_2^2 \quad \text{với điều kiện} \quad \|\mathbf{d}\|_2 = 1. \tag{2.83}$$

Cách viết trên là cách trực tiếp nhất để biểu diễn bài toán, nhưng chưa thật sự "đẹp"về mặt ký hiệu toán học. Biểu thức $\mathbf{d}^T\mathbf{x}^{(i)}$ xuất hiện ở bên phải vector $\mathbf{d}$, trong khi theo quy ước thông thường, các hệ số vô hướng nên được viết ở bên trái vector mà chúng tác động lên. Do đó, ta có thể viết lại bài toán theo dạng:

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \sum_i \|\mathbf{x}^{(i)} - \mathbf{d}^T\mathbf{x}^{(i)}\mathbf{d}\|_2^2 \quad \text{với điều kiện} \quad \|\mathbf{d}\|_2 = 1. \tag{2.84}$$

Hoặc khai thác thực tế rằng một số vô hướng luôn bằng với chuyển vị của chính nó:

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \sum_i \|\mathbf{x}^{(i)} - \mathbf{x}^{(i)T}\mathbf{d}\mathbf{d}\|_2^2 \quad \text{với điều kiện} \quad \|\mathbf{d}\|_2 = 1. \tag{2.85}$$

Việc làm quen với các cách viết khác nhau như trên sẽ giúp ích cho việc đọc và hiểu các tài liệu toán học.

Để thuận tiện hơn, ta có thể viết lại bài toán theo dạng ma trận thay vì tổng theo từng vector riêng lẻ. Gọi $\mathbf{X} \in \mathbb{R}^{m \times n}$ là ma trận chứa các điểm dữ liệu, trong đó hàng thứ i của $\mathbf{X}$ là $\mathbf{x}^{(i)T}$. Khi đó, bài toán có thể viết lại dưới dạng:

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \|\mathbf{X} - \mathbf{X}\mathbf{d}\mathbf{d}^T\|_F^2 \quad \text{với điều kiện} \quad \mathbf{d}^T\mathbf{d} = 1. \tag{2.86}$$

Bỏ qua điều kiện ràng buộc tạm thời, ta có thể đơn giản hóa phần chuẩn Frobenius thành:

$$\arg\min_{\mathbf{d}} \|\mathbf{X} - \mathbf{X}\mathbf{d}\mathbf{d}^T\|_F^2. \tag{2.87}$$

$$= \arg\min_{d} \text{Tr}\left((X - Xdd^T)^T(X - Xdd^T)\right) \tag{2.88}$$

Dựa vào công thức vết ma trận (1.62), ta có:

$$= \arg\min_d \text{Tr}(X^T X - X^T X dd^T - dd^T X^T X + dd^T X^T X dd^T) \tag{2.89}$$

$$= \arg\min_d \text{Tr}(X^T X) - \text{Tr}(X^T X dd^T) - \text{Tr}(dd^T X^T X) + \text{Tr}(dd^T X^T X dd^T) \tag{2.90}$$

Loại bỏ các phần không chứa d vì chúng không ảnh hưởng đến giá trị tối ưu:

$$= \arg\min_d -2\text{Tr}(X^T X dd^T) + \text{Tr}(X^T X dd^T dd^T) \tag{2.91}$$

Sử dụng tính chất chu kỳ của dấu vết (1.65):

$$= \arg\min_d -2\text{Tr}(X^T X dd^T) + \text{Tr}(X^T X dd^T) \tag{2.92}$$

Tại thời điểm này, ta đưa trở lại ràng buộc:

$$\arg\min_d -2\,\text{Tr}(\mathbf{X}^T \mathbf{X} dd^T) + \text{Tr}(\mathbf{X}^T \mathbf{X} dd^T) \quad \text{với điều kiện} \quad d^T d = 1. \tag{2.93}$$

$$= \arg\min_d -2\,\text{Tr}(\mathbf{X}^T \mathbf{X} dd^T) + \text{Tr}(\mathbf{X}^T \mathbf{X} dd^T) \quad \text{với điều kiện} \quad d^T d = 1. \tag{2.94}$$

Do điều kiện ràng buộc, ta có:

$$= \arg\min_d -\text{Tr}(\mathbf{X}^T \mathbf{X} dd^T) \quad \text{với điều kiện} \quad d^T d = 1. \tag{2.95}$$

$$= \arg\max_d \text{Tr}(\mathbf{X}^T \mathbf{X} dd^T) \quad \text{với điều kiện} \quad d^T d = 1. \tag{2.96}$$

$$= \arg\max_d \text{Tr}(d^T \mathbf{X}^T \mathbf{X} d) \quad \text{với điều kiện} \quad d^T d = 1. \tag{2.97}$$

Bài toán tối ưu này có thể được giải bằng phương pháp phân tích giá trị riêng (eigendecomposition). Cụ thể, véc-tơ tối ưu d được cho bởi véc-tơ riêng của ma trận $\mathbf{X}^T \mathbf{X}$ tương ứng với giá trị riêng lớn nhất.

Phương pháp này áp dụng cho trường hợp $l = 1$ và chỉ tìm ra thành phần chính thứ nhất. Tổng quát hơn, khi ta muốn tìm một hệ cơ sở của các thành phần chính, ma trận $\mathbf{D}$ được xác định bởi l véc-tơ riêng ứng với l giá trị riêng lớn nhất. Điều này có thể được chứng minh bằng phương pháp quy nạp. Chúng tôi khuyến khích bạn tự viết lại chứng minh này như một bài tập.

Đại số tuyến tính là một trong những lĩnh vực toán học nền tảng cần thiết để hiểu về học sâu. Một lĩnh vực toán học quan trọng khác thường xuyên xuất hiện trong học máy là lý thuyết xác suất, sẽ được trình bày tiếp theo.

CHƯƠNG 3. LÝ THUYẾT XÁC XUẤT VÀ THÔNG TIN

Trong chương này, chúng ta sẽ tìm hiểu về lý thuyết xác suất và lý thuyết thông tin.

Lý thuyết xác suất là một khuôn khổ toán học để biểu diễn các phát biểu không chắc chắn. Nó cung cấp phương pháp định lượng mức độ không chắc chắn cũng như các tiên đề để suy luận ra những phát biểu mới. Trong các ứng dụng trí tuệ nhân tạo, chúng ta sử dụng lý thuyết xác suất theo hai cách chính. Thứ nhất, các định luật của xác suất cho chúng ta biết hệ thống AI nên suy luận như thế nào, từ đó thiết kế các thuật toán để tính toán hoặc xấp xỉ các biểu thức khác nhau dựa trên xác suất. Thứ hai, ta có thể sử dụng xác suất và thống kê để phân tích lý thuyết về hành vi của các hệ thống AI được đề xuất.

Lý thuyết xác suất là một công cụ nền tảng của nhiều ngành khoa học và kỹ thuật. Chúng tôi cung cấp chương này nhằm đảm bảo rằng những độc giả có nền tảng chủ yếu về kỹ thuật phần mềm, nhưng ít tiếp xúc với xác suất, có thể hiểu được các nội dung trong cuốn sách này.

Trong khi lý thuyết xác suất cho phép chúng ta đưa ra các phát biểu không chắc chắn và suy luận trong điều kiện bất định, thì lý thuyết thông tin giúp định lượng mức độ không chắc chắn trong một phân bố xác suất.

Nếu bạn đã quen thuộc với lý thuyết xác suất và lý thuyết thông tin, bạn có thể bỏ qua chương này, ngoại trừ Mục 3.14, nơi mô tả các đồ thị được sử dụng để mô hình hóa các mô hình xác suất có cấu trúc trong học máy. Nếu bạn chưa từng có kinh nghiệm với những chủ đề này, chương này sẽ cung cấp đủ kiến thức để thực hiện các nghiên cứu sâu về học sâu. Tuy nhiên, chúng tôi khuyến nghị bạn tham khảo thêm các tài liệu khác, chẳng hạn như *Jaynes (2003)*.

3.1 Tại sao lại dùng xác suất?

Nhiều lĩnh vực trong khoa học máy tính làm việc với các thực thể mang tính tất định và chắc chắn. Ví dụ, một lập trình viên có thể yên tâm rằng một CPU sẽ thực thi chính xác một lệnh máy mà không có lỗi. Sai sót phần cứng có thể xảy ra, nhưng hiếm đến mức phần mềm không cần thiết kế để đối phó với chúng. Vì vậy, với một môi trường hoạt động mang tính xác định cao, có thể thấy ngạc nhiên khi học máy lại dựa nhiều vào lý thuyết xác suất.

Học máy thường xuyên phải làm việc với các đại lượng không chắc chắn và đôi khi có tính ngẫu nhiên (stochastic). Việc định lượng sự bất định bằng xác suất đã được đề xuất từ những năm 1980 và nhiều lập luận trong lĩnh vực này lấy cảm hứng từ công trình của *Pearl (1988)*.

Hầu hết mọi hoạt động đều yêu cầu một mức độ suy luận trong điều kiện bất định. Trên thực tế, hầu như không có phát biểu toán học nào là chắc chắn tuyệt đối hoặc một sự kiện nào đó chắc chắn xảy ra.

Có ba nguồn gốc chính của sự bất định:

1. Tính ngẫu nhiên nội tại trong hệ thống được mô hình hóa. Ví dụ, hầu hết các cách diễn giải cơ học lượng tử mô tả chuyển động của hạt dưới dạng xác suất. Chúng ta cũng có thể xây dựng các kịch bản lý thuyết trong đó giả định một hệ động lực học ngẫu nhiên, chẳng hạn như trò chơi xáo trộn bài ngẫu nhiên.
2. Không thể quan sát đầy đủ. Ngay cả những hệ thống tất định cũng có thể xuất hiện yếu tố ngẫu nhiên khi không thể quan sát hết các biến ảnh hưởng đến hành vi của hệ thống. Ví dụ, trong bài toán Monty Hall, người tham gia chọn một trong ba cánh cửa có giấu một phần thưởng. Khi người dẫn chương trình mở một cánh cửa, lựa chọn của người chơi có vẻ ngẫu nhiên từ góc nhìn của họ, mặc dù trên thực tế nó hoàn toàn tất định.
3. Mô hình hóa không hoàn hảo. Khi sử dụng mô hình, ta buộc phải bỏ qua một phần thông tin. Việc này dẫn đến sự không chắc chắn trong dự đoán. Ví dụ, một robot sử dụng bản đồ rời rạc sẽ không thể xác định chính xác vị trí của một vật thể nếu vị trí đó không khớp với các ô lưới trên bản đồ.

Trong nhiều trường hợp, việc sử dụng một quy tắc đơn giản nhưng không chắc chắn lại thực tế hơn so với một quy tắc phức tạp nhưng hoàn toàn chính xác, ngay cả khi sự thật mang tính tất định và mô hình của chúng ta đủ linh hoạt để mô phỏng một quy tắc phức tạp. Ví dụ, quy tắc đơn giản *"Hầu hết các loài chim đều biết bay"* dễ hiểu và hữu ích trong nhiều tình huống, trong khi quy tắc *"Các loài chim biết bay, ngoại trừ chim non chưa học bay, chim ốm hoặc bị thương mất khả năng bay, các loài chim không bay như đà điểu, chim cánh cụt và kiwi…"* thì rất phức tạp, đòi hỏi nhiều công sức để xây dựng, truyền đạt và duy trì. Dù có cố gắng đến đâu, quy tắc này vẫn có thể chưa đầy đủ và dễ sai sót.

Mặc dù rõ ràng rằng chúng ta cần một phương pháp để biểu diễn và suy luận trong điều kiện bất định, nhưng không phải lúc nào lý thuyết xác suất cũng cung cấp tất cả công cụ cần thiết cho ứng dụng trí tuệ nhân tạo. Ban đầu, lý thuyết xác suất được phát triển để phân tích tần suất của các sự kiện. Ta có thể dễ dàng thấy cách nó áp dụng vào các bài toán như xác suất chia bài trong trò chơi poker, nơi các sự kiện có thể lặp lại nhiều lần. Khi nói một kết quả có xác suất p xảy ra, điều đó có nghĩa là nếu lặp lại thí nghiệm (ví dụ, chia một bộ bài) vô hạn lần, thì tỷ lệ p trong số các lần thử sẽ dẫn đến kết quả đó.

Tuy nhiên, cách suy luận này không phù hợp với các phát biểu không thể lặp lại. Ví dụ, nếu một bác sĩ chẩn đoán một bệnh nhân và nói rằng người đó có 40% khả năng mắc cúm, điều này có ý nghĩa rất khác — chúng ta không thể tạo ra vô hạn bản sao của bệnh nhân, cũng không có lý do gì để tin rằng các bệnh nhân khác có cùng triệu chứng nhưng trong điều kiện khác nhau sẽ cho cùng một kết quả.

Trong trường hợp bác sĩ chẩn đoán bệnh, chúng ta sử dụng xác suất để biểu diễn mức độ tin tưởng (*degree of belief*), với giá trị 1 thể hiện sự chắc chắn tuyệt đối rằng bệnh nhân mắc cúm và 0 thể hiện sự chắc chắn tuyệt đối rằng bệnh nhân không mắc cúm. Hình thức

đầu tiên của xác suất, liên quan trực tiếp đến tỷ lệ xuất hiện của các sự kiện, được gọi là xác suất tần suất (*frequentist probability*), trong khi hình thức thứ hai, liên quan đến mức độ tin cậy định tính, được gọi là xác suất Bayes (*Bayesian probability*).

Nếu chúng ta liệt kê một số tính chất mà chúng ta mong muốn sự suy luận hợp lý về tính bất định phải có, thì cách duy nhất để thỏa mãn những tính chất đó là xem xác suất Bayes hành xử giống hệt như xác suất tần suất.

Ví dụ, nếu chúng ta muốn tính xác suất một người chơi thắng một ván bài poker khi cô ấy có một tập hợp bài nhất định, chúng ta sử dụng chính xác cùng một công thức như khi tính xác suất một bệnh nhân mắc một căn bệnh dựa trên các triệu chứng của họ.

Để biết thêm chi tiết về lý do tại sao một tập hợp nhỏ các giả định phổ biến lại ngụ ý rằng cùng một hệ tiên đề phải kiểm soát cả hai loại xác suất, xem *Ramsey (1926)*.

Xác suất có thể được xem như một phần mở rộng của lôgic để xử lý tính bất định. Lôgic cung cấp một tập hợp các quy tắc chính thức để xác định một mệnh đề được suy ra là đúng hay sai, dựa trên giả định rằng một tập hợp mệnh đề khác là đúng hoặc sai.

Lý thuyết xác suất mở rộng tập hợp quy tắc chính thức này để xác định mức độ khả thi của một mệnh đề là đúng, dựa trên khả năng của các mệnh đề khác. Điều này giúp chúng ta không chỉ làm việc với những kết luận chắc chắn mà còn có thể định lượng mức độ tin cậy của chúng trong bối cảnh thực tế.

3.2 Biến Ngẫu Nhiên

Một biến ngẫu nhiên là một biến có thể nhận các giá trị khác nhau một cách ngẫu nhiên. Chúng ta thường ký hiệu biến ngẫu nhiên bằng một chữ cái viết thường với kiểu chữ thông thường, và các giá trị mà nó có thể nhận được bằng chữ viết thường với kiểu chữ viết tay

Ví dụ, x_1 và x_2 là các giá trị có thể có của biến ngẫu nhiên x. Nếu biến là một vectơ, chúng ta ký hiệu biến ngẫu nhiên là $\mathbf{x}$ và một giá trị cụ thể mà nó có thể nhận là $\mathbf{x}$.

Tuy nhiên, bản thân một biến ngẫu nhiên chỉ là một mô tả về các trạng thái có thể xảy ra. Để xác định xác suất xảy ra của mỗi trạng thái này, chúng ta cần một phân phối xác suất đi kèm với nó.

Biến ngẫu nhiên có thể chia thành hai loại chính:

- Biến ngẫu nhiên rời rạc (discrete random variable): Đây là loại biến chỉ có thể nhận một số lượng giá trị hữu hạn hoặc vô hạn đếm được. Các giá trị này không nhất thiết phải là số nguyên mà có thể là các trạng thái được đặt tên (ví dụ: *Trời mưa, Trời nắng, Trời âm u*), miễn là chúng có thể được liệt kê.
- Biến ngẫu nhiên liên tục (continuous random variable): Đây là loại biến có thể nhận bất kỳ giá trị nào trên một khoảng của tập số thực. Nói cách khác, giá trị của nó không bị giới hạn ở một tập hợp đếm được mà trải dài liên tục.

3.3 Phân phối xác suất

Một phân phối xác suất mô tả mức độ xảy ra của một biến ngẫu nhiên hoặc một tập hợp các biến ngẫu nhiên trên tất cả các giá trị có thể của chúng. Cách mô tả phân phối xác suất phụ thuộc vào việc các biến đó là rời rạc hay liên tục.

3.3.1 Biến rời rạc và hàm khối xác suất

Một phân phối xác suất trên các biến rời rạc có thể được mô tả bằng hàm khối xác suất (PMF). Chúng ta thường ký hiệu các hàm khối xác suất bằng chữ cái viết hoa P. Thông thường, mỗi biến ngẫu nhiên được liên kết với một hàm khối xác suất khác nhau, và người đọc cần suy luận hàm PMF nào được sử dụng dựa trên ký hiệu của biến ngẫu nhiên thay vì tên của hàm. Ví dụ, $P(x)$ khác với $P(y)$.

Hàm khối xác suất ánh xạ từ một trạng thái của biến ngẫu nhiên sang xác suất để biến ngẫu nhiên đó nhận giá trị cụ thể. Xác suất để x nhận giá trị x được ký hiệu là $P(x)$. Nếu $x = x$ xảy ra chắc chắn, xác suất sẽ là 1. Ngược lại, nếu $x = x$ không thể xảy ra, xác suất sẽ là 0. Đôi khi, để tránh nhầm lẫn giữa các hàm PMF khác nhau, ta có thể viết rõ ràng tên của biến ngẫu nhiên như $P(X = x)$. Đôi khi, ta còn sử dụng ký hiệu $x \sim P(x)$ để chỉ ra rằng x được lấy từ một phân phối xác suất nhất định.

Hàm khối xác suất có thể hoạt động trên nhiều biến cùng lúc. Một phân phối xác suất trên nhiều biến ngẫu nhiên được gọi là phân phối xác suất chung. Ví dụ, $P(x = a, y = b)$ biểu diễn xác suất để $x = a$ và $y = b$ xảy ra đồng thời. Đôi khi, ta có thể viết gọn là $P(x, y)$.

Để một hàm $P(x)$ là một hàm khối xác suất hợp lệ, nó cần thỏa mãn các điều kiện sau:

- Miền xác định của P phải là tập hợp tất cả các giá trị có thể có của x.
- Với mọi giá trị x, xác suất phải nằm trong khoảng $0 \leq P(x) \leq 1$. Một sự kiện không thể xảy ra sẽ có xác suất bằng 0, trong khi một sự kiện chắc chắn xảy ra sẽ có xác suất bằng 1. Không có giá trị nào có thể nhỏ hơn 0 hoặc lớn hơn 1.
- Tổng tất cả các xác suất phải bằng 1:

$$\sum_{x} P(x) = 1. \tag{3.1}$$

  Điều này đảm bảo rằng phân phối xác suất được chuẩn hóa. Nếu không có tính chất này, ta có thể nhận được xác suất không hợp lệ bằng cách tính tổng xác suất của nhiều sự kiện.

Ví dụ, xét một biến ngẫu nhiên rời rạc x có k trạng thái khác nhau. Nếu ta đặt một phân phối đồng đều trên x, nghĩa là mỗi trạng thái có xác suất bằng nhau, khi đó ta định nghĩa hàm khối xác suất như sau:

$$P(x = x_i) = \frac{1}{k}. \tag{3.2}$$

Vì $\frac{1}{k}$ luôn là số dương khi k là một số nguyên dương, ta thấy rằng:

$$\sum_i P(x = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1. \tag{3.3}$$

Do đó, phân phối được chuẩn hóa hợp lệ.

3.3.2 Biến liên tục và Hàm mật độ xác suất

Khi làm việc với các biến ngẫu nhiên liên tục, ta sử dụng hàm mật độ xác suất (PDF) thay vì hàm khối xác suất. Để một hàm trở thành hàm mật độ xác suất hợp lệ, nó phải thỏa mãn các điều kiện sau:

- Miền xác định của $p(x)$ phải bao gồm tất cả các giá trị có thể có của x.
- Với mọi giá trị x, ta có $p(x) \geq 0$. Tuy nhiên, khác với PMF, có thể xảy ra trường hợp $p(x) > 1$.
- Tổng xác suất trên toàn bộ không gian phải bằng 1:

$$\int p(x)dx = 1. \tag{3.4}$$

Hàm mật độ xác suất $p(x)$ không cho ta trực tiếp xác suất của một giá trị cụ thể. Thay vào đó, xác suất để x rơi vào một khoảng rất nhỏ δx được tính bằng $p(x)\delta x$.

Chúng ta có thể tích phân hàm mật độ xác suất để tính xác suất thực tế trên một khoảng. Cụ thể, xác suất để x nằm trong khoảng $[a, b]$ được tính bằng:

$$P(a \leq x \leq b) = \int_a^b p(x)dx. \tag{3.5}$$

Một ví dụ về PDF là phân phối đồng đều trên một đoạn $[a, b]$. Ta có thể định nghĩa một hàm mật độ xác suất $u(x; a, b)$ sao cho a và b là biên của đoạn. Trong trường hợp này, giá trị của hàm $u(x; a, b)$ là một hằng số được chọn sao cho tổng xác suất trên đoạn $[a, b]$ bằng 1:

$$u(x; a, b) = \frac{1}{b - a}, \quad \text{với } x \in [a, b]. \tag{3.6}$$

Ta thấy rằng $u(x; a, b) \geq 0$ và tích phân của nó trên đoạn $[a, b]$ bằng 1, do đó phân phối này được chuẩn hóa đúng cách. Ký hiệu thông thường để biểu diễn một biến ngẫu nhiên có phân phối đồng đều trên $[a, b]$ là:

$$x \sim U(a, b). \tag{3.7}$$

3.4 Xác suất biên

Đôi khi, chúng ta biết phân phối xác suất trên một tập hợp biến và muốn biết phân phối xác suất chỉ trên một tập con của chúng. Phân phối xác suất trên tập con này được gọi là xác suất biên.

Ví dụ, giả sử chúng ta có các biến ngẫu nhiên rời rạc x và y, và biết $P(x, y)$. Khi đó, ta có thể tìm $P(x)$ bằng quy tắc tổng:

$$forallx \in X, \quad P(x = x_i) = \sum_y P(x = x_i, y = y_j). \tag{3.8}$$

Tên gọi *xác suất biên* xuất phát từ cách tính xác suất biên trên giấy. Khi các giá trị của $P(x, y)$ được viết thành một lưới với các giá trị x trên các hàng và y trên các cột, việc cộng tổng theo hàng sẽ giúp tìm $P(x)$, và giá trị này thường được viết ở lề phải của bảng, tạo thành xác suất biên.

Với các biến liên tục, ta sử dụng tích phân thay vì tổng:

$$p(x) = \int p(x, y)dy. \tag{3.9}$$

Biểu thức này biểu diễn quá trình lấy tích phân của hàm mật độ xác suất chung $p(x, y)$ theo y, từ đó thu được xác suất biên của x. Đây là một nguyên tắc quan trọng trong xác suất và thống kê, đặc biệt khi làm việc với phân phối liên tục.

3.5 Xác Suất Có Điều Kiện

Trong nhiều trường hợp, chúng ta quan tâm đến xác suất của một sự kiện xảy ra khi biết rằng một sự kiện khác đã xảy ra. Đây được gọi là xác suất có điều kiện. Chúng ta ký hiệu xác suất có điều kiện rằng $\mathrm{y} = y$ khi biết $\mathrm{x} = x$ là $P(\mathrm{y} = y \mid \mathrm{x} = x)$. Công thức để tính xác suất có điều kiện là:

$$P(\mathrm{y} = y \mid \mathrm{x} = x) = \frac{P(\mathrm{y} = y, \mathrm{x} = x)}{P(\mathrm{x} = x)}. \tag{3.10}$$

Xác suất có điều kiện chỉ được xác định khi $P(\mathrm{x} = x) > 0$. Chúng ta không thể tính xác suất có điều kiện dựa trên một sự kiện không bao giờ xảy ra.

Cần lưu ý rằng xác suất có điều kiện khác với việc tính toán điều gì sẽ xảy ra nếu một hành động nào đó được thực hiện. Ví dụ, xác suất có điều kiện rằng một người đến từ Đức khi biết rằng họ nói tiếng Đức có thể cao. Tuy nhiên, nếu một người ngẫu nhiên được chọn để học tiếng Đức, quốc gia xuất xứ của họ không thay đổi. Việc đánh giá hậu quả của một hành động được gọi là truy vấn can thiệp (ntervention query), thuộc lĩnh vực mô hình nhân quả (causal modeling), nhưng chúng ta sẽ không đi sâu vào chủ đề này trong cuốn sách này.

3.6 Quy Tắc Chuỗi Của Xác Suất Có Điều Kiện

Bất kỳ phân phối xác suất chung nào trên nhiều biến ngẫu nhiên đều có thể được phân rã thành các phân phối có điều kiện trên từng biến như sau:

$$P(x^{(1)}, \ldots, x^{(n)}) = P(x^{(1)}) \prod_{i=2}^{n} P(x^{(i)} \mid x^{(1)}, \ldots, x^{(i-1)}). \tag{3.11}$$

Kết quả này được gọi là quy tắc chuỗi hoặc quy tắc tích của xác suất. Nó có thể suy ra trực tiếp từ định nghĩa của xác suất có điều kiện trong phương trình 1.10.

Ví dụ, áp dụng định nghĩa nhiều lần, chúng ta có:

$$\begin{aligned} P(a, b, c) &= P(a \mid b, c)P(b, c) \\ P(b, c) &= P(b \mid c)P(c) \\ P(a, b, c) &= P(a \mid b, c)P(b \mid c)P(c). \end{aligned}$$

Công thức trên cho thấy cách xác suất của một tập hợp biến có thể được tính toán bằng cách sử dụng xác suất có điều kiện, giúp đơn giản hóa việc tính toán trong nhiều bài toán thực tế.

3.7 Tính độc lập và độc lập có điều kiện

Hai biến ngẫu nhiên x và y được gọi là độc lập (independent) nếu phân phối xác suất của chúng có thể được biểu diễn dưới dạng tích của hai thành phần, một thành phần chỉ phụ thuộc vào x và một thành phần chỉ phụ thuộc vào y:

$$forallx \in X, y \in Y, \quad p(x = x, y = y) = p(x = x)p(y = y). \tag{3.12}$$

Hai biến ngẫu nhiên x và y được gọi là độc lập có điều kiện khi biết một biến ngẫu nhiên z, nếu phân phối xác suất có điều kiện của chúng có thể được phân rã theo cách tương tự với mọi giá trị của z:

$$forallx \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z}, \quad p(\mathrm{x} = x, \mathrm{y} = y \mid \mathrm{z} = z) = p(\mathrm{x} = x \mid \mathrm{z} = z)p(\mathrm{y} = y \mid \mathrm{z} = z). \tag{3.13}$$

Chúng ta có thể sử dụng ký hiệu gọn để biểu diễn tính độc lập và độc lập có điều kiện như sau:

- $\mathrm{x} \perp \mathrm{y}$ có nghĩa là x và y độc lập.
- $\mathrm{x} \perp \mathrm{y} \mid \mathrm{z}$ có nghĩa là x và y độc lập có điều kiện theo z.

3.8 Kỳ vọng, Phương sai và Hiệp phương sai

Kỳ vọng (hay giá trị kỳ vọng) của một hàm số $f(x)$ theo một phân phối xác suất $P(x)$ chính là giá trị trung bình mà f nhận được khi x được lấy mẫu từ P. Đối với biến rời rạc,

công thức tính kỳ vọng là:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x). \tag{3.14}$$

Đối với biến liên tục, kỳ vọng được tính bằng tích phân:

$$\mathbb{E}_{x\sim P}[f(x)] = \int p(x)f(x)dx. \tag{3.15}$$

Khi phân phối đã được xác định rõ ràng trong ngữ cảnh, chúng ta có thể viết trực tiếp tên của biến ngẫu nhiên mà kỳ vọng được tính toán, ví dụ như $\mathbb{E}_x[f(x)]$. Nếu biến ngẫu nhiên được nhắc đến một cách rõ ràng, ta có thể bỏ qua chỉ số và viết gọn thành $\mathbb{E}[f(x)]$.

Mặc định, toán tử kỳ vọng $\mathbb{E}[\cdot]$ tính giá trị trung bình của tất cả các biến ngẫu nhiên bên trong dấu ngoặc vuông. Khi không có sự mơ hồ, ta có thể lược bỏ dấu ngoặc vuông để viết gọn biểu thức.

Kỳ vọng là một toán tử tuyến tính, có nghĩa là:

$$\mathbb{E}_x[\alpha f(x) + \beta g(x)] = \alpha\mathbb{E}_x[f(x)] + \beta\mathbb{E}_x[g(x)], \tag{3.16}$$

khi α và β không phụ thuộc vào x.

Phương sai là một đại lượng đo lường mức độ phân tán của giá trị một hàm số của biến ngẫu nhiên x khi ta lấy mẫu từ phân phối xác suất của nó:

$$\text{Var}(f(x)) = \mathbb{E}\Big[(f(x) - \mathbb{E}[f(x)])^2\Big]. \tag{3.17}$$

Khi phương sai nhỏ, các giá trị của $f(x)$ tập trung gần giá trị kỳ vọng của nó. Căn bậc hai của phương sai được gọi là độ lệch chuẩn.

Hiệp phương sai đo lường mức độ liên hệ tuyến tính giữa hai đại lượng, cũng như phạm vi giá trị của chúng:

$$\text{Cov}(f(x), g(y)) = \mathbb{E}\Big[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])\Big]. \tag{3.18}$$

Các giá trị tuyệt đối lớn của hiệp phương sai có nghĩa là các giá trị thay đổi mạnh và cùng lệch xa giá trị trung bình của chúng tại cùng một thời điểm. Nếu hiệp phương sai dương, hai biến có xu hướng đạt giá trị cao đồng thời. Nếu hiệp phương sai âm, một biến có xu hướng đạt giá trị cao khi biến kia đạt giá trị thấp và ngược lại.

Các đại lượng khác như hệ số tương quan chuẩn hóa mức đóng góp của mỗi biến để đo lường chính xác mức độ liên quan của hai biến mà không bị ảnh hưởng bởi độ lớn của chúng.

Khái niệm hiệp phương sai và độ phụ thuộc có liên quan nhưng khác biệt. Hai biến độc lập có hiệp phương sai bằng không, nhưng nếu hai biến có hiệp phương sai khác không thì chúng phụ thuộc nhau. Tuy nhiên, độ phụ thuộc là một khái niệm rộng hơn hiệp phương sai. Để hai biến có hiệp phương sai bằng không, giữa chúng không được có mối quan hệ tuyến tính. Tuy nhiên, điều kiện độc lập mạnh hơn vì nó cũng loại bỏ cả mối quan hệ phi tuyến.

Một ví dụ về hai biến phụ thuộc nhưng có hiệp phương sai bằng không: Giả sử ta lấy mẫu một số thực x từ phân phối đều trên đoạn $[-1, 1]$, sau đó chọn ngẫu nhiên một biến s với xác suất $\frac{1}{2}$ để nhận giá trị 1 hoặc -1. Khi đó, đặt $y = sx$, ta thấy rằng x và y không độc lập, vì x quyết định hoàn toàn giá trị của y. Tuy nhiên, ta có:

$$\text{Cov}(x, y) = 0. \tag{3.19}$$

Điều này minh họa rằng hai biến có thể phụ thuộc nhau nhưng vẫn có hiệp phương sai bằng không.

Ma trận hiệp phương sai của một vector ngẫu nhiên $\mathbf{x} \in \mathbb{R}^n$ là một ma trận kích thước $n \times n$, được định nghĩa như sau:

$$\text{Cov}(\mathbf{x})_{i,j} = \text{Cov}(x_i, x_j). \tag{3.20}$$

Các phần tử trên đường chéo của ma trận hiệp phương sai chính là phương sai của từng biến:

$$\text{Cov}(x_i, x_i) = \text{Var}(x_i). \tag{3.21}$$

3.9 Các phân bố xác suất phổ biến

Trong học máy, một số phân phối xác suất đơn giản có vai trò quan trọng trong nhiều bối cảnh khác nhau. Dưới đây là một trong những phân phối cơ bản nhất.

3.9.1 Phân phối Bernoulli

Phân phối Bernoulli là một phân phối xác suất trên một biến ngẫu nhiên nhị phân. Nó được xác định bởi một tham số duy nhất $\phi \in [0, 1]$, biểu thị xác suất để biến ngẫu nhiên nhận giá trị bằng 1. Các tính chất quan trọng của phân phối Bernoulli bao gồm:

$$P(x = 1) = \phi \quad (3.16) \tag{3.22}$$

$$P(x = 0) = 1 - \phi \quad (3.17) \tag{3.23}$$

$$P(x = x) = \phi^x (1 - \phi)^{1-x} \quad (3.18) \tag{3.24}$$

$$\mathbb{E}_x[x] = \phi \quad (3.19) \tag{3.25}$$

$$\text{Var}_x(x) = \phi(1 - \phi) \quad (3.20) \tag{3.26}$$

3.9.2 Phân phối Multinoulli

Phân phối Multinoulli (phân phối phân loại), là một phân phối xác suất trên một biến rời rạc có k trạng thái khác nhau, với k là một số hữu hạn.[1]

Phân phối Multinoulli được tham số hóa bởi một vector xác suất $\mathbf{p} \in [0, 1]^{k-1}$, trong đó p_i biểu diễn xác suất xảy ra của trạng thái thứ i. Xác suất của trạng thái cuối cùng, tức trạng thái thứ k, được tính bằng $1 - \mathbf{1}^T\mathbf{p}$. Điều kiện ràng buộc cần thỏa mãn là:

$$\mathbf{1}^T\mathbf{p} \leq 1. \tag{3.27}$$

Phân phối Multinoulli thường được sử dụng để mô hình hóa phân phối trên các danh mục đối tượng, nên ta không nhất thiết phải giả định rằng trạng thái 1 có giá trị số là 1, trạng thái 2 là 2, v.v. Vì lý do này, thông thường ta không cần tính kỳ vọng hay phương sai của biến ngẫu nhiên phân phối Multinoulli.

Phân phối Bernoulli và Multinoulli có thể mô tả bất kỳ phân phối nào trên miền của chúng. Chúng có khả năng mô tả mọi phân phối trên miền rời rạc không phải vì chúng mạnh mẽ đặc biệt, mà bởi vì miền của chúng đơn giản — chúng mô hình hóa các biến rời rạc mà ta có thể liệt kê tất cả trạng thái. Khi làm việc với biến liên tục, số trạng thái là vô hạn, nên bất kỳ phân phối nào được mô tả bằng một số nhỏ các tham số đều phải có các giới hạn nghiêm ngặt trên phân phối của nó.

3.9.3 Phân phối Gaussian

Phân phối được sử dụng phổ biến nhất trên tập số thực chính là phân phối chuẩn, còn được gọi là phân phối Gaussian:

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.28}$$

Hai tham số của phân phối này là $\mu \in \mathbb{R}$ và $\sigma \in (0, \infty)$. Trong đó:

- μ là trung tâm của phân phối, cũng chính là kỳ vọng của phân phối: $\mathbb{E}[X] = \mu$.
- σ là độ lệch chuẩn, còn phương sai được ký hiệu là σ^2.

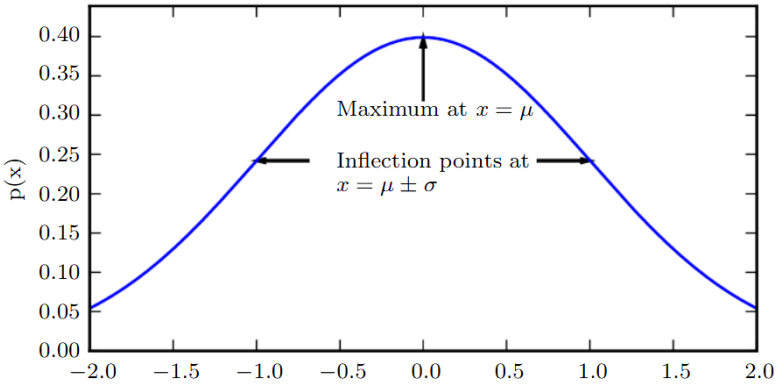
Xem hình 3.1 để thấy đồ thị của hàm mật độ xác suất của phân phối chuẩn.

Khi tính mật độ xác suất (PDF), chúng ta cần bình phương và đảo ngược σ. Để thuận tiện hơn, ta có thể sử dụng tham số $\beta \in (0, \infty)$, gọi là độ chính xác (precision), hay nghịch đảo phương sai:

[1] Thuật ngữ "Multinoulli"được Gustavo Lacerda đặt ra và phổ biến bởi *Murphy (2012)*. Phân phối Multinoulli là một trường hợp đặc biệt của phân phối đa thức (Multinomial Distribution). Một phân phối đa thức là phân phối trên các vector trong $\{0, \ldots, n\}^k$, biểu diễn số lần mỗi danh mục trong k danh mục được chọn khi lấy n mẫu từ một phân phối đa thức. Nhiều tài liệu sử dụng thuật ngữ "Multinomial"để chỉ phân phối Multinoulli mà không làm rõ rằng họ chỉ đang xét trường hợp $n = 1$.

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x-\mu)^2\right). \tag{3.29}$$

Phân phối chuẩn là một lựa chọn hợp lý trong nhiều ứng dụng. Khi không có thông tin cụ thể về dạng phân phối trên tập số thực, phân phối chuẩn thường được chọn mặc định vì hai lý do quan trọng.



Hình 3.1: Phân phối chuẩn. Phân phối chuẩn $\mathcal{N}(x; \mu, \sigma^2)$ có dạng "đường cong hình chuông" đặc trưng, với tọa độ đỉnh trung tâm được xác định bởi μ và độ rộng của nó được kiểm soát bởi σ. Trong ví dụ này, chúng ta minh họa phân phối chuẩn tắc, với $\mu = 0$ và $\sigma = 1$.

Nhiều phân phối trong thực tế có xu hướng gần với phân phối chuẩn. Điều này xuất phát từ định lý giới hạn trung tâm, một kết quả quan trọng trong xác suất thống kê. Định lý này khẳng định rằng tổng của nhiều biến ngẫu nhiên độc lập sẽ xấp xỉ theo phân phối chuẩn, bất kể phân phối ban đầu của chúng là gì.

Điều này có ý nghĩa quan trọng trong thực tế: nhiều hệ thống phức tạp có thể được mô hình hóa như nhiễu có phân phối chuẩn, ngay cả khi hệ thống có thể được chia thành nhiều thành phần có cấu trúc hơn.

Trong số tất cả các phân phối có cùng phương sai, phân phối chuẩn là phân phối mã hóa lượng bất định lớn nhất trên tập số thực. Nói cách khác, nếu không có thông tin ưu tiên nào khác, phân phối chuẩn là mô hình hợp lý nhất để sử dụng. Việc chứng minh và biện luận cho nhận định này đòi hỏi các công cụ toán học cao hơn và sẽ được trình bày chi tiết ở mục 19.4.2.

Phân phối chuẩn có thể được mở rộng cho không gian nhiều chiều $\mathbb{R}^n$, trong trường hợp này, nó được gọi là phân phối chuẩn nhiều chiều (multivariate normal distribution). Phân phối này được xác định bởi một vector trung bình $\boldsymbol{\mu}$ và một ma trận hiệp phương sai $\boldsymbol{\Sigma}$ đối xứng xác định dương. Công thức xác suất của phân phối này được cho bởi:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right). \tag{3.30}$$

Tham số $\boldsymbol{\mu}$ vẫn đại diện cho giá trị trung bình của phân phối, nhưng giờ đây nó là một vector. Tham số $\boldsymbol{\Sigma}$ là ma trận hiệp phương sai của phân phối. Tuy nhiên, trong nhiều trường hợp, việc sử dụng $\boldsymbol{\Sigma}$ không hiệu quả về mặt tính toán do cần phải nghịch đảo ma trận này để tính mật độ xác suất (PDF). Vì vậy, ta có thể sử dụng ma trận chính xác $\boldsymbol{\beta}$ thay thế:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \sqrt{\frac{\det(\boldsymbol{\beta})}{(2\pi)^n}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\beta}(\mathbf{x}-\boldsymbol{\mu})\right). \tag{3.31}$$

Thường thì ta cố định ma trận hiệp phương sai ở dạng đường chéo để đơn giản hóa tính toán. Một trường hợp đặc biệt đơn giản hơn là phân phối Gaussian đẳng hướng (isotropic Gaussian distribution), trong đó ma trận hiệp phương sai là một bội số của ma trận đơn vị.

3.9.4 Phân phối Mũ và Phân phối Laplace

Trong bối cảnh học sâu, chúng ta thường muốn có một phân phối xác suất có điểm sắc nét tại $x = 0$. Điều này có thể đạt được bằng phân phối mũ:

$$p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x). \tag{3.32}$$

Phân phối mũ sử dụng hàm chỉ thị $\mathbf{1}_{x \geq 0}$ để gán xác suất bằng 0 cho tất cả giá trị âm của x.

Một phân phối xác suất liên quan mật thiết cho phép chúng ta đặt một đỉnh sắc nét tại một điểm tùy ý μ là phân phối Laplace:

$$\text{Laplace}(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x-\mu|}{\gamma}\right). \tag{3.33}$$

3.9.5 Phân phối Dirac và Phân phối Thực nghiệm

Trong một số trường hợp, chúng ta muốn chỉ rõ rằng toàn bộ khối lượng trong một phân phối xác suất tập trung tại một điểm duy nhất. Điều này có thể đạt được bằng cách định nghĩa hàm mật độ xác suất (PDF) sử dụng hàm delta Dirac, $\delta(x)$:

$$p(x) = \delta(x - \mu). \tag{3.34}$$

Hàm delta Dirac được định nghĩa sao cho nó có giá trị bằng 0 ở mọi nơi, ngoại trừ tại điểm 0, nhưng tích phân của nó trên toàn bộ không gian lại bằng 1. Hàm delta Dirac không phải là một hàm số thông thường ánh xạ mỗi giá trị x thành một giá trị thực, mà thay vào đó, nó là một loại hàm tổng quát (generalized function) được định nghĩa thông qua các tính chất khi lấy tích phân.

Chúng ta có thể coi hàm delta Dirac là giới hạn của một dãy hàm số dồn mật độ vào điểm 0 ngày càng nhiều. Nếu định nghĩa một hàm xác suất $p(x)$ bằng cách dịch chuyển

delta Dirac bởi $-\mu$, ta sẽ thu được một đỉnh xác suất rất hẹp và cao vô hạn tại $x = \mu$.

Một ứng dụng phổ biến của hàm delta Dirac là trong phân phối thực nghiệm (empirical distribution). Giả sử chúng ta có một tập dữ liệu gồm m điểm $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, phân phối thực nghiệm của tập dữ liệu này được định nghĩa như sau:

$$\hat{p}(x) = \frac{1}{m} \sum_{i=1}^{m} \delta(x - x^{(i)}) \tag{3.35}$$

Công thức trên cho thấy mỗi điểm dữ liệu đóng góp một khối lượng xác suất $\frac{1}{m}$ vào phân phối thực nghiệm. Đối với các biến liên tục, ta cần sử dụng hàm delta Dirac để xác định phân phối thực nghiệm, nhưng với các biến rời rạc, cách tiếp cận đơn giản hơn: phân phối thực nghiệm có thể được xem như một phân phối đa thức, trong đó xác suất của mỗi giá trị đầu vào chính là tần suất thực nghiệm (empirical frequency) của giá trị đó trong tập huấn luyện.

Nhìn từ góc độ khác, phân phối thực nghiệm mô tả xác suất của tập dữ liệu mà chúng ta lấy mẫu trong quá trình huấn luyện mô hình. Một khía cạnh quan trọng khác là phân phối thực nghiệm chính là mật độ xác suất tối đa hóa khả năng hợp lý của tập dữ liệu huấn luyện (xem mục 5.5).

3.9.6 Phân phối hỗn hợp

Một cách phổ biến để định nghĩa các phân phối xác suất là kết hợp các phân phối đơn giản hơn để tạo thành một phân phối hỗn hợp (mixture distribution). Một phân phối hỗn hợp bao gồm nhiều phân phối thành phần. Trong mỗi lần lấy mẫu, việc lựa chọn phân phối thành phần nào để tạo ra mẫu dữ liệu được xác định bằng cách lấy mẫu từ một phân phối multinoulli:

$$P(x) = \sum_{i} P(c = i) P(x \mid c = i), \tag{3.36}$$

trong đó $P(c)$ là phân phối multinoulli trên các thành phần phân phối.

Một ví dụ về phân phối hỗn hợp là phân phối thực nghiệm trên các biến liên tục. Cụ thể, phân phối thực nghiệm có thể được xem là một phân phối hỗn hợp với mỗi thành phần là một hàm delta Dirac ứng với từng mẫu huấn luyện.

Mô hình hỗn hợp (mixture model) là một chiến lược đơn giản để kết hợp các phân phối xác suất nhằm tạo ra một phân phối phong phú hơn. Trong Chương 16, chúng ta sẽ tìm hiểu cách xây dựng các phân phối phức tạp từ những phân phối đơn giản hơn một cách chi tiết.

Mô hình hỗn hợp cho phép chúng ta tiếp cận một khái niệm quan trọng: biến ẩn (latent variable). Biến ẩn là một biến ngẫu nhiên mà chúng ta không thể quan sát trực tiếp. Một ví dụ là biến c, đại diện cho danh tính của thành phần trong mô hình hỗn hợp. Các biến

ẩn có thể liên quan đến biến quan sát $\mathbf{x}$ thông qua phân phối kết hợp:

$$P(\mathbf{x}, c) = P(\mathbf{x} \mid c)P(c). \tag{3.37}$$

Phân phối $P(c)$ trên biến ẩn và phân phối $P(\mathbf{x} \mid c)$ mô tả mối quan hệ giữa biến ẩn và biến quan sát. Mặc dù có thể biểu diễn trực tiếp $P(\mathbf{x})$ mà không cần đề cập đến biến ẩn, nhưng chính những thành phần này quyết định hình dạng của phân phối tổng thể. Biến ẩn sẽ được phân tích rõ hơn trong mục 16.5

Một dạng phổ biến và mạnh mẽ của mô hình hỗn hợp là Mô hình Hỗn hợp Gaussian (Gaussian Mixture Model - GMM), trong đó các thành phần $p(\mathbf{x} \mid c = i)$ là các phân phối Gaussian. Mỗi thành phần có trung bình $\boldsymbol{\mu}^{(i)}$ và ma trận hiệp phương sai $\boldsymbol{\Sigma}^{(i)}$ riêng biệt. Một số mô hình có thể áp đặt các ràng buộc bổ sung, chẳng hạn như các ma trận hiệp phương sai có thể được chia sẻ giữa các thành phần:

$$\boldsymbol{\Sigma}^{(i)} = \boldsymbol{\Sigma}, \quad forall i. \tag{3.38}$$

Giống như một phân phối Gaussian đơn lẻ, mô hình hỗn hợp Gaussian cũng có thể áp đặt các ràng buộc về hiệp phương sai, chẳng hạn như yêu cầu các ma trận hiệp phương sai phải là đường chéo hoặc đẳng hướng.

Bên cạnh trung bình và hiệp phương sai, một mô hình hỗn hợp Gaussian còn yêu cầu xác suất tiên nghiệm $\alpha_i = P(c = i)$ cho từng thành phần i. Thuật ngữ *tiên nghiệm* thể hiện niềm tin của mô hình về c trước khi quan sát $\mathbf{x}$. Ngược lại, xác suất hậu nghiệm $P(c \mid \mathbf{x})$ được tính toán *sau* khi quan sát dữ liệu.

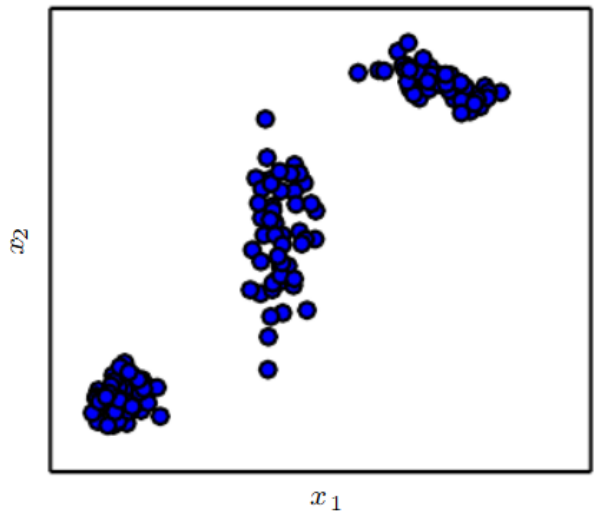
Một tính chất quan trọng của mô hình hỗn hợp Gaussian là tính xấp xỉ phổ quát (universal approximation), nghĩa là bất kỳ phân phối nào đủ trơn đều có thể được xấp xỉ với sai số tùy ý nhỏ bằng mô hình hỗn hợp Gaussian với đủ số lượng thành phần.

Hình 3.2 là ví dụ về mô hình hỗn hợp Gaussian

3.10 Các Tính Chất Hữu Ích của Một Số Hàm Phổ Biến

Trong khi làm việc với các phân phối xác suất, đặc biệt là các phân phối được sử dụng trong mô hình học sâu, một số hàm xuất hiện thường xuyên. Một trong số đó là hàm sigmoid logistic:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{3.39}$$



Hình 3.2: Ví dụ về mô hình hỗn hợp Gaussian. Trong ví dụ này, có ba thành phần. Từ trái sang phải, thành phần đầu tiên có ma trận hiệp phương sai đẳng hướng, nghĩa là phương sai trong mỗi hướng là như nhau. Thành phần thứ hai có ma trận hiệp phương sai đường chéo, cho phép nó kiểm soát phương sai riêng biệt dọc theo mỗi trục tọa độ. Trường hợp này có phương sai dọc theo trục x_2 lớn hơn so với trục x_1. Thành phần thứ ba có ma trận hiệp phương sai đầy đủ hạng, cho phép nó kiểm soát phương sai theo bất kỳ hướng nào một cách độc lập.

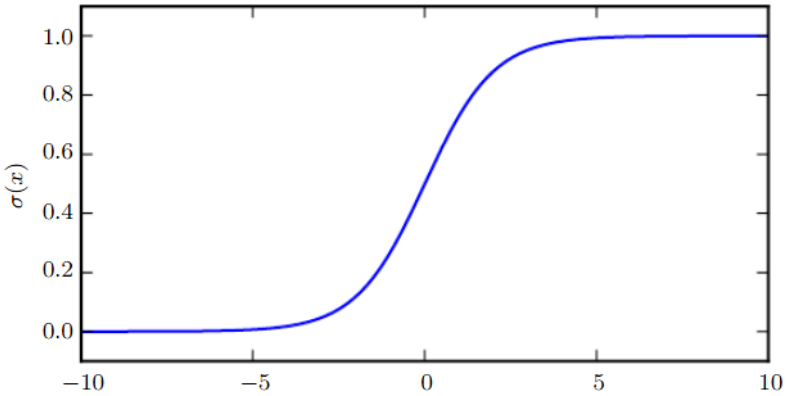
Một hàm thường gặp khác là hàm softplus (*Dugas et al., 2001*), được định nghĩa như sau:

$$\zeta(x) = \log(1 + \exp(x)). \tag{3.40}$$

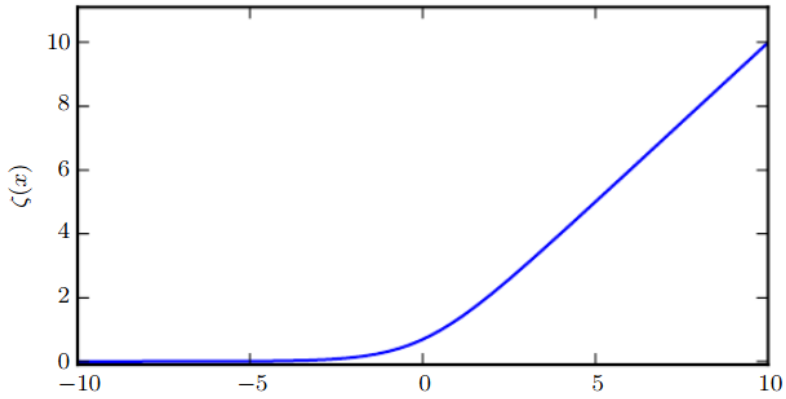
Hàm softplus có thể hữu ích trong việc tính toán tham số β hoặc σ của phân phối chuẩn vì miền giá trị của nó là $(0, \infty)$. Hàm này cũng thường xuất hiện khi xử lý các biểu thức liên quan đến sigmoid. Tên gọi "softplus" xuất phát từ việc đây là một phiên bản *'mượt hơn'* của hàm:

$$x^+ = \max(0, x). \tag{3.41}$$

Xem Hình 3.4 để xem đồ thị của hàm softplus.



Hình 3.3: Đồ thị của hàm logistic sigmoid



Hình 3.4: Đồ thị của hàm softplus

Những tính chất sau đây rất quan trọng và hữu ích, nên ghi nhớ:

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)} \tag{3.42}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) \tag{3.43}$$

$$1 - \sigma(x) = \sigma(-x) \tag{3.44}$$

$$\log \sigma(x) = -\zeta(-x) \tag{3.45}$$

$$\frac{d}{dx}\zeta(x) = \sigma(x) \tag{3.46}$$

$$\sigma^{-1}(x) = \log\left(\frac{x}{1-x}\right), \quad forallx \in (0,1) \tag{3.47}$$

$$\zeta^{-1}(x) = \log(\exp(x) - 1), \quad forallx > 0 \tag{3.48}$$

$$\zeta(x) = \int_{-\infty}^{x} \sigma(y)dy \tag{3.49}$$

$$\zeta(x) - \zeta(-x) = x \tag{3.50}$$

Hàm $\sigma^{-1}(x)$ thường được gọi là logit trong thống kê, nhưng thuật ngữ này ít được sử dụng trong học máy.

Phương trình 3.50 cho ta thấy sự liên hệ giữa $\zeta(x)$ và $\zeta(-x)$, từ đó có thể giải thích cho tên gọi softplus. Hàm softplus là một phiên bản làm mượt của hàm phần dương:

$$x^+ = \max(0, x) \tag{3.51}$$

Hàm phần dương có một hàm đối ứng là hàm phần âm:

$$x^- = \max(0, -x) \tag{3.52}$$

Hàm $\zeta(x)$ có thể xem như một phiên bản làm mượt của x^+, trong khi $\zeta(-x)$ là phiên bản làm mượt của x^-. Vì x có thể được khôi phục từ x^+ và x^- thông qua:

$$x = x^+ - x^- \tag{3.53}$$

nên ta cũng có thể khôi phục x từ $\zeta(x)$ và $\zeta(-x)$ thông qua:

$$\zeta(x) - \zeta(-x) = x \tag{3.54}$$

Phương trình 3.50 cho thấy cách mà softplus có thể được sử dụng như một cách làm mượt của các hàm cực trị như ReLU trong học sâu.

3.11 Quy tắc Bayes

Chúng ta thường gặp tình huống đã biết xác suất có điều kiện $P(y \mid x)$ nhưng cần tính $P(x \mid y)$. May mắn thay, nếu biết $P(x)$, ta có thể tính giá trị mong muốn bằng quy tắc Bayes:

$$P(x \mid y) = \frac{P(x)P(y \mid x)}{P(y)}. \tag{3.55}$$

Lưu ý rằng mặc dù $P(y)$ xuất hiện trong công thức, nhưng thông thường ta có thể tính $P(y)$ bằng:

$$P(y) = \sum_x P(y \mid x)P(x), \tag{3.56}$$

nên không cần phải biết trước $P(y)$.

Quy tắc Bayes có thể được suy ra dễ dàng từ định nghĩa xác suất có điều kiện, nhưng việc biết tên công thức này rất quan trọng vì nhiều tài liệu sử dụng nó một cách trực tiếp. Công thức này được đặt theo tên của Reverend Thomas Bayes, người đầu tiên khám phá ra một trường hợp đặc biệt của nó. Phiên bản tổng quát như đã trình bày ở đây được phát hiện độc lập bởi Pierre-Simon Laplace.

3.12 Các khía cạnh kỹ thuật của biến ngẫu nhiên liên tục

Để hiểu một cách chặt chẽ về biến ngẫu nhiên liên tục và hàm mật độ xác suất, ta cần phát triển lý thuyết xác suất dựa trên một nhánh của toán học gọi là lý thuyết độ đo (measure theory). Mặc dù lý thuyết độ đo nằm ngoài phạm vi của cuốn sách này, nhưng chúng ta có thể phác thảo sơ lược một số vấn đề mà lý thuyết độ đo giúp giải quyết.

Trong mục 3.3.2, ta đã thấy rằng xác suất để một vector ngẫu nhiên liên tục $\mathbf{x}$ thuộc vào một tập hợp $\mathbb{S}$ được xác định bởi tích phân của $p(\mathbf{x})$ trên tập hợp $\mathbb{S}$. Tuy nhiên, một số lựa chọn của tập hợp $\mathbb{S}$ có thể dẫn đến những nghịch lý. Ví dụ, có thể tồn tại hai tập $\mathbb{S}_1$ và $\mathbb{S}_2$ sao cho:

$$p(\mathbf{x} \in \mathbb{S}_1) + p(\mathbf{x} \in \mathbb{S}_2) > 1, \tag{3.57}$$

nhưng $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$. Những tập hợp như vậy thường được tạo ra bằng cách khai thác độ

chính xác vô hạn của số thực, chẳng hạn như bằng cách tạo các tập hợp có cấu trúc fractal hoặc tập hợp được xây dựng bằng cách biến đổi tập hợp các số hữu tỷ.

Một trong những đóng góp quan trọng của lý thuyết độ đo là cung cấp một cách đặc trưng hóa các tập hợp để có thể tính toán xác suất mà không gặp phải nghịch lý. Trong cuốn sách này, chúng ta chỉ xét các tập hợp có mô tả đơn giản, do đó khía cạnh này của lý thuyết độ đo không trở thành một vấn đề quan trọng.

Trong phạm vi ứng dụng của cuốn sách này, lý thuyết độ đo chủ yếu hữu ích để mô tả các định lý áp dụng cho hầu hết các điểm trong không gian $\mathbb{R}^n$, nhưng không áp dụng cho một số tập hợp nhỏ. Lý thuyết độ đo cung cấp một cách mô tả chặt chẽ về việc một tập hợp điểm nhỏ đến mức nào. Một tập hợp như vậy được gọi là có độ đo bằng không (measure zero).

Chúng ta không định nghĩa chặt chẽ khái niệm này trong sách, mà chỉ cần trực giác rằng một tập hợp có độ đo bằng không sẽ không chiếm thể tích trong không gian mà ta đang đo lường. Ví dụ, trong $\mathbb{R}^2$, một đường thẳng có độ đo bằng không, trong khi một đa giác hữu hạn có độ đo dương. Tương tự, một điểm đơn lẻ cũng có độ đo bằng không. Hơn nữa, hợp của vô hạn đếm được các tập hợp có độ đo bằng không vẫn có độ đo bằng không (chẳng hạn, tập hợp tất cả các số hữu tỷ trong một khoảng vẫn có độ đo bằng không).

Một khía cạnh kỹ thuật khác liên quan đến biến ngẫu nhiên liên tục là cách xử lý các biến ngẫu nhiên có quan hệ hàm số xác định với nhau. Giả sử ta có hai biến ngẫu nhiên $\mathbf{x}$ và $\mathbf{y}$, với mối quan hệ:

$$\mathbf{y} = g(\mathbf{x}), \tag{3.58}$$

trong đó g là một phép biến đổi khả nghịch, liên tục và khả vi. Một cách trực quan, ta có thể kỳ vọng mật độ xác suất của $\mathbf{y}$ được tính bằng:

$$p_y(\mathbf{y}) = p_x(g^{-1}(\mathbf{y})). \tag{3.59}$$

Tuy nhiên, điều này không hoàn toàn chính xác. Việc chuyển đổi giữa các biến ngẫu nhiên liên tục cần được xử lý cẩn thận, đặc biệt khi xét đến sự thay đổi của không gian xác suất qua phép biến đổi g.

Giả sử chúng ta có hai biến ngẫu nhiên vô hướng x và y, với $y = \frac{x}{2}$ và $x \sim U(0, 1)$ (biến ngẫu nhiên đều trên khoảng $(0, 1)$). Nếu chúng ta áp dụng quy tắc

$$p_y(y) = p_x(2y) \tag{3.60}$$

thì $p_y(y)$ sẽ bằng 0 ở mọi nơi trừ khoảng $[0, \frac{1}{2}]$, và bằng 1 trên khoảng đó. Điều này dẫn đến:

$$\int p_y(y)dy = \frac{1}{2}, \tag{3.61}$$

mâu thuẫn với định nghĩa của một hàm mật độ xác suất (tổng xác suất phải bằng 1).

Lỗi sai ở đây là chúng ta chưa tính đến sự biến đổi không gian do hàm số g tạo ra. Xác suất để x nằm trong một khoảng rất nhỏ với thể tích dx được cho bởi $p_x(x)dx$. Tuy nhiên, do ánh xạ g có thể co giãn không gian, nên thể tích vô cùng nhỏ tương ứng quanh x có thể khác với thể tích trong không gian y.

Để sửa lỗi, ta cần đảm bảo tính chất bảo toàn xác suất:

$$|p_y(g(x))dy| = |p_x(x)dx| \,. \tag{3.62}$$

Giải phương trình trên theo $p_y(y)$, ta thu được công thức chuyển đổi:

$$p_y(y) = p_x(g^{-1}(y)) \left|\frac{dx}{dy}\right| . \tag{3.63}$$

Hay tương đương:

$$p_x(x) = p_y(g(x)) \left|\frac{dg(x)}{dx}\right| . \tag{3.64}$$

Trong không gian nhiều chiều, ta tổng quát hoá công thức này bằng định thức của ma trận Jacobian:

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x})) \left|\det\left(\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\right)\right| . \tag{3.65}$$

Trong đó, ma trận Jacobian J có phần tử:

$$J_{i,j} = \frac{\partial x_i}{\partial y_j}. \tag{3.66}$$

Như vậy, để chuyển đổi hàm mật độ xác suất qua một ánh xạ g, ta cần nhân với trị tuyệt đối của định thức ma trận Jacobian của ánh xạ đó.

3.13 Lý thuyết Thông tin

Lý thuyết thông tin là một nhánh của toán ứng dụng tập trung vào việc định lượng thông tin có trong một tín hiệu. Ban đầu, lý thuyết này được phát minh để nghiên cứu việc truyền tin qua các bảng chữ cái rời rạc trên một kênh nhiễu, chẳng hạn như truyền thông qua sóng vô tuyến. Trong bối cảnh này, lý thuyết thông tin chỉ ra cách thiết kế mã tối ưu và tính toán độ dài kỳ vọng của các thông điệp được lấy mẫu từ các phân bố xác suất cụ thể bằng cách sử dụng các sơ đồ mã hóa khác nhau.

Trong bối cảnh học máy, chúng ta cũng có thể áp dụng lý thuyết thông tin vào các biến liên tục, nơi mà một số cách diễn giải về độ dài thông điệp không còn áp dụng nữa. Lĩnh vực này rất quan trọng trong nhiều lĩnh vực của kỹ thuật điện và khoa học máy tính. Trong cuốn sách này, chúng ta sẽ chủ yếu sử dụng một số ý tưởng chính của lý thuyết thông tin để đặc trưng hóa các phân bố xác suất hoặc để đo độ tương đồng giữa các phân bố xác suất.

Để tìm hiểu chi tiết hơn về lý thuyết thông tin, có thể tham khảo *Cover and Thomas*

(2006) hoặc *MacKay (2003)*.

Khái niệm cốt lõi đằng sau lý thuyết thông tin là việc học được một sự kiện ít xảy ra mang lại nhiều thông tin hơn so với học được một sự kiện có khả năng cao xảy ra. Ví dụ, một thông điệp nói rằng *"mặt trời mọc vào sáng nay"* là quá hiển nhiên đến mức không cần phải gửi đi, trong khi một thông điệp nói rằng *"sáng nay có nhật thực"* chứa rất nhiều thông tin.

Chúng ta muốn định lượng thông tin theo cách chính thức hóa trực giác này:

- Những sự kiện có khả năng cao nên có lượng thông tin thấp, và trong trường hợp cực đoan, những sự kiện chắc chắn xảy ra không mang lại thông tin gì cả.
- Những sự kiện ít có khả năng xảy ra nên có lượng thông tin cao hơn.
- Những sự kiện độc lập nên có thông tin cộng tính. Ví dụ, việc biết rằng một đồng xu tung lên xuất hiện mặt ngửa hai lần nên mang lại gấp đôi lượng thông tin so với việc biết rằng đồng xu xuất hiện mặt ngửa một lần.

Để thỏa mãn ba tính chất quan trọng trong lý thuyết thông tin, ta định nghĩa lượng tin của một sự kiện $\mathrm{x} = x$ như sau:

$$I(x) = -\log P(x). \tag{3.67}$$

Trong công thức này, $P(x)$ là xác suất xảy ra của sự kiện x. Hàm logarithm được sử dụng ở đây là log tự nhiên (logarit cơ số e). Vì vậy, đơn vị đo của $I(x)$ trong trường hợp này là nat. Một nat chính là lượng thông tin thu được khi quan sát một sự kiện có xác suất bằng $\frac{1}{e}$.

Trong một số tài liệu khác, người ta sử dụng logarit cơ số 2 để đo lượng tin, khi đó đơn vị đo thông tin sẽ là bit (hoặc shannon). Giá trị đo bằng bit chỉ là một cách thể hiện khác của lượng tin, được chuyển đổi từ đơn vị nat.

Khi x là một biến ngẫu nhiên liên tục, ta vẫn có thể sử dụng định nghĩa lượng tin một cách tương tự. Tuy nhiên, một số tính chất từ trường hợp rời rạc không còn đúng nữa. Ví dụ, một sự kiện có mật độ xác suất đơn vị vẫn có lượng tin bằng 0, mặc dù nó không phải là một sự kiện chắc chắn xảy ra.

Lượng tin tự thân chỉ xét một kết quả đơn lẻ. Để đo lường mức độ bất định của toàn bộ một phân phối xác suất, ta sử dụng entropy Shannon, được định nghĩa như sau:

$$H(x) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log P(x)], \tag{3.68}$$

còn được ký hiệu là $H(P)$. Nói cách khác, entropy Shannon của một phân phối là giá trị kỳ vọng của lượng tin khi rút ra một sự kiện từ phân phối đó. Nó cho ta một cận dưới về số bit trung bình cần thiết để mã hóa các ký hiệu được chọn từ phân phối P (nếu logarithm

là cơ số 2, với các cơ số khác thì đơn vị thay đổi).

Các phân phối có tính chất gần như tất định (tức là kết quả gần như chắc chắn) sẽ có entropy thấp. Ngược lại, các phân phối gần như đều có entropy cao (Hình 3.5). Khi x là một biến liên tục, entropy Shannon còn được gọi là entropy vi phân.

Khi có hai phân phối xác suất $P(x)$ và $Q(x)$ trên cùng một biến ngẫu nhiên x, ta có thể đo lường sự khác biệt giữa hai phân phối này bằng độ phân kỳ Kullback-Leibler (KL divergence):

$$D_{KL}(P \parallel Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)]. \tag{3.69}$$

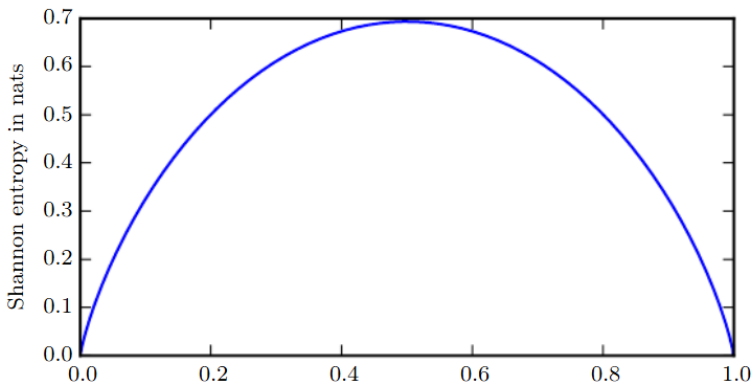
Trong trường hợp biến rời rạc, độ phân kỳ KL đo lượng thông tin dư thừa (tính theo bit nếu dùng log cơ số 2, hoặc theo nat nếu dùng log tự nhiên) cần thiết để gửi một thông điệp chứa các ký hiệu được lấy từ phân phối P, khi ta sử dụng một mã tối ưu hóa cho phân phối Q.

Độ phân kỳ KL có nhiều tính chất quan trọng, trong đó đáng chú ý nhất là tính không âm: $D_{KL}(P \parallel Q) \geq 0$, và chỉ bằng 0 khi $P = Q$ (trong trường hợp rời rạc) hoặc "hầu như bằng nhau"trong trường hợp liên tục.

Tuy nhiên, độ phân kỳ KL không đối xứng, tức là:

$$D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P). \tag{3.70}$$

Điều này có ý nghĩa quan trọng khi lựa chọn sử dụng $D_{KL}(P \parallel Q)$ hay $D_{KL}(Q \parallel P)$ trong thực tế. Xem Hình 3.6 để hiểu rõ hơn



Hình 3.5: Entropy Shannon của biến ngẫu nhiên nhị phân, cho thấy cách phân phối gần như xác định có entropy thấp, trong khi phân phối gần đều có entropy cao. Trục hoành biểu diễn p, xác suất để biến ngẫu nhiên bằng 1. Entropy được tính bởi công thức $H(p) = (p-1)\log(p-1) - p\log p$. Khi p gần 0 hoặc 1, phân phối gần như xác định, do đó entropy thấp. Khi $p = 0.5$, phân phối đều, do đó entropy đạt giá trị lớn nhất.

Một đại lượng liên quan chặt chẽ đến độ phân kỳ KL là entropy chéo $H(P, Q) = H(P) + D_{KL}(P||Q)$, tương tự như độ phân kỳ KL nhưng thiếu số hạng bên trái:

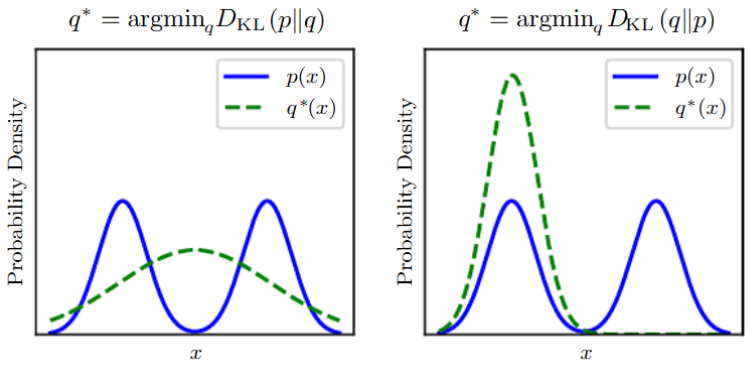
$$H(P, Q) = -\mathbb{E}_{x\sim P} \log Q(x). \tag{3.71}$$

Việc giảm thiểu entropy chéo đối với Q tương đương với việc giảm thiểu độ phân kỳ KL, vì Q không tham gia vào số hạng bị bỏ qua.

Khi tính toán nhiều đại lượng này, thường gặp các biểu thức dạng $0 \log 0$. Theo quy ước, trong ngữ cảnh lý thuyết thông tin, ta xem các biểu thức này như $\lim_{x\to 0} x \log x = 0$.

3.14 Mô Hình Xác Suất Có Cấu Trúc

Các thuật toán học máy thường liên quan đến phân phối xác suất trên một số lượng lớn các biến ngẫu nhiên. Thông thường, các phân phối xác suất này bao gồm các tương tác trực tiếp giữa một số biến tương đối ít. Việc sử dụng một hàm đơn lẻ để mô tả toàn bộ phân phối xác suất chung có thể rất kém hiệu quả (cả về mặt tính toán và thống kê).



Hình 3.6: Độ phân kì KL không đối xứng. Giả sử chúng ta có một phân phối $p(x)$ và muốn xấp xỉ nó bằng một phân phối khác $q(x)$. Chúng ta có thể chọn tối thiểu hóa $D_{KL}(P||q)$ hoặc $D_{KL}(q||P)$. Chúng ta minh họa ảnh hưởng của lựa chọn này bằng cách sử dụng hỗn hợp hai Gaussian cho p và một Gaussian đơn cho q. Lựa chọn hướng của độ phân kỳ KL phụ thuộc vào bài toán. Một số ứng dụng yêu cầu một xấp xỉ thường đặt xác suất cao ở bất kỳ đâu mà phân phối thực sự đặt xác suất cao, trong khi các ứng dụng khác yêu cầu một xấp xỉ hiếm khi đặt xác suất cao ở bất kỳ đâu mà phân phối thực sự đặt xác suất thấp. Lựa chọn hướng của độ phân kỳ KL phản ánh xem xét nào trong số này được ưu tiên cho mỗi ứng dụng. (Bên trái) Ảnh hưởng của việc tối thiểu hóa $D_{KL}(P||q)$. Trong trường hợp này, chúng ta chọn một q có xác suất cao ở nơi p có xác suất cao. Khi p có nhiều mode, q chọn làm mờ các mode lại với nhau, để đặt khối lượng xác suất cao vào tất cả chúng. (Bên phải) Ảnh hưởng của việc tối thiểu hóa $D_{KL}(q||p)$. Trong trường hợp này, chúng ta chọn một q có xác suất thấp ở nơi p có xác suất thấp. Khi p có nhiều mode được tách biệt đủ rộng, như trong hình này, độ phân kỳ KL được tối thiểu hóa bằng cách chọn một mode đơn, để tránh đặt khối lượng xác suất trong các vùng xác suất thấp giữa các mode của p. Ở đây, chúng ta minh họa kết quả khi q được chọn để nhấn mạnh mode bên trái. Chúng ta cũng có thể đạt được giá trị tương đương của độ phân kỳ KL bằng cách chọn mode bên phải. Nếu các mode không được tách biệt bởi một vùng xác suất thấp đủ mạnh, thì hướng này của độ phân kỳ KL vẫn có thể chọn làm mờ các mode.

Thay vì dùng một hàm duy nhất để biểu diễn một phân phối xác suất, ta có thể tách phân phối đó thành nhiều thành phần nhỏ hơn rồi nhân chúng lại với nhau. Chẳng hạn, giả sử ta có ba biến ngẫu nhiên: a, b và c. Giả sử a ảnh hưởng đến giá trị của b, và b ảnh hưởng đến giá trị của c, nhưng a và c độc lập với nhau khi biết b. Khi đó, ta có thể biểu

diễn phân phối xác suất trên cả ba biến này dưới dạng tích của các phân phối xác suất trên hai biến:

$$p(a, b, c) = p(a)p(b \mid a)p(c \mid b). \tag{3.72}$$

Những phân rã này giúp giảm đáng kể số lượng tham số cần thiết để mô tả phân phối. Mỗi thành phần chỉ dùng số lượng tham số theo cấp số mũ của số biến trong thành phần đó. Do đó, ta có thể giảm chi phí biểu diễn một phân phối nếu tìm được cách phân rã thành các phân phối trên ít biến hơn.

Ta có thể mô tả những kiểu phân rã này bằng đồ thị. Ở đây, "đồ thị"được hiểu theo nghĩa lý thuyết đồ thị: một tập hợp các đỉnh (node) có thể được kết nối với nhau bởi các cạnh (edge). Khi ta biểu diễn sự phân rã của một phân phối xác suất bằng đồ thị, ta gọi đó là mô hình xác suất có cấu trúc hoặc mô hình đồ thị (graphical model).

Có hai loại chính của mô hình xác suất có cấu trúc: có hướng (directed) và vô hướng (undirected). Cả hai loại mô hình đồ thị đều sử dụng một đồ thị $\mathcal{G}$, trong đó mỗi nút tương ứng với một biến ngẫu nhiên, và một cạnh kết nối hai biến ngẫu nhiên biểu thị rằng phân phối xác suất có thể biểu diễn được tương tác trực tiếp giữa hai biến đó.

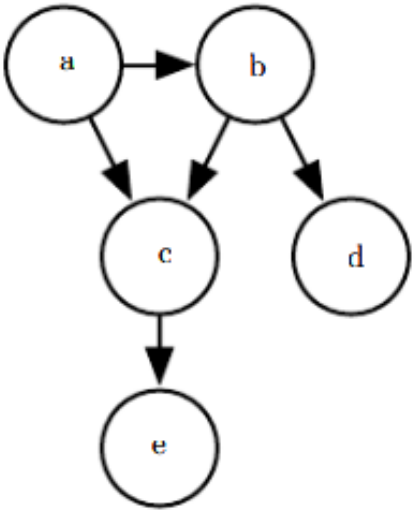
Mô hình có hướng sử dụng các cạnh có hướng (directed edges), và chúng biểu diễn việc phân rã thành các phân phối xác suất điều kiện như trong ví dụ ở trên. Cụ thể, một mô hình có hướng chứa một thành phần cho mỗi biến ngẫu nhiên x_i trong phân phối, và thành phần đó bao gồm phân phối điều kiện trên x_i khi biết các "cha"(parents) của x_i, được ký hiệu là $Pa_{\mathcal{G}}(x_i)$:

$$p(\mathbf{x}) = \prod_i p\left(x_i \mid Pa_{\mathcal{G}}(x_i)\right). \tag{3.53}$$

Xem Hình 3.7 để có một ví dụ về đồ thị có hướng và việc phân rã phân phối xác suất mà nó biểu diễn.

Mô hình vô hướng sử dụng các đồ thị với các cạnh không có hướng, và chúng biểu diễn sự phân rã thành một tập hợp các hàm số. Không giống như trường hợp có hướng, các hàm này thường không phải là các phân phối xác suất.

Bất kỳ tập hợp các nút nào trong đồ thị $\mathcal{G}$ mà tất cả các nút đều được kết nối trực tiếp với nhau được gọi là một clique. Mỗi clique $\mathcal{C}^{(i)}$ trong một mô hình vô hướng sẽ được gán với một hệ số (factor) $\phi^{(i)}(\mathcal{C}^{(i)})$. Các hệ số này chỉ là các hàm số, không nhất thiết phải là phân phối xác suất.



Hình 3.7: Mô hình đồ thị có hướng trên các biến ngẫu nhiên a, b, c, d và e. Đồ thị này tương ứng với các phân phối xác suất có thể được phân rã thành các thành phần như sau:

$$p(a, b, c, d, e) = p(a)p(b \mid a)p(c \mid a, b)p(d \mid b)p(e \mid c). \tag{3.73}$$

Mô hình đồ thị này giúp ta nhanh chóng nhận thấy một số thuộc tính của phân phối. Ví dụ, a và c tương tác trực tiếp, nhưng a và b chỉ tương tác gián tiếp thông qua c.

Đầu ra của mỗi thành phần (factor) phải không âm, nhưng không có ràng buộc nào yêu cầu tổng hay tích phân của thành phần đó phải bằng 1 như trong phân phối xác suất.

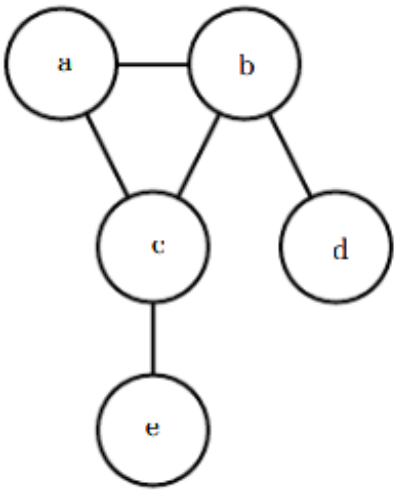
Xác suất của một cấu hình của các biến ngẫu nhiên tỷ lệ thuận với tích của tất cả các thành phần — những gán giá trị tạo ra giá trị thành phần lớn hơn thì càng có xác suất cao hơn. Tất nhiên, không có gì đảm bảo rằng tích này sẽ bằng 1. Do đó, ta chia cho một hằng số chuẩn hóa Z, được định nghĩa là tổng hoặc tích phân qua tất cả các trạng thái của tích các hàm ϕ để thu được một phân phối xác suất chuẩn hóa:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)}\left(c^{(i)}\right). \tag{3.74}$$

Xem Hình 3.8 để thấy ví dụ về một đồ thị vô hướng và sự phân tích thành các thành phần của phân phối xác suất mà nó biểu diễn.

Hãy nhớ rằng các biểu diễn đồ thị của sự phân tích thành nhân tử là một ngôn ngữ để mô tả các phân phối xác suất. Chúng không phải là những họ phân phối xác suất loại trừ lẫn nhau. Việc có hướng hay vô hướng không phải là thuộc tính của một phân phối xác suất; đó là thuộc tính của một mô tả cụ thể về một phân phối xác suất, nhưng bất kỳ phân phối xác suất nào cũng có thể được mô tả theo cả hai cách.

Trong suốt phần 1 và 2 của cuốn sách này, chúng tôi sử dụng các mô hình xác suất có cấu trúc.



Hình 3.8: Mô hình đồ thị vô hướng trên các biến ngẫu nhiên a, b, c, d và e. Đồ thị này tương ứng với các phân phối xác suất có thể được phân tích thành nhân tử như sau:

$$p(a, b, c, d, e) = \frac{1}{Z}\phi^{(1)}(a, b, c)\phi^{(2)}(b, d)\phi^{(3)}(c, e). \tag{3.75}$$

Mô hình đồ thị này cho phép chúng ta nhanh chóng nhận ra một số thuộc tính của phân phối. Ví dụ, a và c tương tác trực tiếp, nhưng a và e chỉ tương tác gián tiếp thông qua c.

Mô hình này chỉ đơn thuần là một ngôn ngữ để mô tả mối quan hệ xác suất trực tiếp mà các thuật toán học máy khác nhau lựa chọn để biểu diễn. Không cần phải hiểu thêm về các mô hình xác suất có cấu trúc cho đến khi thảo luận về các chủ đề nghiên cứu, trong phần 3, nơi chúng ta khám phá các mô hình xác suất có cấu trúc chi tiết hơn nhiều.

Chương này đã tổng hợp những khái niệm cơ bản của lý thuyết xác suất có ý nghĩa nhất đối với học sâu. Vẫn còn một nhóm công cụ toán học quan trọng khác cần khám phá: các phương pháp số.

CHƯƠNG 4. TÍNH TOÁN SỐ

Các thuật toán học máy thường yêu cầu một lượng lớn các phép tính số. Điều này thường đề cập đến các thuật toán giải quyết các bài toán toán học bằng những phương pháp cập nhật ước lượng của nghiệm thông qua một quá trình lặp, thay vì tìm kiếm một công thức biểu diễn nghiệm chính xác theo cách giải tích.

Các phép toán phổ biến bao gồm tối ưu hoá (tìm giá trị của đối số sao cho một hàm đạt cực tiểu hoặc cực đại) và giải hệ phương trình tuyến tính. Thậm chí, chỉ cần đánh giá một hàm toán học trên máy tính số cũng có thể gặp khó khăn khi hàm đó liên quan đến các số thực, bởi vì chúng không thể được biểu diễn chính xác hoàn toàn bằng một lượng bộ nhớ hữu hạn.

4.1 Tràn Số và Mất Dữ Liệu

Khó khăn cơ bản khi thực hiện các phép toán liên tục trên máy tính số là việc phải biểu diễn vô số các số thực bằng một số lượng bit hữu hạn. Điều này dẫn đến việc hầu hết các số thực khi được biểu diễn trong máy tính sẽ gặp một số sai số xấp xỉ. Trong nhiều trường hợp, đây chỉ là sai số làm tròn. Tuy nhiên, sai số này trở nên nghiêm trọng khi tích lũy qua nhiều phép toán, và có thể làm cho các thuật toán vốn hoạt động tốt về mặt lý thuyết lại thất bại trong thực tế nếu chúng không được thiết kế để giảm thiểu sự tích lũy của sai số làm tròn.

Một dạng sai số làm tròn đặc biệt nguy hiểm là mất dữ liệu (underflow). Hiện tượng này xảy ra khi các số có giá trị rất nhỏ gần 0 bị làm tròn thành 0. Nhiều hàm số sẽ cho kết quả khác biệt đáng kể khi đầu vào của chúng là 0 thay vì một giá trị dương rất nhỏ. Ví dụ, chúng ta thường muốn tránh phép chia cho 0 (trong một số môi trường phần mềm, điều này sẽ kích hoạt một ngoại lệ, hoặc trả về một giá trị đặc biệt như "không phải là số" — NaN). Tương tự, việc lấy logarit của 0 thường được xem là $-\infty$, điều này có thể dẫn đến các giá trị không xác định khi tham gia vào nhiều phép toán khác.

Một dạng sai số số học khác cũng gây hậu quả nghiêm trọng là tràn số (overflow). Tràn số xảy ra khi các số có độ lớn rất lớn được xấp xỉ thành ∞ hoặc $-\infty$. Các phép toán tiếp theo trên những giá trị này thường sẽ biến chúng thành các giá trị không xác định (NaN).

Một ví dụ về hàm cần được ổn định để tránh underflow và overflow là hàm softmax. Hàm softmax thường được sử dụng để dự đoán xác suất liên quan đến phân phối multinoulli. Softmax được định nghĩa là:

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}. \tag{4.1}$$

Xét trường hợp tất cả các x_i đều bằng một hằng số c. Về mặt lý thuyết, ta có thể thấy tất cả các đầu ra của softmax sẽ bằng $\frac{1}{n}$. Tuy nhiên, về mặt tính toán, điều này có thể không xảy ra khi c có độ lớn lớn.

Giả sử c rất lớn và mang giá trị âm, khi đó $\exp(c)$ sẽ tiến về 0, dẫn đến mẫu số của softmax bằng 0, khiến kết quả trở nên không xác định. Ngược lại, khi c rất lớn và dương, $\exp(c)$ sẽ tràn số (overflow), một lần nữa làm cho biểu thức trở nên không xác định.

Cả hai vấn đề này đều có thể được giải quyết bằng cách đánh giá softmax theo công thức:

$$\text{softmax}(\mathbf{z}) \text{ với } \mathbf{z} = \mathbf{x} - \max_i x_i \tag{4.2}$$

Theo phép biến đổi đại số đơn giản, giá trị của hàm softmax không thay đổi khi ta cộng hoặc trừ một hằng số vào toàn bộ vector đầu vào. Việc trừ $\max_i x_i$ sẽ đảm bảo rằng đối số lớn nhất của hàm mũ $\exp$ là 0, loại bỏ khả năng tràn số (overflow). Đồng thời, ít nhất một số hạng trong mẫu số sẽ có giá trị là 1, loại bỏ khả năng xảy ra hiện tượng hụt số (underflow) trong mẫu số dẫn đến chia cho 0.

Đây vẫn còn là một vấn đề nhỏ: hiện tượng underflow ở tử số có thể khiến biểu thức trả về kết quả là 0. Nếu ta tính $\log \text{softmax}(\mathbf{x})$ bằng cách chạy softmax trước rồi mới lấy log, ta có thể nhận kết quả $-\infty$ một cách sai lầm.

Do đó, cần phải xây dựng một hàm riêng để tính toán log softmax một cách ổn định về mặt số học. Hàm log softmax có thể được ổn định bằng mẹo tương tự như khi ổn định hàm softmax.

Phần lớn, chúng tôi không trình bày chi tiết tất cả các vấn đề về tính toán số liên quan đến việc triển khai các thuật toán khác nhau được mô tả trong cuốn sách này. Các nhà phát triển thư viện cấp thấp nên lưu ý đến các vấn đề số học khi triển khai các thuật toán học sâu. Phần lớn người đọc cuốn sách này có thể dựa vào các thư viện cấp thấp đã cung cấp các bản cài đặt ổn định. Trong một số trường hợp, có thể triển khai một thuật toán mới và cho phép bản cài đặt mới đó tự động ổn định. Theano *(Bergstra et al., 2010; Bastien et al., 2012)* là một ví dụ về gói phần mềm có khả năng tự động phát hiện và ổn định nhiều biểu thức thường xuyên không ổn định về mặt số học trong bối cảnh học sâu.

4.2 Điều kiện kém (Poor Conditioning)

Điều kiện (Conditioning) đề cập đến mức độ mà một hàm thay đổi khi có một sự thay đổi nhỏ trong đầu vào. Các hàm thay đổi nhanh khi đầu vào bị xáo trộn một chút có thể gây ra vấn đề trong tính toán khoa học, vì lỗi làm tròn trong đầu vào có thể dẫn đến thay đổi lớn trong kết quả đầu ra.

Xét hàm số $f(\mathbf{x}) = A^{-1}\mathbf{x}$. Khi $A \in \mathbb{R}^{n\times n}$ có phân rã giá trị riêng (eigenvalue decomposition), chỉ số điều kiện (condition number) của nó được định nghĩa như sau:

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \tag{4.3}$$

Đây là tỷ số giữa trị tuyệt đối của trị riêng lớn nhất và trị riêng nhỏ nhất. Khi chỉ số này lớn, việc nghịch đảo ma trận đặc biệt nhạy cảm với các sai số trong đầu vào.

Tính nhạy cảm này là một thuộc tính nội tại của chính ma trận, chứ không phải do lỗi làm tròn trong quá trình nghịch đảo ma trận. Các ma trận có điều kiện kém sẽ khuếch đại những sai số có sẵn khi ta nhân với nghịch đảo thực của ma trận. Trên thực tế, sai số còn trở nên trầm trọng hơn do các lỗi số học trong chính quá trình nghịch đảo.

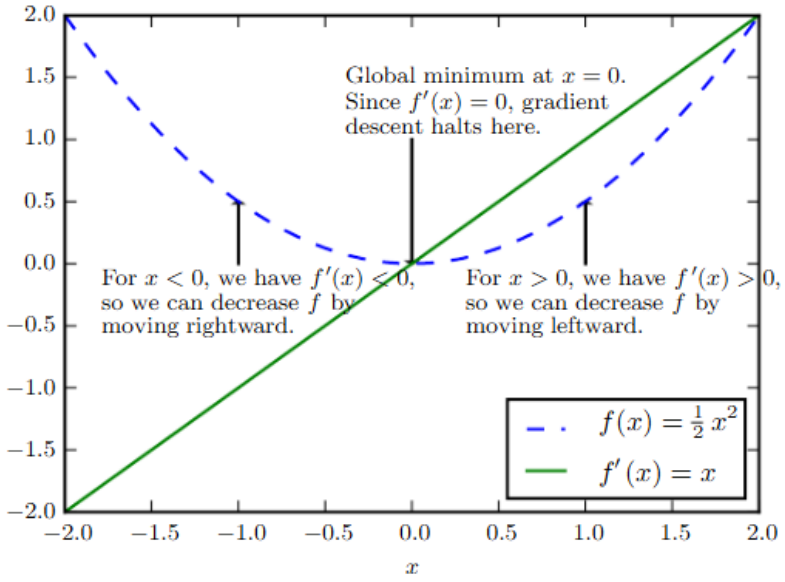
4.3 Tối ưu hóa dựa trên Gradient (Gradient-Based Optimization)

Hầu hết các thuật toán học sâu đều liên quan đến một dạng tối ưu hóa nào đó. Tối ưu hóa đề cập đến nhiệm vụ tối thiểu hóa hoặc tối đa hóa một hàm số $f(\mathbf{x})$ bằng cách thay đổi $\mathbf{x}$. Thông thường, chúng ta biểu diễn hầu hết các bài toán tối ưu hóa dưới dạng bài toán tối thiểu hóa $f(\mathbf{x})$. Việc tối đa hóa có thể được thực hiện thông qua một thuật toán tối thiểu hóa bằng cách cực tiểu hóa $-f(\mathbf{x})$.

Hàm mà ta muốn tối thiểu hóa hoặc tối đa hóa được gọi là hàm mục tiêu (objective function) hoặc tiêu chí (criterion). Khi ta tối thiểu hóa nó, ta cũng có thể gọi là hàm chi phí (cost function), hàm mất mát (loss function), hoặc hàm lỗi (error function). Trong cuốn sách này, chúng tôi sử dụng các thuật ngữ này thay thế cho nhau, mặc dù một số tài liệu về học máy gán ý nghĩa đặc biệt cho từng thuật ngữ.

Chúng ta thường ký hiệu giá trị cực tiểu hoặc cực đại của một hàm bằng dấu mũ $*$. Ví dụ, ta có thể viết:

$$\mathbf{x}^* = \arg\min f(\mathbf{x}). \tag{4.4}$$



Hình 4.1: Gradient descent. Minh họa cách thuật toán gradient descent sử dụng đạo hàm của một hàm số để lần theo hướng dốc xuống và tìm đến điểm cực tiểu.

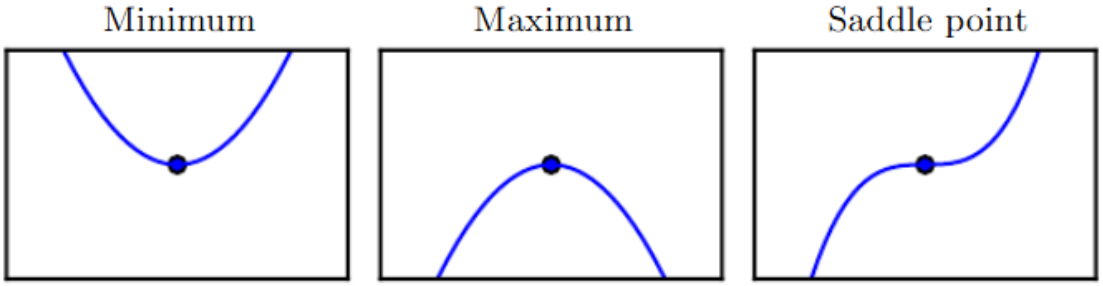
Giả sử ta có một hàm số $y = f(x)$, trong đó cả x và y đều là các số thực. Đạo hàm của hàm này được ký hiệu là $f'(x)$ hoặc $\frac{dy}{dx}$. Đạo hàm $f'(x)$ cho biết độ dốc của $f(x)$ tại điểm x. Nói cách khác, nó xác định cách thức thay đổi nhỏ ở đầu vào sẽ tác động đến đầu ra

như thế nào:

$$f(x+\epsilon) \approx f(x) + \epsilon f'(x). \tag{4.5}$$

Do đó, đạo hàm rất hữu ích trong việc cực tiểu hóa một hàm vì nó cho ta biết cần thay đổi x như thế nào để tạo ra sự cải thiện nhỏ trong y. Ví dụ, ta biết rằng $f(x-\epsilon \operatorname{sign}(f'(x)))$ nhỏ hơn $f(x)$ khi ϵ đủ nhỏ. Vì vậy, ta có thể giảm $f(x)$ bằng cách điều chỉnh x theo những bước nhỏ theo hướng ngược lại với dấu của đạo hàm. Kỹ thuật này được gọi là gradient descent (*Cauchy, 1847*). Xem Hình 4.1 để có ví dụ minh họa cho kỹ thuật này.

Khi $f'(x) = 0$, đạo hàm không cung cấp thông tin về hướng di chuyển. Các điểm mà $f'(x) = 0$ được gọi là điểm tới hạn (*critical points*) hoặc điểm dừng (*stationary points*). Một cực tiểu địa phương (*local minimum*) là một điểm mà $f(x)$ nhỏ hơn tất cả các điểm lân cận, vì vậy không thể giảm thêm $f(x)$ bằng cách thực hiện những bước nhỏ. Ngược lại, một cực đại địa phương (*local maximum*) là điểm mà $f(x)$ lớn hơn tất cả các điểm lân cận.



Hình 4.2: Các loại điểm tới hạn. Ví dụ về ba loại điểm tới hạn trong không gian một chiều. Một điểm tới hạn là điểm có độ dốc bằng không. Điểm này có thể là cực tiểu địa phương (local minimum), nơi giá trị nhỏ hơn so với các điểm lân cận; cực đại địa phương (local maximum), nơi giá trị lớn hơn so với các điểm lân cận; hoặc điểm yên ngựa (saddle point), nơi các điểm lân cận có giá trị vừa lớn hơn vừa nhỏ hơn chính nó.

Trong quá trình tìm cực trị của một hàm số $f(x)$, có những điểm mà tại đó không thể tăng giá trị của $f(x)$ bằng cách thực hiện những bước nhỏ vô hạn. Những điểm như vậy được gọi là điểm tới hạn.

Không phải tất cả các điểm tới hạn đều là cực đại hoặc cực tiểu. Một số điểm tới hạn có tính chất đặc biệt, gọi là điểm yên ngựa (saddle points). Tại những điểm này, hàm số có thể giảm theo một hướng và tăng theo một hướng khác, tạo ra sự không ổn định trong quá trình tối ưu hóa. Xem Hình 4.2 để thấy ví dụ về từng loại điểm tới hạn.

Một điểm mà tại đó $f(x)$ đạt giá trị thấp nhất tuyệt đối được gọi là cực tiểu toàn cục (global minimum). Một hàm số có thể chỉ có một cực tiểu toàn cục hoặc có nhiều cực tiểu toàn cục. Ngoài ra, có thể tồn tại các cực tiểu địa phương không phải là cực tiểu toàn cục.

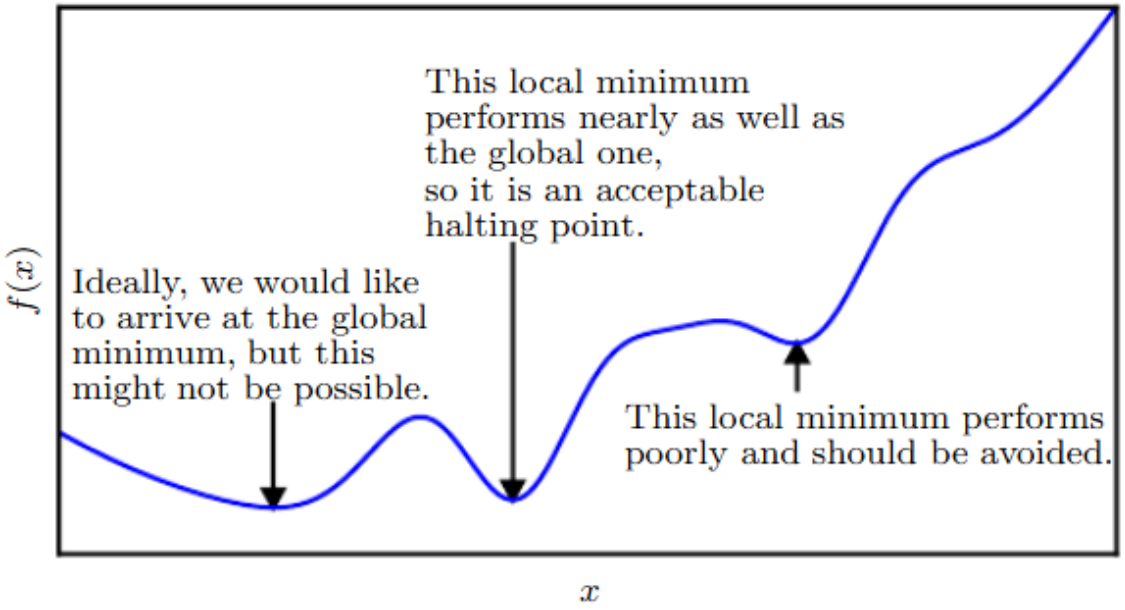
Trong bối cảnh học sâu, chúng ta tối ưu hóa các hàm số có thể có nhiều cực tiểu địa

phương không tối ưu và nhiều điểm yên ngựa được bao quanh bởi các vùng rất phẳng. Tất cả những điều này làm cho việc tối ưu hóa trở nên khó khăn, đặc biệt là khi đầu vào của hàm số là đa chiều. Do đó, chúng ta thường chấp nhận tìm một giá trị của f đủ thấp nhưng không nhất thiết phải là giá trị nhỏ nhất theo nghĩa chặt chẽ. Xem Hình 4.3 để biết ví dụ minh họa.

Chúng ta thường tối ưu hóa các hàm số có nhiều đầu vào: $f : \mathbb{R}^n \to \mathbb{R}$. Để khái niệm *"tối ưu hóa"* có ý nghĩa, đầu ra của hàm số đó phải là một giá trị vô hướng duy nhất.

Với các hàm số có nhiều đầu vào, chúng ta cần sử dụng khái niệm đạo hàm riêng (partial derivatives). Đạo hàm riêng $\frac{\partial}{\partial x_i} f(x)$ đo lường sự thay đổi của f khi chỉ có biến x_i thay đổi tại điểm x.

Gradient là sự tổng quát hóa khái niệm đạo hàm cho trường hợp đạo hàm theo một vector: gradient của f là vector chứa tất cả các đạo hàm riêng, được ký hiệu là $\nabla_x f(x)$. Phần tử thứ i của gradient là đạo hàm riêng của f theo x_i. Trong không gian nhiều chiều, các điểm tới hạn là những điểm mà mọi phần tử của gradient đều bằng không.



Hình 4.3: Tối ưu hóa xấp xỉ. Các thuật toán tối ưu hóa có thể không tìm được cực tiểu toàn cục khi tồn tại nhiều cực tiểu cục bộ hoặc các vùng bình nguyên. Trong bối cảnh học sâu, chúng ta thường chấp nhận các nghiệm như vậy dù chúng không thực sự là cực tiểu tuyệt đối, miễn là chúng tương ứng với các giá trị đủ thấp của hàm chi phí.

Đạo hàm theo hướng (directional derivative) theo hướng $\mathbf{u}$ (một vector đơn vị) cho biết độ dốc của hàm f theo hướng $\mathbf{u}$. Nói cách khác, đạo hàm theo hướng là đạo hàm của hàm $f(\mathbf{x} + \alpha\mathbf{u})$ theo α, được tính tại $\alpha = 0$. Sử dụng quy tắc chuỗi, ta có:

$$\frac{\partial}{\partial \alpha} f(\mathbf{x} + \alpha\mathbf{u})\Big|_{\alpha=0} = \mathbf{u}^\top \nabla_x f(\mathbf{x}) \tag{4.6}$$

Để tối thiểu hóa f, ta muốn tìm hướng mà f giảm nhanh nhất. Có thể làm điều này bằng

cách sử dụng đạo hàm theo hướng:

$$\min_{\mathbf{u},\mathbf{u}^\top\mathbf{u}=1} \mathbf{u}^\top \nabla_x f(\mathbf{x}) \tag{4.7}$$

$$= \min_{\mathbf{u},\mathbf{u}^\top\mathbf{u}=1} \|\mathbf{u}\|_2 \|\nabla_x f(\mathbf{x})\|_2 \cos\theta \tag{4.8}$$

Trong đó, θ là góc giữa $\mathbf{u}$ và gradient. Với điều kiện $\|\mathbf{u}\|_2 = 1$ và bỏ qua các yếu tố không phụ thuộc vào $\mathbf{u}$, bài toán trở thành $\min_{\mathbf{u}} \cos\theta$. Bài toán này đạt giá trị nhỏ nhất khi $\mathbf{u}$ chỉ theo hướng ngược lại với gradient. Nói cách khác, gradient trỏ theo hướng leo dốc, trong khi gradient âm trỏ theo hướng đi xuống.

Do đó, ta có thể giảm f bằng cách di chuyển theo hướng của gradient âm. Phương pháp này được gọi là phương pháp hạ dốc nhất (method of steepest descent), hay thường được biết đến là gradient descent.

Phương pháp hạ gradient đề xuất một điểm mới như sau:

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_x f(\mathbf{x}) \tag{4.9}$$

Trong đó, ϵ là tốc độ học (learning rate), một hằng số dương xác định kích thước của bước di chuyển. Chúng ta có thể chọn ϵ theo nhiều cách khác nhau. Một cách phổ biến là đặt ϵ là một hằng số nhỏ. Đôi khi, ta có thể giải để tìm kích thước bước làm cho đạo hàm theo hướng triệt tiêu. Một cách tiếp cận khác là đánh giá $f(\mathbf{x} - \epsilon\nabla_x f(\mathbf{x}))$ với nhiều giá trị ϵ khác nhau và chọn giá trị cho kết quả nhỏ nhất của hàm mục tiêu. Chiến lược này được gọi là tìm kiếm theo đường thẳng (line search).

Phương pháp hạ gradient hội tụ khi mọi phần tử của gradient bằng không (hoặc trong thực tế là rất gần bằng không). Trong một số trường hợp, ta có thể tránh việc lặp lại thuật toán và nhảy trực tiếp đến điểm tới hạn bằng cách giải phương trình:$\nabla_x f(\mathbf{x}) = 0$ theo x

Mặc dù phương pháp gradient descent bị giới hạn trong việc tối ưu hóa trên không gian liên tục, ý tưởng lặp lại việc thực hiện một bước nhỏ (gần như là bước đi tốt nhất) để tiến tới cấu hình tốt hơn có thể được khái quát hóa cho không gian rời rạc. Việc leo lên giá trị của một hàm mục tiêu trong không gian tham số rời rạc được gọi là leo đồi (hill climbing) *(Russel and Norvig, 2003)*.

4.3.1 Vượt ra ngoài Gradient: Ma trận Jacobian và Hessian

Đôi khi, chúng ta cần tìm tất cả các đạo hàm riêng của một hàm mà đầu vào và đầu ra đều là các vector. Ma trận chứa tất cả các đạo hàm riêng đó được gọi là ma trận Jacobian. Cụ thể, nếu ta có một hàm $f : \mathbb{R}^m \to \mathbb{R}^n$, thì ma trận Jacobian $\mathbf{J} \in \mathbb{R}^{n\times m}$ của f được định nghĩa là $J_{i,j} = \frac{\partial}{\partial x_j} f(x)_i$.

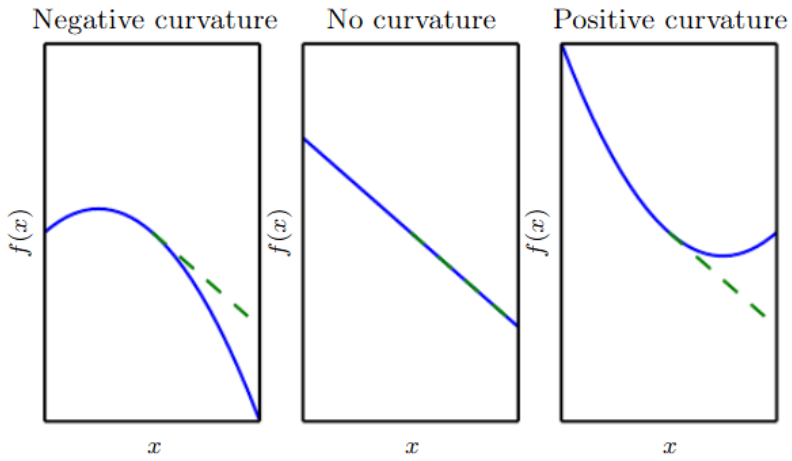
Đôi khi, chúng ta cũng quan tâm đến đạo hàm của một đạo hàm, hay còn gọi là đạo

hàm bậc hai. Ví dụ, với một hàm $f : \mathbb{R}^n \to \mathbb{R}$, đạo hàm theo x_i của đạo hàm của f theo x_j được ký hiệu là $\frac{\partial^2}{\partial x_i \partial x_j} f$.

Trong trường hợp một chiều, ta có thể ký hiệu $\frac{d^2}{dx^2} f$ là $f''(x)$. Đạo hàm bậc hai cho chúng ta biết đạo hàm bậc nhất sẽ thay đổi như thế nào khi ta thay đổi đầu vào. Điều này quan trọng vì nó cho biết liệu một bước theo gradient có cải thiện nhiều như mong đợi hay không dựa trên gradient đơn thuần.

Ta có thể nghĩ về đạo hàm bậc hai như là phép đo độ cong (curvature). Giả sử ta có một hàm bậc hai (nhiều hàm trong thực tế không phải là bậc hai nhưng có thể xấp xỉ là bậc hai ở phạm vi nhỏ). Nếu một hàm như vậy có đạo hàm bậc hai bằng không, thì không có độ cong nào cả — đồ thị hàm là một đường phẳng hoàn hảo và giá trị của nó có thể được dự đoán chỉ bằng cách sử dụng gradient.

Nếu gradient là 1, ta có thể thực hiện một bước với kích thước ϵ dọc theo hướng gradient âm, và giá trị hàm số sẽ giảm đi ϵ. Nếu đạo hàm bậc hai là âm, hàm sẽ cong xuống, do đó giá trị hàm số sẽ giảm nhiều hơn ϵ. Cuối cùng, nếu đạo hàm bậc hai là dương, hàm sẽ cong lên, làm cho giá trị hàm giảm ít hơn ϵ.



Hình 4.4: Đạo hàm bậc hai xác định độ cong của một hàm. Ở đây, chúng ta minh họa các hàm bậc hai với những độ cong khác nhau. Đường nét đứt cho thấy giá trị của hàm mất mát mà ta mong đợi dựa trên thông tin từ gradient khi thực hiện một bước theo hướng giảm dần của gradient. Với độ cong âm, hàm mất mát thực sự giảm nhanh hơn so với dự đoán của gradient. Khi không có độ cong, gradient dự đoán chính xác mức giảm. Với độ cong dương, hàm giảm chậm hơn so với mong đợi và cuối cùng bắt đầu tăng lên. Do đó, nếu các bước đi quá lớn, giá trị hàm có thể vô tình tăng lên.

Nhìn Hình 4.4 để thấy được cách mà các dạng độ cong khác nhau ảnh hưởng đến mối quan hệ giữa giá trị của hàm mất mát dự đoán bởi gradient và giá trị thực

Khi một hàm số có nhiều biến đầu vào, chúng ta có thể tính đạo hàm bậc hai theo từng cặp biến. Các đạo hàm bậc hai này được gom lại thành một ma trận gọi là ma trận Hessian. Ma trận Hessian $\mathbf{H}(f)(\mathbf{x})$ được định nghĩa như sau:

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}). \tag{4.10}$$

Tương đương, Hessian có thể xem là ma trận Jacobian của gradient.

Nếu các đạo hàm bậc hai của hàm số liên tục, thứ tự lấy đạo hàm có thể hoán đổi:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\mathbf{x}). \tag{4.11}$$

Điều này có nghĩa là $H_{i,j} = H_{j,i}$, nên ma trận Hessian là đối xứng tại những điểm đó. Trong lĩnh vực học sâu, phần lớn các hàm số mà ta làm việc có ma trận Hessian đối xứng gần như ở mọi nơi.

Vì Hessian là ma trận thực và đối xứng, ta có thể phân rã nó thành một tập giá trị riêng thực và một cơ sở trực chuẩn của vector riêng. Đạo hàm bậc hai theo một hướng nhất định, được biểu diễn bởi vector đơn vị $\mathbf{d}$, được tính bằng $\mathbf{d}^T\mathbf{H}\mathbf{d}$. Nếu $\mathbf{d}$ là một vector riêng của $\mathbf{H}$, thì đạo hàm bậc hai theo hướng đó chính là giá trị riêng tương ứng. Nếu $\mathbf{d}$ không phải là vector riêng, đạo hàm bậc hai theo hướng $\mathbf{d}$ là một trung bình có trọng số của tất cả các giá trị riêng, với trọng số từ 0 đến 1. Các vector riêng có góc nhỏ hơn với $\mathbf{d}$ sẽ có trọng số cao hơn. Giá trị riêng lớn nhất xác định độ cong lớn nhất của hàm (đạo hàm bậc hai lớn nhất), còn giá trị riêng nhỏ nhất xác định độ cong nhỏ nhất.

Đạo hàm bậc hai theo một hướng nhất định cho ta biết mức độ hiệu quả của bước nhảy trong thuật toán Gradient Descent. Chúng ta có thể xấp xỉ Taylor bậc hai của hàm số $f(x)$ quanh điểm hiện tại $x^{(0)}$:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^T\mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(0)})^T\mathbf{H}(\mathbf{x} - \mathbf{x}^{(0)}), \tag{4.12}$$

trong đó $\mathbf{g}$ là gradient và $\mathbf{H}$ là Hessian tại $x^{(0)}$. Nếu ta sử dụng một tốc độ học ϵ, thì điểm mới x sẽ được thay thế thành $x^{(0)} - \epsilon\mathbf{g}$., ta có:

$$f(x^{(0)} - \epsilon\mathbf{g}) \approx f(x^{(0)}) - \epsilon\mathbf{g}^T\mathbf{g} + \frac{1}{2}\epsilon^2\mathbf{g}^T\mathbf{H}\mathbf{g}. \tag{4.13}$$

Có ba thành phần trong phương trình này:

- Giá trị ban đầu của hàm số.

- Sự cải thiện do gradient quyết định.

- Sự điều chỉnh do độ cong của hàm số quyết định.

Nếu thành phần cuối quá lớn, Gradient Descent có thể di chuyển ngược hướng tối ưu. Khi $\mathbf{g}^T\mathbf{H}\mathbf{g}$ bằng không hoặc âm, việc tăng ϵ sẽ làm cho hàm số tăng lên mãi mãi. Trong thực tế, Taylor series chỉ chính xác trong một khoảng nhỏ, nên ta cần chọn ϵ hợp lý. Khi

$\mathbf{g}^T\mathbf{H}\mathbf{g}$ dương, giá trị ϵ tối ưu là:

$$\epsilon^* = \frac{\mathbf{g}^T\mathbf{g}}{\mathbf{g}^T\mathbf{H}\mathbf{g}}. \tag{4.14}$$

Trong trường hợp xấu nhất, khi $\mathbf{g}$ thẳng hàng với vector riêng của $\mathbf{H}$ ứng với giá trị riêng lớn nhất $\lambda_{\max}$, thì kích bước nhảy tối ưu sẽ cho bởi $\epsilon^* = \frac{1}{\lambda_{\max}}$. Ở mức độ mà hàm chúng ta cần tối ưu có thể được xấp xỉ tốt bởi một hàm bậc hai, các giá trị riêng của ma trận Hessian sẽ quyết định phạm vi của tốc độ học.

Đạo hàm bậc hai có thể được sử dụng để xác định liệu một điểm tới hạn có phải là cực đại địa phương, cực tiểu địa phương hay điểm yên ngựa hay không. Nhắc lại rằng tại một điểm tới hạn, ta có $f'(x) = 0$. Nếu đạo hàm bậc hai thoả mãn $f''(x) > 0$, thì đạo hàm bậc nhất $f'(x)$ tăng dần khi di chuyển về bên phải và giảm dần khi di chuyển về bên trái. Điều này có nghĩa là tại điểm x, hàm số có xu hướng đi xuống ở bên trái và đi lên ở bên phải, tạo thành một điểm cực tiểu địa phương. Tương tự, nếu $f'(x) = 0$ và $f''(x) < 0$, thì đạo hàm bậc nhất giảm khi đi về bên phải và tăng khi đi về bên trái, dẫn đến điểm cực đại địa phương. Điều này được gọi là kiểm tra đạo hàm bậc hai. Tuy nhiên, nếu $f''(x) = 0$, phép thử không thể đưa ra kết luận được, và điểm x có thể là một điểm yên ngựa hoặc một phần của miền bằng phẳng.

Trong không gian nhiều chiều, để xác định một điểm tới hạn x có phải là cực đại, cực tiểu hay điểm yên ngựa, ta cần kiểm tra tất cả các đạo hàm bậc hai của hàm số. Cách tiếp cận phổ biến là sử dụng ma trận Hessian $H(x)$ và phân tích các trị riêng của nó.

Tại một điểm tới hạn x, tức là $\nabla f(x) = 0$, ta có thể xác định bản chất của điểm này dựa trên các trị riêng của ma trận Hessian:

- Nếu Hessian là dương xác định (tất cả các trị riêng đều dương), x là một điểm cực tiểu.
- Nếu Hessian là âm xác định (tất cả các trị riêng đều âm), x là một điểm cực đại.
- Nếu Hessian có cả trị riêng dương và âm, x là một điểm yên ngựa.
- Nếu có ít nhất một trị riêng bằng 0, kiểm tra này không kết luận được và ta cần sử dụng phương pháp khác.

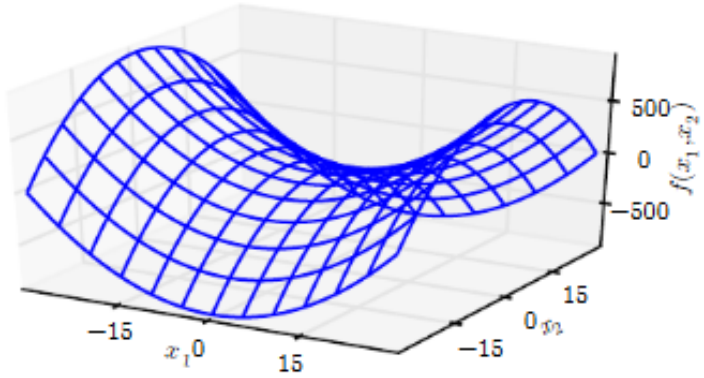
Trong không gian nhiều chiều, giá trị của các trị riêng giúp ta hiểu về độ cong của hàm số theo các hướng khác nhau:

- Nếu tất cả các hướng đều cong lên (trị riêng dương), hàm số có một cực tiểu.
- Nếu tất cả các hướng đều cong xuống (trị riêng âm), hàm số có một cực đại.
- Nếu có hướng cong lên và hướng cong xuống, ta đang ở điểm yên ngựa.
- Nếu có ít nhất một trị riêng bằng 0, ta cần kiểm tra thêm vì có thể tồn tại vùng phẳng.

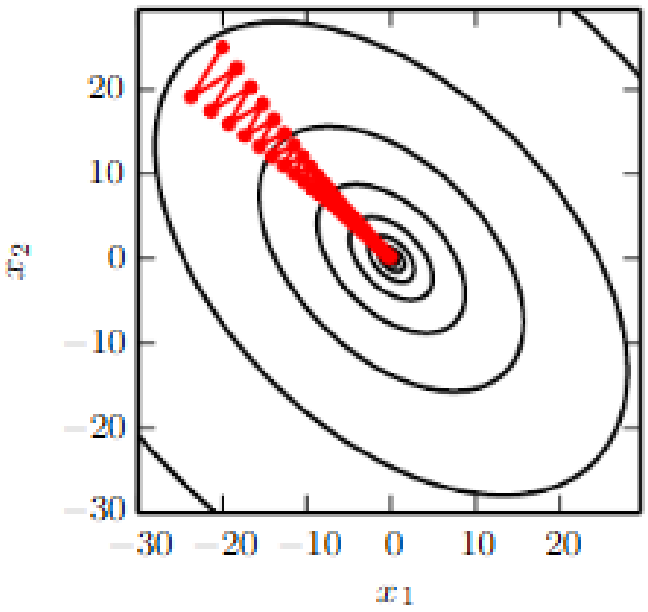
Ví dụ minh họa ở Hình 4.5

Cuối cùng, kiểm tra đạo hàm bậc hai trong không gian nhiều chiều có thể không đưa ra kết luận, giống như trong trường hợp một biến số. Việc kiểm tra này không kết luận được khi tất cả các trị riêng khác 0 có cùng dấu, nhưng ít nhất một trị riêng bằng 0. Điều này xảy ra vì trong mặt cắt tương ứng với trị riêng bằng 0, kiểm tra đạo hàm bậc hai một biến cũng không đưa ra kết luận rõ ràng.

Trong không gian nhiều chiều, tại một điểm, tồn tại nhiều đạo hàm bậc hai khác nhau theo từng hướng. Số điều kiện của ma trận Hessian tại điểm đó đo lường mức độ khác biệt giữa các đạo hàm bậc hai này. Khi ma trận Hessian có số điều kiện kém (tức là có giá trị riêng lớn và nhỏ chênh lệch đáng kể), phương pháp hạ dốc (gradient descent) hoạt động kém hiệu quả. Nguyên nhân là do đạo hàm theo một hướng có thể tăng nhanh, trong khi theo hướng khác lại tăng chậm. Gradient descent không nhận thức được sự khác biệt này, nên không biết rằng cần ưu tiên khám phá theo hướng mà đạo hàm vẫn còn âm trong thời gian dài hơn. Số điều kiện kém cũng khiến việc chọn kích thước bước (step size) trở nên khó khăn. Kích thước bước phải đủ nhỏ để tránh vượt quá cực tiểu và đi lên ở những hướng có độ cong dương mạnh. Tuy nhiên, điều này thường dẫn đến bước nhảy quá nhỏ, khiến tiến trình bị chậm lại ở những hướng có độ cong yếu hơn. Ví dụ minh họa ở Hình 4.6



Hình 4.5: Một điểm yên ngựa là một điểm trong không gian mà tại đó hàm số thể hiện cả độ cong dương và độ cong âm theo các hướng khác nhau. Một ví dụ điển hình là hàm số $f(x) = x_1^2 - x_2^2$. Dọc theo trục x_1, hàm số cong lên trên, thể hiện đặc trưng của một cực tiểu cục bộ. Trục này là một trị riêng của ma trận Hessian với trị riêng dương. Trong khi đó, dọc theo trục x_2, hàm số cong xuống dưới, cho thấy đặc trưng của một cực đại cục bộ. Trục này là một trị riêng của Hessian với trị riêng âm. Tên gọi *điểm yên ngựa* xuất phát từ hình dạng của đồ thị hàm số này, giống như một chiếc yên ngựa. Đây là một ví dụ tiêu biểu của một hàm số có điểm yên ngựa. Trong không gian nhiều chiều, một điểm yên ngựa không nhất thiết phải có một trị riêng bằng không. Điều quan trọng là tại điểm đó tồn tại cả trị riêng dương và trị riêng âm. Chúng ta có thể hình dung một điểm yên ngựa như một điểm mà trong một mặt cắt nào đó, nó là cực đại cục bộ, nhưng trong một mặt cắt khác, nó lại là cực tiểu cục bộ.



Hình 4.6: Thuật toán hạ gradient (Gradient Descent) không tận dụng được đầy đủ thông tin về độ cong có trong ma trận Hessian. Giả sử ta sử dụng hạ gradient để tối ưu một hàm bậc hai $f(x)$ có ma trận Hessian với số điều kiện bằng 5. Điều này có nghĩa là hướng có độ cong lớn nhất gấp năm lần hướng có độ cong nhỏ nhất. Trong trường hợp này, độ cong lớn nhất nằm theo hướng $[1, 1]^T$, trong khi độ cong nhỏ nhất nằm theo hướng $[1, -1]^T$. Các đường màu đỏ chỉ ra quỹ đạo của quá trình hạ gradient. Hàm bậc hai này có dạng rất thuôn dài, giống như một hẻm núi dài. Do đó, hạ gradient thường lãng phí thời gian khi liên tục đi xuống hai bên vách hẻm vì chúng có độ dốc lớn nhất. Khi bước nhảy quá lớn, thuật toán có xu hướng vượt qua đáy của hàm số và cần phải đi xuống vách đối diện trong lần lặp tiếp theo. Trị riêng dương lớn của Hessian theo hướng này cho thấy rằng đạo hàm theo hướng đó tăng nhanh, do đó, một thuật toán tối ưu dựa trên Hessian có thể dự đoán rằng hướng có độ dốc lớn nhất thực ra không phải là hướng tìm kiếm tốt nhất trong trường hợp này.

Một trong những phương pháp tối ưu hóa sử dụng thông tin từ ma trận Hessian là phương pháp Newton. Phương pháp này dựa trên khai triển Taylor bậc hai để xấp xỉ một hàm số $f(\mathbf{x})$ gần một điểm $\mathbf{x}^{(0)}$:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^T \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(0)})^T \mathbf{H}(f)(\mathbf{x}^{(0)})(\mathbf{x} - \mathbf{x}^{(0)}). \quad (4.15)$$

Giải phương trình tìm điểm dừng của hàm này, ta thu được:

$$\mathbf{x}^* = \mathbf{x}^{(0)} - \mathbf{H}(f)(\mathbf{x}^{(0)})^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}). \quad (4.16)$$

Khi hàm số f là một hàm bậc hai xác định dương, phương pháp Newton có thể tìm cực tiểu của hàm số này chỉ sau một lần cập nhật bằng phương trình 4.16. Trong trường hợp f không hoàn toàn là hàm bậc hai nhưng có thể xấp xỉ cục bộ bằng một hàm bậc hai xác định dương, phương pháp Newton sẽ được lặp đi lặp lại nhiều lần bằng cách áp

dụng phương trình 4.16 nhiều lần. Việc liên tục cập nhật xấp xỉ và nhảy tới điểm cực tiểu của hàm số giúp phương pháp Newton hội tụ nhanh hơn nhiều so với thuật toán gradient descent thông thường. Đây là một ưu điểm đáng kể khi điểm tới gần là một điểm cực tiểu địa phương. Tuy nhiên, phương pháp này có thể gặp vấn đề khi điểm tới gần là điểm yên ngựa (saddle point). Như đã đề cập trong mục (8.2.3), phương pháp Newton chỉ đảm bảo hoạt động hiệu quả khi điểm tới gần là một điểm cực tiểu (tất cả các trị riêng của ma trận Hessian đều dương). Trong khi đó, thuật toán gradient descent không bị hút về điểm yên ngựa, trừ khi gradient trực tiếp hướng về phía chúng.

Các thuật toán tối ưu hóa sử dụng thông tin về gradient, chẳng hạn như gradient descent, được gọi là thuật toán tối ưu bậc nhất. Trong khi đó, các thuật toán tối ưu hóa sử dụng thêm ma trận Hessian, như phương pháp Newton, được gọi là thuật toán tối ưu bậc hai (*Nocedal and Wright, 2006*).

Trong hầu hết các ngữ cảnh được đề cập trong cuốn sách này, các thuật toán tối ưu hóa có thể áp dụng cho nhiều loại hàm khác nhau, nhưng chúng thường không đi kèm với đảm bảo chắc chắn về hiệu suất. Các thuật toán học sâu cũng vậy: chúng thường không có sự đảm bảo chặt chẽ vì các hàm trong học sâu rất phức tạp và khó phân tích một cách chính xác.

Trong nhiều lĩnh vực khác, cách tiếp cận phổ biến để tối ưu hóa là thiết kế thuật toán tối ưu hóa cho một nhóm hàm cụ thể, giúp việc phân tích và đảm bảo tính đúng đắn trở nên dễ dàng hơn. Tuy nhiên, trong học sâu, chúng ta phải đối mặt với một không gian hàm rất rộng và phi tuyến tính, dẫn đến việc khó có thể đảm bảo rằng thuật toán tối ưu hóa luôn hội tụ hoặc đạt hiệu suất tốt.

Do đó, thay vì tìm kiếm một thuật toán tối ưu hóa hoàn hảo, học sâu tập trung vào việc phát triển các phương pháp thực nghiệm, tinh chỉnh siêu tham số và tận dụng sức mạnh tính toán để đạt được hiệu suất tốt nhất có thể.

Trong bối cảnh học sâu, đôi khi chúng ta có thể đạt được một số đảm bảo bằng cách giới hạn các hàm được sử dụng vào các hàm liên tục Lipschitz hoặc có đạo hàm liên tục Lipschitz. Một hàm f được gọi là liên tục Lipschitz nếu tốc độ thay đổi của nó bị chặn bởi một hằng số Lipschitz $\mathcal{L}$:

$$forall\mathbf{x}, \mathbf{y},\ |f(\mathbf{x}) - f(\mathbf{y})| \leq \mathcal{L}\|\mathbf{x} - \mathbf{y}\|_2. \tag{4.17}$$

Thuộc tính này rất hữu ích vì nó giúp chúng ta định lượng giả thuyết rằng một thay đổi nhỏ trong đầu vào do một thuật toán như gradient descent thực hiện sẽ dẫn đến một thay đổi nhỏ tương ứng trong đầu ra. Hơn nữa, tính liên tục Lipschitz là một ràng buộc khá yếu, và nhiều bài toán tối ưu trong học sâu có thể được làm cho liên tục Lipschitz với một số điều chỉnh nhỏ.

Một trong những lĩnh vực thành công nhất của tối ưu hóa chuyên biệt là tối ưu lồi. Các thuật toán tối ưu lồi có thể đưa ra nhiều đảm bảo hơn bằng cách áp đặt các ràng buộc chặt

chẽ hơn. Các thuật toán này chỉ áp dụng cho các hàm lồi—những hàm mà Hessian luôn là nửa xác định dương ở mọi nơi.

Những hàm này có tính chất đặc biệt tốt vì chúng không có điểm yên ngựa (saddle points), và mọi điểm cực tiểu cục bộ của chúng đều là điểm cực tiểu toàn cục. Tuy nhiên, hầu hết các bài toán trong học sâu đều khó biểu diễn theo dạng tối ưu lồi. Trên thực tế, tối ưu lồi thường chỉ được sử dụng như một phần nhỏ trong một số thuật toán học sâu.

Các ý tưởng từ phân tích tối ưu lồi có thể hữu ích để chứng minh sự hội tụ của các thuật toán học sâu, nhưng nhìn chung, vai trò của tối ưu lồi bị giảm đi đáng kể trong bối cảnh học sâu.

Để tìm hiểu thêm về tối ưu lồi, bạn có thể tham khảo *Boyd and Vandenberghe (2004)* hoặc *Rockafellar (1997)*.

4.4 Tối Ưu Có Ràng Buộc

Đôi khi, chúng ta không chỉ muốn tối đa hóa hoặc tối thiểu hóa một hàm $f(\mathbf{x})$ trên toàn bộ miền giá trị của $\mathbf{x}$, mà thay vào đó, ta muốn tìm giá trị cực đại hoặc cực tiểu của $f(\mathbf{x})$ khi $\mathbf{x}$ thuộc một tập hợp giới hạn $\mathbb{S}$. Đây được gọi là tối ưu có ràng buộc. Các điểm $\mathbf{x}$ nằm trong tập hợp $\mathbb{S}$ được gọi là điểm khả thi trong thuật ngữ tối ưu hóa có ràng buộc.

Chúng ta thường muốn tìm một nghiệm có kích thước nhỏ theo một tiêu chí nào đó. Một cách phổ biến để làm điều này là áp đặt một ràng buộc về chuẩn, chẳng hạn như $\|\mathbf{x}\| \leq 1$.

Một cách tiếp cận đơn giản để tối ưu có ràng buộc là điều chỉnh thuật toán hạ dốc (gradient descent) để tính đến ràng buộc. Nếu chúng ta sử dụng một bước nhảy nhỏ ϵ, ta có thể thực hiện các bước gradient descent như bình thường, sau đó chiếu kết quả về lại miền khả thi $\mathbb{S}$. Nếu sử dụng tìm kiếm đường (line search), ta có thể chỉ tìm kiếm các kích thước bước ϵ sao cho điểm mới $\mathbf{x}$ vẫn nằm trong miền khả thi, hoặc chiếu mỗi điểm trên đường đi về lại miền khả thi. Khi có thể, phương pháp này có thể được làm hiệu quả hơn bằng cách chiếu gradient vào không gian tiếp tuyến của miền khả thi trước khi thực hiện bước nhảy hoặc bắt đầu tìm kiếm đường (*Rosen, 1960*)

Một cách tiếp cận tiên tiến hơn để giải bài toán tối ưu có ràng buộc là biến đổi nó thành một bài toán về tối ưu hóa không ràng buộc bằng cách cực tiểu hóa $g(\theta) = f([\cos\theta, \sin\theta]^\top)$ theo θ, sau đó lấy nghiệm $[\cos\theta, \sin\theta]$ làm lời giải cho bài toán ban đầu. Cách tiếp cận này đòi hỏi sự sáng tạo vì việc chuyển đổi giữa các bài toán tối ưu phải được thiết kế đặc biệt cho từng trường hợp cụ thể.

Phương pháp Karush-Kuhn-Tucker (KKT) cung cấp một cách tiếp cận [1] tổng quát để giải bài toán tối ưu có ràng buộc. Trong phương pháp này, ta sử dụng một hàm mới gọi là Lagrangian tổng quát (generalized Lagrangian) hay hàm Lagrange tổng quát.

Để định nghĩa hàm Lagrange, trước tiên ta cần mô tả tập nghiệm $\mathcal{S}$ dưới dạng các

[1] Phương pháp KKT mở rộng phương pháp nhân tử Lagrange (Lagrange multipliers), phương pháp này chỉ áp dụng cho các ràng buộc phương trình mà không hỗ trợ các ràng buộc bất đẳng thức.

phương trình và bất đẳng thức. Cụ thể, giả sử ta có m hàm $g^{(i)}(\mathbf{x})$ và n hàm $h^{(j)}(\mathbf{x})$, khi đó tập nghiệm có thể được mô tả như sau:

$$\mathcal{S} = \{\mathbf{x} \mid forall i, g^{(i)}(\mathbf{x}) = 0 \text{ và } forall j, h^{(j)}(\mathbf{x}) \leq 0\}.$$

Trong đó:

- Các phương trình $g^{(i)}(\mathbf{x}) = 0$ được gọi là ràng buộc phương trình (equality constraints).

- Các bất đẳng thức $h^{(j)}(\mathbf{x}) \leq 0$ được gọi là ràng buộc bất đẳng thức (inequality constraints).

Chúng ta giới thiệu các biến mới λ_i và α_j cho từng ràng buộc, được gọi là các nhân tử KKT (KKT multipliers). Khi đó, Lagrangian tổng quát được định nghĩa là:

$$L(x, \lambda, \alpha) = f(x) + \sum_i \lambda_i g^{(i)}(x) + \sum_j \alpha_j h^{(j)}(x). \tag{4.18}$$

Với Lagrangian tổng quát, ta có thể chuyển bài toán tối ưu có ràng buộc về một bài toán tối ưu không ràng buộc. Miễn là tồn tại ít nhất một điểm khả thi và hàm mục tiêu $f(x)$ không nhận giá trị vô cực, ta có:

$$\min_x \max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha) \tag{4.19}$$

có cùng giá trị tối ưu và tập hợp nghiệm tối ưu x như bài toán gốc:

$$\min_{x \in S} f(x). \tag{4.20}$$

Điều này đúng vì nếu mọi ràng buộc đều được thỏa mãn, ta có:

$$\max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha) = f(x), \tag{4.21}$$

trong khi nếu có bất kỳ ràng buộc nào bị vi phạm, ta có:

$$\max_\lambda \max_{\alpha, \alpha \geq 0} L(x, \lambda, \alpha) = \infty. \tag{4.22}$$

Những tính chất này đảm bảo rằng không có điểm bất khả thi nào có thể trở thành tối ưu, đồng thời nghiệm tối ưu trong tập các điểm khả thi vẫn được bảo toàn.

Để thực hiện bài toán tối ưu có ràng buộc với mục tiêu cực đại, ta có thể xây dựng hàm Lagrange tổng quát dựa trên $-f(x)$. Khi đó, bài toán tối ưu trở thành:

$$\min_x \max_\lambda \max_{\alpha,\alpha\geq 0} -f(x) + \sum_i \lambda_i g^{(i)}(x) + \sum_j \alpha_j h^{(j)}(x). \tag{4.23}$$

Chúng ta cũng có thể chuyển bài toán này về dạng có cực đại ở vòng lặp ngoài:

$$\max_\lambda \min_x \min_{\alpha,\alpha\geq 0} f(x) + \sum_i \lambda_i g^{(i)}(x) - \sum_j \alpha_j h^{(j)}(x). \tag{4.24}$$

Dấu của hạng tử trong ràng buộc phương trình không quan trọng; chúng ta có thể định nghĩa chúng với phép cộng hoặc trừ tùy ý. Điều này là do bài toán tối ưu có thể tự do lựa chọn dấu phù hợp cho từng λ_i.

Các ràng buộc bất đẳng thức đóng vai trò đặc biệt quan trọng trong tối ưu hóa có ràng buộc. Ta nói rằng một ràng buộc $h^{(i)}(x)$ là đang hoạt động (active) nếu $h^{(i)}(x^*) = 0$. Nếu một ràng buộc không hoạt động, thì nghiệm của bài toán tìm được với ràng buộc này vẫn sẽ là một nghiệm cục bộ nếu bỏ ràng buộc đó đi. Trong một số trường hợp, một ràng buộc không hoạt động có thể loại bỏ một số nghiệm khác.

Chẳng hạn, với một bài toán lồi có một vùng lớn chứa toàn bộ các nghiệm tối ưu toàn cục (một vùng phẳng rộng có cùng giá trị chi phí), một phần của vùng này có thể bị loại bỏ do ràng buộc. Ngoài ra, với một bài toán không lồi, có thể có những nghiệm dừng tốt hơn nhưng bị loại trừ do một ràng buộc không hoạt động tại thời điểm hội tụ. Tuy nhiên, nghiệm tại điểm hội tụ vẫn là một điểm dừng bất kể ràng buộc không hoạt động có được đưa vào hay không.

Vì một ràng buộc không hoạt động $h^{(i)}(x)$ có giá trị âm, nên nghiệm của bài toán:

$$\min_x \max_\lambda \max_{\alpha,\alpha\geq 0} L(x, \lambda, \alpha) \tag{4.25}$$

sẽ có $\alpha_i = 0$. Do đó, tại nghiệm tối ưu, ta có:

$$\boldsymbol{\alpha} \odot h(x) = \mathbf{0}. \tag{4.26}$$

Nói cách khác, với mọi i, ít nhất một trong hai điều kiện $\alpha_i \geq 0$ hoặc $h^{(i)}(x) \leq 0$ phải được thỏa mãn tại nghiệm. Hiểu một cách trực quan, hoặc là nghiệm nằm trên biên của ràng buộc và ta cần sử dụng số nhân KKT để ảnh hưởng đến nghiệm x, hoặc ràng buộc không có ảnh hưởng nào đến nghiệm và ta biểu diễn điều này bằng cách đặt số nhân KKT tương ứng bằng 0.

Một tập hợp các thuộc tính mô tả điểm tối ưu của bài toán tối ưu có ràng buộc. Các thuộc tính này được gọi là điều kiện Karush-Kuhn-Tucker (KKT) *(Karush, 1939; Kuhn and Tucker, 1951)*. Đây là các điều kiện cần (và trong một số trường hợp cũng là điều kiện đủ) để một điểm là tối ưu. Các điều kiện này bao gồm:

- Gradient của hàm Lagrange tổng quát bằng 0.
- Tất cả các ràng buộc trên x và số nhân KKT đều được thỏa mãn.
- Các ràng buộc bất đẳng thức tuân theo tính chất bù trừ lỏng (complementary slackness):

$$\boldsymbol{\alpha} \odot h(x) = \mathbf{0}. \tag{4.27}$$

Để tìm hiểu thêm về phương pháp KKT, xem thêm tài liệu của *Nocedal and Wright (2006)*.

4.5 Ví dụ: Bình phương tối thiểu tuyến tính

Giả sử chúng ta muốn tìm giá trị của x để cực tiểu hóa:

$$f(x) = \frac{1}{2}\|Ax - b\|_2^2. \tag{4.28}$$

Các thuật toán đại số tuyến tính chuyên biệt có thể giải bài toán này một cách hiệu quả. Tuy nhiên, ta cũng có thể khảo sát cách giải quyết bài toán này bằng phương pháp tối ưu hóa dựa trên gradient như một ví dụ đơn giản để minh họa cách hoạt động của các kỹ thuật này.

Trước tiên, chúng ta cần tính gradient:

$$\nabla_x f(x) = A^\top(Ax - b) = A^\top Ax - A^\top b. \tag{4.29}$$

Sau đó, ta có thể đi theo hướng gradient đi xuống bằng cách thực hiện các bước nhỏ. Xem thuật toán 4.1 để biết thêm chi tiết.

Algorithm 4.1 Thuật toán cực tiểu hóa $f(x) = \frac{1}{2}\|Ax - b\|_2^2$ bằng phương pháp gradient descent.

1: Chọn kích thước bước (ϵ) và ngưỡng hội tụ (δ) là các số dương nhỏ.
2: **while** $\left\|A^\top Ax - A^\top b\right\|_2 > \delta$ **do**
3: Cập nhật: $x \leftarrow x - \epsilon(A^\top Ax - A^\top b)$
4: **end while**

Phương pháp Newton cũng là một phương pháp tối ưu hóa hiệu quả, đặc biệt phù hợp với các hàm có dạng bậc hai. Khi áp dụng cho một hàm có dạng bậc hai, phương pháp này có thể tìm được nghiệm tối ưu chỉ sau một bước. Điều này là do phép xấp xỉ bậc hai của phương pháp Newton trong trường hợp này là chính xác.

Giả sử chúng ta muốn cực tiểu hóa cùng một hàm mục tiêu nhưng có thêm ràng buộc:

$$\mathbf{x}^\top \mathbf{x} \leq 1. \tag{4.30}$$

Trong trường hợp này, ta có thể sử dụng phương pháp nhân tử Lagrange để chuyển bài toán tối ưu có ràng buộc thành bài toán tối ưu không ràng buộc. Ta định nghĩa hàm Lagrange như sau:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda \left(\mathbf{x}^\top \mathbf{x} - 1\right). \tag{4.31}$$

Sau đó, bài toán tối ưu có thể được biểu diễn dưới dạng:

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda). \tag{4.32}$$

Lời giải có chuẩn nhỏ nhất cho bài toán bình phương tối thiểu không ràng buộc có thể được tìm bằng cách sử dụng giả nghịch đảo Moore-Penrose: $\boldsymbol{x} = \boldsymbol{A}^+\boldsymbol{b}$. Nếu điểm này thỏa mãn ràng buộc, thì nó chính là nghiệm của bài toán có ràng buộc. Nếu không, ta cần tìm một nghiệm khác thoả mãn điều kiện ràng buộc.

Để tìm nghiệm khi ràng buộc có hiệu lực, ta sử dụng phương pháp nhân tử Lagrange. Bằng cách lấy đạo hàm của hàm Lagrange theo biến $\mathbf{x}$, ta thu được phương trình:

$$A^\top A\mathbf{x} - A^\top \mathbf{b} + 2\lambda \mathbf{x} = 0. \tag{4.33}$$

Giải phương trình này, ta thu được nghiệm có dạng:

$$\mathbf{x} = (A^\top A + 2\lambda I)^{-1} A^\top \mathbf{b}. \tag{4.34}$$

Khi giải bài toán tối ưu có ràng buộc, chúng ta cần chọn một giá trị phù hợp cho hệ số λ sao cho nghiệm tìm được thỏa mãn ràng buộc. Một phương pháp để tìm giá trị này là sử dụng kỹ thuật *gradient ascent* trên λ. Hãy quan sát đạo hàm của hàm Lagrange theo λ:

$$\frac{\partial}{\partial \lambda} L(\mathbf{x}, \lambda) = \mathbf{x}^T \mathbf{x} - 1. \tag{4.35}$$

Nếu chuẩn của $\mathbf{x}$ vượt quá 1, thì đạo hàm này dương. Do đó, để tối đa hóa Lagrangian theo λ, ta cần tăng λ. Khi λ tăng, hệ số phạt $\mathbf{x}^T\mathbf{x}$ cũng tăng theo, dẫn đến nghiệm $\mathbf{x}$ có độ lớn nhỏ hơn. Quá trình này tiếp tục cho đến khi $\mathbf{x}$ đạt chuẩn mong muốn và đạo hàm bằng 0.

Kết quả là một phương pháp lặp giúp điều chỉnh giá trị λ để đảm bảo ràng buộc được thỏa mãn một cách chính xác. Đây là một trong những nguyên lý quan trọng trong việc tối ưu hóa có ràng buộc, được sử dụng rộng rãi trong học máy và các thuật toán tối ưu hiện đại.

CHƯƠNG 5. HỌC MÁY CƠ BẢN

Học Sâu (Deep Learning) là một nhánh của Học Máy (Machine Learning). Để hiểu rõ về Học Sâu, trước tiên chúng ta cần nắm vững những nguyên tắc cơ bản của Học Máy. Chương này cung cấp một cái nhìn tổng quan về những nguyên lý quan trọng nhất, giúp bạn có nền tảng vững chắc trước khi đi sâu vào các nội dung phức tạp hơn của cuốn sách. Nếu bạn là người mới tìm hiểu hoặc muốn có một cái nhìn tổng quát hơn, bạn nên tham khảo thêm các sách chuyên sâu về Học Máy như *Murphy (2012)* hoặc *Bishop (2006)* để có một nền tảng vững chắc hơn. Nếu bạn đã quen thuộc với các khái niệm cơ bản, bạn có thể bỏ qua phần này và chuyển ngay đến mục 5.11, nơi cung cấp những góc nhìn về các kỹ thuật Học Máy truyền thống và cách chúng ảnh hưởng đến sự phát triển của các thuật toán Học Sâu hiện đại.

Chúng ta bắt đầu với định nghĩa về thuật toán học máy và lấy một ví dụ cụ thể: thuật toán hồi quy tuyến tính. Tiếp theo, ta sẽ tìm hiểu về thách thức trong việc khớp dữ liệu huấn luyện với mô hình, cũng như việc làm thế nào để mô hình có thể tổng quát hóa với dữ liệu mới. Hầu hết các thuật toán học máy đều có các tham số điều chỉnh, gọi là *siêu tham số* (hyperparameters), cần phải được xác định bên ngoài thuật toán học. Chúng ta sẽ thảo luận cách thiết lập chúng bằng cách sử dụng thêm dữ liệu. Học máy thực chất là một dạng thống kê ứng dụng, nhưng nhấn mạnh vào việc sử dụng máy tính để ước lượng các hàm phức tạp một cách thống kê, hơn là tìm kiếm khoảng tin cậy xung quanh các hàm đó. Do đó, ta sẽ đề cập đến hai phương pháp thống kê quan trọng: ước lượng theo tần suất (frequentist estimators) và suy diễn Bayes (Bayesian inference). Hầu hết các thuật toán học máy có thể được chia thành hai nhóm: học có giám sát (supervised learning) và học không giám sát (unsupervised learning). Chúng ta sẽ phân tích sự khác biệt giữa hai nhóm này, đồng thời đưa ra ví dụ về các thuật toán học máy đơn giản thuộc mỗi nhóm. Phần lớn các thuật toán học sâu được xây dựng dựa trên một phương pháp tối ưu hóa quan trọng, gọi là *gradient descent* (hạ gradient ngẫu nhiên). Chúng ta sẽ thảo luận cách kết hợp các thành phần quan trọng trong một thuật toán học máy, bao gồm thuật toán tối ưu hóa, hàm mất mát, mô hình và tập dữ liệu để tạo ra một mô hình hoàn chỉnh. Cuối cùng, trong mục 5.11, chúng ta sẽ tìm hiểu về các yếu tố hạn chế khả năng tổng quát hóa của các thuật toán học máy truyền thống. Những thách thức này là động lực thúc đẩy sự phát triển của các thuật toán học sâu nhằm khắc phục những nhược điểm trên.

5.1 Thuật Toán Học Máy

Một thuật toán học máy là một thuật toán có khả năng học từ dữ liệu. Nhưng chính xác thì *học* nghĩa là gì? Theo *Mitchell (1997)*, học máy có thể được định nghĩa như sau:

> "Một chương trình máy tính được gọi là có khả năng học từ kinh nghiệm E với một tập các tác vụ T và thước đo hiệu suất P, nếu hiệu suất của nó trong các tác vụ T, được đo bằng P, cải thiện theo thời gian khi có thêm kinh nghiệm E."

Chúng ta có thể tưởng tượng ra nhiều dạng kinh nghiệm E, tác vụ T, và các thước đo

hiệu suất P khác nhau. Tuy nhiên, trong cuốn sách này, chúng ta sẽ không cố gắng định nghĩa một cách cứng nhắc về từng khái niệm trên. Thay vào đó, trong các phần tiếp theo, chúng ta sẽ đưa ra các mô tả trực quan và ví dụ minh họa về các loại tác vụ khác nhau, các tiêu chí đánh giá hiệu suất, cũng như những kinh nghiệm có thể được sử dụng để xây dựng thuật toán học máy.

5.1.1 Tác Vụ Học Máy (T)

Học máy giúp chúng ta giải quyết những tác vụ quá phức tạp để có thể giải bằng các chương trình cố định do con người thiết kế. Từ góc độ khoa học và triết học, học máy rất thú vị vì phát triển hiểu biết về lĩnh vực này cũng đồng nghĩa với việc khám phá các nguyên lý nền tảng của trí tuệ nhân tạo.

Trong định nghĩa chính thức, quá trình học tập không phải là tác vụ cần thực hiện. Học chỉ là cách thức giúp chúng ta đạt được khả năng thực hiện tác vụ đó. Ví dụ, nếu chúng ta muốn một con robot có thể đi bộ, thì tác vụ ở đây chính là đi bộ. Chúng ta có thể lập trình để robot tự học cách đi, hoặc viết trực tiếp một chương trình hướng dẫn robot đi theo từng bước cụ thể.

Các tác vụ học máy thường được mô tả dựa trên cách hệ thống học máy xử lý một ví dụ (example). Một ví dụ là tập hợp các đặc trưng (features), tức là những thông tin được đo lường định lượng từ một đối tượng hoặc sự kiện mà ta muốn hệ thống học máy xử lý. Chúng ta thường biểu diễn một ví dụ dưới dạng một vector $\mathbf{x} \in \mathbb{R}^n$, trong đó mỗi phần tử x_i của vector đại diện cho một đặc trưng cụ thể. Ví dụ, đối với một hình ảnh, các đặc trưng của nó có thể là giá trị của từng pixel trong ảnh.

Một trong những tác vụ phổ biến nhất trong học máy là:

- Phân loại (Classification): là một dạng bài toán trong học máy, trong đó chương trình máy tính được yêu cầu xác định một đầu vào thuộc về danh mục nào trong k danh mục có sẵn. Để giải quyết bài toán này, thuật toán học máy thường được yêu cầu xây dựng một hàm ánh xạ:

  $$f : \mathbb{R}^n \to \{1, \ldots, k\}$$

  Trong đó, với mỗi đầu vào được biểu diễn dưới dạng một vector $\mathbf{x}$, mô hình sẽ gán nó vào một danh mục xác định bởi một mã số $y = f(\mathbf{x})$.

  Một ví dụ phổ biến của bài toán phân loại là nhận dạng đối tượng trong ảnh. Đầu vào của bài toán là một hình ảnh (có thể biểu diễn dưới dạng ma trận giá trị độ sáng của các điểm ảnh), và đầu ra là một mã số xác định đối tượng trong ảnh. Chẳng hạn, robot Willow Garage PR2 có thể hoạt động như một nhân viên phục vụ, nhận diện và phân loại các loại đồ uống khác nhau để giao cho khách hàng *(Good-fellow et al., 2010)*. Hiện nay, các phương pháp nhận diện đối tượng hiện đại được thực hiện tốt nhất bằng học sâu *(Krizhevsky et al., 2012; Ioffe and Szegedy, 2015)*. Công nghệ nhận diện đối tượng cũng được sử dụng trong các ứng dụng nhận diện khuôn mặt, giúp tự động gắn thẻ ảnh *(Taigman et al., 2014)*.

- Phân loại với đầu vào bị thiếu (Classification with missing inputs): Bài toán phân loại trở nên khó khăn hơn khi không đảm bảo rằng tất cả các thành phần trong vector đầu vào luôn có sẵn. Trong trường hợp này, thay vì học một hàm phân loại duy nhất, mô hình cần học một tập hợp các hàm phân loại, mỗi hàm tương ứng với một tập con các đầu vào bị thiếu.

  Tình huống này thường xảy ra trong lĩnh vực chẩn đoán y khoa, do nhiều loại xét nghiệm có thể đắt đỏ hoặc xâm lấn, dẫn đến việc không phải lúc nào cũng có đầy đủ dữ liệu đầu vào. Một cách hiệu quả để xử lý bài toán này là học một phân phối xác suất trên toàn bộ các biến liên quan, sau đó giải bài toán phân loại bằng cách lấy tích phân (marginalizing out) các biến bị thiếu. Với n biến đầu vào, có thể có tối đa 2^n hàm phân loại tương ứng với các trường hợp khác nhau của tập hợp biến bị thiếu. Tuy nhiên, thay vì học tất cả các hàm riêng lẻ này, mô hình chỉ cần học một hàm duy nhất mô tả phân phối xác suất *(Goodfellow et al. (2013b)*.

  Một ví dụ điển hình của mô hình phân loại xác suất có thể xử lý đầu vào bị thiếu là Mạng Bayes (Bayesian Network), cho phép mô hình ước lượng xác suất của từng nhãn dựa trên thông tin quan sát được. Điều này giúp mô hình có khả năng xử lý linh hoạt hơn khi dữ liệu thực tế không đầy đủ.

- Hồi quy (Regression): Hồi quy (Regression) là một dạng bài toán trong học máy, trong đó chương trình máy tính được yêu cầu dự đoán một giá trị số dựa trên đầu vào. Để giải quyết bài toán này, thuật toán học máy cần xây dựng một hàm ánh xạ f: $\mathbb{R}^n \rightarrow \mathbb{R}$. Bài toán hồi quy tương tự như bài toán phân loại, nhưng đầu ra của nó là một giá trị liên tục thay vì một danh mục rời rạc. Một ví dụ điển hình của bài toán hồi quy là dự đoán số tiền yêu cầu bồi thường bảo hiểm dự kiến của một cá nhân (dùng để xác định mức phí bảo hiểm), hoặc dự đoán giá cổ phiếu trong tương lai. Những loại dự đoán này cũng được sử dụng trong giao dịch thuật toán.

- Chuyển đổi văn bản (Transcription): là một dạng bài toán trong học máy, trong đó hệ thống cần quan sát một dạng dữ liệu không có cấu trúc rõ ràng và chuyển nó thành dạng văn bản rời rạc. Ví dụ, trong bài toán nhận dạng ký tự quang học (OCR - Optical Character Recognition), chương trình máy tính được cung cấp một hình ảnh chứa văn bản và phải trả về chuỗi ký tự tương ứng (ví dụ, ở định dạng ASCII hoặc Unicode).

  Một ứng dụng phổ biến của OCR là Google Street View sử dụng học sâu để nhận dạng số địa chỉ trên đường phố *(Goodfellowet al., 2014d)*. Một ví dụ khác là nhận dạng giọng nói, trong đó chương trình máy tính được cung cấp một dạng sóng âm thanh và xuất ra chuỗi ký tự hoặc mã từ mô tả nội dung của âm thanh đó. Học sâu là một thành phần quan trọng trong các hệ thống nhận dạng giọng nói hiện đại, được sử dụng tại các công ty lớn như Microsoft, IBM và Google *(Hintonet al., 2012b)*.

- Dịch máy (Machine translation): là một bài toán trong học máy, trong đó đầu vào là một chuỗi ký hiệu trong một ngôn ngữ, và chương trình máy tính cần chuyển đổi nó

thành một chuỗi ký hiệu trong một ngôn ngữ khác. Ứng dụng phổ biến nhất của bài toán này là dịch giữa các ngôn ngữ tự nhiên, chẳng hạn như từ tiếng Anh sang tiếng Pháp.

Gần đây, học sâu đã có tác động quan trọng đến lĩnh vực dịch máy. Các mô hình dựa trên mạng nơ-ron như Seq2Seq với cơ chế Attention đã cải thiện đáng kể chất lượng bản dịch so với các phương pháp truyền thống *(Sutskever et al., 2014; Bahdanau et al., 2015)*. Những mô hình này hiện đang được sử dụng trong các công cụ dịch thuật tự động phổ biến như Google Dịch.

- Đầu ra có cấu trúc (Structured Output): Các tác vụ đầu ra có cấu trúc bao gồm bất kỳ tác vụ nào mà đầu ra là một vector (hoặc một cấu trúc dữ liệu khác chứa nhiều giá trị) với các mối quan hệ quan trọng giữa các phần tử khác nhau. Đây là một danh mục rộng và bao gồm các tác vụ phiên âm và dịch thuật đã được đề cập ở trên, cũng như nhiều tác vụ khác. Một ví dụ là phân tích cú pháp—chuyển một câu ngôn ngữ tự nhiên thành một cây mô tả cấu trúc ngữ pháp của nó bằng cách gán nhãn các nút của cây là động từ, danh từ, trạng từ, v.v. Xem *Collobert (2011)* để biết một ví dụ về việc ứng dụng học sâu vào bài toán phân tích cú pháp. Một ví dụ khác là phân đoạn hình ảnh theo từng pixel, trong đó một chương trình máy tính gán mỗi pixel trong ảnh vào một danh mục cụ thể.

  Ví dụ, học sâu có thể được sử dụng để chú thích vị trí của các con đường trong ảnh chụp từ trên không (*Mnih and Hinton, 2010*). Đầu ra không nhất thiết phải phản ánh cấu trúc của đầu vào một cách chặt chẽ như trong các tác vụ chú thích này. Chẳng hạn, trong bài toán chú thích ảnh, chương trình máy tính quan sát một hình ảnh và tạo ra một câu mô tả hình ảnh đó (*Kiroset al., 2014a,b; Mao et al., 2015; Vinyals et al., 2015b; Donahue et al., 2014;Karpathy and Li, 2015; Fang et al., 2015; Xu et al., 2015*). Các tác vụ này được gọi là tác vụ đầu ra có cấu trúc vì chương trình phải tạo ra nhiều giá trị liên kết chặt chẽ với nhau. Ví dụ, các từ được tạo ra bởi một chương trình chú thích ảnh phải tạo thành một câu có nghĩa.

- Phát Hiện Dị Thường (Anomaly detection): Trong loại bài toán này, chương trình máy tính phân tích một tập hợp các sự kiện hoặc đối tượng và xác định những yếu tố nào là bất thường hoặc không điển hình. Một ví dụ về bài toán phát hiện dị thường là phát hiện gian lận thẻ tín dụng. Bằng cách mô hình hoá thói quen mua sắm của bạn, công ty thẻ tín dụng có thể phát hiện việc sử dụng thẻ không hợp lệ. Nếu kẻ gian đánh cắp thẻ tín dụng hoặc thông tin thẻ của bạn, các giao dịch của kẻ đó thường sẽ khác biệt với phân phối xác suất của các loại giao dịch thông thường của bạn. Công ty thẻ tín dụng có thể ngăn chặn gian lận bằng cách khóa tài khoản ngay khi phát hiện giao dịch bất thường. Xem *Chandola et al. (2009)* để biết thêm về các phương pháp phát hiện dị thường.

- Tổng Hợp và Lấy Mẫu (Synthesis and sampling): Trong loại bài toán này, thuật toán máy học được yêu cầu tạo ra các ví dụ mới tương tự với những ví dụ trong tập dữ liệu

huấn luyện. Quá trình tổng hợp và lấy mẫu bằng máy học có thể rất hữu ích cho các ứng dụng truyền thông khi cần tạo ra khối lượng lớn nội dung mà nếu làm thủ công sẽ rất tốn kém, nhàm chán, hoặc mất nhiều thời gian. Ví dụ, trong các trò chơi điện tử, thuật toán có thể tự động tạo ra các kết cấu (textures) cho các đối tượng lớn hoặc khung cảnh thay vì yêu cầu các nghệ sĩ gán nhãn từng điểm ảnh một cách thủ công *(Luo et al., 2013)*.

Trong một số trường hợp, quá trình lấy mẫu hoặc tổng hợp cần tạo ra một loại đầu ra cụ thể dựa trên đầu vào. Ví dụ, trong bài toán tổng hợp giọng nói, chúng ta cung cấp một câu viết và yêu cầu chương trình phát ra một dạng sóng âm thanh chứa phiên bản nói của câu đó. Đây là một dạng bài toán có cấu trúc đầu ra, nhưng với yêu cầu bổ sung là không có một kết quả duy nhất đúng cho mỗi đầu vào, và chúng ta mong muốn đầu ra có sự đa dạng lớn để làm cho kết quả trông tự nhiên và thực tế hơn.

- Bổ Sung Giá Trị Thiếu (Imputation of Missing Values): Trong loại bài toán này, thuật toán máy học nhận một ví dụ mới $x \in \mathbb{R}^n$, nhưng một số thành phần x_i của x bị thiếu. Thuật toán phải cung cấp dự đoán về các giá trị bị thiếu đó.

- Khử Nhiễu (Denoising): Trong loại bài toán này, thuật toán máy học được cung cấp một đầu vào là một ví dụ bị nhiễu $\tilde{x} \in \mathbb{R}^n$ được tạo ra từ một quá trình gây nhiễu không xác định từ một ví dụ sạch $x \in \mathbb{R}^n$. Nhiệm vụ của thuật toán là dự đoán ví dụ sạch x từ phiên bản bị nhiễu $\tilde{x}$, hoặc dự đoán phân phối xác suất có điều kiện $p(x \mid \tilde{x})$.

- Ước Lượng Mật Độ (Density estimation hoặc probability mass function estimation): Trong bài toán ước lượng mật độ, thuật toán máy học được yêu cầu học một hàm $p_{\text{model}} : \mathbb{R}^n \to \mathbb{R}$, trong đó $p_{\text{model}}(\mathbf{x})$ có thể được hiểu là một hàm mật độ xác suất (nếu $\mathbf{x}$ là liên tục) hoặc một hàm khối xác suất (nếu $\mathbf{x}$ là rời rạc) trên không gian mà các ví dụ được lấy ra.

  Để làm tốt nhiệm vụ này, thuật toán cần học được cấu trúc của dữ liệu mà nó quan sát được. Nó phải biết các ví dụ tập trung chặt chẽ ở đâu và nơi nào chúng ít có khả năng xuất hiện. Hầu hết các bài toán được mô tả ở trên đều yêu cầu thuật toán học ít nhất là nắm bắt ngầm cấu trúc của phân phối xác suất. Ước lượng mật độ cho phép ta nắm bắt rõ ràng phân phối đó, từ đó có thể thực hiện các phép tính trên phân phối để giải quyết các bài toán khác.

  Ví dụ, nếu chúng ta đã thực hiện ước lượng mật độ để thu được một phân phối xác suất $p(\mathbf{x})$, ta có thể sử dụng phân phối đó để giải quyết bài toán bổ sung giá trị thiếu. Nếu một giá trị x_i bị thiếu và tất cả các giá trị khác, được ký hiệu là $\mathbf{x}_{-i}$, đã được biết, thì ta có thể suy ra phân phối của x_i khi biết $\mathbf{x}_{-i}$ thông qua $p(x_i \mid \mathbf{x}_{-i})$.

  Trên thực tế, ước lượng mật độ không phải lúc nào cũng cho phép ta giải quyết tất cả các bài toán liên quan, vì trong nhiều trường hợp, các phép tính cần thiết trên $p(\mathbf{x})$ là không khả thi về mặt tính toán.

Tất nhiên, còn rất nhiều tác vụ và loại tác vụ khác trong học máy. Những ví dụ được liệt kê ở đây chỉ nhằm mục đích minh họa cho những gì mà học máy có thể làm được, chứ không phải để xây dựng một hệ thống phân loại cứng nhắc về các loại tác vụ.

5.1.2 Thước đo hiệu suất, *P*

Để đánh giá khả năng của một thuật toán machine learning, chúng ta cần thiết kế một thước đo định lượng về hiệu suất của nó. Thông thường, thước đo hiệu suất *P* sẽ được thiết kế phù hợp với từng nhiệm vụ cụ thể mà hệ thống đảm nhiệm.

Với các bài toán như phân loại, phân loại với dữ liệu thiếu, hoặc nhận diện giọng nói, chúng ta thường đo lường độ chính xác (*accuracy*) của mô hình. Độ chính xác được định nghĩa là tỉ lệ các mẫu được mô hình dự đoán đúng trên tổng số mẫu kiểm thử.

Bên cạnh đó, ta cũng có thể đánh giá mô hình thông qua tỉ lệ lỗi (*error rate*), tức là tỉ lệ các mẫu mà mô hình dự đoán sai trên tổng số mẫu. Tỉ lệ lỗi thường được gọi là hàm mất mát 0-1 (*0-1 loss*). Cụ thể, giá trị của hàm mất mát 0-1 trên một mẫu sẽ là 0 nếu mẫu đó được phân loại đúng và 1 nếu phân loại sai.

Tuy nhiên, đối với các bài toán như ước lượng mật độ xác suất, việc sử dụng độ chính xác hay tỉ lệ lỗi là không hợp lý. Thay vào đó, chúng ta cần một thước đo hiệu suất khác cho phép mô hình đưa ra một điểm số liên tục cho mỗi mẫu. Phương pháp phổ biến nhất là báo cáo xác suất logarit trung bình mà mô hình gán cho các mẫu.

Thông thường, chúng ta quan tâm đến việc thuật toán machine learning hoạt động tốt như thế nào trên dữ liệu mà nó chưa từng thấy trước đây, vì điều này quyết định mức độ hiệu quả của nó khi triển khai trong thực tế. Do đó, chúng ta đánh giá các thước đo hiệu suất bằng cách sử dụng một tập kiểm tra (*test set*) gồm những dữ liệu tách biệt với dữ liệu đã dùng để huấn luyện hệ thống machine learning.

Việc chọn thước đo hiệu suất có vẻ đơn giản và khách quan, nhưng thực tế, việc lựa chọn một thước đo phù hợp với hành vi mong muốn của hệ thống lại không hề dễ dàng.

Trong một số trường hợp, điều này xảy ra vì khó xác định được điều gì cần đo lường. Ví dụ, khi thực hiện một tác vụ chuyển giọng nói thành văn bản, chúng ta nên đo lường độ chính xác của hệ thống khi chuyển toàn bộ câu, hay sử dụng một thước đo chi tiết hơn cho phép tính điểm một phần khi mô hình nhận diện đúng một vài từ trong câu? Đối với bài toán hồi quy (*regression*), chúng ta có nên phạt hệ thống nặng hơn khi mắc lỗi ở mức trung bình hay khi hiếm khi mắc lỗi lớn? Những quyết định thiết kế này phụ thuộc nhiều vào ứng dụng cụ thể.

Trong các trường hợp khác, mặc dù chúng ta biết chính xác đại lượng cần đo lường, nhưng việc đo lường trực tiếp là không khả thi. Ví dụ, điều này thường gặp trong bối cảnh ước lượng mật độ xác suất. Nhiều mô hình xác suất tốt nhất chỉ biểu diễn phân phối xác suất một cách gián tiếp. Việc tính toán giá trị xác suất thực sự gán cho một điểm cụ thể trong không gian đối với nhiều mô hình như vậy là không thể thực hiện được. Trong những trường hợp này, ta cần thiết kế một tiêu chí thay thế vẫn phản ánh được mục tiêu

thiết kế, hoặc xây dựng một xấp xỉ tốt cho tiêu chí mong muốn.

5.1.3 Trải nghiệm, E

Các thuật toán máy học có thể được phân loại rộng rãi thành không giám sát hoặc giám sát dựa trên loại trải nghiệm mà chúng được phép có trong quá trình học.

Hầu hết các thuật toán học trong cuốn sách này có thể được hiểu là được phép trải nghiệm toàn bộ tập dữ liệu. Một tập dữ liệu là một tập hợp của nhiều ví dụ, như được định nghĩa trong mục 5.1.1. Đôi khi chúng ta gọi các ví dụ là điểm dữ liệu.

Một trong những tập dữ liệu lâu đời nhất được các nhà thống kê và các nhà nghiên cứu máy học nghiên cứu là tập dữ liệu Iris (*Fisher, 1936*). Đây là một tập hợp các phép đo của các bộ phận khác nhau của 150 cây iris. Mỗi cây riêng lẻ tương ứng với một ví dụ. Các đặc trưng trong mỗi ví dụ là các phép đo của từng bộ phận của cây: chiều dài đài hoa, chiều rộng đài hoa, chiều dài cánh hoa và chiều rộng cánh hoa. Tập dữ liệu cũng ghi lại loài mà mỗi cây thuộc về. Ba loài khác nhau được biểu diễn trong tập dữ liệu.

Học không giám sát (Unsupervised Learning) là một phương pháp học máy trong đó mô hình tiếp xúc với một tập dữ liệu chứa nhiều đặc trưng, nhưng không có nhãn hoặc mục tiêu cụ thể. Mô hình sẽ tự động khám phá các quy luật tiềm ẩn trong dữ liệu. Trong lĩnh vực học sâu (Deep Learning), học không giám sát thường nhằm mục tiêu học được phân phối xác suất của dữ liệu, có thể là một cách tường minh (ví dụ: ước lượng mật độ) hoặc ngầm định (ví dụ: tạo dữ liệu mới hoặc khử nhiễu). Một số thuật toán học không giám sát phổ biến bao gồm phân cụm (Clustering) để chia tập dữ liệu thành các nhóm dựa trên sự tương đồng, hoặc giảm chiều dữ liệu (Dimensionality Reduction) để tìm ra những đặc trưng quan trọng nhất.

Học có giám sát (Supervised Learning) là một phương pháp học máy trong đó mỗi mẫu dữ liệu không chỉ chứa các đặc trưng mà còn đi kèm với một nhãn (label) hoặc mục tiêu (target). Ví dụ, tập dữ liệu *Iris* chứa thông tin về các loài hoa Iris với các đặc trưng như chiều dài và chiều rộng cánh hoa, trong đó nhãn là loài của từng bông hoa. Một thuật toán học có giám sát có thể học từ dữ liệu này để phân loại hoa Iris thành ba loài khác nhau dựa trên các đặc trưng đó. Sự khác biệt chính giữa hai phương pháp này nằm ở việc học có giám sát cần có nhãn để hướng dẫn, còn học không giám sát thì không có nhãn mà phải tự tìm hiểu cấu trúc dữ liệu.

Học không giám sát (unsupervised learning) về cơ bản là quá trình quan sát nhiều ví dụ của một vectơ ngẫu nhiên $\mathbf{x}$ và cố gắng học một cách tường minh hoặc ngầm định về phân phối xác suất $p(\mathbf{x})$ hoặc những thuộc tính quan trọng của phân phối đó. Ngược lại, học có giám sát (supervised learning) liên quan đến việc quan sát nhiều ví dụ của một vectơ ngẫu nhiên $\mathbf{x}$ cùng với một giá trị hoặc một vectơ kết hợp $\mathbf{y}$. Mục tiêu là học cách dự đoán $\mathbf{y}$ từ $\mathbf{x}$, thường bằng cách ước lượng phân phối xác suất có điều kiện $p(\mathbf{y}|\mathbf{x})$. Thuật ngữ học có giám sát bắt nguồn từ quan điểm rằng giá trị đích $\mathbf{y}$ được cung cấp bởi một người hướng dẫn hoặc giáo viên, người cho hệ thống học máy biết cần phải làm gì. Trong học không giám sát, không có người hướng dẫn hay giáo viên, và thuật toán phải tự tìm ra cấu trúc

hoặc ý nghĩa của dữ liệu mà không có sự hướng dẫn này.

Học có giám sát và học không giám sát không phải là hai khái niệm tách biệt hoàn toàn mà có thể liên kết với nhau. Nhiều kỹ thuật học máy có thể được sử dụng để thực hiện cả hai nhiệm vụ này.

Ví dụ, theo quy tắc chuỗi của xác suất, phân phối kết hợp của một vectơ $\mathbf{x} \in \mathbb{R}^n$ có thể được phân rã thành:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i | x_1, \ldots, x_{i-1}). \tag{5.1}$$

Điều này có nghĩa là chúng ta có thể giải quyết vấn đề học không giám sát bằng cách chia nhỏ nó thành n bài toán học có giám sát.

Ngược lại, chúng ta cũng có thể giải quyết bài toán học có giám sát $p(\mathbf{y}|\mathbf{x})$ bằng cách sử dụng các kỹ thuật học không giám sát để học phân phối kết hợp $p(\mathbf{x}, \mathbf{y})$, sau đó suy luận bằng công thức:

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}. \tag{5.2}$$

Ngoài học có giám sát và học không giám sát, còn có nhiều biến thể khác của mô hình học tập. Ví dụ, trong *học bán giám sát* (semi-supervised learning), một số mẫu dữ liệu có nhãn giám sát, trong khi các mẫu khác thì không. Một dạng khác là *học đa thể* (multi-instance learning), trong đó một tập hợp các ví dụ được gán nhãn theo cụm, nhưng từng phần tử riêng lẻ trong tập hợp đó thì không được gán nhãn cụ thể.

Một số thuật toán học máy không chỉ đơn thuần làm việc với một tập dữ liệu cố định. Ví dụ, các thuật toán học củng cố (reinforcement learning) tương tác với môi trường và có một cơ chế phản hồi giữa hệ thống học và kinh nghiệm của nó. Các thuật toán này nằm ngoài phạm vi của cuốn sách này, nhưng bạn có thể tham khảo thêm ở Sutton and Barto (1998) hoặc Bertsekas and Tsitsiklis (1996) về lý thuyết học củng cố, và Mnih et al. (2013) về cách tiếp cận học sâu trong học củng cố. Hầu hết các thuật toán học máy làm việc với tập dữ liệu cố định. Một tập dữ liệu có thể được mô tả theo nhiều cách khác nhau. Trong mọi trường hợp, một tập dữ liệu bao gồm nhiều ví dụ, và mỗi ví dụ lại là một tập hợp các đặc trưng (features).

Một cách phổ biến để mô tả tập dữ liệu là sử dụng ma trận thiết kế (design matrix). Đây là một ma trận trong đó mỗi hàng tương ứng với một ví dụ, và mỗi cột tương ứng với một đặc trưng. Ví dụ, tập dữ liệu Iris chứa 150 mẫu, mỗi mẫu có 4 đặc trưng. Điều này có nghĩa là ta có thể biểu diễn tập dữ liệu dưới dạng một ma trận thiết kế $\mathbf{X} \in \mathbb{R}^{150 \times 4}$, trong đó:

- $X_{i,1}$ là chiều dài đài hoa của cây thứ i.
- $X_{i,2}$ là chiều rộng đài hoa của cây thứ i.
- $X_{i,3}$ là chiều dài cánh hoa của cây thứ i.

- $X_{i,4}$ là chiều rộng cánh hoa của cây thứ i.

Trong cuốn sách này, hầu hết các thuật toán học máy được mô tả theo cách chúng hoạt động trên các tập dữ liệu có dạng ma trận thiết kế.

Để mô tả một tập dữ liệu dưới dạng ma trận thiết kế (design matrix), ta cần biểu diễn mỗi mẫu dữ liệu dưới dạng một vector và tất cả các vector này phải có cùng kích thước. Tuy nhiên, điều này không phải lúc nào cũng khả thi. Ví dụ, nếu ta có một tập hợp ảnh với các kích thước khác nhau (chiều rộng và chiều cao khác nhau), thì số lượng pixel trong từng ảnh cũng sẽ khác nhau. Do đó, không thể biểu diễn tất cả các ảnh bằng cùng một kích thước vector.

Trong những trường hợp như vậy, thay vì mô tả tập dữ liệu dưới dạng một ma trận có m hàng, ta mô tả nó như một tập hợp chứa m phần tử:

$$\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\} \tag{5.3}$$

Ký hiệu này không hàm ý rằng hai vector bất kỳ $\mathbf{x}^{(i)}$ và $\mathbf{x}^{(j)}$ có cùng kích thước.

Trong Mục 9.7 và Chương 10, chúng ta sẽ thảo luận cách xử lý các loại dữ liệu không đồng nhất này.

Trong học có giám sát (supervised learning), mỗi mẫu dữ liệu không chỉ chứa các đặc trưng (features) mà còn đi kèm với một nhãn (label) hoặc mục tiêu. Ví dụ, nếu chúng ta muốn sử dụng thuật toán học máy để nhận diện đối tượng trong ảnh, ta cần xác định đối tượng xuất hiện trong từng bức ảnh. Điều này có thể được thực hiện bằng một mã số: 0 có thể biểu diễn con người, 1 biểu diễn ô tô, 2 biểu diễn con mèo, v.v. Ngay cả khi làm việc với một tập dữ liệu chứa ma trận thiết kế của các quan sát đặc trưng $\mathbf{X}$, ta cũng cần có một vector nhãn $\mathbf{y}$, trong đó y_i là nhãn của mẫu thứ i.

Đôi khi, nhãn không chỉ là một số đơn lẻ. Ví dụ, nếu chúng ta muốn huấn luyện hệ thống nhận diện giọng nói để chuyển giọng nói thành văn bản, thì nhãn của mỗi câu sẽ là một chuỗi từ thay vì một số duy nhất.

Giống như không có định nghĩa chính thức nào về học có giám sát và không giám sát, cũng không có một phân loại cứng nhắc nào về tập dữ liệu hay trải nghiệm học. Các cấu trúc được mô tả trong chương này bao quát hầu hết các trường hợp, nhưng vẫn có thể thiết kế những cấu trúc mới cho các ứng dụng khác nhau.

5.1.4 Ví dụ: Hồi quy tuyến tính

Định nghĩa của một thuật toán học máy là một thuật toán có khả năng cải thiện hiệu suất của một chương trình máy tính trong một tác vụ nào đó thông qua kinh nghiệm. Để làm rõ hơn khái niệm này, chúng ta xét một thuật toán học máy đơn giản: hồi quy tuyến tính (linear regression). Chúng ta sẽ quay lại ví dụ này nhiều lần khi giới thiệu thêm các khái niệm về học máy giúp hiểu rõ hơn về cách hoạt động của thuật toán.

Như tên gọi của nó, hồi quy tuyến tính giải quyết một bài toán hồi quy. Mục tiêu là xây dựng một hệ thống có thể nhận đầu vào là một vector $\mathbf{x} \in \mathbb{R}^n$ và dự đoán giá trị của một số vô hướng $y \in \mathbb{R}$ làm đầu ra. Đầu ra của hồi quy tuyến tính là một hàm tuyến tính của đầu vào. Gọi $\hat{y}$ là giá trị dự đoán của mô hình đối với y, chúng ta định nghĩa đầu ra như sau:

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \tag{5.4}$$

trong đó $\mathbf{w} \in \mathbb{R}^n$ là một vector tham số (parameters).

Trong hồi quy tuyến tính, tham số của mô hình được biểu diễn bằng vector $\mathbf{w}$. Các tham số này kiểm soát hành vi của hệ thống. Cụ thể, mỗi w_i là hệ số nhân với đặc trưng x_i trước khi tổng hợp đóng góp từ tất cả các đặc trưng. Ta có thể xem $\mathbf{w}$ như một tập hợp trọng số (weights), quyết định mức độ ảnh hưởng của từng đặc trưng lên kết quả dự đoán.

Mỗi đặc trưng có tác động nhất định đến kết quả dự đoán:

- Nếu một đặc trưng x_i có trọng số w_i dương, việc tăng giá trị của đặc trưng đó sẽ làm tăng giá trị dự đoán $\hat{y}$.

- Nếu một đặc trưng x_i có trọng số w_i âm, việc tăng giá trị của đặc trưng đó sẽ làm giảm giá trị dự đoán $\hat{y}$.

- Nếu một đặc trưng có trọng số có độ lớn lớn, nó có ảnh hưởng mạnh đến dự đoán.

- Nếu trọng số của một đặc trưng bằng 0, đặc trưng đó không ảnh hưởng đến dự đoán.

Chúng ta có thể phát biểu bài toán của mình như sau:

$$\hat{y} = \mathbf{w}^\top \mathbf{x}. \tag{5.5}$$

Tiếp theo, ta cần định nghĩa một thước đo hiệu suất của mô hình, được ký hiệu là P.

Giả sử ta có một ma trận thiết kế chứa m mẫu dữ liệu đầu vào, nhưng chỉ được sử dụng để đánh giá mô hình chứ không dùng để huấn luyện. Ngoài ra, ta có một vector chứa các giá trị đúng của y tương ứng với từng mẫu dữ liệu này. Do tập dữ liệu này chỉ dùng để kiểm tra mô hình, ta gọi nó là tập kiểm tra (test set). Chúng ta ký hiệu:

- Ma trận thiết kế của tập kiểm tra: $\mathbf{X}^{(\text{test})}$

- Vector chứa nhãn thực tế của tập kiểm tra: $\mathbf{y}^{(\text{test})}$.

Một cách để đánh giá hiệu suất của mô hình là tính lỗi trung bình bình phương (MSE) trên tập kiểm tra. Nếu $\hat{y}^{(\text{test})}$ là dự đoán của mô hình trên tập kiểm tra, thì MSE được tính theo công thức:

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i (\hat{y}_i^{(\text{test})} - y_i^{(\text{test})})_i^2. \tag{5.6}$$

Công thức này có thể được viết lại như sau:

$$\text{MSE}_{\text{test}} = \frac{1}{m}\|\hat{y}^{(\text{test})} - \mathbf{y}^{(\text{test})}\|_2^2. \tag{5.7}$$

Trực quan, ta thấy rằng lỗi này giảm về 0 khi $\hat{y}^{(\text{test})} = \mathbf{y}^{(\text{test})}$. Ngược lại, lỗi sẽ tăng khi khoảng cách Euclid giữa dự đoán và giá trị thực tăng lên.

Để xây dựng một thuật toán học máy, ta cần thiết kế một thuật toán cập nhật trọng số $\mathbf{w}$ sao cho giá trị MSE_{test} giảm đi. Quá trình này diễn ra khi thuật toán quan sát tập huấn luyện $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$. Một cách tiếp cận phổ biến là giảm MSE trên tập huấn luyện, tức là tối thiểu hóa $\text{MSE}_{\text{train}}$.

Mục tiêu của hồi quy tuyến tính là tìm vector trọng số $\mathbf{w}$ sao cho hàm mất mát MSE (Mean Squared Error) trên tập huấn luyện đạt giá trị nhỏ nhất. Hàm mất mát MSE được định nghĩa như sau:

$$\text{MSE}_{\text{train}} = \frac{1}{m}\|\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\|_2^2 \tag{5.8}$$

Trong đó:

- m là số lượng mẫu huấn luyện,
- $\hat{\mathbf{y}}^{(\text{train})} = \mathbf{X}^{(\text{train})}\mathbf{w}$ là dự đoán của mô hình,
- $\mathbf{y}^{(\text{train})}$ là nhãn thực tế.

Để tìm $\mathbf{w}$ tối ưu, ta giải phương trình đạo hàm bằng 0:

$$\nabla_{\mathbf{w}}\text{MSE}_{\text{train}} = 0 \tag{5.9}$$

$$\Rightarrow \nabla_{\mathbf{w}}\frac{1}{m}\|\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2 = 0 \tag{5.10}$$

Ta triển khai đạo hàm như sau:

$$\frac{1}{m}\nabla_{\mathbf{w}}\|\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2 = 0 \tag{5.11}$$

$$\Rightarrow \nabla_{\mathbf{w}}\left(\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\right)^T\left(\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\right) = 0 \tag{5.12}$$

Khai triển biểu thức:

$$\nabla_{\mathbf{w}}\left(\mathbf{w}^T\mathbf{X}^{(\text{train})T}\mathbf{X}^{(\text{train})}\mathbf{w} - 2\mathbf{w}^T\mathbf{X}^{(\text{train})T}\mathbf{y}^{(\text{train})} + \mathbf{y}^{(\text{train})T}\mathbf{y}^{(\text{train})}\right) = 0 \tag{5.13}$$

Lấy đạo hàm:

$$2\mathbf{X}^{(\text{train})T}\mathbf{X}^{(\text{train})}\mathbf{w} - 2\mathbf{X}^{(\text{train})T}\mathbf{y}^{(\text{train})} = 0 \tag{5.14}$$

Giải ra nghiệm tối ưu:

$$\mathbf{w} = \left(\mathbf{X}^{(\text{train})T}\mathbf{X}^{(\text{train})}\right)^{-1}\mathbf{X}^{(\text{train})T}\mathbf{y}^{(\text{train})} \tag{5.15}$$

Phương trình trên được gọi là phương trình chuẩn (normal equations). Việc tính toán nghiệm theo phương trình này là một thuật toán học máy đơn giản.

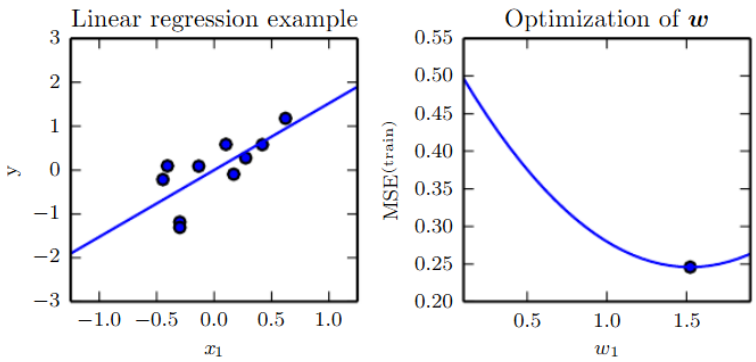
Ngoài ra, thuật ngữ hồi quy tuyến tính (linear regression) thường được dùng để chỉ một mô hình mở rộng hơn, trong đó có thêm một tham số chặn b. Khi đó, mô hình được biểu diễn dưới dạng:

$$\hat{y} = \mathbf{w}^T\mathbf{x} + b, \tag{5.16}$$

với b là hệ số chặn (intercept term).

Do đó, ánh xạ từ tham số $\mathbf{w}$ tới dự đoán vẫn là một hàm tuyến tính, nhưng ánh xạ từ đặc trưng $\mathbf{x}$ tới dự đoán thực tế lại là một hàm affine (hàm tuyến tính có thêm thành phần dịch chuyển). Điều này có nghĩa là đồ thị của mô hình vẫn là một đường thẳng, nhưng không nhất thiết phải đi qua gốc tọa độ.

Thay vì thêm tham số b, ta có thể tiếp tục sử dụng mô hình chỉ với trọng số bằng cách mở rộng vector đặc trưng $\mathbf{x}$ với một phần tử bổ sung luôn có giá trị bằng 1. Trọng số tương ứng với phần tử 1 này đóng vai trò là tham số b. Trong suốt cuốn sách này, chúng tôi thường sử dụng thuật ngữ tuyến tính (linear) khi đề cập đến các hàm affine.



Hình 5.1: Minh họa một bài toán hồi quy tuyến tính với tập dữ liệu huấn luyện gồm 10 điểm dữ liệu, mỗi điểm chỉ có một đặc trưng (feature). Vì chỉ có một đặc trưng, nên vector trọng số $\mathbf{w}$ chỉ chứa một tham số duy nhất cần học, ký hiệu là w_1. (Hình trái) Ta có thể quan sát rằng thuật toán hồi quy tuyến tính sẽ tìm giá trị w_1 sao cho đường thẳng $y = w_1 x$ càng gần với các điểm dữ liệu huấn luyện càng tốt. (Hình Phải) Điểm được đánh dấu thể hiện giá trị w_1 được tìm thấy bằng phương trình chuẩn (normal equations), đảm bảo rằng lỗi bình phương trung bình (MSE) trên tập huấn luyện là nhỏ nhất.

Thuật ngữ chặn b thường được gọi là tham số độ chệch của phép biến đổi afin. Thuật ngữ này xuất phát từ quan điểm rằng đầu ra của phép biến đổi bị lệch về phía b khi không có đầu vào. Thuật ngữ này khác với khái niệm độ chệch trong thống kê, trong đó ước lượng kỳ vọng của một thuật toán ước lượng thống kê đối với một đại lượng không bằng với giá trị thực của đại lượng đó.

Hồi quy tuyến tính tất nhiên là một thuật toán học máy đơn giản và có giới hạn, nhưng nó cung cấp một ví dụ về cách một thuật toán học có thể hoạt động. Trong các phần tiếp theo, chúng tôi mô tả một số nguyên tắc cơ bản trong thiết kế thuật toán học và chứng minh cách những nguyên tắc này có thể được sử dụng để xây dựng các thuật toán học phức tạp hơn.

5.2 Capacity, Overfitting and Underfitting

Thách thức trung tâm trong học máy (machine learning) là thuật toán của chúng ta phải hoạt động tốt trên các đầu vào *mới, chưa từng thấy trước đây* — không chỉ trên những dữ liệu mà mô hình đã được huấn luyện. Khả năng hoạt động tốt trên các đầu vào chưa được quan sát trước đó được gọi là khả năng tổng quát hóa (generalization).

Thông thường, khi huấn luyện một mô hình học máy, chúng ta có một tập dữ liệu huấn luyện (training set); từ đó, có thể tính toán một độ lỗi nào đó trên tập huấn luyện, được gọi là lỗi huấn luyện (training error), và chúng ta tìm cách giảm lỗi này. Cho đến nay, những gì chúng ta mô tả chỉ đơn giản là một bài toán tối ưu hóa. Điều làm cho học máy khác với tối ưu hóa là chúng ta muốn lỗi tổng quát hóa (generalization error), còn được gọi là lỗi kiểm tra (test error), cũng phải thấp. Lỗi tổng quát hóa được định nghĩa là giá trị kỳ vọng của lỗi trên một đầu vào mới. Ở đây, kỳ vọng được lấy trên các đầu vào khác nhau, được lấy mẫu từ phân phối mà hệ thống dự kiến sẽ gặp phải trong thực tế.

Chúng ta thường ước lượng lỗi tổng quát hóa của một mô hình học máy bằng cách đo lường hiệu suất của nó trên một tập kiểm tra (test set), bao gồm các mẫu được thu thập riêng biệt với tập huấn luyện.

Trong bài toán hồi quy tuyến tính, chúng ta huấn luyện mô hình bằng cách tối ưu hóa lỗi huấn luyện, được định nghĩa như sau:

$$\frac{1}{m_{\text{train}}} \|\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2. \tag{5.17}$$

Tuy nhiên, mục tiêu thực sự của chúng ta là giảm thiểu lỗi kiểm tra (test error), được biểu diễn như:

$$\frac{1}{m_{\text{test}}} \|\mathbf{X}^{(\text{test})}\mathbf{w} - \mathbf{y}^{(\text{test})}\|_2^2. \tag{5.18}$$

Vậy làm thế nào để đảm bảo mô hình hoạt động tốt trên tập kiểm tra khi chúng ta chỉ quan sát tập huấn luyện? Đây là câu hỏi trọng tâm của Lý thuyết học thống kê (statistical learning theory). Nếu tập huấn luyện và tập kiểm tra được thu thập một cách ngẫu nhiên không có mối liên hệ nào, chúng ta không thể làm gì nhiều. Nhưng nếu có thể đưa ra một số giả định về cách chúng được lấy mẫu, chúng ta có thể phân tích và cải thiện hiệu suất mô hình.

Một trong những giả định quan trọng trong Machine Learning là dữ liệu huấn luyện

và kiểm tra đều được lấy từ cùng một quá trình sinh dữ liệu (data-generating process). Cụ thể, chúng ta thường giả định rằng các mẫu trong tập dữ liệu độc lập với nhau và tuân theo cùng một phân phối xác suất. Giả định này được gọi là i.i.d. (independent and identically distributed - độc lập và phân phối giống nhau).

Điều này có nghĩa là tập huấn luyện và tập kiểm tra đều được lấy từ cùng một phân phối xác suất nền tảng, gọi là p_{data}. Giả định này giúp chúng ta sử dụng mô hình toán học để nghiên cứu mối quan hệ giữa lỗi huấn luyện và lỗi kiểm tra.

Một trong những mối liên hệ quan trọng giữa lỗi huấn luyện và lỗi kiểm tra là giá trị kỳ vọng của lỗi huấn luyện của một mô hình được chọn ngẫu nhiên sẽ bằng với kỳ vọng của lỗi kiểm tra của mô hình đó. Giả sử chúng ta có một phân phối xác suất $p(\mathbf{x}, \mathbf{y})$ và lấy mẫu từ nó để tạo ra tập huấn luyện và tập kiểm tra. Với một giá trị cố định của $\mathbf{w}$, lỗi kỳ vọng trên tập huấn luyện sẽ bằng với lỗi kỳ vọng trên tập kiểm tra, vì cả hai đều được sinh ra từ cùng một quá trình lấy mẫu dữ liệu. Sự khác biệt duy nhất giữa hai điều kiện này là tên mà chúng ta gán cho tập dữ liệu mà chúng ta lấy mẫu.

Tuy nhiên, trong thực tế, khi sử dụng một thuật toán machine learning, chúng ta không cố định tham số trước, mà sẽ lấy mẫu tập huấn luyện, sau đó sử dụng nó để điều chỉnh tham số nhằm giảm lỗi trên tập huấn luyện, rồi mới đánh giá trên tập kiểm tra. Quá trình này khiến kỳ vọng lỗi trên tập kiểm tra lớn hơn hoặc bằng kỳ vọng lỗi trên tập huấn luyện.

Mức độ hiệu quả của thuật toán machine learning phụ thuộc vào hai yếu tố chính:

1. Làm cho lỗi huấn luyện nhỏ.
2. Làm cho khoảng cách giữa lỗi huấn luyện và lỗi kiểm tra nhỏ.

Hai yếu tố này liên quan trực tiếp đến hai thách thức quan trọng trong machine learning: underfitting và overfitting.

Underfitting xảy ra khi mô hình không thể đạt được lỗi huấn luyện đủ thấp, nghĩa là mô hình chưa học được các đặc trưng quan trọng của dữ liệu. Điều này có thể do mô hình quá đơn giản hoặc chưa được huấn luyện đủ lâu.

Overfitting xảy ra khi mô hình học quá mức vào tập huấn luyện, dẫn đến khoảng cách lớn giữa lỗi huấn luyện và lỗi kiểm tra. Khi đó, mô hình hoạt động rất tốt trên tập huấn luyện nhưng lại kém hiệu quả trên dữ liệu thực tế.

Chúng ta có thể điều chỉnh khả năng overfitting hay underfitting của một mô hình thông qua độ phức tạp (capacity). Nói một cách đơn giản, độ phức tạp của mô hình là khả năng của nó trong việc học một tập hợp rộng các hàm. Mô hình có độ phức tạp thấp có thể gặp khó khăn trong việc học tập dữ liệu huấn luyện, trong khi mô hình có độ phức tạp cao có nguy cơ overfitting bằng cách ghi nhớ dữ liệu huấn luyện mà không tổng quát hóa tốt trên dữ liệu mới.

Một cách để kiểm soát độ phức tạp của một thuật toán học máy là chọn không gian giả thuyết (hypothesis space), tức là tập hợp các hàm mà thuật toán có thể chọn làm lời giải.

Ví dụ, thuật toán hồi quy tuyến tính có không gian giả thuyết là tập hợp tất cả các hàm tuyến tính của đầu vào. Nếu chúng ta mở rộng hồi quy tuyến tính để bao gồm các đa thức thay vì chỉ các hàm tuyến tính, không gian giả thuyết sẽ lớn hơn, làm tăng độ phức tạp của mô hình.

Một mô hình hồi quy bậc 1 chính là mô hình hồi quy tuyến tính mà chúng ta đã quen thuộc, với công thức dự đoán như sau:

$$\hat{y} = b + wx. \tag{5.19}$$

Bằng cách thêm x^2 làm một đặc trưng bổ sung vào mô hình hồi quy tuyến tính, chúng ta có thể học được một mô hình có dạng bậc hai theo biến x:

$$\hat{y} = b + w_1 x + w_2 x^2. \tag{5.20}$$

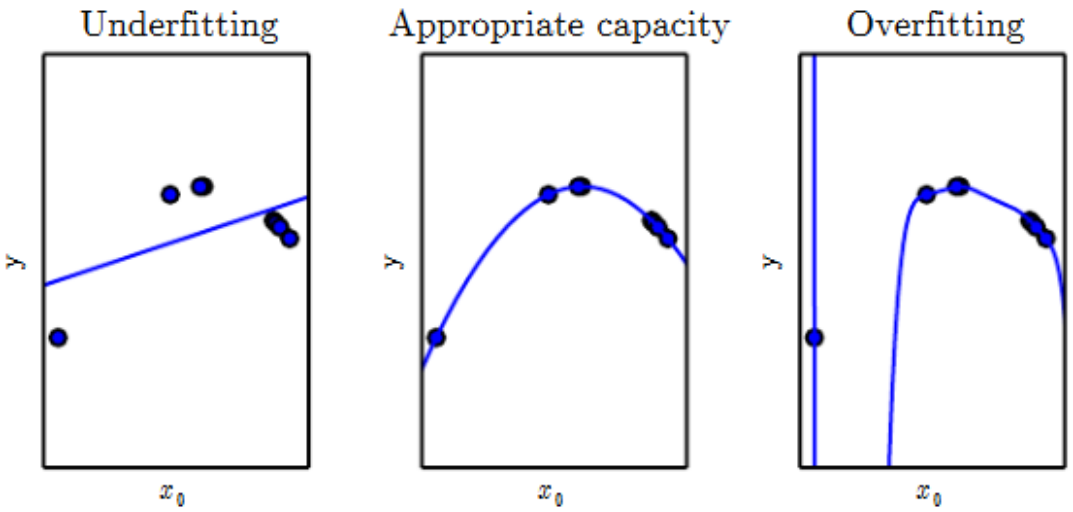
Mặc dù mô hình trên có dạng một hàm bậc hai theo đầu vào (x), nhưng đầu ra vẫn là một hàm tuyến tính theo các tham số (b, w_1, w_2). Do đó, chúng ta vẫn có thể sử dụng các phương pháp giải đóng (closed-form solution) như bình phương tối thiểu để huấn luyện mô hình.

Chúng ta có thể tiếp tục mở rộng bằng cách thêm nhiều lũy thừa của x làm đặc trưng mới. Ví dụ, để có một mô hình bậc 9:

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i. \tag{5.21}$$

Các thuật toán học máy sẽ hoạt động tốt nhất khi năng lực của mô hình phù hợp với độ phức tạp thực sự của bài toán. Nếu mô hình có năng lực quá thấp, nó sẽ không thể nắm bắt được quy luật phức tạp trong dữ liệu và dẫn đến hiện tượng bỏ sót (underfitting). Ngược lại, nếu mô hình có năng lực quá cao so với yêu cầu của bài toán, nó có thể học quá mức vào dữ liệu huấn luyện và dẫn đến quá khớp (overfitting).

Hình 5.2 minh họa nguyên tắc này trong thực tế. Chúng tôi so sánh ba mô hình dự đoán: tuyến tính, bậc hai và bậc 9 trên một bài toán mà hàm số thực sự có dạng bậc hai. Hàm tuyến tính không thể nắm bắt được độ cong của bài toán thực sự, do đó nó mắc lỗi dưới khớp. Mô hình bậc 9 có khả năng biểu diễn chính xác hàm số thực sự, nhưng đồng thời cũng có thể biểu diễn vô số hàm số khác đi qua chính xác các điểm huấn luyện, bởi vì số lượng tham số lớn hơn số lượng mẫu huấn luyện. Khi có quá nhiều nghiệm khác nhau phù hợp với dữ liệu huấn luyện, chúng ta khó có thể chọn được một nghiệm tổng quát hóa tốt. Trong ví dụ này, mô hình bậc hai khớp chính xác với cấu trúc thực sự của bài toán, vì vậy nó có khả năng tổng quát hóa tốt trên dữ liệu mới.



Hình 5.2: Chúng tôi thử nghiệm ba mô hình trên tập dữ liệu huấn luyện này. Dữ liệu huấn luyện được tạo ra một cách tổng hợp bằng cách lấy mẫu ngẫu nhiên các giá trị x và xác định y theo một hàm bậc hai.*(Trái)*: Một mô hình tuyến tính khớp với dữ liệu bị lỗi dưới khớp (*underfitting*)—nó không thể nắm bắt được độ cong thực sự của dữ liệu. *(Giữa)*: Một mô hình bậc hai khớp với dữ liệu có khả năng tổng quát hóa tốt trên các điểm chưa thấy trước đó. Nó không gặp phải vấn đề quá khớp (*overfitting*) hay dưới khớp.*(Phải)*: Một đa thức bậc 9 khớp với dữ liệu bị lỗi quá khớp. Trong trường hợp này, chúng tôi sử dụng nghịch đảo giả Moore-Penrose để giải hệ phương trình chuẩn không xác định. Nghiệm thu được đi qua tất cả các điểm huấn luyện, nhưng lại không trích xuất đúng cấu trúc thực sự. Mô hình này tạo ra một vùng lõm sâu giữa hai điểm huấn luyện, điều này không xuất hiện trong hàm thực sự. Hơn nữa, nó tăng mạnh ở phía bên trái dữ liệu, trong khi hàm thực sự lại có xu hướng giảm tại đó.

Trong học máy, một cách để thay đổi độ phức tạp của mô hình là tăng hoặc giảm số lượng đầu vào và các tham số liên quan. Tuy nhiên, mô hình không chỉ phụ thuộc vào cấu trúc của nó mà còn vào tập hợp các hàm mà thuật toán học máy có thể lựa chọn để tối ưu hóa hàm mất mát. Điều này được gọi là khả năng biểu diễn của mô hình.

Những ý tưởng hiện đại về việc cải thiện khả năng tổng quát hóa của mô hình học máy thực chất là sự phát triển từ những tư tưởng triết học từ thời cổ đại, ít nhất là từ thời Ptolemy. Nhiều học giả thời kỳ đầu đã áp dụng nguyên lý giản ước (parsimony), hiện nay được biết đến rộng rãi với tên gọi Occam's razor (khoảng năm 1287–1347). Nguyên tắc này phát biểu rằng: trong số các giả thuyết có thể giải thích hiện tượng quan sát được một cách tương đương, ta nên chọn giả thuyết đơn giản nhất. Nguyên tắc này đã được hệ thống hóa và làm rõ hơn trong thế kỷ 20 bởi những người sáng lập lý thuyết học thống kê, tiêu biểu là Vapnik và Chervonenkis.

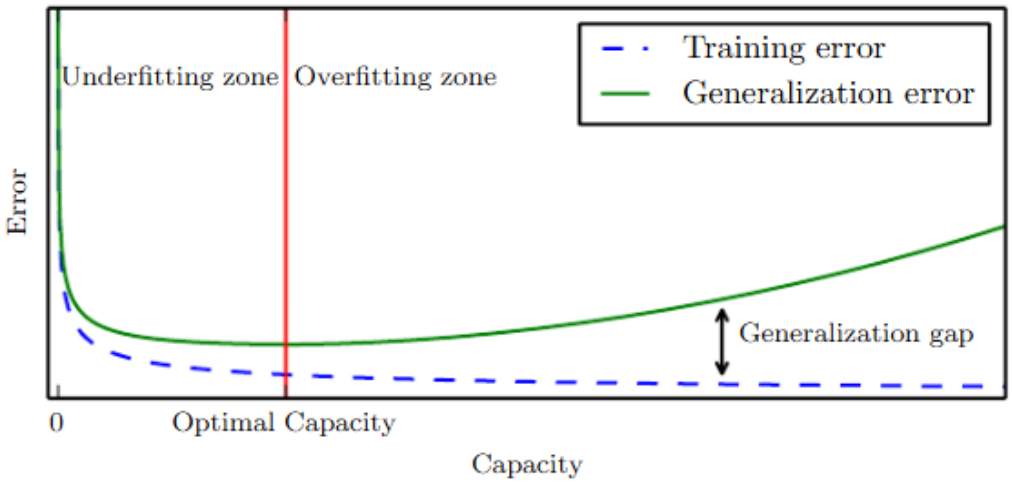
Lý thuyết học thống kê cung cấp nhiều cách để đo lường độ phức tạp của mô hình, trong đó nổi bật nhất là độ đo Vapnik-Chervonenkis (VC dimension). VC dimension là một thước đo khả năng của một bộ phân loại nhị phân. Nó được định nghĩa là số lượng điểm tối đa m mà mô hình có thể gán nhãn theo mọi cách có thể.

Việc định lượng độ phức tạp của mô hình giúp lý thuyết học thống kê đưa ra các dự đoán định lượng. Một trong những kết quả quan trọng của lý thuyết này cho thấy sự chênh lệch giữa lỗi huấn luyện và lỗi tổng quát bị chặn trên bởi một đại lượng tăng khi độ phức

tạp của mô hình tăng nhưng giảm khi số lượng mẫu huấn luyện tăng. Tuy nhiên, trong học sâu, các giới hạn này ít được sử dụng trong thực tế vì chúng thường khá lỏng lẻo và khó áp dụng.

Vấn đề xác định độ phức tạp của một mô hình học sâu còn phức tạp hơn vì khả năng hiệu quả của mô hình bị giới hạn bởi khả năng của thuật toán tối ưu hóa. Chúng ta hiện chưa có sự hiểu biết lý thuyết đầy đủ về các vấn đề tối ưu hóa phi lồi tổng quát trong học sâu.

Mặc dù các hàm đơn giản có khả năng tổng quát hóa tốt hơn (tức là có khoảng cách nhỏ giữa lỗi huấn luyện và lỗi kiểm tra), chúng ta vẫn cần chọn một giả thuyết đủ phức tạp để đạt được lỗi huấn luyện thấp. Thông thường, lỗi huấn luyện giảm dần và tiệm cận giá trị tối thiểu khi độ phức tạp mô hình tăng. Ngược lại, lỗi tổng quát hóa có dạng đường cong hình chữ U khi xét theo độ phức tạp mô hình. Mô tả ở Hình 5.3



Hình 5.3: Minh họa mối quan hệ điển hình giữa độ phức tạp của mô hình và sai số. Sai số trên tập huấn luyện và tập kiểm tra có xu hướng thay đổi khác nhau. Ở phía bên trái của đồ thị, khi mô hình có độ phức tạp thấp, cả sai số huấn luyện và sai số tổng quát đều cao. Đây được gọi là hiện tượng underfitting (mô hình chưa đủ phức tạp để học tốt dữ liệu). Khi tăng độ phức tạp của mô hình, sai số huấn luyện giảm dần, nhưng khoảng cách giữa sai số huấn luyện và sai số tổng quát bắt đầu tăng lên. Nếu độ phức tạp của mô hình tiếp tục tăng vượt quá một mức tối ưu, khoảng cách này trở nên quá lớn, dẫn đến hiện tượng overfitting (mô hình quá phức tạp, học thuộc dữ liệu huấn luyện nhưng không tổng quát hóa tốt cho dữ liệu mới). Mức độ phức tạp tối ưu được gọi là optimal capacity, tại đó mô hình đạt được sự cân bằng tốt nhất giữa lỗi huấn luyện và lỗi tổng quát.

Để đạt tới trường hợp cực hạn với khả năng mô hình cao tùy ý, ta có thể sử dụng mô hình phi tham số (nonparametric models). Cho đến nay, chúng ta mới chỉ xét các mô hình tham số, chẳng hạn như hồi quy tuyến tính. Các mô hình tham số học một hàm được mô tả bởi một vector tham số có kích thước hữu hạn và được cố định trước khi quan sát dữ liệu.

Trong một số trường hợp, các mô hình phi tham số (nonparametric models) chỉ mang tính lý thuyết, ví dụ như một thuật toán tìm kiếm qua tất cả các phân bố xác suất có thể có. Tuy nhiên, trong thực tế, chúng ta có thể thiết kế các mô hình phi tham số thực tiễn

bằng cách làm cho độ phức tạp của chúng phụ thuộc vào kích thước của tập huấn luyện. Một ví dụ điển hình của mô hình này là hồi quy láng giềng gần nhất (nearest neighbor regression).

Không giống như hồi quy tuyến tính, trong đó mô hình có một tập trọng số cố định, hồi quy láng giềng gần nhất chỉ cần lưu trữ dữ liệu đầu vào $\mathbf{X}$ và nhãn đầu ra $\mathbf{y}$ từ tập huấn luyện. Khi cần phân loại một điểm kiểm tra $\mathbf{x}$, mô hình sẽ tìm điểm gần nhất trong tập huấn luyện và trả về nhãn hồi quy tương ứng. Cụ thể:

$$\hat{y} = y_i \quad \text{với } i = \arg\min \|\mathbf{X}_i - \mathbf{x}\|_2^2. \tag{5.22}$$

Thuật toán này cũng có thể được mở rộng bằng cách sử dụng các độ đo khoảng cách khác thay vì chuẩn L_2, chẳng hạn như các độ đo học được từ dữ liệu *(Goldberger et al., 2005)*. Nếu có nhiều điểm dữ liệu có cùng khoảng cách gần nhất, ta có thể tính giá trị trung bình của các y_i tương ứng để giảm sai số. Trong trường hợp này, thuật toán sẽ đạt được lỗi huấn luyện nhỏ nhất có thể. Tuy nhiên, lỗi này có thể lớn hơn 0 nếu có hai điểm đầu vào giống nhau nhưng có nhãn đầu ra khác nhau trong tập dữ liệu hồi quy.

Cuối cùng, chúng ta cũng có thể tạo ra một thuật toán học phi tham số bằng cách bao bọc một thuật toán học có tham số bên trong một thuật toán khác, giúp tăng số lượng tham số khi cần. Ví dụ, ta có thể tưởng tượng một vòng lặp học tập bên ngoài điều chỉnh bậc của đa thức được học bởi hồi quy tuyến tính trên một phép mở rộng đa thức của đầu vào.

Mô hình lý tưởng là một tiên tri (oracle) biết chính xác phân phối xác suất thật sự của dữ liệu. Ngay cả mô hình này cũng sẽ mắc một số lỗi do có thể tồn tại nhiễu trong phân phối. Trong học có giám sát, ánh xạ từ $\mathbf{x}$ đến y có thể mang tính ngẫu nhiên, hoặc y có thể là một hàm tất định phụ thuộc vào nhiều biến khác ngoài $\mathbf{x}$. Lỗi mà một tiên tri mắc phải khi dự đoán từ phân phối thật $p(\mathbf{x}, y)$ được gọi là lỗi Bayes.

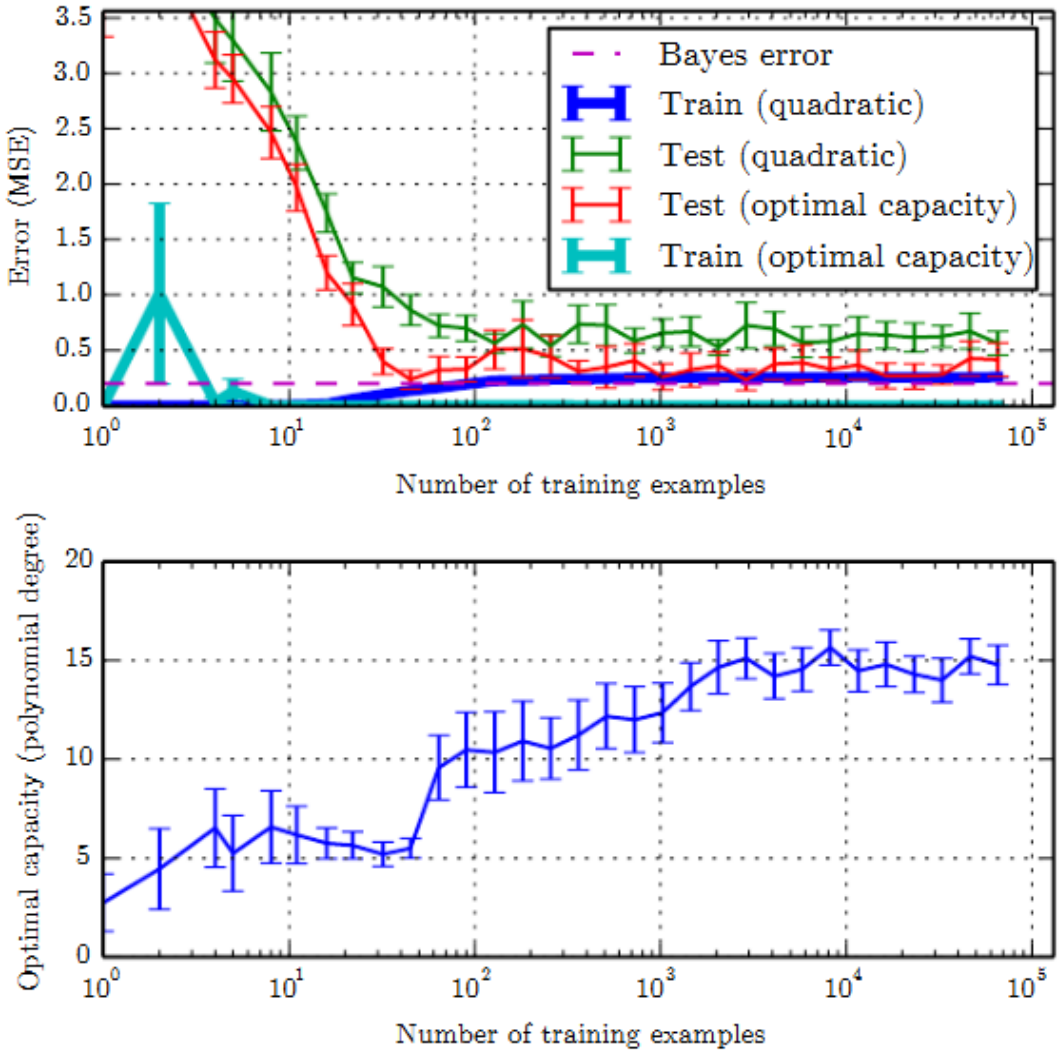
Lỗi huấn luyện và lỗi khái quát hóa thay đổi theo kích thước của tập huấn luyện. Lỗi khái quát hóa kỳ vọng không bao giờ tăng khi số lượng mẫu huấn luyện tăng. Đối với các mô hình phi tham số, càng nhiều dữ liệu thì khả năng khái quát hóa càng tốt, cho đến khi đạt được lỗi tối ưu nhất có thể. Bất kỳ mô hình tham số cố định nào có dung lượng nhỏ hơn mức tối ưu sẽ có lỗi hội tụ đến một giá trị lớn hơn lỗi Bayes (Nhìn hình 5.4). Chúng ta cũng lưu ý rằng, một mô hình có thể có dung lượng tối ưu nhưng vẫn có khoảng cách lớn giữa lỗi huấn luyện và lỗi khái quát hóa. Trong trường hợp này, ta có thể giảm khoảng cách đó bằng cách thu thập thêm dữ liệu huấn luyện.

5.2.1 Định lý Không Có Bữa Trưa Miễn Phí (No Free Lunch Theorem)

Lý thuyết học máy cho rằng một thuật toán học máy có thể khái quát tốt từ một tập dữ liệu huấn luyện hữu hạn. Điều này dường như mâu thuẫn với các nguyên tắc cơ bản của logic. Lập luận quy nạp (inductive reasoning), tức là suy ra quy tắc tổng quát từ một tập hợp hữu hạn các ví dụ, không hợp lệ theo logic thuần túy. Để suy luận logic một quy tắc áp dụng cho mọi phần tử trong tập dữ liệu, ta cần có thông tin về tất cả các phần tử đó.

Học máy phần nào tránh được vấn đề này bằng cách chỉ đưa ra các quy tắc mang tính xác suất, thay vì các quy tắc chắc chắn tuyệt đối như trong lập luận logic thuần túy. Học máy tìm kiếm các quy tắc có xác suất đúng cao đối với hầu hết các phần tử trong tập dữ liệu.

Tuy nhiên, ngay cả điều này cũng không giải quyết triệt để vấn đề. Định lý Không Có Bữa Trưa Miễn Phí (No Free Lunch Theorem) trong học máy *(Wolpert, 1996)* chỉ ra rằng, khi tính trung bình trên tất cả các phân phối sinh dữ liệu có thể có, mọi thuật toán phân loại đều có cùng một tỷ lệ lỗi khi phân loại các điểm chưa quan sát trước đó. Nói cách khác, không có thuật toán học máy nào tốt hơn thuật toán khác trong mọi trường hợp. Ngay cả thuật toán tinh vi nhất cũng có hiệu suất trung bình (trên tất cả các tác vụ có thể có) ngang với việc chỉ đơn giản dự đoán rằng mọi điểm thuộc cùng một lớp.



Hình 5.4: Hình 5.4 mô tả ảnh hưởng của kích thước tập dữ liệu huấn luyện đến lỗi huấn luyện, lỗi kiểm tra và độ phức tạp tối ưu của mô hình. Một bài toán hồi quy tổng hợp được xây dựng bằng cách thêm nhiễu vào đa thức bậc 5, sau đó tạo ra một tập dữ liệu duy nhất và chia thành các tập huấn luyện có kích thước khác nhau. Với mỗi kích thước, 40 tập huấn luyện khác nhau được tạo để vẽ thanh lỗi với mức tin cậy 95%. (Trên): Sai số trung bình bình phương (MSE) trên tập huấn luyện và tập kiểm tra cho hai mô hình: mô hình bậc hai và mô hình có bậc được chọn để tối ưu hóa lỗi kiểm tra. Với mô hình bậc hai, lỗi huấn luyện tăng theo kích thước tập huấn luyện do tập dữ liệu lớn hơn khó khớp hơn. Đồng thời, lỗi kiểm tra giảm do số lượng giả thuyết sai phù hợp với dữ liệu huấn luyện ít hơn. Tuy nhiên, mô hình bậc hai không đủ khả năng để giải quyết bài toán, do đó lỗi kiểm tra tiệm cận đến một giá trị cao. Trong khi đó, mô hình có độ phức tạp tối ưu có thể đạt lỗi kiểm tra tiệm cận đến lỗi Bayes. Thậm chí, lỗi huấn luyện có thể nhỏ hơn lỗi Bayes do thuật toán có thể ghi nhớ các mẫu cụ thể. Khi kích thước tập huấn luyện tiến tới vô hạn, lỗi huấn luyện của bất kỳ mô hình cố định nào cũng phải lớn hơn hoặc bằng lỗi Bayes. (Dưới): Khi kích thước tập huấn luyện tăng, độ phức tạp tối ưu của mô hình (biểu diễn qua bậc của mô hình hồi quy đa thức) cũng tăng. Tuy nhiên, độ phức tạp tối ưu sẽ bão hòa sau khi đạt đủ mức để giải quyết bài toán.

May mắn thay, những kết quả này chỉ đúng khi chúng ta tính trung bình trên *tất cả* các phân phối sinh dữ liệu có thể có. Nếu chúng ta đưa ra giả định về các loại phân phối xác suất mà ta gặp phải trong các ứng dụng thực tế, thì chúng ta có thể thiết kế các thuật toán học máy hoạt động tốt trên những phân phối này.

Điều này có nghĩa là mục tiêu của nghiên cứu học máy không phải là tìm kiếm một

thuật toán học tập mang tính phổ quát hoặc thuật toán học tập tốt nhất tuyệt đối. Thay vào đó, mục tiêu của chúng ta là hiểu các loại phân phối nào liên quan đến "thế giới thực"mà một tác tử AI trải nghiệm, và những thuật toán học máy nào hoạt động hiệu quả trên dữ liệu được lấy từ các loại phân phối sinh dữ liệu mà chúng ta quan tâm.

5.2.2 Regularization - Chính quy hóa

Định lý *No Free Lunch* cho thấy rằng không có thuật toán học máy nào có thể hoạt động tốt trên tất cả các bài toán. Do đó, chúng ta cần thiết kế thuật toán sao cho phù hợp với từng bài toán cụ thể. Điều này được thực hiện bằng cách đưa các ưu tiên vào thuật toán học. Khi những ưu tiên này phù hợp với bài toán cần giải quyết, thuật toán sẽ hoạt động hiệu quả hơn.

Một cách cơ bản để điều chỉnh thuật toán học là thay đổi khả năng biểu diễn của mô hình. Điều này có thể thực hiện bằng cách thêm hoặc loại bỏ các hàm trong không gian giả thuyết - tập hợp các lời giải mà thuật toán có thể lựa chọn. Ví dụ, trong bài toán hồi quy, chúng ta có thể thay đổi bậc của đa thức để điều chỉnh độ phức tạp của mô hình. Tuy nhiên, cách tiếp cận này còn khá đơn giản và chưa toàn diện.

Hiệu suất của thuật toán không chỉ bị ảnh hưởng bởi kích thước của không gian giả thuyết mà còn bởi loại hàm cụ thể trong đó. Chẳng hạn, trong hồi quy tuyến tính, không gian giả thuyết bao gồm các hàm tuyến tính của đầu vào. Những hàm này rất hữu ích khi quan hệ giữa đầu vào và đầu ra là tuyến tính. Tuy nhiên, nếu bài toán có quan hệ phi tuyến, như dự đoán $\sin(x)$ từ x, thì hồi quy tuyến tính sẽ không hoạt động tốt.

Chính quy hóa cho phép kiểm soát hiệu suất của thuật toán bằng cách chọn loại hàm mà thuật toán được phép sử dụng và giới hạn số lượng hàm này. Điều này giúp mô hình học tốt hơn và tránh hiện tượng quá khớp.

Trong không gian giả thuyết của một thuật toán học máy, có thể có nhiều hàm mục tiêu hợp lệ. Tuy nhiên, đôi khi ta muốn ưu tiên một lời giải hơn lời giải khác. Giải pháp không được ưu tiên chỉ được chọn nếu nó khớp với dữ liệu huấn luyện tốt hơn đáng kể so với giải pháp được ưu tiên.

Một cách để làm điều này trong hồi quy tuyến tính là sử dụng giảm trọng số (weight decay). Cụ thể, chúng ta có thể điều chỉnh tiêu chí huấn luyện bằng cách thêm một thành phần ưu tiên các trọng số nhỏ hơn. Hàm mất mát mới có dạng:

$$J(\mathbf{w}) = \text{MSE}_{\text{train}} + \lambda \mathbf{w}^T \mathbf{w}, \tag{5.23}$$

trong đó λ là một hệ số được chọn trước để kiểm soát mức độ ưu tiên các trọng số nhỏ hơn. Khi $\lambda = 0$, không có sự ưu tiên nào, nhưng khi λ tăng, các trọng số bị ép phải nhỏ hơn. Việc tối ưu hàm $J(\mathbf{w})$ giúp cân bằng giữa độ khớp của mô hình với dữ liệu huấn luyện và độ phức tạp của mô hình. Điều này dẫn đến những nghiệm có độ dốc nhỏ hơn hoặc có ít trọng số khác 0 hơn.

Ví dụ, khi thực hiện hồi quy đa thức bậc cao, ta có thể kiểm soát hiện tượng *quá khớp* (overfitting) hoặc *chưa khớp* (underfitting) bằng cách điều chỉnh giá trị λ. Hình 5.5 minh họa kết quả với các giá trị khác nhau của λ.

Tổng quát hơn, ta có thể chuẩn hóa (regularization) một mô hình học hàm $f(x;\theta)$ bằng cách thêm một thành phần gọi là *bộ chuẩn hóa* (regularizer) vào hàm mất mát. Trong trường hợp giảm trọng số, bộ chuẩn hóa là:

$$\Omega(\mathbf{w}) = \mathbf{w}^T\mathbf{w}. \tag{5.24}$$

Trong chương 7, chúng ta sẽ thấy rằng còn nhiều dạng bộ chuẩn hóa khác có thể được áp dụng để cải thiện khả năng tổng quát của mô hình.

Việc ưu tiên một hàm mục tiêu hơn các hàm khác là một cách tổng quát để kiểm soát độ phức tạp của mô hình. Thay vì loại bỏ hoàn toàn một số hàm ra khỏi không gian giả thuyết, ta có thể coi việc loại trừ một hàm là thể hiện một sự ưu tiên vô hạn chống lại hàm đó.

Trong ví dụ về giảm trọng số, chúng ta thể hiện sự ưu tiên cho các mô hình hồi quy tuyến tính có trọng số nhỏ hơn bằng cách thêm một điều kiện ràng buộc vào hàm mất mát. Ngoài giảm trọng số, có nhiều cách khác nhau để thể hiện sự ưu tiên đối với các nghiệm khác nhau, cả một cách tường minh lẫn ngầm định. Những phương pháp này được gọi chung là chuẩn hóa (regularization).

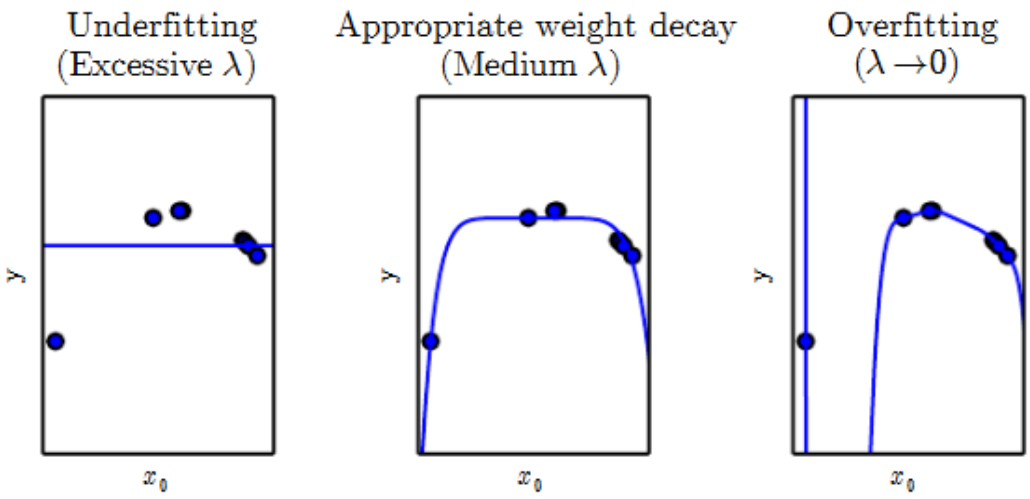
Regularization là bất kỳ điều chỉnh nào đối với thuật toán học nhằm giảm lỗi khái quát hóa mà không làm giảm lỗi huấn luyện. Regularization là một trong những vấn đề quan trọng nhất trong lĩnh vực học máy, chỉ đứng sau tối ưu hóa.

Định lý "Không có bữa trưa miễn phí"(No Free Lunch Theorem) đã chỉ ra rằng không có thuật toán học máy nào là tối ưu cho mọi bài toán, và đặc biệt là không có một dạng regularization nào tốt nhất cho mọi trường hợp. Thay vào đó, chúng ta cần chọn một dạng regularization phù hợp với bài toán cụ thể mà chúng ta đang giải quyết.

5.3 Các Siêu Tham Số và Tập Kiểm Định (Hyperparameters and Validation Sets)

Hầu hết các thuật toán học máy đều có siêu tham số (hyperparameters), là các thiết lập giúp ta kiểm soát hành vi của thuật toán. Giá trị của các siêu tham số không được thuật toán học máy tự động điều chỉnh mà cần được xác định từ bên ngoài (mặc dù ta có thể thiết kế một quy trình học lồng nhau trong đó một thuật toán học sẽ tìm ra các siêu tham số tối ưu cho một thuật toán học khác).

Ví dụ, trong bài toán hồi quy đa thức được minh họa trong hình 5.2, có một siêu tham số duy nhất: bậc của đa thức. Đây là một siêu tham số quyết định độ phức tạp (capacity) của mô hình. Ngoài ra, giá trị λ trong phương pháp giảm trọng số cũng là một ví dụ khác về siêu tham số, vì nó kiểm soát mức độ ưu tiên của trọng số nhỏ hơn trong thuật toán học.



Hình 5.5: Hình 5.5 minh họa việc áp dụng mô hình hồi quy đa thức bậc cao vào tập huấn luyện trong hình 5.2. Hàm thực sự là một hàm bậc hai, nhưng ở đây ta chỉ sử dụng mô hình bậc 9. Để ngăn chặn mô hình phức tạp này bị overfitting, ta điều chỉnh giá trị của weight decay (λ). *(Trái)* Khi λ rất lớn, mô hình bị ép buộc học một hàm gần như không có độ dốc, dẫn đến hiện tượng underfitting vì nó chỉ có thể biểu diễn một hàm hằng. *(Giữa)* Khi λ ở mức trung bình, thuật toán có thể khôi phục một đường cong có hình dạng tổng quát đúng. Mặc dù mô hình có khả năng biểu diễn các hàm phức tạp hơn, nhưng weight decay đã khuyến khích nó sử dụng một hàm đơn giản hơn với các hệ số nhỏ hơn. *(Phải)* Khi λ tiến dần về 0 (tức là sử dụng pseudo-inverse Moore-Penrose để giải bài toán kém xác định với mức regularization tối thiểu), mô hình hồi quy bậc 9 bị overfit đáng kể, như đã thấy trong hình 5.2.

Khi xây dựng một mô hình học máy, chúng ta cần tối ưu hóa các tham số và siêu tham số (hyperparameters) để đạt hiệu suất cao nhất. Tuy nhiên, nếu chúng ta điều chỉnh siêu tham số trực tiếp trên tập huấn luyện (training set), mô hình có thể bị overfitting—tức là học quá mức vào dữ liệu huấn luyện nhưng không tổng quát hóa tốt trên dữ liệu mới.

Ví dụ, nếu chúng ta sử dụng một mô hình hồi quy đa thức, ta có thể luôn khớp hoàn hảo dữ liệu huấn luyện bằng cách tăng bậc của đa thức. Tuy nhiên, điều này không có nghĩa là mô hình sẽ hoạt động tốt trên dữ liệu mới. Để giải quyết vấn đề này, chúng ta cần một tập dữ liệu validation set mà mô hình không được huấn luyện trực tiếp trên đó.

Tập kiểm tra (test set) là tập dữ liệu chỉ được sử dụng sau khi quá trình học hoàn tất để đánh giá khả năng tổng quát hóa của mô hình. Quan trọng là test set không được sử dụng trong quá trình tinh chỉnh siêu tham số hay lựa chọn mô hình, vì điều này có thể dẫn đến việc vô tình tối ưu hóa mô hình theo test set, làm mất ý nghĩa đánh giá.

Thay vào đó, ta cần validation set—được trích từ tập huấn luyện ban đầu—để điều chỉnh siêu tham số. Cách làm phổ biến là chia tập huấn luyện thành hai phần:

- Training set: Dùng để học các tham số của mô hình.
- Validation set: Dùng để đánh giá lỗi tổng quát hóa trong quá trình huấn luyện và tinh chỉnh mô hình.

Thông thường, người ta sử dụng khoảng 80% dữ liệu huấn luyện cho training set và 20% còn lại làm validation set. Do validation set được sử dụng để tối ưu hóa siêu tham số,

lỗi trên validation set thường nhỏ hơn lỗi trên test set nhưng vẫn cao hơn lỗi trên training set.

Sau khi hoàn tất quá trình tinh chỉnh siêu tham số, mô hình cuối cùng sẽ được đánh giá trên test set để có một thước đo chính xác về hiệu suất trên dữ liệu chưa từng thấy.

Trong thực tế, nếu một tập test được sử dụng quá nhiều lần để đánh giá hiệu suất của các mô hình khác nhau trong nhiều năm, thì các thuật toán mới có thể vô tình tối ưu hóa theo tập test này thay vì tổng quát hóa tốt hơn cho dữ liệu thực tế. Điều này khiến kết quả đánh giá trên test set trở nên lạc quan một cách giả tạo.

Hơn nữa, nếu cộng đồng nghiên cứu liên tục tối ưu hóa mô hình để đạt hiệu suất cao nhất trên một tập test cũ, thì tập test đó sẽ dần trở nên lỗi thời và không còn phản ánh chính xác hiệu suất thực tế của hệ thống khi triển khai. Do đó, các benchmark thường được cập nhật với các tập dữ liệu mới, lớn hơn và phức tạp hơn để đảm bảo việc đánh giá mô hình vẫn khách quan và chính xác.

5.3.1 Cross-Validation

Một vấn đề khi chia tập dữ liệu cố định thành tập huấn luyện và tập kiểm tra là nếu tập kiểm tra quá nhỏ, nó sẽ làm tăng độ bất định trong việc ước tính lỗi tổng quát hóa. Điều này dẫn đến khó khăn trong việc khẳng định thuật toán A tốt hơn thuật toán B trên một bài toán cụ thể.

Khi tập dữ liệu có hàng trăm nghìn mẫu trở lên, điều này không phải là vấn đề lớn. Tuy nhiên, nếu tập dữ liệu nhỏ, các phương pháp thay thế có thể giúp tận dụng toàn bộ dữ liệu để ước tính lỗi trung bình mà không cần giữ lại một phần cố định làm test set. Các phương pháp này dựa trên ý tưởng lặp lại quá trình huấn luyện và kiểm tra trên các tập con khác nhau của tập dữ liệu ban đầu.

Một trong những phương pháp phổ biến nhất là k-fold cross-validation. Trong phương pháp này, tập dữ liệu được chia thành k tập con không trùng lặp. Sau đó, mô hình được huấn luyện k lần, mỗi lần sử dụng một tập con khác nhau làm tập kiểm tra, trong khi phần còn lại dùng để huấn luyện. Lỗi kiểm tra trung bình qua k lần lặp được dùng để ước tính lỗi tổng quát hóa.

Một hạn chế của phương pháp này là không có bộ ước lượng không thiên lệch cho phương sai của lỗi trung bình thu được (Bengio và Grandvalet, 2004). Tuy nhiên, trong thực tế, các ước lượng xấp xỉ vẫn thường được sử dụng.

5.4 Ước lượng, Độ chệch và Phương sai (Estimators, Bias and Variance)

Thống kê cung cấp nhiều công cụ giúp chúng ta đạt được mục tiêu của học máy: không chỉ giải quyết bài toán trên tập huấn luyện mà còn có khả năng tổng quát hóa cho dữ liệu mới.

Các khái niệm cốt lõi như ước lượng tham số, độ chệch (*bias*) và phương sai (*variance*) đóng vai trò quan trọng trong việc đánh giá khả năng tổng quát hóa của mô hình. Hiểu rõ về độ chệch và phương sai giúp ta phân tích được hiện tượng dưới khớp (*underfitting*) và

quá khớp (*overfitting*), từ đó có chiến lược điều chỉnh mô hình phù hợp.

5.4.1 Ước lượng điểm

Ước lượng điểm là phương pháp nhằm cung cấp một giá trị dự đoán "tốt nhất"cho một đại lượng quan tâm. Trong thực tế, đại lượng này có thể là một tham số đơn lẻ hoặc một vector tham số trong một mô hình tham số, chẳng hạn như trọng số trong bài toán hồi quy tuyến tính được đề cập ở mục 5.1.4, hoặc thậm chí là một hàm số hoàn chỉnh.

Để phân biệt giữa giá trị ước lượng và giá trị thực của tham số, ta sử dụng ký hiệu $\hat{\theta}$ để biểu diễn ước lượng điểm của một tham số θ.

Giả sử tập dữ liệu gồm m quan sát độc lập và phân phối giống hệt nhau được ký hiệu là $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$.

Thuật toán 5.1 Thuật toán kiểm tra chéo k-fold. Nó có thể được sử dụng để ước tính lỗi khái quát hóa của một thuật toán học máy A khi tập dữ liệu $\mathbb{D}$ quá nhỏ để chia thành tập huấn luyện/kiểm tra hoặc tập huấn luyện/xác thực đơn giản nhằm mang lại ước tính chính xác về lỗi khái quát hóa, bởi vì trung bình của một hàm mất mát L trên một tập kiểm tra nhỏ có thể có phương sai cao. Tập dữ liệu $\mathbb{D}$ chứa các mẫu trừu tượng $z^{(i)}$ (cho mẫu thứ i), có thể là cặp (đầu vào, nhãn) $z^{(i)} = (x^{(i)}, y^{(i)})$ trong học có giám sát hoặc chỉ là đầu vào $z^{(i)} = x^{(i)}$ trong học không giám sát. Thuật toán trả về một vector các lỗi e cho mỗi mẫu trong $\mathbb{D}$, trong đó trung bình của nó là lỗi khái quát hóa ước tính. Các lỗi trên từng mẫu có thể được sử dụng để tính toán khoảng tin cậy xung quanh trung bình (phương trình 5.47). Mặc dù những khoảng tin cậy này không được chứng minh đầy đủ sau khi sử dụng kiểm tra chéo, nhưng vẫn là thực tế phổ biến khi sử dụng chúng để tuyên bố rằng thuật toán A tốt hơn thuật toán B chỉ khi khoảng tin cậy của lỗi của thuật toán A nằm dưới và không giao với khoảng tin cậy của thuật toán B.

Algorithm 5.1 KFoldXV

Require: $\mathbb{D}$, tập dữ liệu cho trước, với các phần tử $z^{(i)}$
Require: A, thuật toán học, được xem như một hàm nhận vào một tập dữ liệu và xuất ra một hàm đã học
Require: L, hàm mất mát, được xem như một hàm từ một hàm đã học f và một mẫu $z^{(i)} \in \mathbb{D}$ tới một giá trị vô hướng $\in \mathbb{R}$
Require: k, số lượng phần chia (folds)
  Chia $\mathbb{D}$ thành k tập con $\mathbb{D}_i$ không giao nhau, sao cho hợp lại là $\mathbb{D}$
  for i từ 1 đến k **do**
    $f_i = A(\mathbb{D} \setminus \mathbb{D}_i)$
    for $z^{(j)}$ trong $\mathbb{D}_i$ **do**
      $e_j = L(f_i, z^{(j)})$
    end for
  end for
  Trả về e

Trong thống kê, một ước lượng điểm (point estimator) hay còn gọi là thống kê (statistic) là một hàm số của dữ liệu:

$$\hat{\theta}_m = g(x^{(1)}, \ldots, x^{(m)}). \tag{5.25}$$

Định nghĩa trên không yêu cầu hàm g phải trả về một giá trị gần với tham số thật θ, hay thậm chí miền giá trị của g phải trùng với tập hợp giá trị có thể có của θ. Điều này cho phép người thiết kế ước lượng có sự linh hoạt lớn. Do đó, hầu như bất kỳ hàm số nào cũng có thể được xem là một ước lượng.

Một ước lượng tốt là một hàm số có đầu ra gần với giá trị thực sự θ đã tạo ra dữ liệu huấn luyện.

Chúng ta sẽ xem xét ước lượng điểm từ góc độ tần suất (frequentist). Điều này có nghĩa là: - Giá trị tham số thật θ là cố định nhưng không xác định được. - Ước lượng điểm $\hat{\theta}$ là một hàm của dữ liệu. - Dữ liệu được lấy ngẫu nhiên từ một phân phối xác suất, do đó $\hat{\theta}$ cũng là một biến ngẫu nhiên.

Ngoài ra, ước lượng điểm còn có thể dùng để ước lượng mối quan hệ giữa đầu vào và đầu ra. Trong trường hợp này, ta gọi chúng là ước lượng hàm số.

Trong một số trường hợp, ta không chỉ quan tâm đến việc ước lượng một tham số mà còn muốn ước lượng một hàm số. Ví dụ, ta muốn dự đoán biến $\mathbf{y}$ dựa trên một đầu vào $\mathbf{x}$. Giả sử có một hàm số $f(x)$ biểu diễn mối quan hệ gần đúng giữa $\mathbf{y}$ và $\mathbf{x}$:

$$\mathbf{y} = f(\mathbf{x}) + \epsilon, \tag{5.26}$$

trong đó ϵ biểu diễn phần nhiễu mà ta không thể dự đoán từ $\mathbf{x}$.

Trong ước lượng hàm số, mục tiêu của ta là tìm một mô hình hoặc một hàm $\hat{f}$ để xấp xỉ f. Có thể xem ước lượng hàm số như một trường hợp đặc biệt của ước lượng điểm, trong đó thay vì ước lượng một tham số θ, ta ước lượng cả một hàm số f.

Ví dụ:

- Hồi quy tuyến tính (được trình bày trong Mục 5.1.4)

- Hồi quy đa thức (được trình bày trong Mục 5.2)

Cả hai phương pháp trên đều minh họa cách tiếp cận để ước lượng một tham số $\mathbf{w}$ hoặc một hàm $\hat{f}$ ánh xạ từ $\mathbf{x}$ sang $\mathbf{y}$.

Tiếp theo, chúng ta sẽ xem xét các tính chất quan trọng nhất của các bộ ước lượng và thảo luận về cách đánh giá chúng.

5.4.2 Độ Chệch Của Bộ Ước Lượng

Độ chệch (bias) của một bộ ước lượng được định nghĩa như sau:

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta, \tag{5.27}$$

trong đó kỳ vọng được lấy trên dữ liệu (xem dữ liệu như là mẫu từ một biến ngẫu nhiên), và θ là giá trị thật của tham số được dùng để xác định phân phối tạo dữ liệu.

Một bộ ước lượng $\hat{\theta}_m$ được gọi là không chệch nếu $\text{bias}(\hat{\theta}_m) = 0$, hay nói cách khác

$\mathbb{E}(\hat{\theta}_m) = \theta$. Một bộ ước lượng được gọi là không chệch tiệm cận nếu:

$$\lim_{m\to\infty} \text{bias}(\hat{\theta}_m) = 0, \tag{5.28}$$

đồng nghĩa với việc:

$$\lim_{m\to\infty} \mathbb{E}(\hat{\theta}_m) = \theta. \tag{5.29}$$

Ví dụ: Phân phối Bernoulli

Xét một tập mẫu $\{x^{(1)}, \ldots, x^{(m)}\}$ được lấy độc lập và phân phối giống nhau theo phân phối Bernoulli với kỳ vọng θ:

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)}}. \tag{5.30}$$

Một bộ ước lượng phổ biến cho tham số θ của phân phối này là trung bình của các mẫu huấn luyện:

$$\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}. \tag{5.31}$$

Để kiểm tra xem ước lượng này có bị chệch hay không, ta thay phương trình (5.22) vào phương trình (5.20):

$$\begin{aligned}
\text{bias}(\hat{\theta}_m) &= \mathbb{E}[\hat{\theta}_m] - \theta && (5.32)\\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta && (5.33)\\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}[x^{(i)}] - \theta && (5.34)\\
&= \frac{1}{m}\sum_{i=1}^{m}\sum_{x^{(i)}=0}^{1} \left(x^{(i)}\theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)}}\right) - \theta && (5.35)\\
&= \frac{1}{m}\sum_{i=1}^{m}(\theta) - \theta && (5.36)\\
&= \theta - \theta = 0. && (5.37)
\end{aligned}$$

Vì $\text{bias}(\hat{\theta}) = 0$, ta kết luận rằng bộ ước lượng $\hat{\theta}$ là không chệch.

Ví Dụ: Bộ Ước Lượng Trung Bình Của Phân Phối Gaussian

Giả sử chúng ta có một tập hợp các mẫu $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ được lấy ngẫu nhiên và độc lập từ một phân phối Gaussian:

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x^{(i)}-\mu)^2}{\sigma^2}\right). \tag{5.38}$$

Một bộ ước lượng phổ biến cho giá trị trung bình μ của phân phối Gaussian là trung bình mẫu:

$$\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}. \tag{5.39}$$

Để xác định độ chệch của trung bình mẫu, ta tính kỳ vọng của nó:

$$\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}[\hat{\mu}_m] - \mu && (5.40)\\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \mu && (5.41)\\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}[x^{(i)}] - \mu && (5.42)\\
&= \frac{1}{m}\sum_{i=1}^{m} \mu - \mu && (5.43)\\
&= \mu - \mu = 0. && (5.44)
\end{aligned}$$

Vậy, ta kết luận rằng trung bình mẫu là một bộ ước lượng không chệch của trung bình Gaussian.

Ví dụ: Bộ Ước Lượng Phương Sai Của Phân Phối Gaussian. Trong ví dụ này, chúng ta so sánh hai bộ ước lượng khác nhau của tham số phương sai σ^2 trong một phân phối Gaussian. Mục tiêu của chúng ta là xác định xem liệu các bộ ước lượng này có bị chệch hay không.

Bây giờ, chúng ta xét bộ ước lượng cho phương sai σ^2 của phân phối Gaussian. Bộ ước lượng đầu tiên được xem xét là phương sai mẫu:

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2. \tag{5.45}$$

Ở đây, $\hat{\mu}_m$ là trung bình mẫu. Chúng ta muốn kiểm tra xem bộ ước lượng này có bị chệch hay không bằng cách tính độ chệch:

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2. \tag{5.46}$$

Trước tiên, ta tính kỳ vọng của $\hat{\sigma}_m^2$:

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] && (5.47)\\
&= \frac{m-1}{m}\sigma^2. && (5.48)
\end{aligned}$$

Từ đó, ta thấy rằng:

$$\text{bias}(\hat{\sigma}_m^2) = -\frac{\sigma^2}{m}. \tag{5.49}$$

Vậy phương sai mẫu là một bộ ước lượng bị chệch của phương sai thực sự của phân phối Gaussian.

Một cách tiếp cận khác là sử dụng bộ ước lượng phương sai không chệch, được định nghĩa như sau:

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2. \tag{5.50}$$

Như tên gọi, bộ ước lượng này là không chệch. Cụ thể, chúng ta chứng minh rằng:

$$\mathbb{E}[\tilde{\sigma}_m^2] = \sigma^2. \tag{5.51}$$

Thực hiện phép toán kỳ vọng:

$$\begin{aligned}
\mathbb{E}[\tilde{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] && (5.52)\\
&= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}_m^2] && (5.53)\\
&= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) && (5.54)\\
&= \sigma^2. && (5.55)
\end{aligned}$$

Vậy, ta kết luận rằng $\tilde{\sigma}_m^2$ là một bộ ước lượng không chệch của phương sai thực sự.

Chúng ta có hai bộ ước lượng: một bị chệch và một không chệch. Mặc dù các bộ ước lượng không chệch thường được ưa chuộng, nhưng không phải lúc nào chúng cũng là bộ ước lượng tốt nhất. Trong thực tế, chúng ta thường sử dụng các bộ ước lượng bị chệch vì chúng có những đặc tính quan trọng khác.

5.4.3 Phương sai và Sai số chuẩn

Một đặc tính khác của bộ ước lượng mà chúng ta cần xem xét là mức độ biến thiên của nó khi dữ liệu mẫu thay đổi. Tương tự như cách chúng ta tính kỳ vọng của bộ ước lượng để xác định độ chệch, chúng ta cũng có thể tính phương sai của nó.

Phương sai của một bộ ước lượng được định nghĩa là:

$$\text{Var}(\hat{\theta}) \tag{5.56}$$

trong đó, biến ngẫu nhiên chính là tập dữ liệu huấn luyện. Ngoài ra, căn bậc hai của phương sai được gọi là Sai số chuẩn (*Standard Error*), ký hiệu là $\text{SE}(\hat{\theta})$.

Phương sai hoặc sai số chuẩn của một bộ ước lượng phản ánh mức độ mà chúng ta

mong đợi giá trị ước lượng thu được từ dữ liệu sẽ thay đổi khi lấy mẫu lại từ cùng một quá trình sinh dữ liệu. Cũng giống như chúng ta muốn một bộ ước lượng có độ chệch thấp, chúng ta cũng mong muốn nó có phương sai thấp.

Khi tính bất kỳ thống kê nào từ một tập hợp hữu hạn mẫu, giá trị ước lượng của tham số thực sự có thể không chắc chắn. Điều này có nghĩa là nếu chúng ta lấy các mẫu khác từ cùng một phân phối, các giá trị thống kê của chúng có thể khác nhau. Mức độ biến thiên kỳ vọng trong bất kỳ bộ ước lượng nào chính là một nguồn sai số mà chúng ta cần định lượng.

Sai số chuẩn của trung bình (Standard Error of the Mean - SEM) giúp đánh giá độ không chắc chắn của ước lượng trung bình mẫu. Công thức tính SEM được định nghĩa như sau:

$$\text{SE}(\hat{\mu}_m) = \sqrt{\text{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right)} = \frac{\sigma}{\sqrt{m}}, \tag{5.57}$$

trong đó σ^2 là phương sai thực của mẫu x^i. Trong thực tế, phương sai này thường không được biết trước và phải ước lượng từ dữ liệu. Một số cách phổ biến là sử dụng phương sai mẫu hoặc căn bậc hai của ước lượng không chệch của phương sai. Tuy nhiên, cả hai phương pháp này thường đánh giá thấp độ lệch chuẩn thực sự, nhưng vẫn được sử dụng phổ biến do tính khả dụng và đơn giản của chúng. Khi kích thước mẫu m lớn, sai số này trở nên nhỏ và phép xấp xỉ trở nên chính xác hơn.

Sai số chuẩn của trung bình rất quan trọng trong các thí nghiệm học máy. Khi đánh giá tổng quát hóa của một mô hình, ta thường tính trung bình lỗi trên tập kiểm tra. Độ chính xác của ước lượng này phụ thuộc vào số lượng mẫu trong tập kiểm tra.

Theo định lý giới hạn trung tâm, nếu số lượng mẫu đủ lớn, trung bình của chúng sẽ phân phối xấp xỉ theo phân phối chuẩn. Do đó, ta có thể sử dụng sai số chuẩn để tính toán khoảng tin cậy cho giá trị trung bình thực sự. Chẳng hạn, khoảng tin cậy 95% cho trung bình $\hat{\mu}_m$ được tính như sau:

$$\left(\hat{\mu}_m - 1.96 \cdot \text{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96 \cdot \text{SE}(\hat{\mu}_m)\right). \tag{5.58}$$

Khoảng tin cậy này có ý nghĩa quan trọng trong so sánh các mô hình học máy. Khi so sánh hai thuật toán A và B, ta có thể coi thuật toán A tốt hơn nếu giới hạn trên của khoảng tin cậy 95% của lỗi của thuật toán A nhỏ hơn giới hạn dưới của khoảng tin cậy 95% của lỗi thuật toán B. Điều này giúp đảm bảo rằng sự khác biệt về hiệu suất không chỉ do nhiễu ngẫu nhiên mà thực sự có ý nghĩa thống kê.

Ví dụ: Phân phối Bernoulli

Xét một tập mẫu $\{x^{(1)}, \ldots, x^{(m)}\}$ được chọn độc lập và giống hệt nhau từ một phân

phối Bernoulli, với xác suất $P(x^{(i)};\theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}$. Giả sử chúng ta muốn tính phương sai của ước lượng $\hat{\theta}_m$, được định nghĩa là:

$$\text{Var}(\hat{\theta}_m) = \text{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \tag{5.59}$$

Sử dụng tính chất của phương sai:

$$= \frac{1}{m^2}\sum_{i=1}^{m} \text{Var}(x^{(i)}) \tag{5.60}$$

Vì mỗi biến $x^{(i)}$ tuân theo phân phối Bernoulli với phương sai $\theta(1-\theta)$, ta có:

$$= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \tag{5.61}$$

$$= \frac{1}{m^2} m\theta(1-\theta) \tag{5.62}$$

$$= \frac{1}{m}\theta(1-\theta) \tag{5.63}$$

Như vậy, phương sai của ước lượng giảm khi kích thước mẫu m tăng. Đây là một tính chất quan trọng của các bộ ước lượng phổ biến mà chúng ta sẽ quay lại khi bàn về tính hội tụ.

5.5 Đánh đổi giữa Bias và Variance để giảm Mean Squared Error

Bias và phương sai (variance) đo lường hai nguồn sai số khác nhau trong một bộ ước lượng. Bias đo lường độ lệch trung bình so với giá trị thật của hàm hoặc tham số cần ước lượng. Trong khi đó, phương sai đo lường mức độ dao động của giá trị ước lượng do sự thay đổi của tập dữ liệu huấn luyện.

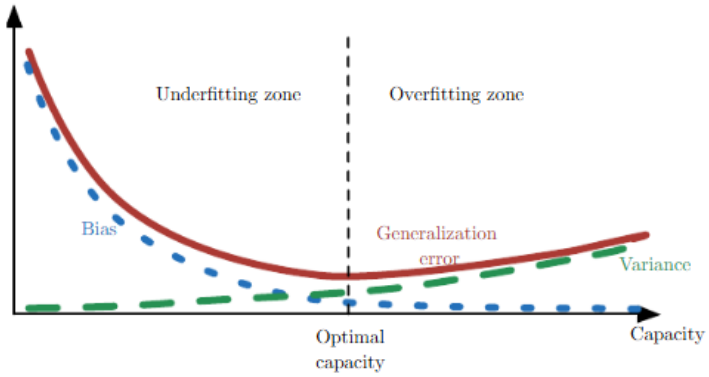
Khi đối mặt với hai bộ ước lượng, một có bias cao và một có phương sai cao, chúng ta nên chọn cái nào? Ví dụ, giả sử chúng ta muốn xấp xỉ một hàm số và chỉ có hai lựa chọn: một mô hình có bias cao và một mô hình có phương sai cao. Làm thế nào để lựa chọn?

Cách tiếp cận phổ biến nhất để xử lý trade-off này là sử dụng cross-validation. Thực nghiệm cho thấy cross-validation hoạt động rất hiệu quả trong nhiều bài toán thực tế. Một cách khác là so sánh sai số bình phương trung bình (Mean Squared Error - MSE) của các bộ ước lượng:

$$MSE = \mathbb{E}[(\hat{\theta}_m - \theta)^2] \tag{5.64}$$

$$= \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m) \tag{5.65}$$

MSE đo lường mức độ sai số kỳ vọng tổng thể giữa giá trị ước lượng và giá trị thật của tham số θ. Công thức trên cho thấy rằng MSE kết hợp cả bias và phương sai. Một bộ ước lượng tốt là bộ có MSE nhỏ, nghĩa là nó giữ bias và phương sai ở mức hợp lý.



Hình 5.6: Khi dung lượng mô hình tăng (trục hoành), bias (đường chấm chấm) có xu hướng giảm, trong khi phương sai (đường gạch đứt) tăng lên, dẫn đến một đường cong hình chữ U cho lỗi tổng quát hóa (đường đậm). Nếu chúng ta điều chỉnh dung lượng mô hình theo một trục, sẽ tồn tại một mức dung lượng tối ưu: dưới mức này, mô hình bị underfitting; trên mức này, mô hình bị overfitting. Mối quan hệ này tương tự như mối liên hệ giữa dung lượng mô hình, underfitting và overfitting đã được đề cập trong Mục **??** và Hình **??**.

Mối quan hệ giữa bias và phương sai có liên hệ mật thiết với các khái niệm về khả năng khái quát hóa (generalization), dung lượng mô hình (capacity), underfitting và overfitting. Khi lỗi tổng quát hóa được đo lường bằng MSE, việc tăng dung lượng mô hình thường làm tăng phương sai nhưng giảm bias. Điều này được minh họa bởi đường cong hình chữ U của lỗi tổng quát hóa theo dung lượng mô hình.

5.5.1 Tính nhất quán (Consistency)

Trong quá trình xây dựng mô hình học máy, chúng ta quan tâm đến tính chất của các ước lượng khi kích thước tập dữ liệu tăng lên. Một ước lượng tốt không chỉ có độ chính xác cao trên tập dữ liệu huấn luyện mà còn cần đảm bảo rằng khi số lượng mẫu m trong tập dữ liệu tăng lên, ước lượng dần hội tụ về giá trị thật của tham số cần ước lượng. Điều này được biểu diễn như sau:

$$\operatorname*{plim}_{m\to\infty} \hat{\theta}_m = \theta. \tag{5.66}$$

Ký hiệu plim biểu thị sự hội tụ theo xác suất, có nghĩa là với mọi $\epsilon > 0$, ta có:

$$P(|\hat{\theta}_m - \theta| > \epsilon) \to 0 \text{ khi } m \to \infty. \tag{5.67}$$

Điều kiện này được gọi là tính nhất quán (consistency) của ước lượng $\hat{\theta}_m$. Trong một số trường hợp, nó còn được gọi là tính nhất quán yếu (weak consistency). Trong khi đó, tính nhất quán mạnh (strong consistency) yêu cầu ước lượng hội tụ gần như chắc chắn về

tham số thật:

$$P\Big(\lim_{m\to\infty}\hat{\theta}_m = \theta\Big) = 1. \tag{5.68}$$

Tính nhất quán đảm bảo rằng sai số của bộ ước lượng giảm dần khi số lượng mẫu tăng lên. Tuy nhiên, điều ngược lại không đúng: một bộ ước lượng không chệch không nhất thiết là một bộ ước lượng nhất quán.

Ví dụ, giả sử chúng ta cần ước lượng kỳ vọng μ của một phân phối chuẩn $\mathcal{N}(x, \mu, \sigma^2)$ với một tập dữ liệu gồm m mẫu: $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$. Một cách đơn giản là chọn phần tử đầu tiên $x^{(1)}$ làm ước lượng:

$$\hat{\theta} = x^{(1)}.$$

Bộ ước lượng này là không chệch vì:

$$\mathbb{E}(\hat{\theta}_m) = \theta.$$

Điều này có nghĩa là dù số lượng mẫu m có tăng lên bao nhiêu, trung bình kỳ vọng của bộ ước lượng vẫn bằng giá trị thực của tham số cần ước lượng. Do đó, bộ ước lượng này là không chệch.

Tuy nhiên, bộ ước lượng này không nhất quán. Một bộ ước lượng được gọi là nhất quán nếu giá trị của nó hội tụ về tham số thực sự khi số lượng mẫu $m \to \infty$. Trong trường hợp này, vì $x^{(1)}$ chỉ là một phần tử ngẫu nhiên của tập dữ liệu, nó không nhất thiết hội tụ về μ khi m tăng. Do đó, bộ ước lượng $\hat{\theta}_m$ không phải là một ước lượng nhất quán.

5.6 Ước lượng hợp lý cực đại (Maximum Likelihood Estimation - MLE)

Một nguyên tắc phổ biến để xác định bộ ước lượng là nguyên tắc hợp lý cực đại (MLE). Thay vì đoán một hàm nào đó là bộ ước lượng tốt rồi phân tích độ chệch và phương sai của nó, MLE giúp ta suy ra một bộ ước lượng cụ thể từ một nguyên tắc thống kê phổ quát.

Giả sử chúng ta có một tập hợp m mẫu:

$$\mathbb{X} = \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$$

được rút ngẫu nhiên từ một phân phối dữ liệu thực $p_{\text{data}}(\mathbf{x})$ chưa biết.

Gọi $p_{\text{model}}(\mathbf{x}; \theta)$ là một họ phân phối xác suất tham số hóa trên không gian đầu vào, được xác định bởi tham số θ. Nói cách khác, $p_{\text{model}}(\mathbf{x}; \theta)$ ánh xạ một điểm dữ liệu $\mathbf{x}$ vào một giá trị xác suất ước tính xác suất thực sự $p_{\text{data}}(\mathbf{x})$.

Bộ ước lượng hợp lý cực đại của θ được xác định bởi:

$$\theta_{\text{ML}} = \arg\max_{\theta} p_{\text{model}}(\mathbb{X}; \theta),$$

$$= \arg\max_{\theta} \prod_{i=1}^{m} p_{\text{model}}(x^{(i)}; \theta).$$

Tuy nhiên, tích của nhiều xác suất có thể gây ra vấn đề số học, chẳng hạn như hiện tượng underflow. Để tránh vấn đề này, ta có thể lấy log của hàm hợp lý mà không làm thay đổi giá trị tối đa hóa:

$$\theta_{\text{ML}} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{\text{model}}(x^{(i)}; \theta).$$

Một cách hiểu khác về ước lượng hợp lý cực đại là xem nó như một cách để giảm sự khác biệt giữa phân phối thực nghiệm $\hat{p}_{\text{data}}$ (định nghĩa bởi tập huấn luyện) và phân phối mô hình, với mức độ chênh lệch đo bằng phân kỳ KL:

$$D_{\text{KL}}(\hat{p}_{\text{data}} || p_{\text{model}}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \left[\log \hat{p}_{\text{data}}(x) - \log p_{\text{model}}(x) \right].$$

Vì vế trái chỉ phụ thuộc vào quá trình tạo dữ liệu chứ không phụ thuộc vào mô hình, khi huấn luyện mô hình để tối ưu D_{KL}, ta chỉ cần tối thiểu hóa:

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(x),$$

đây chính là bài toán cực đại hóa đã được đề cập trước đó.

Do đó, tối ưu phân kỳ KL chính là tối ưu hàm cross-entropy giữa hai phân phối. Trong nhiều tài liệu, thuật ngữ "cross-entropy" thường được sử dụng để chỉ log-likelihood âm của Bernoulli hoặc softmax, nhưng thực tế, mọi hàm mất mát chứa log-likelihood âm đều là một dạng của cross-entropy. Ví dụ, sai số bình phương trung bình (MSE) là cross-entropy giữa phân phối thực nghiệm và một phân phối Gaussian.

Như vậy, phương pháp hợp lý cực đại thực chất là một cách để làm cho phân phối mô hình tiệm cận với phân phối thực nghiệm $\hat{p}_{\text{data}}$. Trong thực tế, ta muốn phân phối mô hình tiệm cận phân phối dữ liệu thực sự p_{data}, nhưng ta không có quyền truy cập trực tiếp vào nó.

Dù θ tối ưu là giống nhau dù ta cực đại hóa hợp lý cực đại hay cực tiểu hóa phân kỳ KL, nhưng giá trị của hai hàm mục tiêu là khác nhau. Do đó, trong phần mềm, ta thường mô tả cả hai bài toán này dưới dạng tối ưu hàm mất mát. Tối ưu hợp lý cực đại tương đương với tối thiểu hóa log-likelihood âm (NLL), hay tối thiểu hóa cross-entropy.

Nhìn từ góc độ cross-entropy giúp bài toán dễ hiểu hơn, vì phân kỳ KL luôn nhận giá trị không âm và có giá trị tối thiểu là 0, trong khi log-likelihood âm có thể trở thành số âm khi x là giá trị thực.

5.6.1 Log-Likelihood có Điều kiện và Sai số Bình phương Trung bình

Trong học máy, ước lượng hợp lý cực đại (Maximum Likelihood Estimation - MLE) có thể được mở rộng để ước lượng xác suất có điều kiện $P(\mathbf{y} \mid \mathbf{x}; \theta)$ nhằm dự đoán $\mathbf{y}$ dựa trên $\mathbf{x}$. Đây là tình huống phổ biến nhất trong học có giám sát vì nó là nền tảng của nhiều mô hình học máy.

Giả sử $\mathbf{X}$ biểu diễn toàn bộ đầu vào và $\mathbf{Y}$ là toàn bộ nhãn quan sát, khi đó ước lượng hợp lý cực đại có điều kiện được xác định bởi:

$$\theta_{ML} = \arg\max_{\theta} P(\mathbf{Y} \mid \mathbf{X}; \theta). \tag{5.69}$$

Nếu giả định rằng các mẫu dữ liệu là độc lập và phân phối giống nhau (i.i.d.), phương trình trên có thể được viết lại dưới dạng tổng log-likelihood:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log P(y^{(i)} \mid \mathbf{x}^{(i)}; \theta). \tag{5.70}$$

Ví dụ: Hồi quy tuyến tính dưới góc nhìn của Maximum Likelihood

Hồi quy tuyến tính có thể được hiểu như một phương pháp ước lượng hợp lý cực đại. Trước đây, ta xem hồi quy tuyến tính như một thuật toán học ánh xạ từ đầu vào $\mathbf{x}$ sang đầu ra dự đoán $\hat{y}$ bằng cách tối thiểu hóa sai số bình phương trung bình. Giờ đây, ta tiếp cận nó từ góc nhìn ước lượng hợp lý cực đại, tức là tìm mô hình sinh ra xác suất có điều kiện $p(y \mid x)$.

Giả sử dữ liệu huấn luyện vô cùng lớn, ta có thể thấy nhiều mẫu dữ liệu có cùng đầu vào $\mathbf{x}$ nhưng giá trị đầu ra y khác nhau. Mục tiêu của thuật toán học là khớp xác suất có điều kiện $p(y \mid x)$ với tất cả các giá trị y có thể có tương ứng với x.

Để thu được thuật toán hồi quy tuyến tính đã biết, ta giả sử $p(y \mid x) = \mathcal{N}(\hat{y}; g(\mathbf{x}, \mathbf{w}), \sigma^2)$, trong đó $\hat{y}$ là dự đoán của mô hình và σ^2 là phương sai cố định.

Dưới giả định dữ liệu là i.i.d., log-likelihood có điều kiện được tính bởi:

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid x^{(i)}; \theta) \tag{5.71}$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{|\hat{y}^{(i)} - y^{(i)}|^2}{2\sigma^2}. \tag{5.72}$$

Do đó, tối đa hóa log-likelihood theo $\mathbf{w}$ tương đương với tối thiểu hóa hàm mất MSE:

$$\text{MSE}_{\text{train}} = \frac{1}{m} \sum_{i=1}^{m} |\hat{y}^{(i)} - y^{(i)}|^2. \tag{5.73}$$

Điều này chứng minh rằng sử dụng MSE trong hồi quy tuyến tính là hợp lý từ quan điểm của ước lượng hợp lý cực đại.

5.6.2 Tính chất của ước lượng hợp lý cực đại

Lợi ích chính của ước lượng hợp lý cực đại là có thể được chứng minh là bộ ước lượng tốt nhất về mặt tiệm cận, khi số lượng mẫu $m \to \infty$, xét tốc độ hội tụ khi m tăng lên.

Một trong những đặc điểm quan trọng của ước lượng hợp lý cực đại là tính nhất quán. Nghĩa là, khi số lượng mẫu huấn luyện m tiến tới vô hạn, ước lượng θ_{ML} sẽ hội tụ về giá trị thực của tham số. Điều kiện cần để đảm bảo tính nhất quán bao gồm:

- Phân phối thực tế p_{data} phải nằm trong mô hình xác suất $p_{\text{model}}(\cdot;\theta)$. Nếu không, ước lượng hợp lý cực đại có thể không tìm ra phân phối chính xác.
- Phân phối thực tế p_{data} phải tương ứng với một giá trị duy nhất của θ. Nếu không, phương pháp này có thể tìm ra p_{data} đúng nhưng không thể xác định giá trị nào của θ đã sinh ra dữ liệu.

Bên cạnh tính nhất quán, ước lượng hợp lý cực đại còn có tính hiệu quả thống kê (statistical efficiency), tức là nó có thể đạt lỗi khái quát hóa thấp hơn so với các bộ ước lượng khác với cùng một số lượng mẫu m. Trong trường hợp tham số hóa (chẳng hạn như hồi quy tuyến tính), sai số bình phương trung bình giữa giá trị ước lượng và tham số thực giảm khi m tăng.

Theo định lý Cramér-Rao, không có bộ ước lượng nhất quán nào có phương sai nhỏ hơn ước lượng hợp lý cực đại.

Tổng hợp lại, do có cả tính nhất quán và hiệu quả, ước lượng hợp lý cực đại là một phương pháp được ưa chuộng trong học máy. Khi số lượng mẫu không đủ lớn để đảm bảo tính hội tụ, các chiến lược chính quy hóa (regularization) thường được sử dụng để đạt được sự đánh đổi tốt giữa độ chệch và phương sai.

5.7 Thống kê Bayes

Cho đến nay, chúng ta đã thảo luận về thống kê tần suất (frequentist statistics) và các phương pháp ước lượng giá trị duy nhất của θ, sau đó sử dụng giá trị này để đưa ra dự đoán. Một cách tiếp cận khác là xem xét tất cả các giá trị có thể của θ khi đưa ra dự đoán. Cách tiếp cận này thuộc về thống kê Bayes (Bayesian statistics).

Như đã đề cập trong mục trước, quan điểm của thống kê tần suất là giá trị thực của tham số θ là cố định nhưng không biết trước, trong khi ước lượng điểm $\hat{\theta}$ được coi là một biến ngẫu nhiên do nó phụ thuộc vào tập dữ liệu (mà tập dữ liệu này được xem là ngẫu nhiên).

Quan điểm của Bayes đối với thống kê thì hoàn toàn khác. Bayes sử dụng xác suất để phản ánh mức độ chắc chắn trong trạng thái tri thức. Tập dữ liệu được quan sát trực tiếp, vì vậy nó không phải là ngẫu nhiên. Mặt khác, tham số thực sự θ là không biết hoặc có sự bất định, do đó nó được xem như một biến ngẫu nhiên.

Trước khi quan sát dữ liệu, chúng ta biểu diễn tri thức của mình về θ thông qua phân phối xác suất tiên nghiệm (prior probability distribution), ký hiệu là $p(\theta)$. Trong nhiều trường hợp, người làm học máy chọn một phân phối tiên nghiệm khá rộng (tức là có entropy cao) để phản ánh mức độ không chắc chắn cao về giá trị của θ trước khi quan sát bất kỳ dữ liệu nào. Ví dụ, ta có thể giả định a priori rằng θ nằm trong một phạm vi hữu hạn hoặc một thể tích hữu hạn, với phân phối đồng đều. Nhiều phân phối tiên nghiệm được thiết kế để phản ánh sự ưu tiên đối với các nghiệm *đơn giản hơn* (ví dụ: hệ số nhỏ hơn, hoặc một hàm gần như hằng số).

Bây giờ, giả sử chúng ta có một tập dữ liệu mẫu $\{x^{(1)}, \ldots, x^{(m)}\}$. Chúng ta có thể cập nhật niềm tin về θ bằng cách kết hợp xác suất của dữ liệu $p(x^{(1)}, \ldots, x^{(m)}|\theta)$ với phân phối tiên nghiệm thông qua quy tắc Bayes:

$$p(\theta|x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)}|\theta)p(\theta)}{p(x^{(1)}, \ldots, x^{(m)})}. \tag{5.74}$$

Trong nhiều bài toán học máy, phương pháp ước lượng Bayes thường được sử dụng khi chúng ta có một phân bố tiên nghiệm (prior) thể hiện sự không chắc chắn ban đầu về tham số. Thông thường, phân bố tiên nghiệm này có thể là phân bố đồng nhất hoặc phân bố Gauss với entropy cao. Khi có thêm dữ liệu quan sát, phân bố hậu nghiệm (posterior) sẽ giảm entropy và tập trung quanh một số giá trị tham số có xác suất cao hơn.

So với phương pháp tối đa hóa hợp lý cực đại (Maximum Likelihood Estimation - MLE), ước lượng Bayes có hai điểm khác biệt quan trọng:

- Trong khi MLE chỉ sử dụng một giá trị điểm của tham số θ để thực hiện dự đoán, thì phương pháp Bayes sử dụng toàn bộ phân bố xác suất của θ.
- Thay vì chọn ra một giá trị tối ưu duy nhất, phương pháp Bayes lấy trung bình xác suất của tất cả các giá trị θ có thể có.

Chẳng hạn, giả sử ta đã quan sát được m mẫu dữ liệu $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$. Khi đó, phân bố dự đoán cho mẫu dữ liệu tiếp theo $x^{(m+1)}$ được tính theo công thức:

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \theta)p(\theta \mid x^{(1)}, \ldots, x^{(m)})d\theta. \tag{5.75}$$

Công thức này cho thấy rằng mỗi giá trị của θ có xác suất khác 0 sẽ đóng góp vào việc dự đoán giá trị tiếp theo, với trọng số được quyết định bởi phân bố hậu nghiệm $p(\theta \mid x^{(1)}, \ldots, x^{(m)})$. Nếu sau khi quan sát m mẫu mà ta vẫn chưa chắc chắn về giá trị chính xác của θ, thì sự không chắc chắn này sẽ được phản ánh trực tiếp vào dự đoán của ta đối với dữ liệu mới.

Trong mục 5.4, chúng ta đã thảo luận về cách tiếp cận của phương pháp tần suất (Frequentist) trong việc xử lý tính bất định của một ước lượng điểm cho tham số θ bằng

cách đánh giá phương sai của nó. Phương sai của ước lượng phản ánh mức độ dao động của ước tính khi thay đổi tập dữ liệu quan sát.

Phương pháp Bayesian đưa ra một cách tiếp cận khác để giải quyết vấn đề này bằng cách tích phân qua toàn bộ phân bố xác suất, giúp giảm thiểu hiện tượng quá khớp (overfitting). Việc này chỉ đơn giản là áp dụng các quy tắc của xác suất, làm cho phương pháp Bayesian dễ dàng được biện minh về mặt lý thuyết. Trong khi đó, phương pháp tần suất lại dựa vào một quyết định mang tính chủ quan khi chỉ sử dụng một ước lượng điểm duy nhất để tổng hợp toàn bộ thông tin trong tập dữ liệu.

Một khác biệt quan trọng giữa phương pháp Bayesian và phương pháp hợp lý cực đại (Maximum Likelihood) nằm ở việc phương pháp Bayesian sử dụng phân bố tiên nghiệm (prior distribution).

Phân bố tiên nghiệm ảnh hưởng đến kết quả bằng cách điều chỉnh phân bố xác suất theo hướng ưu tiên những giá trị tham số có khả năng cao hơn trước khi quan sát dữ liệu. Trong thực tế, tiên nghiệm thường thể hiện sự ưu tiên đối với những mô hình đơn giản và trơn tru hơn. Điều này khiến một số người chỉ trích rằng phương pháp Bayesian bị ảnh hưởng bởi tính chủ quan của con người khi lựa chọn tiên nghiệm.

Phương pháp Bayesian có khả năng tổng quát hóa tốt hơn trong trường hợp dữ liệu huấn luyện hạn chế. Tuy nhiên, nhược điểm lớn nhất của nó là chi phí tính toán rất cao khi kích thước của tập dữ liệu huấn luyện tăng lên đáng kể.

Ví dụ: Hồi quy tuyến tính Bayesian

Ở đây, chúng ta xem xét phương pháp ước lượng Bayesian để học các tham số trong mô hình hồi quy tuyến tính. Trong hồi quy tuyến tính, ta học một ánh xạ tuyến tính từ một vector đầu vào $\mathbf{x} \in \mathbb{R}^n$ để dự đoán giá trị của một đại lượng vô hướng $y \in \mathbb{R}$. Dự đoán được tham số hóa bởi vector trọng số $\mathbf{w} \in \mathbb{R}^n$:

$$\hat{y} = \mathbf{w}^T \mathbf{x}. \tag{5.76}$$

Với tập dữ liệu huấn luyện gồm m mẫu $(\mathbf{X}^{\text{train}}, \mathbf{y}^{\text{train}})$, ta có thể biểu diễn dự đoán của y trên toàn bộ tập huấn luyện dưới dạng:

$$\hat{\mathbf{y}}^{\text{train}} = \mathbf{X}^{\text{train}} \mathbf{w}. \tag{5.77}$$

Biểu diễn dưới dạng một phân phối Gaussian có điều kiện trên $\mathbf{y}^{\text{train}}$, ta có:

$$p(\mathbf{y}^{\text{train}} | \mathbf{X}^{\text{train}}, \mathbf{w}) = \mathcal{N}(\mathbf{y}^{\text{train}}; \mathbf{X}^{\text{train}} \mathbf{w}, \mathbf{I}) \tag{5.78}$$

$$\propto \exp\left(-\frac{1}{2} (\mathbf{y}^{\text{train}} - \mathbf{X}^{\text{train}} \mathbf{w})^T (\mathbf{y}^{\text{train}} - \mathbf{X}^{\text{train}} \mathbf{w}) \right), \tag{5.79}$$

trong đó ta sử dụng công thức lỗi bình phương trung bình (MSE) tiêu chuẩn với giả định rằng phương sai Gaussian của y bằng 1. Để đơn giản ký hiệu, ta sẽ viết $(\mathbf{X}^{\text{train}}, \mathbf{y}^{\text{train}})$ thành $(\mathbf{X}, \mathbf{y})$.

Để xác định phân phối hậu nghiệm của vector tham số mô hình $\mathbf{w}$, trước tiên ta cần xác định phân phối tiên nghiệm. Phân phối tiên nghiệm thể hiện niềm tin ban đầu của chúng ta về giá trị của các tham số này.

Thông thường, chúng ta giả định một phân phối rộng, phản ánh mức độ không chắc chắn cao về tham số $\boldsymbol{\theta}$. Đối với các tham số có giá trị thực, một giả định phổ biến là sử dụng phân phối Gauss:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \boldsymbol{\mu}_0, \Lambda_0) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right), \tag{5.80}$$

trong đó $\boldsymbol{\mu}_0$ là vector trung bình tiên nghiệm và Λ_0 là ma trận hiệp phương sai của phân phối tiên nghiệm.[1]

Với phân phối tiên nghiệm đã xác định, ta có thể suy ra phân phối hậu nghiệm của các tham số mô hình bằng công thức Bayes:

$$p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) p(\mathbf{w}). \tag{5.81}$$

Nếu giả định rằng sai số trong mô hình tuân theo phân phối Gauss, hàm hợp lý có dạng:

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) \propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})\right). \tag{5.82}$$

Kết hợp với phân phối tiên nghiệm, ta thu được phân phối hậu nghiệm:

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})\right) \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right). \tag{5.83}$$

$$\propto \exp\left(-\frac{1}{2}\left(-2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \Lambda_0^{-1}\mathbf{w} - 2\boldsymbol{\mu}_0^\top \Lambda_0^{-1}\mathbf{w}\right)\right). \tag{5.84}$$

Chúng ta định nghĩa các biến quan trọng trong phương pháp hồi quy Bayesian như sau:

$$\boldsymbol{\Lambda}_m = \left(\mathbf{X}^T\mathbf{X} + \boldsymbol{\Lambda}_0^{-1}\right)^{-1}, \quad \boldsymbol{\mu}_m = \boldsymbol{\Lambda}_m \left(\mathbf{X}^T\mathbf{y} + \boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0\right). \tag{5.85}$$

[1] Trừ khi có lý do cụ thể để sử dụng một cấu trúc hiệp phương sai đặc biệt, thông thường chúng ta giả định ma trận hiệp phương sai là đường chéo: $\Lambda_0 = \text{diag}(\lambda_0)$.

Sử dụng các biến này, ta có thể viết lại phân phối hậu nghiệm dưới dạng phân phối Gaussian:

$$p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}) \propto \exp\left(-\frac{1}{2}(\mathbf{w}-\boldsymbol{\mu}_m)^T\boldsymbol{\Lambda}_m^{-1}(\mathbf{w}-\boldsymbol{\mu}_m) + \frac{1}{2}\boldsymbol{\mu}_m^T\boldsymbol{\Lambda}_m^{-1}\boldsymbol{\mu}_m\right) \tag{5.86}$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{w}-\boldsymbol{\mu}_m)^T\boldsymbol{\Lambda}_m^{-1}(\mathbf{w}-\boldsymbol{\mu}_m)\right). \tag{5.87}$$

Tất cả các hằng số không phụ thuộc vào vector tham số $\mathbf{w}$ đã bị lược bỏ, vì chúng không ảnh hưởng đến hình dạng của phân phối hậu nghiệm. Để đảm bảo phân phối này hợp lệ, ta cần chuẩn hóa sao cho tổng xác suất bằng 1.

Việc phân tích phân phối hậu nghiệm giúp ta hiểu rõ hơn về ảnh hưởng của suy luận Bayesian.

- Trong hầu hết các tình huống, ta thường đặt $\boldsymbol{\mu}_0 = 0$.
- Nếu ma trận hiệp phương sai ban đầu $\boldsymbol{\Lambda}_0 = \frac{1}{\alpha}\mathbf{I}$, thì giá trị kỳ vọng $\boldsymbol{\mu}_m$ của phân phối hậu nghiệm chính là nghiệm của hồi quy tuyến tính với một ràng buộc suy giảm trọng số có dạng $\alpha\mathbf{w}^T\mathbf{w}$.
- Điểm khác biệt quan trọng là nếu $\alpha = 0$, Bayesian regression không xác định vì nó tương đương với việc có một phân phối tiên nghiệm có phương sai vô hạn.
- Quan trọng hơn, ước lượng Bayesian không chỉ đưa ra một giá trị trung bình $\boldsymbol{\mu}_m$ của $\mathbf{w}$, mà còn cung cấp ma trận hiệp phương sai $\boldsymbol{\Lambda}_m$, thể hiện mức độ không chắc chắn về các giá trị của $\mathbf{w}$.

Điều này cho phép chúng ta không chỉ dự đoán giá trị của trọng số mô hình mà còn hiểu được độ tin cậy của những dự đoán đó, một lợi thế quan trọng của suy luận Bayesian so với các phương pháp ước lượng tần suất (frequentist estimation).

5.7.1 Ước lượng hậu nghiệm cực đại (MAP)

Mặc dù cách tiếp cận nguyên tắc nhất trong dự đoán là sử dụng toàn bộ phân phối hậu nghiệm Bayesian trên tham số θ, nhưng đôi khi ta vẫn muốn có một ước lượng điểm duy nhất. Một lý do phổ biến để cần một ước lượng điểm là vì hầu hết các phép toán liên quan đến phân phối hậu nghiệm Bayesian trong các mô hình phức tạp đều không khả thi về mặt tính toán. Do đó, một ước lượng điểm cung cấp một xấp xỉ có thể tính toán được.

Thay vì chỉ quay lại ước lượng hợp lý cực đại (MLE), chúng ta có thể tận dụng lợi thế của phương pháp Bayesian bằng cách cho phép phân phối tiên nghiệm ảnh hưởng đến lựa chọn ước lượng điểm. Một cách hợp lý để làm điều này là chọn ước lượng hậu nghiệm cực đại (Maximum a Posteriori - MAP).

Ước lượng MAP chọn điểm có xác suất hậu nghiệm cực đại (hoặc mật độ xác suất cực đại trong trường hợp θ là một biến liên tục):

$$\theta_{MAP} = \arg\max_{\theta} p(\theta \mid \mathbf{x}) = \arg\max_{\theta} \log p(\mathbf{x} \mid \theta) + \log p(\theta). \tag{5.88}$$

Ước lượng MAP có thể được coi là sự kết hợp giữa phương pháp tối đa hóa hợp lý cực đại (MLE) và phân phối tiên nghiệm trong Bayesian:

- Nếu phân phối tiên nghiệm $p(\theta)$ là phân phối đều (không có thông tin tiên nghiệm), thì ước lượng MAP trùng với MLE.
- Nếu phân phối tiên nghiệm có thông tin, MAP kết hợp cả thông tin từ dữ liệu và tiên nghiệm để đưa ra một ước lượng hợp lý hơn.
- MAP giúp tránh hiện tượng overfitting bằng cách thêm một ràng buộc vào ước lượng, đặc biệt hữu ích trong trường hợp dữ liệu ít hoặc nhiễu.

Điều này làm cho MAP trở thành một phương pháp mạnh mẽ trong các mô hình học máy Bayesian.

Ước lượng MAP có thể được coi là sự kết hợp giữa phương pháp tối đa hóa hợp lý cực đại (MLE) và phân phối tiên nghiệm trong Bayesian:

- Nếu phân phối tiên nghiệm $p(\theta)$ là phân phối đều (không có thông tin tiên nghiệm), thì ước lượng MAP trùng với MLE.
- Nếu phân phối tiên nghiệm có thông tin, MAP kết hợp cả thông tin từ dữ liệu và tiên nghiệm để đưa ra một ước lượng hợp lý hơn.
- MAP giúp tránh hiện tượng overfitting bằng cách thêm một ràng buộc vào ước lượng, đặc biệt hữu ích trong trường hợp dữ liệu ít hoặc nhiễu.

Xét một mô hình hồi quy tuyến tính với tiên nghiệm Gaussian trên trọng số $\mathbf{w}$. Nếu tiên nghiệm này tuân theo phân phối $\mathcal{N}(\mathbf{w}; \mathbf{0}, \frac{1}{\lambda} I)$, thì log của phân phối tiên nghiệm có dạng tỷ lệ với thành phần $\lambda \mathbf{w}^T \mathbf{w}$, tức là một dạng regularization weight decay quen thuộc. Khi đó, suy luận Bayesian với tiên nghiệm Gaussian trên trọng số tương đương với áp dụng weight decay.

Giống như suy luận Bayesian đầy đủ, ước lượng MAP tận dụng thông tin từ tiên nghiệm mà không thể tìm thấy trong dữ liệu huấn luyện. Điều này giúp giảm phương sai của ước lượng MAP so với MLE, nhưng đổi lại sẽ làm tăng độ chệch (bias).

Nhiều chiến lược regularization, chẳng hạn như học có regularization bằng weight decay, có thể được hiểu là một xấp xỉ của MAP đối với Bayesian inference. Điều này đặc biệt đúng khi regularization tương ứng với một thành phần bổ sung trong hàm mục tiêu có dạng $\log p(\theta)$. Tuy nhiên, không phải tất cả regularization đều tương đương với MAP. Một số dạng regularization không thể được biểu diễn dưới dạng log của một phân phối xác suất.

Suy luận Bayesian theo phương pháp MAP cung cấp một cách tiếp cận rõ ràng để thiết kế các điều khoản regularization phức tạp nhưng dễ giải thích. Ví dụ, thay vì sử dụng tiên nghiệm Gaussian đơn lẻ, ta có thể sử dụng một mixture của nhiều Gaussian để tạo ra các regularization phong phú hơn (Nowlan and Hinton, 1992).

5.8 Thuật toán học có giám sát

Như đã đề cập trong mục trước, thuật toán học có giám sát là các thuật toán học máy có khả năng liên kết một đầu vào với một đầu ra dựa trên một tập dữ liệu huấn luyện chứa các cặp ví dụ (input $\mathbf{x}$ và output $\mathbf{y}$).

Trong nhiều trường hợp, các đầu ra $\mathbf{y}$ có thể khó thu thập tự động và phải được cung cấp bởi con người. Điều này dẫn đến việc cần có một "giám sát viên" (supervisor) để gán nhãn cho dữ liệu huấn luyện. Tuy nhiên, thuật ngữ này vẫn được sử dụng ngay cả khi các nhãn của tập dữ liệu được thu thập một cách tự động.

5.8.1 Học có giám sát theo xác suất

Hầu hết các thuật toán học có giám sát trong sách này đều dựa trên việc ước lượng một phân phối xác suất $p(y|\mathbf{x})$. Chúng ta có thể thực hiện điều này bằng cách sử dụng ước lượng hợp lý cực đại (Maximum Likelihood Estimation - MLE) để tìm tham số tối ưu $\boldsymbol{\theta}$ cho một họ phân phối tham số $p(y|\mathbf{x};\boldsymbol{\theta})$.

Chúng ta đã thấy rằng hồi quy tuyến tính (Linear Regression) tương ứng với mô hình phân phối:

$$p(y|\mathbf{x};\boldsymbol{\theta}) = \mathcal{N}(y;\boldsymbol{\theta}^T\mathbf{x}, I). \tag{5.80}$$

Chúng ta có thể tổng quát hóa hồi quy tuyến tính sang bài toán phân loại bằng cách sử dụng một họ phân phối xác suất khác. Nếu bài toán có hai lớp (class 0 và class 1), ta chỉ cần xác định xác suất của một trong hai lớp. Vì tổng xác suất của hai lớp phải bằng 1, nên xác suất của class 1 suy ra xác suất của class 0.

Phân phối Gauss trong hồi quy tuyến tính được tham số hóa theo giá trị trung bình. Với biến nhị phân, giá trị trung bình phải nằm trong khoảng (0,1). Một cách để giải quyết vấn đề này là sử dụng hàm sigmoid logistic để chuẩn hóa đầu ra của mô hình tuyến tính thành khoảng (0,1) và diễn giải kết quả này như một xác suất:

$$p(y = 1|\mathbf{x};\boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^T\mathbf{x}). \tag{5.81}$$

Phương pháp này được gọi là hồi quy logistic (logistic regression), mặc dù nó được sử dụng cho phân loại thay vì hồi quy.

Trong hồi quy tuyến tính, chúng ta có thể tìm trọng số tối ưu bằng cách giải phương trình chuẩn. Trong hồi quy logistic, bài toán khó hơn vì không có nghiệm đóng (closed-form solution) cho trọng số tối ưu. Thay vào đó, ta cần tìm nghiệm bằng cách cực đại hóa hàm log-likelihood, thường thực hiện thông qua thuật toán tối ưu hóa như gradient descent.

Chiến lược này có thể áp dụng cho hầu hết các bài toán học có giám sát bằng cách sử dụng một mô hình tham số hóa cho phân phối xác suất có điều kiện phù hợp với kiểu dữ liệu đầu vào và đầu ra.

5.9 Máy vector hỗ trợ (Support Vector Machines)

Máy vector hỗ trợ (SVM) là một trong những phương pháp học có giám sát quan trọng nhất trong học máy [1], [2]. Mô hình này tương tự như hồi quy logistic ở chỗ nó sử dụng một hàm tuyến tính $\mathbf{w}^T\mathbf{x} + b$ để đưa ra quyết định phân loại. Tuy nhiên, khác với hồi quy logistic, SVM không cung cấp xác suất dự đoán mà chỉ trả về một nhãn lớp. Nếu $\mathbf{w}^T\mathbf{x} + b$ dương, mẫu $\mathbf{x}$ được phân vào lớp dương. Ngược lại, nếu $\mathbf{w}^T\mathbf{x} + b$ âm, mẫu $\mathbf{x}$ được phân vào lớp âm.

Một cải tiến quan trọng của SVM là thủ thuật kernel. Thủ thuật này dựa trên quan sát rằng nhiều thuật toán học máy có thể được biểu diễn chỉ bằng tích vô hướng của các mẫu huấn luyện. Cụ thể, có thể chứng minh rằng hàm tuyến tính của SVM có thể được viết lại dưới dạng:

$$\mathbf{w}^T\mathbf{x} + b = b + \sum_{i=1}^{m} \alpha_i \mathbf{x}^T\mathbf{x}^{(i)} \tag{5.89}$$

trong đó:

- $\mathbf{x}^{(i)}$ là mẫu huấn luyện,
- α là vector hệ số.

Việc viết lại này cho phép ta thay thế $\mathbf{x}$ bằng đầu ra của một hàm ánh xạ đặc trưng $\phi(\mathbf{x})$. Khi đó, tích vô hướng được thay thế bởi một hàm:

$$k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}^{(i)}) \tag{5.90}$$

Hàm k được gọi là hàm kernel. Toán tử "$\cdot$"đại diện cho một tích vô hướng tương tự như tích vô hướng trong không gian vector.

Trong một số không gian đặc trưng, tích vô hướng có thể không thực sự là một tích vector thông thường mà có thể là một tích vô hướng trong không gian vô hạn chiều, được định nghĩa thông qua tích phân thay vì tổng. Việc nghiên cứu đầy đủ về các loại tích vô hướng này nằm ngoài phạm vi của chương này.

Sau khi thay thế tích vô hướng bằng đánh giá kernel, ta có thể biểu diễn hàm dự đoán như sau:

$$f(\mathbf{x}) = b + \sum_{i} \alpha_i k(\mathbf{x}, \mathbf{x}^{(i)}) \tag{5.91}$$

Hàm này là phi tuyến đối với $\mathbf{x}$, nhưng mối quan hệ giữa $\phi(\mathbf{x})$ và $f(\mathbf{x})$ lại là tuyến tính. Tương tự, quan hệ giữa α và $f(\mathbf{x})$ cũng là tuyến tính. Do đó, một hàm dựa trên kernel có thể xem như một bước tiền xử lý dữ liệu bằng cách ánh xạ $\phi(\mathbf{x})$ cho mọi đầu vào, sau đó học một mô hình tuyến tính trong không gian đã biến đổi.

Thủ thuật kernel đặc biệt mạnh mẽ vì hai lý do. Thứ nhất, nó giúp chúng ta học được các mô hình phi tuyến tính theo $\mathbf{x}$ bằng các kỹ thuật tối ưu lồi, đảm bảo hội tụ nhanh. Điều này có thể thực hiện được vì ta coi ϕ là cố định và chỉ tối ưu α. Nói cách khác, thuật

toán tối ưu có thể xem hàm quyết định như một mô hình tuyến tính trong một không gian khác. Thứ hai, hàm kernel k cho phép triển khai tính toán hiệu quả hơn so với cách tiếp cận ngây thơ là xây dựng hai vector $\phi(\mathbf{x})$ rồi tính tích vô hướng trực tiếp.

Trong một số trường hợp, $\phi(\mathbf{x})$ thậm chí có thể là một không gian vô hạn chiều, dẫn đến chi phí tính toán rất lớn nếu tiếp cận theo cách thông thường. Thay vào đó, nhiều kernel phi tuyến có thể được tính toán hiệu quả ngay cả khi $\phi(\mathbf{x})$ không thể biểu diễn tường minh. Ví dụ, một ánh xạ đặc trưng trong không gian vô hạn có thể được xây dựng trên các số nguyên không âm x, với đầu ra là một vector chứa x và vô hạn số không. Khi đó, một hàm kernel có thể được định nghĩa như:

$$k(x, x^{(i)}) = \min(x, x^{(i)}) \tag{5.92}$$

Hàm này tương đương với tích vô hướng trong một không gian vô hạn chiều.

Hàm kernel được sử dụng phổ biến nhất là kernel Gauss, còn được gọi là hàm nền tảng bán kính (RBF):

$$k(u, v) = \mathcal{N}(u - v; 0, \sigma^2 I), \tag{5.93}$$

trong đó $\mathcal{N}(x; \mu, \Sigma)$ là phân phối chuẩn. Hàm này đo khoảng cách giữa các điểm trong không gian biến đổi, giúp phát hiện các mẫu phức tạp hơn so với không gian ban đầu.

Một cách hiểu khác về kernel Gauss là nó hoạt động như một phương pháp so khớp mẫu. Khi một điểm kiểm tra x' gần với một mẫu huấn luyện x, hàm kernel sẽ có giá trị lớn, cho thấy rằng x' rất giống x. Do đó, trọng số lớn hơn sẽ được gán cho nhãn tương ứng với mẫu huấn luyện gần nhất.

Mặc dù hiệu quả, nhưng một hạn chế lớn của SVM với kernel là chi phí tính toán cao. Khi số lượng mẫu huấn luyện lớn, việc tính toán kernel với từng mẫu trở nên tốn kém. Giải pháp cho vấn đề này là tìm một tập hợp nhỏ các mẫu huấn luyện có trọng số α_i khác không, được gọi là vector hỗ trợ.

Mặc dù SVM với kernel rất mạnh, nhưng các mô hình này gặp khó khăn khi tổng quát hóa trên các tập dữ liệu lớn. Điều này dẫn đến sự ra đời của học sâu (Deep Learning), nơi mà mạng nơ-ron có thể học được các đặc trưng phức tạp mà không cần thiết kế kernel thủ công. Hinton et al. (2006) đã chứng minh rằng mạng nơ-ron có thể vượt trội so với SVM với kernel RBF trên bài toán MNIST.

5.10 Các Thuật Toán Học Có Giám Sát Đơn Giản Khác

5.10.1 Hồi Quy Hàng Xóm Gần Nhất

Trong phần này, chúng ta sẽ thảo luận về một thuật toán học có giám sát phi xác suất khác: hồi quy hàng xóm gần nhất (nearest neighbor regression). Tổng quát hơn, phương pháp *k-nearest neighbors* (k-NN) là một họ các kỹ thuật có thể được sử dụng cho cả bài toán phân loại và hồi quy. Là một thuật toán học phi tham số, k-NN không bị giới hạn bởi

số lượng tham số cố định.

Chúng ta thường nghĩ về thuật toán k-NN như một phương pháp không có tham số, mà chỉ thực hiện một ánh xạ đơn giản từ dữ liệu huấn luyện. Thực tế, thuật toán này không có một giai đoạn huấn luyện thực sự. Thay vào đó, khi cần dự đoán đầu ra y cho một đầu vào mới $\mathbf{x}$, thuật toán sẽ tìm k điểm gần nhất với $\mathbf{x}$ trong tập dữ liệu huấn luyện $\mathcal{X}$. Sau đó, giá trị đầu ra y sẽ được tính bằng trung bình của các giá trị y tương ứng trong tập huấn luyện.

Nguyên tắc này có thể áp dụng cho bất kỳ bài toán học có giám sát nào mà ta có thể tính trung bình trên các giá trị y. Trong bài toán phân loại, ta có thể tính trung bình trên các vector mã hóa one-hot $\mathbf{c}$ với $c_y = 1$ và $c_i = 0$ cho các nhãn khác. Khi đó, giá trị trung bình trên các vector one-hot này có thể được diễn giải như một phân phối xác suất trên các lớp.

5.10.2 Đặc Điểm của k-NN

Là một thuật toán học phi tham số, k-NN có thể đạt dung lượng (capacity) rất cao. Ví dụ, giả sử ta có một bài toán phân loại đa lớp và đánh giá hiệu suất bằng hàm mất mát 0-1. Khi số lượng mẫu huấn luyện tiến đến vô hạn, thuật toán 1-NN sẽ hội tụ về một giá trị gấp đôi sai số Bayes. Sai số này chủ yếu xuất phát từ việc chọn một hàng xóm duy nhất bằng cách ngẫu nhiên trong trường hợp có nhiều hàng xóm có khoảng cách bằng nhau.

Khi tập dữ liệu huấn luyện vô hạn, mọi điểm kiểm tra đều sẽ có vô hạn điểm lân cận có khoảng cách bằng 0. Nếu ta cho phép tất cả các điểm lân cận này tham gia vào quá trình dự đoán (thay vì chọn ngẫu nhiên), thuật toán sẽ hội tụ về tỉ lệ lỗi Bayes.

Dung lượng cao của k-NN cho phép nó đạt độ chính xác cao khi có tập dữ liệu huấn luyện lớn. Tuy nhiên, phương pháp này yêu cầu tính toán rất nhiều và có thể khái quát hóa kém khi tập dữ liệu huấn luyện nhỏ.

5.10.3 Hạn Chế của k-NN

Một nhược điểm quan trọng của k-NN là nó không thể học được mức độ quan trọng của từng đặc trưng. Ví dụ, giả sử ta có một bài toán hồi quy với $\mathbf{x} \in \mathbb{R}^{100}$ được lấy từ một phân phối Gauss đẳng hướng, nhưng chỉ có một biến x_1 là có ý nghĩa đối với đầu ra. Nếu đầu ra y chỉ đơn giản là một ánh xạ của x_1, tức là $y = x_1$ trong tất cả các trường hợp, thì hồi quy hàng xóm gần nhất sẽ không thể phát hiện được mối quan hệ này.

Do đó, mặc dù k-NN có thể đạt kết quả tốt khi có dữ liệu lớn, nhưng nó không phải lúc nào cũng là lựa chọn tối ưu do chi phí tính toán cao và khả năng tổng quát hóa kém với dữ liệu nhỏ.

5.10.4 Cây Quyết Định

Một loại thuật toán học khác cũng chia không gian đầu vào thành các vùng và sử dụng tham số riêng biệt cho mỗi vùng là *cây quyết định* (decision tree) [3] và các biến thể của nó. Mỗi nút trong cây quyết định tương ứng với một vùng trong không gian đầu vào, và các nút bên trong sẽ chia vùng này thành các phân vùng nhỏ hơn (thường là bằng cách cắt

dọc theo trục tọa độ). Không gian đầu vào được phân chia thành các vùng không chồng lấn, với mỗi lá của cây tương ứng với một vùng đầu vào nhất định. Các nút lá thường gán cùng một đầu ra cho tất cả các điểm trong vùng của nó.

Cây quyết định thường được huấn luyện bằng các thuật toán chuyên biệt, nằm ngoài phạm vi của cuốn sách này. Nếu cho phép cây phát triển đến kích thước tùy ý, cây quyết định có thể được xem là một thuật toán phi tham số, mặc dù trong thực tế chúng thường được giới hạn kích thước để trở thành mô hình tham số.

Cây quyết định thường sử dụng cách chia theo trục tọa độ và gán giá trị hằng số trong mỗi nút. Điều này có thể gây khó khăn trong việc học một số bài toán đơn giản mà hồi quy logistic có thể giải quyết dễ dàng. Ví dụ, nếu ta có một bài toán phân loại hai lớp, trong đó lớp dương xảy ra khi $x_2 > x_1$, thì biên quyết định không thẳng hàng với trục tọa độ. Cây quyết định sẽ phải sử dụng nhiều nút để xấp xỉ biên quyết định bằng một hàm bậc thang, đi qua lại xung quanh biên quyết định thực sự bằng các bước rời rạc dọc theo trục tọa độ.

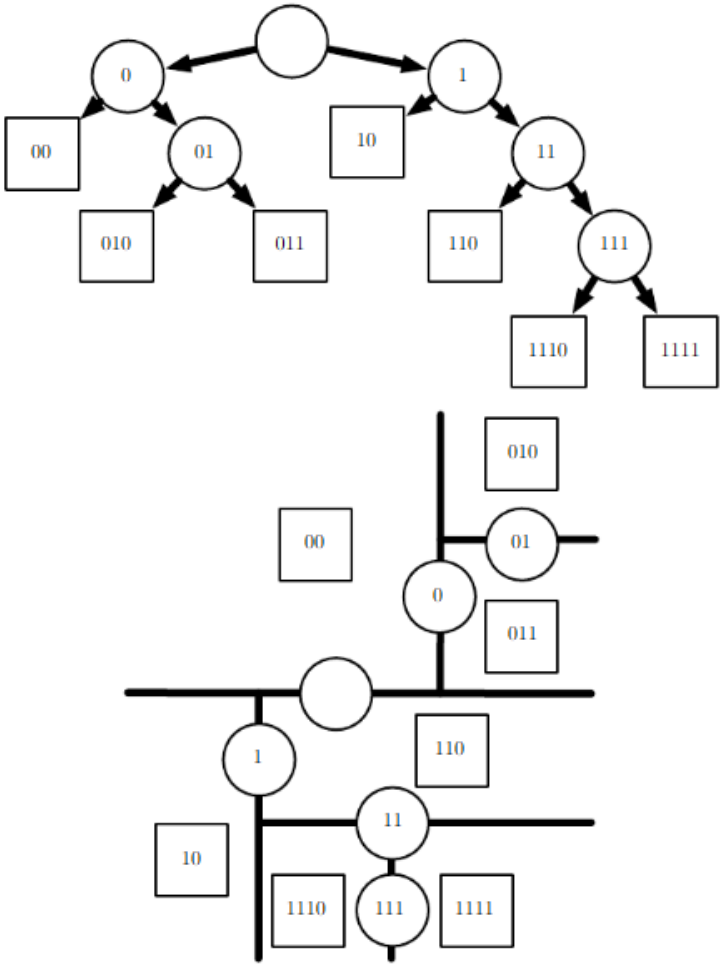
5.10.5 Kết Luận

Như chúng ta đã thấy, cả k-NN và cây quyết định đều có những hạn chế nhất định. Tuy nhiên, chúng vẫn là các thuật toán học hữu ích khi tài nguyên tính toán bị hạn chế. Hơn nữa, việc so sánh sự tương đồng và khác biệt giữa các thuật toán học đơn giản như k-NN hoặc cây quyết định có thể giúp xây dựng trực giác về các thuật toán học phức tạp hơn.

Để tìm hiểu thêm về các thuật toán học có giám sát truyền thống, bạn có thể tham khảo **murphy2012**, [4], [5].

5.11 Thuật Toán Học Không Giám Sát

Hãy nhớ lại từ mục 5.1.3 rằng các thuật toán không giám sát là những thuật toán chỉ dựa vào các *đặc trưng* mà không có tín hiệu giám sát. Sự khác biệt giữa học có giám sát và không giám sát không được xác định một cách rõ ràng và cứng nhắc, vì không có một bài kiểm tra khách quan nào để phân biệt giá trị nào là đặc trưng hay là mục tiêu do con người gán nhãn.



Hình 5.7: Minh họa cách cây quyết định hoạt động. Cây quyết định là một thuật toán học có giám sát phổ biến trong học máy. Mỗi nút trong cây quyết định chịu trách nhiệm phân loại một điểm dữ liệu vào nhánh con bên trái (gán nhãn 0) hoặc nhánh con bên phải (gán nhãn 1). Các nút bên trong được vẽ dưới dạng hình tròn, trong khi các nút lá được biểu diễn bằng hình vuông. Mỗi nút được gán một chuỗi nhị phân để biểu thị vị trí của nó trong cây, được tạo ra bằng cách thêm một bit vào định danh của nút cha (0 = đi về bên trái, 1 = đi về bên phải). Cây quyết định có thể chia không gian thành các vùng khác nhau. Hình ảnh trên minh họa cách cây quyết định phân chia không gian hai chiều $\mathbb{R}^2$. Các nút bên trong cây được đặt dọc theo các đường phân chia không gian dựa trên các thuộc tính dữ liệu. Các nút lá nằm ở trung tâm vùng mà chúng đại diện. Kết quả của cây quyết định là một hàm có dạng hằng từng đoạn (piecewise-constant function), trong đó mỗi đoạn ứng với một nút lá. Do mỗi nút lá yêu cầu ít nhất một ví dụ huấn luyện để xác định, nên cây quyết định không thể học một hàm có số cực đại cục nhiều hơn số lượng ví dụ huấn luyện.

Một cách không chính thức, học không giám sát thường được hiểu là việc trích xuất thông tin từ dữ liệu mà không yêu cầu con người gán nhãn thủ công. Các thuật toán này thường liên quan đến các nhiệm vụ như ước lượng mật độ, lấy mẫu từ một phân phối, khử nhiễu dữ liệu, tìm kiếm đa tạp ẩn chứa trong dữ liệu hoặc phân cụm dữ liệu thành các nhóm liên quan.

Một trong những nhiệm vụ chính của học không giám sát là tìm ra biểu diễn tốt nhất của dữ liệu. "Tốt nhất"có thể được định nghĩa theo nhiều cách khác nhau, nhưng nhìn chung, chúng ta tìm kiếm một biểu diễn giữ được càng nhiều thông tin về dữ liệu x càng tốt, đồng thời tuân theo một số ràng buộc hoặc quy tắc giúp cho biểu diễn trở nên *đơn*

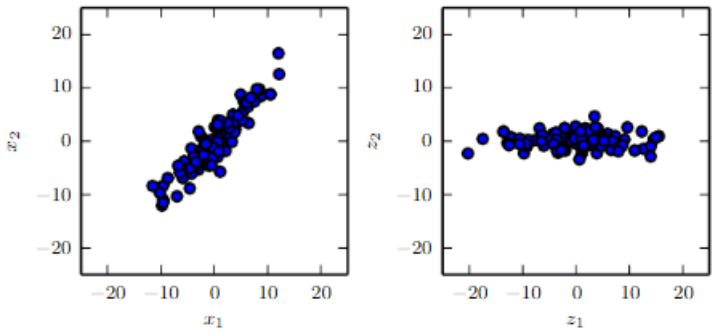
giản hơn hoặc *dễ hiểu hơn* so với dữ liệu gốc.

Có nhiều cách khác nhau để định nghĩa một biểu diễn đơn giản. Ba phương pháp phổ biến nhất bao gồm:

- Biểu diễn chiều thấp: Giảm số chiều của dữ liệu trong khi vẫn giữ lại nhiều thông tin nhất có thể.
- Biểu diễn thưa (Sparse Representation): Mã hóa dữ liệu thành dạng mà hầu hết các phần tử có giá trị bằng không. Điều này giúp biểu diễn trở nên dễ dàng hơn, đồng thời vẫn giữ được cấu trúc quan trọng của dữ liệu.
- Biểu diễn độc lập (Independent Representation): Cố gắng tách biệt các nguồn biến thiên trong dữ liệu sao cho các thành phần của biểu diễn là độc lập về mặt thống kê.

Các phương pháp này không hoàn toàn loại trừ lẫn nhau. Ví dụ, các biểu diễn chiều thấp thường có thể giúp phát hiện và loại bỏ các thành phần dư thừa trong dữ liệu, từ đó giúp tăng hiệu suất của các thuật toán nén dữ liệu.

Khái niệm biểu diễn là một trong những chủ đề trung tâm trong học sâu. Trong phần này, chúng ta sẽ phát triển một số ví dụ đơn giản về các thuật toán học biểu diễn. Các thuật toán này giúp minh họa cách thức hiện thực hóa ba tiêu chí chính đã đề cập ở trên. Hầu hết các phần còn lại của cuốn sách sẽ mở rộng và giới thiệu thêm các thuật toán học biểu diễn khác nhằm phát triển hoặc cải thiện những tiêu chí này.



Hình 5.8: PCA học một phép chiếu tuyến tính sao cho hướng có phương sai lớn nhất được căn chỉnh với các trục của không gian mới. (Trái) Dữ liệu ban đầu bao gồm các mẫu của $\mathbf{x}$. Trong không gian này, phương sai của dữ liệu có thể không thẳng hàng với các trục tọa độ. (Phải) Dữ liệu được biến đổi $\mathbf{z} = \mathbf{x}^T\mathbf{W}$ bằng ma trận trọng số $\mathbf{W}$ của PCA. Sau khi biến đổi, dữ liệu chủ yếu thay đổi theo trục z_1, hướng có phương sai lớn nhất, trong khi phương sai lớn thứ hai nằm dọc theo trục z_2.

5.11.1 Phân Tích Thành Phần Chính

Trong mục 2.12, chúng ta đã thấy rằng thuật toán phân tích thành phần chính (PCA) cung cấp một phương pháp để nén dữ liệu. Chúng ta cũng có thể xem PCA như một thuật toán học không giám sát, học cách biểu diễn dữ liệu. Biểu diễn này dựa trên hai tiêu chí để có một biểu diễn đơn giản đã được mô tả trước đó. PCA học một biểu diễn có số chiều thấp hơn so với đầu vào ban đầu. Nó cũng học một biểu diễn mà các phần tử không có

tương quan tuyến tính với nhau. Đây là một bước đầu tiên để đáp ứng tiêu chí học biểu diễn mà các phần tử là độc lập về mặt thống kê. Để đạt được sự độc lập hoàn toàn, một thuật toán học biểu diễn cũng phải loại bỏ các mối quan hệ phi tuyến tính giữa các biến.

PCA học một phép biến đổi tuyến tính trực giao của dữ liệu để chiếu đầu vào $\mathbf{x}$ sang biểu diễn $\mathbf{z}$, như được minh họa trong hình 5.8. Trong mục 2.12, chúng ta đã thấy rằng có thể học một biểu diễn một chiều tốt nhất tái tạo dữ liệu gốc (theo nghĩa lỗi bình phương trung bình) và biểu diễn này thực sự tương ứng với thành phần chính đầu tiên của dữ liệu. Do đó, chúng ta có thể sử dụng PCA như một phương pháp giảm chiều đơn giản và hiệu quả, giúp bảo toàn nhiều nhất có thể thông tin trong dữ liệu (cũng được đo bằng lỗi tái tạo bình phương nhỏ nhất). Trong phần tiếp theo, chúng ta sẽ nghiên cứu cách biểu diễn PCA làm mất đi sự tương quan của dữ liệu gốc $\mathbf{X}$.

Giả sử chúng ta xem xét ma trận thiết kế $m \times n$ $\mathbf{X}$. Chúng ta giả định rằng dữ liệu có kỳ vọng bằng không, tức là $\mathbb{E}[\mathbf{x}] = \mathbf{0}$. Nếu không phải như vậy, dữ liệu có thể được trung tâm hóa dễ dàng bằng cách trừ đi giá trị trung bình từ tất cả các mẫu trong một bước tiền xử lý.

Ma trận hiệp phương sai mẫu không chệch liên quan đến $\mathbf{X}$ được xác định bởi:

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1}\mathbf{X}^\top\mathbf{X}. \tag{5.94}$$

PCA tìm một biểu diễn (thông qua phép biến đổi tuyến tính) $\mathbf{z} = \mathbf{W}^\top\mathbf{x}$, trong đó $\text{Var}[\mathbf{z}]$ là đường chéo.

Trong mục 2.12, chúng ta đã thấy rằng các thành phần chính của ma trận thiết kế $\mathbf{X}$ được xác định bởi các vectơ riêng của $\mathbf{X}^\top\mathbf{X}$. Từ góc nhìn này, ta có:

$$\mathbf{X}^\top\mathbf{X} = \mathbf{W}\boldsymbol{\Lambda}\mathbf{W}^\top. \tag{5.95}$$

Trong phần này, chúng ta sẽ khai thác một cách tiếp cận khác để tìm các thành phần chính của dữ liệu, thông qua phân rã giá trị kỳ dị (SVD - Singular Value Decomposition). Các thành phần chính có thể được xác định bằng cách sử dụng các vector kỳ dị bên phải của ma trận dữ liệu $\mathbf{X}$. Cụ thể, nếu ta ký hiệu $\mathbf{W}$ là các vector kỳ dị bên phải trong phân rã:

$$\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top, \tag{5.96}$$

thì phương trình giá trị riêng ban đầu có thể được viết lại với $\mathbf{W}$ là hệ cơ sở:

$$\mathbf{X}^\top\mathbf{X} = (\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top)^\top\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top = \mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^\top. \tag{5.97}$$

SVD rất hữu ích để chứng minh rằng PCA tạo ra một ma trận phương sai chéo. Bằng

cách sử dụng phân rã SVD của $\mathbf{X}$, ta có thể biểu diễn phương sai của $\mathbf{X}$ như sau:

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1}\mathbf{X}^\top\mathbf{X} \tag{5.98}$$
$$= \frac{1}{m-1}(\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top)^\top\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top \tag{5.99}$$
$$= \frac{1}{m-1}\mathbf{W}\boldsymbol{\Sigma}^\top\mathbf{U}^\top\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top \tag{5.100}$$
$$= \frac{1}{m-1}\mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^\top, \tag{5.101}$$

trong đó ta sử dụng tính chất $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ do $\mathbf{U}$ là ma trận trực giao. Điều này chứng minh rằng phương sai của $\mathbf{z}$ là chéo:

$$\text{Var}[\mathbf{z}] = \frac{1}{m-1}\mathbf{Z}^\top\mathbf{Z} \tag{5.102}$$
$$= \frac{1}{m-1}\mathbf{W}^\top\mathbf{X}^\top\mathbf{X}\mathbf{W} \tag{5.103}$$
$$= \frac{1}{m-1}\mathbf{W}^\top\mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^\top\mathbf{W} \tag{5.104}$$
$$= \frac{1}{m-1}\boldsymbol{\Sigma}^2, \tag{5.105}$$

trong đó ta sử dụng thêm tính chất $\mathbf{W}^\top\mathbf{W} = \mathbf{I}$ từ định nghĩa của SVD.

Phân tích trên cho thấy khi ta chiếu dữ liệu $\mathbf{x}$ sang $\mathbf{z}$ bằng phép biến đổi tuyến tính $\mathbf{W}$, ma trận hiệp phương sai của dữ liệu mới trở thành ma trận đường chéo (được cho bởi $\boldsymbol{\Sigma}^2$). Điều này có nghĩa là các phần tử của $\mathbf{z}$ là không tương quan với nhau.

Tính chất này của PCA rất quan trọng, vì nó giúp chuyển đổi dữ liệu sang một dạng mà các thành phần không còn bị phụ thuộc lẫn nhau. Điều này có thể hiểu như một cách để "tách rời các nhân tố biến đổi chưa biết" trong dữ liệu. Với PCA, việc tách rời này được thực hiện bằng cách tìm một phép quay của không gian đầu vào (được xác định bởi $\mathbf{W}$), sao cho trục chính của phương sai khớp với hệ cơ sở của không gian mới được biểu diễn bởi $\mathbf{z}$.

Mặc dù tương quan là một dạng quan trọng của sự phụ thuộc giữa các phần tử trong dữ liệu, nhưng chúng ta cũng quan tâm đến việc học các biểu diễn có thể tách biệt các dạng phụ thuộc phức tạp hơn giữa các đặc trưng. Để làm được điều này, chúng ta cần nhiều hơn những gì có thể đạt được chỉ với một phép biến đổi tuyến tính đơn giản.

5.12 k-means Clustering

Một ví dụ khác về thuật toán học biểu diễn đơn giản là phân cụm k-means. Thuật toán k-means chia tập dữ liệu thành k cụm khác nhau, trong đó các điểm dữ liệu gần nhau sẽ được nhóm vào cùng một cụm. Ta có thể xem thuật toán này như một phương pháp tạo mã hóa dạng one-hot với một vector k chiều $\mathbf{h}$ để biểu diễn đầu vào $\mathbf{x}$. Nếu $\mathbf{x}$ thuộc về cụm i, thì $h_i = 1$, và tất cả các phần tử khác của $\mathbf{h}$ sẽ là 0.

Mã hóa one-hot từ phân cụm k-means là một ví dụ của biểu diễn thưa (sparse representation), vì hầu hết các phần tử trong vector đều có giá trị bằng 0. Sau này, chúng ta có thể phát triển các thuật toán học biểu diễn linh hoạt hơn, cho phép nhiều phần tử trong vector mã hóa có thể có giá trị khác 0. Tuy nhiên, mã hóa one-hot là một dạng biểu diễn cực đoan, làm mất đi nhiều lợi ích của biểu diễn phân tán (distributed representation). Dù vậy, nó vẫn có một số ưu điểm về mặt thống kê, chẳng hạn như giúp nhóm các điểm dữ liệu trong cùng một cụm có sự tương đồng với nhau, đồng thời mang lại lợi ích về mặt tính toán khi biểu diễn chỉ cần sử dụng một số nguyên đơn giản.

Thuật toán k-means hoạt động bằng cách khởi tạo k tâm cụm (centroids) khác nhau $\{\mu^{(1)}, \ldots, \mu^{(k)}\}$ với các giá trị ngẫu nhiên, sau đó lặp lại hai bước luân phiên cho đến khi hội tụ:

1. Gán mỗi điểm dữ liệu vào cụm gần nhất i, trong đó i là chỉ số của tâm cụm gần nhất $\mu^{(i)}$.

2. Cập nhật mỗi tâm cụm $\mu^{(i)}$ thành giá trị trung bình của tất cả các điểm dữ liệu $\mathbf{x}^{(j)}$ thuộc về cụm đó.

Một trong những thách thức của phân cụm là vấn đề xác định mức độ phù hợp của kết quả phân cụm với dữ liệu thực tế. Không có một tiêu chí duy nhất nào để đo lường điều này một cách khách quan. Một cách phổ biến là đo lường chất lượng phân cụm dựa trên khoảng cách Euclid giữa tâm cụm và các điểm trong cụm đó. Cách tiếp cận này giúp đánh giá mức độ khớp của dữ liệu với phân cụm, nhưng không thể xác định chính xác mức độ phản ánh các thuộc tính thực tế của dữ liệu.

Ví dụ, giả sử ta chạy thuật toán phân cụm trên một tập dữ liệu chứa ảnh của xe tải màu xám, xe tải màu đỏ, xe hơi màu xám và xe hơi màu đỏ. Nếu yêu cầu thuật toán phân cụm thành hai nhóm, một cách phân cụm có thể là tách ra thành nhóm "xe hơi"và nhóm "xe tải". Tuy nhiên, một cách phân cụm khác có thể chia thành nhóm "xe màu xám"và nhóm "xe màu đỏ". Nếu ta cho phép thuật toán tự xác định số cụm tối ưu, nó có thể phân cụm thành bốn nhóm: "xe hơi đỏ", "xe hơi xám", "xe tải đỏ"và "xe tải xám". Cách phân cụm này phản ánh đồng thời cả hai thuộc tính (màu sắc và loại phương tiện), nhưng không thể hiện được mức độ tương đồng giữa các đối tượng.

Chẳng hạn, xe hơi màu đỏ và xe tải màu đỏ thuộc về các cụm khác nhau, giống như cách mà xe tải màu xám và xe tải màu đỏ cũng thuộc về các cụm khác nhau. Tuy nhiên, từ kết quả phân cụm, ta không thể kết luận rằng "xe hơi màu đỏ gần gũi với xe hơi màu xám hơn là với xe tải màu đỏ". Kết quả chỉ đơn thuần thể hiện sự khác biệt giữa các nhóm mà không cung cấp thông tin về mối quan hệ giữa chúng.

Những vấn đề này minh họa lý do tại sao ta có thể muốn sử dụng biểu diễn phân tán (distributed representation) thay vì biểu diễn one-hot. Biểu diễn phân tán có thể gán nhiều thuộc tính cho mỗi đối tượng—chẳng hạn, một thuộc tính đại diện cho màu sắc và một thuộc tính đại diện cho loại xe. Điều này giúp thuật toán có thể học được cách biểu diễn tối ưu để phản ánh các thuộc tính quan trọng của dữ liệu. Tuy nhiên, một thách thức đặt ra

là làm sao để thuật toán có thể biết được những thuộc tính nào là quan trọng (ví dụ: màu sắc hay loại xe). Việc có nhiều thuộc tính làm giảm gánh nặng trong việc phải đoán xem thuộc tính nào là quan trọng, đồng thời cho phép mô hình đo lường sự tương đồng giữa các đối tượng một cách chi tiết hơn bằng cách so sánh nhiều thuộc tính cùng lúc.

5.13 Stochastic Gradient Descent

Hầu hết các phương pháp học sâu đều được xây dựng dựa trên một thuật toán quan trọng: Stochastic Gradient Descent (SGD). SGD là một biến thể của phương pháp hạ gradient đã được giới thiệu trước đó.

Một vấn đề phổ biến trong học máy là tập dữ liệu huấn luyện lớn có thể giúp mô hình tổng quát hóa tốt hơn, nhưng cũng làm tăng đáng kể chi phí tính toán. Do đó, việc tối ưu hóa trên một tập dữ liệu lớn đặt ra thách thức về mặt tính toán.

Hàm mất mát trong học máy thường được biểu diễn dưới dạng tổng của các mất mát riêng lẻ trên từng mẫu dữ liệu. Ví dụ, trong bài toán học có giám sát, hàm mất mát có thể được viết như sau:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\text{data}} L(x, y, \theta) = \frac{1}{m} \sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta) \tag{5.106}$$

trong đó $L(x, y, \theta)$ là hàm mất mát trên từng mẫu dữ liệu. Ví dụ với bài toán phân loại, ta có thể sử dụng hàm mất mát log-likelihood:

$$L(x, y, \theta) = -\log p(y \mid x; \theta). \tag{5.107}$$

Để tối ưu hóa $J(\theta)$, phương pháp hạ gradient tiêu chuẩn yêu cầu tính gradient trung bình trên toàn bộ tập dữ liệu:

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta). \tag{5.108}$$

Chi phí tính toán của phép toán này là $O(m)$. Khi tập dữ liệu có kích thước rất lớn, việc tính gradient đầy đủ sẽ trở nên không khả thi.

SGD giải quyết vấn đề trên bằng cách xấp xỉ gradient trung bình bằng cách sử dụng một minibatch nhỏ gồm m' mẫu dữ liệu, thay vì toàn bộ tập dữ liệu:

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta). \tag{5.109}$$

Thông thường, m' có thể chỉ là vài chục hoặc vài trăm mẫu dữ liệu, giúp giảm đáng kể chi phí tính toán.

Ý tưởng quan trọng của SGD là gradient thực chất là một kỳ vọng toán học. Do đó, ta có thể ước lượng nó bằng cách lấy một mẫu nhỏ từ tập dữ liệu. Cụ thể, ở mỗi bước của

thuật toán, ta chọn ngẫu nhiên một minibatch $\mathbb{B} = \{x^{(1)}, \ldots, x^{(m')}\}$ từ tập huấn luyện.

Minibatch thường có kích thước nhỏ, khoảng vài chục đến vài trăm mẫu. Đặc biệt, kích thước m' thường được giữ cố định ngay cả khi kích thước tập huấn luyện m tăng lên đáng kể. Điều này giúp ta có thể huấn luyện trên các tập dữ liệu khổng lồ bằng cách sử dụng chỉ một số lượng nhỏ mẫu tại mỗi lần cập nhật.

Sau khi tính toán ước lượng gradient, SGD thực hiện bước cập nhật tham số:

$$\theta \leftarrow \theta - \epsilon g, \tag{5.110}$$

trong đó ϵ là tốc độ học (learning rate).

Hạ gradient nói chung thường bị coi là chậm hoặc không đáng tin cậy. Trước đây, việc áp dụng nó cho các bài toán tối ưu hóa phi lồi thường bị coi là liều lĩnh. Tuy nhiên, ngày nay, các mô hình học máy huấn luyện bằng gradient descent hoạt động rất hiệu quả. Thuật toán tối ưu hóa này có thể không đảm bảo hội tụ đến cực tiểu địa phương, nhưng nó thường tìm được giá trị nhỏ của hàm mất mát trong thời gian hợp lý.

SGD có nhiều ứng dụng quan trọng ngoài học sâu. Nó là phương pháp chính để huấn luyện các mô hình tuyến tính trên tập dữ liệu lớn. Khi mô hình có kích thước cố định, chi phí cho mỗi lần cập nhật SGD không phụ thuộc vào kích thước tập huấn luyện m. Số lượng cập nhật cần thiết để hội tụ thường tăng khi m tăng. Tuy nhiên, khi m tiến đến vô hạn, mô hình có thể hội tụ về lỗi kiểm tra tốt nhất trước khi SGD lặp qua toàn bộ tập huấn luyện. Vì vậy, chi phí tiệm cận của SGD theo m có thể coi là $O(1)$.

Trước khi học sâu phát triển, các mô hình phi tuyến chủ yếu được huấn luyện bằng kernel trick kết hợp với mô hình tuyến tính. Các thuật toán kernel thường yêu cầu xây dựng ma trận $G_{i,j} = k(x^{(i)}, x^{(j)})$ có chi phí tính toán $O(m^2)$, điều này không khả thi với tập dữ liệu lớn. Từ năm 2006, học sâu bắt đầu thu hút sự quan tâm vì khả năng tổng quát hóa tốt trên dữ liệu mới khi huấn luyện trên tập có quy mô trung bình. Sau đó, ngành công nghiệp quan tâm nhiều hơn đến học sâu vì khả năng mở rộng để huấn luyện mô hình phi tuyến trên tập dữ liệu lớn.

SGD và các cải tiến của nó sẽ được trình bày chi tiết hơn trong Chương 8.

5.14 Xây dựng thuật toán học máy

Hầu hết các thuật toán học sâu có thể được mô tả như các trường hợp cụ thể của một công thức tổng quát, bao gồm: Định nghĩa một tập dữ liệu, một hàm chi phí (cost function), một mô hình học máy, một thuật toán tối ưu hóa.

Ví dụ, bài toán hồi quy tuyến tính có các thành phần sau bao gồm các biến đầu vào X và đầu ra, hàm chi phí:

$$J(w, b) = -\mathbb{E}_{x,y\sim p_{\text{data}}} \log p_{\text{model}}(y \mid x), \tag{5.111}$$

Mô hình hồi quy tuyến tính: $p_{\text{model}}(y \mid x) = \mathcal{N}(y; x^T w + b, 1)$. Thuật toán tối ưu hóa:

Trong trường hợp này, ta có thể tìm nghiệm bằng cách giải đạo hàm của hàm chi phí theo công thức nghiệm chính quy.

Một điểm quan trọng là chúng ta có thể thay thế từng thành phần trên mà không làm ảnh hưởng đến các thành phần còn lại, từ đó có thể xây dựng nhiều thuật toán khác nhau.

Hàm chi phí thường bao gồm ít nhất một thành phần giúp quá trình học có thể ước lượng thống kê một cách chính xác. Hàm chi phí phổ biến nhất là độ hợp lý tối đa (negative log-likelihood), giúp tối ưu mô hình dựa trên nguyên tắc cực đại hóa hợp lý.

Hàm chi phí cũng có thể chứa các thành phần bổ sung như điều chuẩn (regularization). Ví dụ, ta có thể thêm hệ số phạt chuẩn L2 (weight decay) vào bài toán hồi quy tuyến tính như sau:

$$J(w, b) = \lambda \|w\|_2^2 - \mathbb{E}_{x,y \sim p_{\text{data}}} \log p_{\text{model}}(y \mid x). \tag{5.112}$$

Điều này vẫn cho phép ta tối ưu bằng nghiệm chính quy.

Nếu mô hình không còn là tuyến tính, việc tìm nghiệm đóng không còn khả thi và chúng ta phải sử dụng các thuật toán tối ưu số, chẳng hạn như phương pháp hạ gradient (gradient descent).

Công thức trên không chỉ áp dụng cho học có giám sát mà còn hỗ trợ học không giám sát. Hồi quy tuyến tính là một ví dụ điển hình của học có giám sát. Trong học không giám sát, ta có thể làm việc chỉ với tập dữ liệu X và sử dụng một hàm chi phí phù hợp. Ví dụ, thuật toán phân tích thành phần chính (PCA) có thể được biểu diễn thông qua hàm chi phí:

$$J(w) = \mathbb{E}_{x \sim p_{\text{data}}} \|x - r(x; w)\|_2^2, \tag{5.113}$$

trong đó mô hình được xác định với w có chuẩn bằng 1, và hàm tái tạo (reconstruction function) được định nghĩa là:

$$r(x) = w^T x w. \tag{5.114}$$

Trong một số trường hợp, hàm chi phí có thể là một hàm mà chúng ta không thể thực sự đánh giá được do các hạn chế về tính toán. Trong những trường hợp như vậy, chúng ta vẫn có thể xấp xỉ và tối ưu hóa nó bằng phương pháp tối ưu số lặp, miễn là có cách để xấp xỉ gradient của nó.

Hầu hết các thuật toán học máy đều sử dụng công thức tổng quát này, mặc dù điều đó có thể không hiển nhiên ngay từ đầu. Nếu một thuật toán học máy trông có vẻ độc đáo hoặc được thiết kế thủ công, thì thường có thể hiểu nó như việc sử dụng một bộ tối ưu hóa đặc biệt. Một số mô hình như cây quyết định (decision trees) và k-means yêu cầu các bộ tối ưu hóa đặc biệt vì hàm chi phí của chúng có những vùng phẳng (flat regions), khiến chúng không phù hợp để tối ưu hóa bằng các phương pháp dựa trên gradient.

Nhận ra rằng hầu hết các thuật toán học máy đều có thể được mô tả theo công thức này giúp chúng ta nhìn thấy sự liên kết giữa các thuật toán khác nhau. Điều này giúp phân loại

chúng theo các phương pháp giải quyết bài toán tương tự thay vì chỉ xem chúng như một danh sách dài các thuật toán riêng biệt với những lý do tồn tại khác nhau.

5.15 Những Thách Thức Thúc Đẩy Deep Learning

Các thuật toán học máy đơn giản được mô tả trong chương này hoạt động tốt trên nhiều vấn đề quan trọng. Tuy nhiên, chúng chưa thành công trong việc giải quyết các vấn đề trung tâm trong trí tuệ nhân tạo (AI), chẳng hạn như nhận diện giọng nói hoặc nhận dạng đối tượng.

Sự phát triển của deep learning một phần được thúc đẩy bởi sự thất bại của các thuật toán truyền thống trong việc tổng quát hóa tốt trên các tác vụ AI như vậy.

Phần này thảo luận về cách mà thách thức tổng quát hóa đối với các ví dụ mới trở nên khó khăn theo cấp số nhân khi làm việc với dữ liệu có số chiều cao, cũng như cách mà các cơ chế được sử dụng để đạt được sự tổng quát hóa trong học máy truyền thống không đủ để học các hàm phức tạp trong không gian có số chiều cao. Những không gian như vậy cũng thường đòi hỏi chi phí tính toán rất cao. Deep learning được thiết kế để khắc phục những trở ngại này và nhiều vấn đề khác.

5.15.1 Lời Nguyền Chiều Không Gian (The Curse of Dimensionality)

Nhiều bài toán học máy trở nên cực kỳ khó khăn khi số chiều của dữ liệu tăng cao. Hiện tượng này được gọi là *lời nguyền chiều không gian* (curse of dimensionality). Điều đáng lo ngại đặc biệt là số lượng cấu hình khác biệt có thể có của một tập hợp biến số tăng theo cấp số nhân khi số lượng biến số tăng lên.

Lời nguyền chiều không gian xuất hiện ở nhiều lĩnh vực trong khoa học máy tính, đặc biệt là trong học máy.

Hình 5.9: Khi số chiều liên quan của dữ liệu tăng lên (từ trái sang phải), số lượng cấu hình quan tâm có thể tăng theo cấp số nhân. (Trái) Trong ví dụ một chiều, chúng ta có một biến số và chỉ cần phân biệt 10 vùng quan trọng. Nếu có đủ số lượng mẫu nằm trong mỗi vùng (mỗi vùng tương ứng với một ô trong hình minh họa), thuật toán học có thể dễ dàng tổng quát hóa chính xác. Một cách đơn giản để tổng quát hóa là ước lượng giá trị của hàm mục tiêu trong mỗi vùng (và có thể nội suy giữa các vùng lân cận). (Giữa) Với hai chiều, việc phân biệt 10 giá trị khác nhau của mỗi biến trở nên khó khăn hơn. Chúng ta cần theo dõi tối đa $10 \times 10 = 100$ vùng, và cần ít nhất số lượng mẫu tương ứng để bao phủ tất cả các vùng này.(Phải) Với ba chiều, số vùng tăng lên thành $10^3 = 1,000$, và cần ít nhất số lượng mẫu tương ứng. Với d chiều và v giá trị cần phân biệt trên mỗi trục, số vùng và số mẫu cần thiết dường như là $O(v^d)$. Đây chính là một ví dụ về lời nguyền chiều không gian. Hình ảnh được cung cấp bởi Nicolas Chapados.

Một trong những thách thức do lời nguyền chiều không gian gây ra là thách thức thống kê. Như minh họa trong Hình 5.9, vấn đề thống kê phát sinh do số lượng cấu hình có thể có của x lớn hơn rất nhiều so với số lượng mẫu huấn luyện.

Để hiểu vấn đề này, hãy xem xét không gian đầu vào được tổ chức thành một lưới, như trong hình. Trong không gian có số chiều thấp, ta có thể mô tả dữ liệu bằng một số ít ô lưới, trong đó phần lớn ô được lấp đầy bởi dữ liệu. Khi tổng quát hóa đến một điểm dữ liệu mới, ta có thể dễ dàng suy luận dựa trên các mẫu huấn luyện nằm trong cùng một ô với đầu vào mới.

- Nếu ước lượng mật độ xác suất tại một điểm x, ta có thể đơn giản trả về số lượng mẫu huấn luyện trong cùng một ô chia cho tổng số mẫu.
- Nếu thực hiện phân loại, ta có thể trả về nhãn phổ biến nhất trong ô đó.
- Nếu thực hiện hồi quy, ta có thể lấy trung bình giá trị mục tiêu của các mẫu trong ô.

Nhưng điều gì xảy ra với các ô mà không có mẫu huấn luyện nào? Trong không gian có số chiều cao, số lượng cấu hình quá lớn, vượt xa số lượng mẫu huấn luyện, dẫn đến hầu hết các ô không có dữ liệu huấn luyện. Làm sao có thể suy luận một cách hợp lý về những cấu hình mới này?

Nhiều thuật toán học máy truyền thống đ ơn giản giả định rằng đầu ra tại một điểm mới sẽ xấp xỉ đầu ra của điểm huấn luyện gần nhất.

5.15.2 Tính trơn và điều chuẩn duy trì tính cục bộ

Để tổng quát hóa tốt, các thuật toán học máy cần có những giả định tiên nghiệm (prior) về loại hàm nào mà chúng ta nên học. Những tiên nghiệm này có thể được biểu diễn rõ ràng dưới dạng phân bố xác suất của các tham số mô hình. Ngoài ra, chúng ta cũng có thể hiểu tiên nghiệm như những ảnh hưởng trực tiếp lên chính bản thân hàm số hoặc lên các tham số mô hình một cách gián tiếp.

Một trong những tiên nghiệm phổ biến nhất là giả định về tính trơn (smoothness) hoặc duy trì tính cục bộ (local constancy). Điều này có nghĩa là hàm chúng ta học không nên thay đổi quá nhiều trong một vùng lân cận nhỏ.

Nhiều thuật toán đơn giản chỉ dựa vào tiên nghiệm này để tổng quát hóa, nhưng kết quả là chúng thường không giải quyết tốt các vấn đề trong AI hiện đại. Trong cuốn sách này, chúng ta sẽ thấy rằng học sâu (deep learning) bổ sung nhiều tiên nghiệm khác (cả tường minh và ngầm định) để giảm lỗi tổng quát hóa trong các bài toán phức tạp.

Có nhiều cách khác nhau để thể hiện tiên nghiệm rằng hàm cần học nên có tính trơn hoặc gần như hằng số cục bộ. Điều này dẫn đến nhiều phương pháp học máy khác nhau nhằm tìm một hàm số f^* thỏa mãn điều kiện:

$$f^*(x) \approx f^*(x + \epsilon) \tag{5.115}$$

cho hầu hết các giá trị x và ϵ nhỏ.

Một trường hợp điển hình của phương pháp duy trì tính cục bộ là thuật toán k-láng giềng gần nhất (k-nearest neighbors - kNN). Các bộ dự đoán này thực chất là không thay đổi trong mỗi vùng chứa tất cả các điểm x có cùng tập k láng giềng gần nhất.

Mặc dù thuật toán kNN có thể dự đoán đầu ra từ các điểm lân cận, nhưng hầu hết các thuật toán kernel hiện đại sử dụng các phương pháp nội suy mượt mà hơn để gán nhãn dữ liệu. Một nhóm quan trọng của các hàm kernel là hàm kernel cục bộ, trong đó $k(x, v)$ có giá trị lớn khi $x = v$ và giảm dần khi x và v cách xa nhau.

Cây quyết định (decision trees) cũng có những hạn chế khi chỉ dựa vào tính trơn. Chúng phân chia không gian đặc trưng thành nhiều vùng rời rạc, mỗi vùng có một tham số riêng biệt. Điều này có nghĩa là để biểu diễn chính xác một cây có n lá, ta cần ít nhất n điểm dữ liệu huấn luyện.

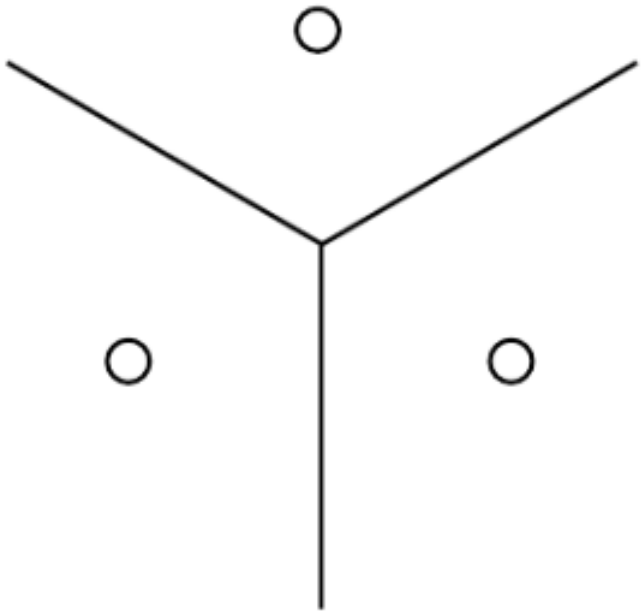
Nhìn chung, để phân biệt $O(k)$ vùng trong không gian đầu vào, hầu hết các phương pháp này yêu cầu $O(k)$ mẫu huấn luyện. Với kNN, mỗi mẫu có thể xác định đúng một vùng duy nhất.

Một vấn đề phức tạp hơn là làm thế nào để học được một hàm chi phí có thể phân biệt nhiều vùng hơn số lượng mẫu huấn luyện. Ví dụ, nếu dữ liệu có cấu trúc như một bàn cờ (checkerboard), thì việc dựa vào tính trơn và tổng quát cục bộ có thể không đủ để học chính xác toàn bộ mô hình.

Giả định về tính trơn tru và các thuật toán học không tham số liên quan hoạt động cực kỳ hiệu quả miễn là có đủ ví dụ để thuật toán học quan sát được các điểm cao trên hầu hết các đỉnh và các điểm thấp trên hầu hết các thung lũng của hàm số thực sự cần học. Điều này thường đúng khi hàm cần học đủ trơn tru và chỉ thay đổi trên một số ít chiều.

Trong không gian có số chiều cao, ngay cả một hàm rất trơn tru cũng có thể thay đổi theo cách khác nhau dọc theo từng chiều. Nếu hàm số còn có hành vi khác biệt ở các vùng khác nhau, thì việc mô tả nó bằng một tập hợp các ví dụ huấn luyện có thể trở nên cực kỳ phức tạp. Nếu hàm số quá phức tạp (chúng ta muốn phân biệt một số lượng lớn vùng so với số lượng ví dụ), liệu có hy vọng nào để tổng quát hóa tốt không?

Câu trả lời cho cả hai câu hỏi trên—liệu có thể biểu diễn một hàm số phức tạp một cách hiệu quả, và liệu hàm ước lượng có thể tổng quát hóa tốt cho các đầu vào mới hay không—là có. Điểm mấu chốt là có thể xác định một số lượng rất lớn vùng, chẳng hạn $O(2^k)$, chỉ với $O(k)$ ví dụ, miễn là chúng ta giới thiệu một số sự phụ thuộc giữa các vùng thông qua các giả định bổ sung về phân phối sinh dữ liệu cơ bản. Theo cách này, chúng ta có thể thực sự tổng quát hóa không chỉ cục bộ mà còn mở rộng phạm vi rộng hơn [6], [7]. Nhiều thuật toán học sâu khác nhau cung cấp các giả định ngầm hoặc tường minh hợp lý cho một loạt các nhiệm vụ AI nhằm tận dụng lợi thế này.



Hình 5.10: Minh họa về cách thuật toán láng giềng gần nhất (nearest neighbor) phân chia không gian đầu vào thành các vùng. Mỗi ví dụ (được biểu diễn bằng một vòng tròn) trong mỗi vùng xác định ranh giới của vùng đó (được biểu diễn bằng các đường). Giá trị được gán cho từng ví dụ xác định đầu ra cho tất cả các điểm trong vùng tương ứng. Các vùng được xác định bởi thuật toán láng giềng gần nhất tạo thành một mô hình hình học gọi là biểu đồ Voronoi. Số lượng các vùng liên tục này không thể tăng nhanh hơn số lượng ví dụ huấn luyện. Mặc dù hình minh họa này mô tả cụ thể hành vi của thuật toán láng giềng gần nhất, nhưng các thuật toán học máy khác cũng dựa vào giả định trơn tru cục bộ để tổng quát hóa có xu hướng thể hiện hành vi tương tự: mỗi ví dụ huấn luyện chỉ cung cấp thông tin cho mô hình về cách tổng quát hóa trong một vùng lân cận ngay xung quanh ví dụ đó.

Các phương pháp khác trong học máy thường đưa ra những giả định mạnh mẽ và đặc thù hơn theo từng tác vụ. Ví dụ, chúng ta có thể dễ dàng giải quyết bài toán bàn cờ bằng cách cung cấp giả định rằng hàm mục tiêu là tuần hoàn. Thông thường, chúng ta không đưa vào mạng nơ-ron những giả định mạnh mẽ và đặc thù như vậy để chúng có thể tổng quát hóa cho nhiều cấu trúc khác nhau. Các nhiệm vụ AI có cấu trúc quá phức tạp để bị giới hạn bởi những thuộc tính đơn giản, được xác định thủ công như tính tuần hoàn, vì vậy chúng ta cần các thuật toán học có thể kết hợp những giả định tổng quát hơn.

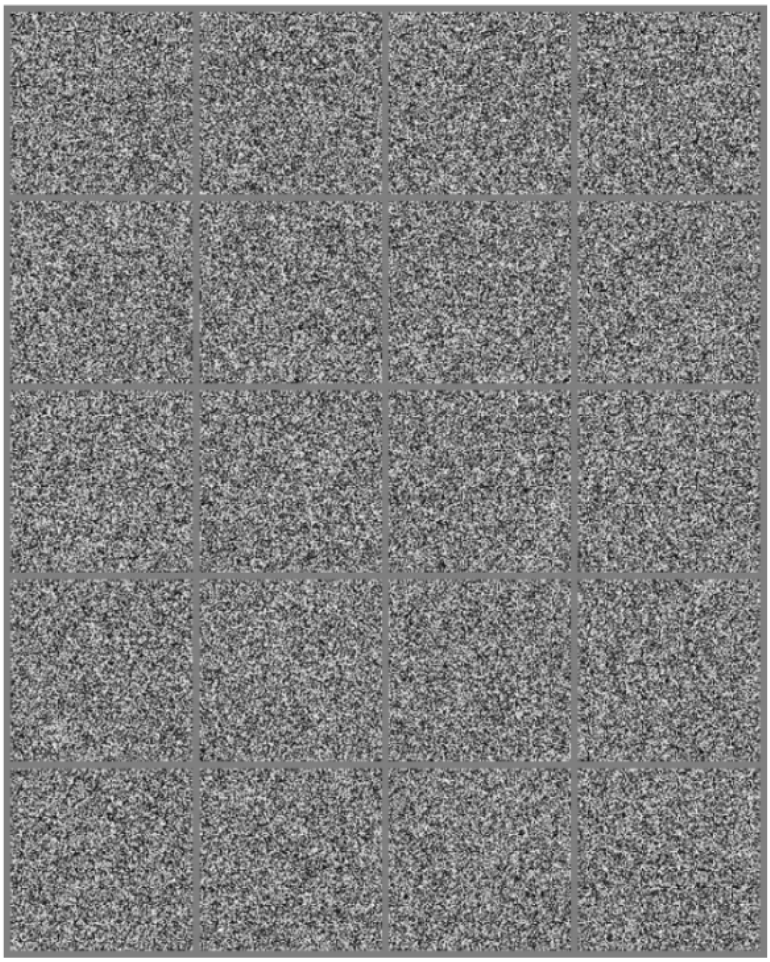
Ý tưởng cốt lõi trong học sâu là giả định rằng dữ liệu được tạo ra bởi sự kết hợp của các nhân tố, hoặc đặc trưng, có thể ở nhiều cấp độ trong một hệ thống phân cấp. Nhiều giả định tương tự khác cũng có thể cải thiện hơn nữa các thuật toán học sâu. Những giả định có vẻ đơn giản này cho phép thu được lợi thế theo cấp số nhân trong mối quan hệ giữa số lượng ví dụ và số lượng vùng có thể phân biệt. Chúng tôi mô tả những lợi thế theo cấp số nhân này một cách chính xác hơn trong các mục 6.4.1, 15.4 và 15.5. Những lợi thế theo cấp số nhân này nhờ vào việc sử dụng các biểu diễn phân tán sâu, giúp chống lại các thách thức theo cấp số nhân do *lời nguyền chiều cao* gây ra.

5.15.3 Học Manifold (Manifold Learning)

Một khái niệm quan trọng trong nhiều ý tưởng về học máy là manifold. Manifold là một vùng liên thông trong không gian. Về mặt toán học, nó là một tập hợp các điểm được liên kết với một vùng lân cận xung quanh mỗi điểm. Từ một điểm bất kỳ, manifold có dạng cục bộ giống không gian Euclid. Trong cuộc sống hàng ngày, chúng ta cảm nhận bề mặt Trái Đất như một mặt phẳng 2D, nhưng thực tế nó là một manifold hình cầu trong không gian 3D.

Khái niệm về vùng lân cận xung quanh mỗi điểm ngụ ý sự tồn tại của các phép biến đổi có thể được áp dụng để di chuyển trên manifold từ một vị trí đến vị trí lân cận. Ví dụ, với bề mặt Trái Đất như một manifold, ta có thể di chuyển theo hướng bắc, nam, đông hoặc tây.

Mặc dù thuật ngữ *manifold* có ý nghĩa toán học chính xác, trong học máy nó thường được sử dụng theo cách linh hoạt hơn để chỉ một tập hợp liên thông các điểm có thể được xấp xỉ tốt bằng cách chỉ xem xét một số ít chiều tự do, hoặc các chiều nhúng trong một không gian có số chiều cao hơn. Mỗi chiều tương ứng với một hướng biến đổi cục bộ. Hình **??** minh họa dữ liệu huấn luyện nằm gần một manifold một chiều được nhúng trong không gian hai chiều.



Hình 5.11: Lấy mẫu hình ảnh một cách ngẫu nhiên (bằng cách chọn ngẫu nhiên mỗi pixel theo phân phối đều) sẽ tạo ra các hình ảnh nhiễu. Mặc dù có một xác suất không bằng 0 để tạo ra một hình ảnh của một khuôn mặt hoặc bất kỳ đối tượng nào khác thường gặp trong các ứng dụng AI, nhưng chúng ta không bao giờ thực sự thấy điều này xảy ra trong thực tế. Điều này gợi ý rằng các hình ảnh gặp phải trong các ứng dụng AI chiếm một tỉ lệ không đáng kể trong không gian hình ảnh.

Trong bối cảnh học máy, ta cho phép số chiều của manifold thay đổi từ điểm này sang điểm khác. Điều này thường xảy ra khi một manifold tự cắt qua chính nó. Ví dụ, một hình số tám (*figure eight*) là một manifold có một chiều tại hầu hết các vị trí, nhưng có hai chiều tại điểm giao nhau ở trung tâm.

Nhiều bài toán học máy có vẻ vô vọng nếu chúng ta kỳ vọng thuật toán học phải tìm ra các hàm có biến đổi phức tạp trên toàn bộ không gian $\mathbb{R}^n$. Các thuật toán học manifold giải quyết trở ngại này bằng cách giả định rằng phần lớn $\mathbb{R}^n$ bao gồm các đầu vào không hợp lệ, và các đầu vào quan trọng chỉ xuất hiện trên một tập hợp các manifold chứa một tập con nhỏ các điểm, với các biến đổi thú vị trong đầu ra của hàm học được chỉ xảy ra dọc theo các hướng nằm trên manifold, hoặc chỉ xuất hiện khi chúng ta di chuyển từ manifold này sang manifold khác.

Học manifold ban đầu được giới thiệu cho dữ liệu liên tục và trong bối cảnh học không giám sát, mặc dù ý tưởng về sự tập trung xác suất này có thể được tổng quát hóa cho cả dữ

liệu rời rạc và học có giám sát. Giả định chính vẫn là khối lượng xác suất tập trung mạnh mẽ vào một số vùng nhất định.

Giả định rằng dữ liệu nằm dọc theo một manifold có số chiều thấp có thể không phải lúc nào cũng đúng hoặc hữu ích. Tuy nhiên, chúng tôi lập luận rằng trong bối cảnh các nhiệm vụ AI, chẳng hạn như xử lý hình ảnh, âm thanh hoặc văn bản, giả định manifold ít nhất cũng đúng một cách xấp xỉ. Bằng chứng ủng hộ giả định này có thể được chia thành hai loại quan sát chính.

Quan sát đầu tiên ủng hộ giả thuyết về manifoled là phân phối xác suất trên hình ảnh, chuỗi văn bản và âm thanh xuất hiện trong đời sống thực là rất tập trung. Nhiễu đồng đều hầu như không bao giờ giống với các đầu vào có cấu trúc từ các lĩnh vực này. Hình 5.12 cho thấy, thay vào đó, các điểm được lấy mẫu đồng đều trông giống như các mẫu nhiễu xuất hiện trên các tivi analog khi không có tín hiệu. Tương tự, nếu bạn tạo ra một tài liệu bằng cách chọn ngẫu nhiên các chữ cái, xác suất bạn sẽ tạo ra một văn bản có nghĩa bằng tiếng Anh là gì? Gần như bằng 0, lại một lần nữa, bởi vì hầu hết các chuỗi chữ cái dài không tương ứng với chuỗi ngôn ngữ tự nhiên: phân phối của các chuỗi ngôn ngữ tự nhiên chiếm một không gian rất nhỏ trong tổng thể không gian các chuỗi chữ cái.

Dĩ nhiên, các phân phối xác suất tập trung không đủ để chứng minh rằng dữ liệu nằm trên một số manifoled có kích thước hợp lý. Chúng ta cũng phải chứng minh rằng các ví dụ chúng ta gặp phải được kết nối với nhau bởi các ví dụ khác, với mỗi ví dụ được bao quanh bởi các ví dụ khác có độ tương đồng cao có thể đạt được bằng cách áp dụng các phép biến đổi để di chuyển trên manifoled. Lập luận thứ hai ủng hộ giả thuyết về manifoled là chúng ta có thể tưởng tượng ra những khu vực và phép biến đổi như vậy, ít nhất là theo cách hình dung. Trong trường hợp hình ảnh, chúng ta chắc chắn có thể nghĩ đến nhiều phép biến đổi có thể cho phép chúng ta vẽ ra một manifoled trong không gian hình ảnh: chúng ta có thể từ từ làm mờ hoặc sáng ánh sáng, từ từ di chuyển hoặc xoay các đối tượng trong hình ảnh, từ từ thay đổi màu sắc trên bề mặt các đối tượng, v.v. Nhiều manifoled có thể tham gia vào hầu hết các ứng dụng. Ví dụ, manifoled của hình ảnh khuôn mặt người có thể không được kết nối với manifoled của hình ảnh khuôn mặt con mèo.

Những thí nghiệm tư duy này mang lại một số lý do trực quan ủng hộ giả thuyết về manifoled. Các thí nghiệm chính thức hơn (Cayton, 2005; Narayanan và Mitter, 2010; Schölkopf et al., 1998; Roweis và Saul, 2000; Tenenbaum et al., 2000; Brand, 2003; Belkin và Niyogi, 2003; Donoho và Grimes, 2003; Weinberger và Saul, 2004) rõ ràng hỗ trợ giả thuyết này đối với một lớp lớn các bộ dữ liệu quan trọng trong AI.

Khi dữ liệu nằm trên một manifoled có chiều thấp, việc biểu diễn dữ liệu dưới dạng tọa độ trên manifoled có thể là cách tự nhiên nhất đối với các thuật toán học máy, thay vì dưới dạng tọa độ trong Rn. Trong đời sống hàng ngày, chúng ta có thể coi các con đường như những manifoled 1 chiều được nhúng trong không gian 3 chiều. Chúng ta chỉ đường đến các địa chỉ cụ thể theo các số nhà trên những con đường 1 chiều này, không phải dưới dạng tọa độ trong không gian 3 chiều. Việc trích xuất các tọa độ trên manifoled này là một

thách thức nhưng hứa hẹn sẽ cải thiện nhiều thuật toán học máy. Nguyên lý tổng quát này được áp dụng trong nhiều bối cảnh. Hình 5.13 cho thấy cấu trúc manifoled của một bộ dữ liệu bao gồm các khuôn mặt. Vào cuối cuốn sách này, chúng ta sẽ phát triển các phương pháp cần thiết để học cấu trúc manifoled như vậy. Trong hình 20.6, chúng ta sẽ thấy cách một thuật toán học máy có thể thành công đạt được mục tiêu này.

Đây là phần kết thúc của phần I, phần đã cung cấp các khái niệm cơ bản trong toán học và học máy được sử dụng trong các phần còn lại của cuốn sách. Bạn đã sẵn sàng để bắt đầu nghiên cứu về học sâu.

Hình 5.12: Các ví dụ huấn luyện từ Bộ dữ liệu Khuôn mặt Đa quan điểm QMUL (Gong et al., 2000), trong đó các đối tượng được yêu cầu di chuyển theo cách sao cho bao phủ được manifoled hai chiều tương ứng với hai góc quay. Chúng ta muốn các thuật toán học có thể khám phá và tách rời các tọa độ của manifoled như vậy. Hình 20.6 minh họa một thành tựu như vậy.

CHƯƠNG 6. MẠNG NƠ-RON HỌC SÂU

Mạng nơ-ron lan truyền xuôi sâu (Deep Feedforward Networks) là một trong những mô hình cơ bản nhất trong học sâu. Từ "học sâu"bắt nguồn từ việc mô hình có nhiều lớp — tức là "sâu"về mặt cấu trúc. Lớp cuối cùng trong mạng được gọi là lớp đầu ra (output layer). Khi huấn luyện mạng nơ-ron, mục tiêu là điều chỉnh hàm $f(x)$ để nó gần giống với một hàm mục tiêu $f^*(x)$. Tập dữ liệu huấn luyện sẽ cung cấp các ví dụ x cùng với nhãn $y \approx f^*(x)$, để từ đó mạng học cách cho ra kết quả gần với y. Chỉ có lớp đầu ra là được hướng dẫn trực tiếp như vậy; các lớp còn lại phải tự điều chỉnh sao cho toàn mạng có thể ước lượng f^* tốt nhất. Do không có nhãn trực tiếp cho chúng, các lớp này được gọi là lớp ẩn (hidden layers).

Mạng này được gọi là "nơ-ron"vì được lấy cảm hứng từ hệ thần kinh. Trong mỗi lớp ẩn, dữ liệu thường được biểu diễn dưới dạng một vector, trong đó độ dài vector thể hiện độ rộng của mô hình. Mỗi phần tử trong vector hoạt động giống như một "nơ-ron", xử lý dữ liệu đầu vào rồi truyền qua một hàm kích hoạt. Dù xuất phát từ sinh học, nhưng các mạng nơ-ron hiện đại lại dựa phần lớn vào toán học và kỹ thuật, chứ không nhằm tái hiện chính xác cách hoạt động của não. Vì vậy, ta có thể xem mạng lan truyền xuôi như một công cụ để ước lượng hàm, hơn là mô hình mô phỏng não bộ.

Để hiểu rõ hơn, ta hãy nhìn lại các mô hình tuyến tính và lý do vì sao chúng chưa đủ. Ví dụ như hồi quy tuyến tính hay hồi quy logistic — dễ tối ưu nhờ các tính chất toán học thì lại bị giới hạn bởi tính tuyến tính. Trong khi đó, các bài toán trong thực tế thường là phi tuyến, đòi hỏi mô hình có khả năng biểu diễn được các hàm phi tuyến.

6.1 Bài toán XOR và mạng nơ-ron truyền thẳng

Mạng nơ-ron truyền thẳng là một trong những cấu trúc phổ biến nhất trong học sâu. Để làm rõ cách hoạt động của mạng, ta sẽ dùng ví dụ bài toán XOR.

6.1.1 Bài toán XOR

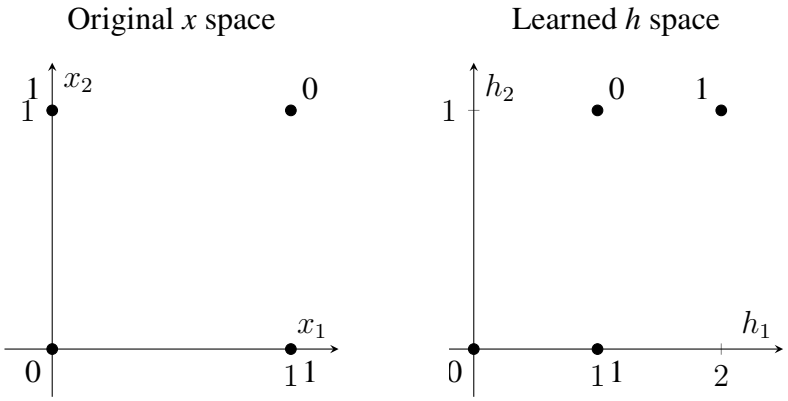
Hàm XOR (Exclusive OR) là một phép toán logic đơn giản trên hai giá trị nhị phân x_1 và x_2:

- Nếu một trong hai giá trị bằng 1, kết quả là 1.
- Nếu cả hai bằng 0 hoặc cả hai bằng 1, kết quả là 0.

Bảng giá trị của hàm XOR:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

Bài toán đặt ra là tìm một hàm $y = f(x; \theta)$ với tham số θ sao cho kết quả của hàm gần giống với XOR.



Nếu dùng một mô hình tuyến tính như sau:

$$f(x; w, b) = x^T w + b \tag{6.1}$$

thì nghiệm tối ưu sẽ cho $w = 0$ và $b = \frac{1}{2}$, nghĩa là luôn dự đoán 0.5 — không giải được bài toán. Lý do là vì XOR là hàm phi tuyến.

Để giải được, ta cần thêm tầng ẩn. Cấu trúc mạng gồm:

- Tầng ẩn: Biến đổi x thành h
- Tầng đầu ra: Dùng mô hình tuyến tính để tính y từ h

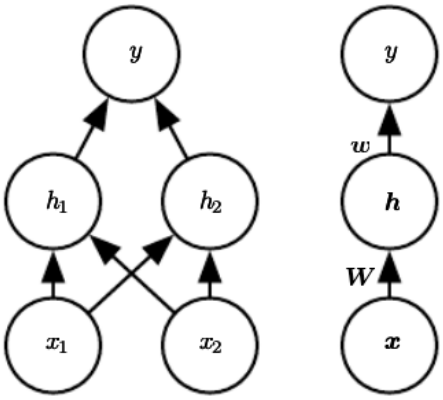
Công thức:

$$h = g(W^T x + c) \tag{6.2}$$
$$y = f(h) = w^T h + b \tag{6.3}$$

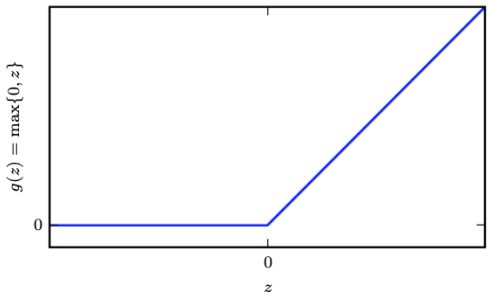
Trong đó:

- W: trọng số từ x đến h
- c: bias
- g: hàm kích hoạt phi tuyến (thường là ReLU hoặc sigmoid)

Nếu không có hàm g, mạng chỉ là một hàm tuyến tính và vẫn không giải được XOR.



Hình 6.1: Mạng nơ-ron dùng để giải bài toán XOR với một tầng ẩn có 2 nút. Hình bên trái vẽ từng nút rõ ràng; hình bên phải vẽ từng tầng như một khối.



Hình 6.2: Hàm ReLU (Rectified Linear Unit) là hàm kích hoạt phổ biến nhất. Nó giữ lại tính đơn giản và khả năng tối ưu tốt của mô hình tuyến tính, đồng thời giúp tạo ra phi tuyến cần thiết cho học sâu.

Với ReLU, ta có:

$$f(x; W, c, w, b) = w^T \max(0, W^T x + c) + b.$$

Dùng các tham số:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b = 0.$$

Tập đầu vào:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Tính toán từng bước:

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}, \quad XW + c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix},$$

$$h = \max(0, XW + c) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}, \quad y = hw = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Kết quả đúng 100

6.1.2 Tổng quan về mạng nơ-ron nhân tạo

Mạng nơ-ron nhân tạo (ANN - Artificial Neural Networks) là mô hình lấy cảm hứng từ cách hoạt động của hệ thần kinh, gồm các đơn vị gọi là "nơ-ron"được tổ chức thành các lớp. Mỗi nơ-ron cơ bản có thể coi là một mô hình hồi quy tuyến tính đơn giản.

Mạng nơ-ron thường gồm ba loại lớp:

- Lớp đầu vào (Input Layer): nhận dữ liệu.
- Lớp ẩn (Hidden Layers): xử lý và biến đổi dữ liệu.
- Lớp đầu ra (Output Layer): cho kết quả cuối cùng.

Mỗi nơ-ron trong mạng nơ-ron có thể được mô tả bằng một công thức đơn giản:

$$h = g(W^\top x + b) \tag{6.4}$$

Trong đó:

- x là đầu vào,
- W là vector trọng số kết nối đầu vào với nơ-ron,
- b là hệ số bù (bias),
- $g(\cdot)$ là hàm kích hoạt.

Hàm kích hoạt rất quan trọng vì nó giúp mạng học được các quan hệ phi tuyến giữa đầu vào và đầu ra. Một số hàm kích hoạt thường dùng là:

- Sigmoid: $g(z) = \frac{1}{1+e^{-z}}$, phổ biến trong phân loại nhị phân nhưng dễ gặp vấn đề gradient biến mất.
- Tanh: $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, cho giá trị đầu ra trong khoảng $[-1, 1]$, giúp trung bình dữ liệu tốt hơn sigmoid.
- ReLU (Rectified Linear Unit): $g(z) = \max(0, z)$, đơn giản và hiệu quả, nhưng có thể dẫn đến hiện tượng "nơ-ron chết"nếu giá trị đầu vào nhỏ hơn 0 quá nhiều.

Mạng nơ-ron học bằng cách điều chỉnh trọng số sao cho đầu ra gần với nhãn đúng nhất. Quá trình học diễn ra qua hai bước chính:

- Lan truyền tiến (Forward Propagation): Dữ liệu đi qua mạng từ đầu vào đến đầu ra, tạo ra dự đoán.
- Lan truyền ngược (Backward Propagation): Dựa vào sai số giữa dự đoán và thực tế, mạng điều chỉnh lại trọng số thông qua thuật toán như Gradient Descent.

Sai số được đo bằng hàm mất mát $\mathcal{L}$. Ví dụ, với phân loại nhị phân, ta thường dùng hàm mất mát entropy chéo:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)] \tag{6.5}$$

Trong quá trình tối ưu, một số biến thể của thuật toán hạ gradient được sử dụng:

- SGD (Stochastic Gradient Descent): Cập nhật trọng số từng mẫu một, nhanh nhưng nhiễu.
- Momentum: Giúp quá trình tối ưu ổn định hơn bằng cách "nhớ"hướng di chuyển trước đó.
- Adam (Adaptive Moment Estimation): Kết hợp ưu điểm của Momentum và tốc độ học điều chỉnh riêng cho từng tham số.

Mạng nơ-ron là nền tảng cốt lõi trong học sâu và được ứng dụng rộng rãi trong thực tế.

Việc lựa chọn đúng kiến trúc mạng, hàm kích hoạt, và thuật toán tối ưu sẽ ảnh hưởng lớn đến hiệu suất mô hình.

6.2 Học dựa trên Gradient

Huấn luyện mạng nơ-ron cũng như nhiều mô hình học máy khác thường dựa vào phương pháp hạ gradient (gradient descent). Điểm khác biệt lớn giữa mạng nơ-ron và các mô hình tuyến tính là tính phi tuyến khiến cho hàm mất mát của mạng nơ-ron thường không lồi (non-convex).

Trong khi các mô hình tuyến tính như hồi quy hay SVM có thể dùng bộ giải tối ưu lồi để tìm nghiệm toàn cục, mạng nơ-ron phải dùng các thuật toán lặp dựa vào gradient để tìm nghiệm gần đúng.

Với mạng nơ-ron, quá trình tối ưu nhắm đến giảm giá trị hàm mất mát càng thấp càng tốt. Do hàm mất mát không lồi, thuật toán có thể bị kẹt ở cực tiểu địa phương và kết quả phụ thuộc vào cách khởi tạo trọng số ban đầu.

Thường thì:

- Trọng số được khởi tạo ngẫu nhiên nhỏ.
- Bias có thể khởi tạo bằng 0 hoặc một giá trị nhỏ dương.

Các thuật toán tối ưu sẽ được trình bày chi tiết ở Chương 8. Ở đây, chỉ cần nắm rằng huấn luyện mạng nơ-ron luôn dựa vào việc tính gradient để cập nhật tham số sao cho giảm sai số đầu ra.

6.2.1 Hàm chi phí

Một thành phần không thể thiếu trong quá trình huấn luyện mạng nơ-ron là hàm chi phí. Nó giúp đo lường độ sai khác giữa dự đoán và giá trị thật.

Nếu mô hình xác định phân phối xác suất có điều kiện $p(y|x;\theta)$, thì thường ta sẽ dùng nguyên lý cực đại hợp lý (Maximum Likelihood) và chọn hàm chi phí là hàm chệch chéo (cross-entropy):

$$L = -\sum_i y_i \log \hat{y}_i, \tag{6.6}$$

Hàm này rất phổ biến trong các bài toán phân loại.

Trong bài toán hồi quy, ta dùng hàm mất mát bình phương trung bình (Mean Squared Error - MSE):

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{6.7}$$

Tổng hàm chi phí dùng để huấn luyện mạng thường gồm:

- Hàm mất mát chính.
- Một thành phần điều chuẩn (regularization) để giảm overfitting.

Ví dụ, kỹ thuật weight decay từng dùng cho hồi quy tuyến tính cũng được áp dụng hiệu quả cho mạng nơ-ron.

Các phương pháp điều chuẩn nâng cao sẽ được trình bày trong Chương 7.

6.2.1.1 Học phân phối có điều kiện bằng Maximum Likelihood

Trong học máy, tối đa hóa hợp lý (Maximum Likelihood Estimation - MLE) là cách phổ biến để tìm tham số tốt nhất cho mô hình bằng cách làm cho mô hình khớp tối đa với dữ liệu quan sát.

Hàm hợp lý (likelihood function) chính là xác suất mà mô hình gán cho dữ liệu thực tế:

$$L(\theta) = \prod_{i=1}^{N} p_{model}(y_i|x_i, \theta) \tag{6.8}$$

Ở đây:

- (x_i, y_i) là các cặp dữ liệu huấn luyện,
- $p_{model}(y_i|x_i, \theta)$ là xác suất mà mô hình gán cho đầu ra y_i tương ứng với đầu vào x_i,
- θ là tập các tham số cần học.

Mục tiêu là tìm θ sao cho $L(\theta)$ là lớn nhất. Tuy nhiên, vì tích của nhiều số nhỏ có thể gây tràn số, ta thường lấy log để chuyển sang log-likelihood:

$$\log L(\theta) = \sum_{i=1}^{N} \log p_{model}(y_i|x_i, \theta) \tag{6.9}$$

Thay vì tối đa log-likelihood, ta sẽ tối thiểu hoá giá trị âm của nó — được gọi là hàm chi phí dựa trên log-likelihood âm:

$$J(\theta) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x) \tag{6.10}$$

Hàm này thực chất chính là entropy chéo — một cách đo sự khác biệt giữa phân phối dữ liệu thật $\hat{p}_{data}$ và phân phối dự đoán p_{model} của mô hình.

Trong một trường hợp cụ thể khi $p_{model}(y|x)$ là phân phối chuẩn (Gaussian), ta có thể liên hệ entropy chéo với lỗi bình phương trung bình (MSE). Giả sử mô hình tuân theo phân phối chuẩn:

$$p_{model}(y|x) = N(y; f(x; \theta), I) \tag{6.11}$$

Công thức của phân phối chuẩn như sau:

$$p(y|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - f(x;\theta))^2}{2\sigma^2}\right) \tag{6.12}$$

Lấy log của biểu thức trên:

$$\log p(y|x) = -\frac{(y - f(x;\theta))^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma^2) \tag{6.13}$$

Tối thiểu hoá $-\log p(y|x)$, ta bỏ qua các hằng số không phụ thuộc vào θ, và thu được hàm lỗi:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{x,y\sim\hat{p}_{data}}||y - f(x;\theta)||^2 \tag{6.14}$$

Đây chính là hàm mất mát bình phương trung bình (MSE).

Ví dụ, giả sử ta có tập dữ liệu gồm ba điểm:

x	y
1	2
2	4
3	6

Giả sử mô hình dự đoán là $f(x) = 2x + 1$, ta có bảng so sánh:

x	y (thực tế)	$\hat{y} = f(x)$
1	2	3
2	4	5
3	6	7

Khi đó, MSE là:

$$MSE = \frac{1}{3}\sum_{i=1}^{3}(y_i - \hat{y}_i)^2 = \frac{(2-3)^2 + (4-5)^2 + (6-7)^2}{3} = \frac{3}{3} = 1. \tag{6.15}$$

6.2.1.2 Học thống kê có điều kiện

Trong một số trường hợp, thay vì tìm toàn bộ phân phối $p(y|x;\theta)$, ta chỉ cần tìm một giá trị đại diện, chẳng hạn như trung bình hoặc trung vị của y khi biết x.

Khi huấn luyện mạng nơ-ron, ta thực chất đang học một hàm $f(x;\theta)$ để mô phỏng mối quan hệ giữa đầu vào và đầu ra. Đây không chỉ là việc điều chỉnh tham số, mà là bài toán tối ưu trên không gian hàm số — nghĩa là ta tìm ra một hàm tối ưu nhất trong vô số hàm

có thể.

Trong toán học, khái niệm này dẫn đến một công cụ gọi là giải tích biến phân (calculus of variations), trong đó hàm mất mát được viết dưới dạng:

$$\mathcal{L}[f] = \int L(x, f(x), f'(x), \dots)\, dx \tag{6.16}$$

Dù không cần dùng đến giải tích biến phân một cách tường minh, ta có thể sử dụng các kết quả quan trọng của nó để hiểu rõ hơn về mục tiêu huấn luyện.

Ví dụ với hàm mất mát MSE:

$$f^* = \arg\min_f \mathbb{E}_{x,y\sim p_{\text{data}}} \|y - f(x)\|^2 \tag{6.17}$$

Nghiệm tối ưu của bài toán trên là:

$$f^*(x) = \mathbb{E}_{y\sim p_{\text{data}}(y|x)}[y] \tag{6.18}$$

Nghĩa là nếu có đủ dữ liệu, tối ưu MSE giúp mô hình học được giá trị kỳ vọng của y khi biết x.

Nếu thay bằng lỗi tuyệt đối trung bình (MAE):

$$f^* = \arg\min_f \mathbb{E}_{x,y\sim p_{\text{data}}} \|y - f(x)\|_1 \tag{6.19}$$

thì nghiệm tối ưu sẽ là trung vị:

$$f^*(x) = \text{median}(y \mid x) \tag{6.20}$$

Trung vị hữu ích khi dữ liệu có nhiều giá trị ngoại lai, vì ít bị ảnh hưởng hơn trung bình.

Tuy nhiên, cả MSE và MAE đều có những hạn chế khi dùng với gradient descent:

- MSE dễ tạo ra gradient nhỏ khi lỗi lớn, khiến cập nhật chậm.
- MAE có gradient không liên tục, gây bất ổn cho việc học.

Do đó, với các bài toán phân loại hoặc khi cần ước lượng xác suất, ta thường dùng hàm mất mát entropy chéo:

$$\mathcal{L} = -\sum_i y_i \log \hat{y}_i \tag{6.21}$$

Hàm này có gradient ổn định hơn và kết hợp tốt với hàm softmax ở đầu ra.

Tóm lại, việc chọn đúng hàm mất mát là yếu tố then chốt trong huấn luyện mạng nơ-ron. Tùy theo mục tiêu của bài toán (dự đoán giá trị trung bình, trung vị hay phân loại), ta sẽ chọn giữa MSE, MAE hay cross-entropy để tối ưu hóa mô hình hiệu quả nhất.

6.2.2 Lớp đầu ra

Việc lựa chọn hàm mất mát luôn gắn liền với cách thiết kế lớp đầu ra của mô hình. Thông thường, ta sử dụng cross-entropy để đo mức độ khác biệt giữa phân phối thật của dữ liệu và phân phối mà mô hình dự đoán. Cách biểu diễn đầu ra sẽ ảnh hưởng trực tiếp đến dạng cụ thể của hàm mất mát này.

Trên thực tế, bất kỳ loại đơn vị nào được dùng ở lớp đầu ra cũng đều có thể sử dụng cho lớp ẩn. Tuy nhiên, trong phần này ta sẽ tập trung vào cách sử dụng các đơn vị này ở lớp đầu ra, còn vai trò của chúng trong lớp ẩn sẽ được trình bày ở Mục 6.3.

Giả sử mô hình mạng lan truyền xuôi tạo ra một biểu diễn đặc trưng ẩn là:

$$h = f(x; \theta)$$

trong đó x là đầu vào và θ là tham số mô hình. Lớp đầu ra có nhiệm vụ biến đổi h thành đầu ra cuối cùng phục vụ cho bài toán cụ thể.

6.2.2.1 Các đơn vị tuyến tính cho phân phối Gaussian

Một loại lớp đầu ra đơn giản là đơn vị tuyến tính — thực hiện phép biến đổi tuyến tính kèm theo một hằng số, mà không sử dụng bất kỳ hàm phi tuyến nào. Cụ thể:

$$\hat{y} = W^\top h + b$$

với W là ma trận trọng số, b là vector hệ số bù.

Đơn vị tuyến tính thường được dùng để mô hình hóa trung bình của một phân phối Gaussian có điều kiện:

$$p(y \mid x) = \mathcal{N}(y; \hat{y}, I)$$

Trong đó $\hat{y}$ là trung bình và I là ma trận hiệp phương sai đơn vị.

Tối đa hóa log-likelihood trong trường hợp này tương đương với việc tối thiểu hóa lỗi bình phương trung bình (MSE).

Ngoài ra, lý thuyết hợp lý cực đại còn cho phép ta học cả ma trận hiệp phương sai, hoặc làm cho nó phụ thuộc vào đầu vào x. Tuy nhiên, để đảm bảo tính hợp lệ, ma trận hiệp phương sai cần luôn là dương xác định. Việc này thường khó thực hiện nếu chỉ dùng lớp đầu ra tuyến tính, nên các kỹ thuật khác sẽ được trình bày ở phần sau.

Một lợi thế của đơn vị tuyến tính là không gây ra hiện tượng bão hòa — đầu ra không bị giới hạn trong một khoảng nhất định. Điều này giúp quá trình học bằng các thuật toán tối ưu dựa trên gradient diễn ra hiệu quả và ổn định hơn.

6.2.2.2 Hàm sigmoid cho phân phối Bernoulli

Khi bài toán yêu cầu dự đoán giá trị của một biến nhị phân y (ví dụ: phân loại nhị phân), ta cần mô hình hóa xác suất $P(y = 1|x)$. Để đảm bảo xác suất nằm trong khoảng $[0, 1]$, cần dùng một hàm kích hoạt phù hợp.

Một giải pháp đơn giản là dùng một đơn vị tuyến tính rồi kẹp đầu ra vào khoảng $[0, 1]$:

$$P(y = 1|x) = \max\left(0, \min\left(1, w^T h + b\right)\right) \tag{6.22}$$

Tuy nhiên, cách này gặp vấn đề khi giá trị $w^T h + b$ vượt khỏi khoảng $[0, 1]$, vì lúc đó đạo hàm theo các tham số bằng 0, làm cho gradient descent không còn hiệu quả.

Giải pháp tốt hơn là dùng hàm sigmoid, vì nó luôn cho giá trị trong $(0, 1)$ và có gradient hợp lý khi đầu ra sai lệch nhiều.

Đơn vị đầu ra sigmoid được định nghĩa như sau:

$$\hat{y} = \sigma(w^T h + b) \tag{6.23}$$

với:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.24}$$

Tức là mô hình tính toán một giá trị $z = w^T h + b$, sau đó dùng sigmoid để chuyển z thành xác suất.

Để hiểu rõ hơn, ta có thể biểu diễn lại xác suất một cách chuẩn hóa:

$$\log \tilde{P}(y) = yz \tag{6.25}$$

$$\tilde{P}(y) = e^{yz} \tag{6.26}$$

$$P(y) = \frac{e^{yz}}{e^z + e^0} \tag{6.27}$$

$$P(y) = \sigma((2y - 1)z) \tag{6.28}$$

Dạng này là một ví dụ điển hình của phân phối xác suất được xác định bởi hàm mũ và chuẩn hóa — rất phổ biến trong thống kê. Ở đây, z còn được gọi là logit.

Hàm mất mát tương ứng khi dùng phương pháp hợp lý cực đại là:

$$J(\theta) = -\log P(y|x) \tag{6.29}$$
$$= -\log \sigma((2y-1)z) \tag{6.30}$$
$$= \zeta((1-2y)z) \tag{6.31}$$

Hàm mất mát này có một đặc tính rất hay: khi mô hình dự đoán đúng (ví dụ $y = 1$ và z lớn dương), gradient sẽ nhỏ. Nhưng khi dự đoán sai, gradient sẽ lớn, giúp mô hình điều chỉnh nhanh hơn. Điều này lý tưởng cho việc huấn luyện.

Ngược lại, nếu dùng MSE trong khi vẫn dùng sigmoid, mô hình có thể gặp vấn đề vì gradient trở nên rất nhỏ khi đầu ra bão hòa — gần 0 hoặc 1. Điều này làm quá trình học chậm và dễ kẹt ở các điểm không tối ưu.

Do đó, khi sử dụng sigmoid ở lớp đầu ra, người ta thường kết hợp với cross-entropy loss (tương đương với negative log-likelihood) để tối ưu hiệu quả.

Một lưu ý khi lập trình là không nên lấy log của $\hat{y} = \sigma(z)$ trực tiếp nếu không xử lý cẩn thận, vì $\hat{y}$ có thể tiến sát 0 và gây lỗi số. Tốt hơn là viết $\log \sigma(z)$ trực tiếp dưới dạng toán học như trên, thay vì lấy log sau khi đã tính sigmoid.

6.2.2.3 Hàm kích hoạt softmax cho phân phối Multinoulli

Trong nhiều bài toán phân loại, đầu ra cần là một phân phối xác suất trên n lớp rời rạc. Hàm softmax thường được dùng trong lớp đầu ra để mô hình hóa phân phối xác suất này. Về cơ bản, softmax là phần mở rộng của sigmoid, từ phân phối Bernoulli (2 lớp) sang phân phối Multinoulli (nhiều lớp).

Ví dụ, thay vì chỉ dự đoán xác suất $P(y = 1|x)$ như trong bài toán nhị phân:

$$\hat{y} = P(y = 1|x), \tag{6.32}$$

ta cần dự đoán một vector $\hat{y}$ với từng phần tử là:

$$\hat{y}_i = P(y = i|x), \tag{6.33}$$

sao cho $\hat{y}_i \in [0, 1]$ và $\sum_i \hat{y}_i = 1$.

Mô hình trước tiên tính toán một vector logit z bằng:

$$z = W^T h + b, \tag{6.34}$$

với $z_i = \log \tilde{P}(y = i|x)$ là các giá trị chưa chuẩn hóa.

Sau đó, hàm softmax được dùng để chuyển z thành phân phối xác suất:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{6.35}$$

Softmax được thiết kế để tối ưu log-likelihood:

$$\log P(y = i; z) = \log \text{softmax}(z)_i, \tag{6.36}$$

và có thể viết lại như:

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j). \tag{6.37}$$

Khi tối ưu, giá trị z_i của lớp đúng được tăng lên, trong khi tổng các z_j của các lớp còn lại bị đẩy xuống. Điều này đảm bảo mô hình học cách tập trung xác suất vào lớp đúng.

Trong thực tế, ta thường xấp xỉ:

$$\log \sum_j \exp(z_j) \approx \max_j z_j \tag{6.38}$$

để hiểu rằng softmax sẽ phạt mạnh khi mô hình dự đoán sai với xác suất cao.

Tuy nhiên, giống như sigmoid, softmax cũng có thể bão hòa: khi các giá trị z có sự chênh lệch lớn, đạo hàm sẽ rất nhỏ. Điều này gây khó khăn cho một số hàm mất mát như MSE. Một tính chất quan trọng là softmax không đổi khi cộng cùng một hằng số c vào toàn bộ z:

$$\text{softmax}(z) = \text{softmax}(z + c). \tag{6.39}$$

Tận dụng điều này, ta thường trừ đi giá trị lớn nhất trong z để tránh tràn số khi tính $\exp(z_i)$:

$$\text{softmax}(z) = \text{softmax}(z - \max_i z_i). \tag{6.40}$$

Nhờ các đặc tính trên, softmax trở thành một lựa chọn mặc định cho các bài toán phân loại nhiều lớp trong học sâu.

6.2.2.4 Các loại đầu ra khác trong mạng nơ-ron

Ba lớp đầu ra phổ biến trong mạng nơ-ron là tuyến tính, sigmoid và softmax. Tuy nhiên, tùy vào bài toán cụ thể, ta có thể xây dựng các loại đầu ra khác. Nguyên tắc chung là dựa vào nguyên lý hợp lý cực đại (maximum likelihood).

Giả sử mô hình xác suất có điều kiện:

$$p(y|x; \theta), \tag{6.41}$$

thì hàm mất mát tương ứng sẽ là:

$$-\log p(y|x; \theta). \tag{6.42}$$

Dự đoán tham số phân phối xác suất

Thay vì dự đoán trực tiếp giá trị của y, mô hình sẽ dự đoán các tham số ω của một phân phối xác suất trên y:

$$f(x;\theta) = \omega. \tag{6.43}$$

Hàm mất mát sẽ là:

$$-\log p(y;\omega(x)). \tag{6.44}$$

Học phương sai trong mô hình Gaussian điều kiện

Ví dụ, nếu y tuân theo phân phối Gaussian với trung bình $\mu(x)$ và phương sai σ^2, thì ước lượng cực đại hợp lý của σ^2 là:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu(x_i))^2. \tag{6.45}$$

Nếu muốn để phương sai phụ thuộc vào đầu vào x, ta có thể dùng thêm một đầu ra:

$$\sigma^2(x) = g(f(x;\theta)), \tag{6.46}$$

trong đó g là hàm đảm bảo đầu ra dương, ví dụ:

$$\zeta(a) = \log(1 + e^a) \tag{6.47}$$

(hàm *softplus*).

Mô hình như vậy gọi là heteroscedastic, vì phương sai thay đổi theo x. Đôi khi, người ta sử dụng ma trận chính xác (precision matrix) thay vì hiệp phương sai để tăng độ ổn định khi huấn luyện.

Mạng nơ-ron với đầu ra hỗn hợp Gaussian (Mixture Density Networks)

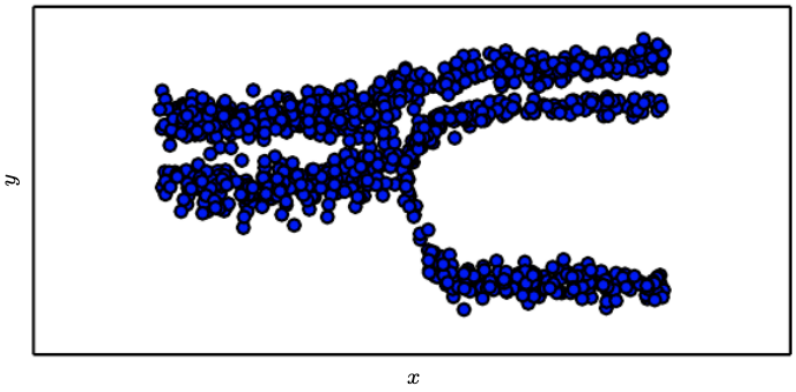
Có những bài toán mà đầu vào x có thể tương ứng với nhiều giá trị y hợp lý. Lúc này, một phân phối Gaussian đơn là không đủ. Giải pháp là sử dụng Gaussian Mixture Model (GMM):

$$p(y|x) = \sum_{i=1}^{n} p(c = i|x)\mathcal{N}(y;\mu^{(i)}(x),\Sigma^{(i)}(x)), \tag{6.48}$$

Trong đó:

- $p(c = i|x)$: xác suất chọn thành phần i, tính bằng softmax,
- $\mu^{(i)}(x)$: trung bình của thành phần Gaussian thứ i,
- $\Sigma^{(i)}(x)$: ma trận hiệp phương sai của thành phần i (thường giả định là đường chéo để đơn giản).

Khi huấn luyện, mạng nơ-ron cần dự đoán:



Hình 6.3: Mẫu đầu ra từ một mạng nơ-ron với lớp cuối là mô hình hỗn hợp Gaussian. Đầu vào x được lấy từ phân phối đều, đầu ra y được lấy mẫu từ $p_{\text{model}}(y|x)$. Mạng học cách ánh xạ đầu vào tới các tham số phân phối đầu ra — bao gồm xác suất của từng thành phần, trung bình và phương sai tương ứng.

1. Xác suất thành phần $p(c = i|x)$ (softmax).
2. Trung bình $\mu^{(i)}(x)$ cho từng thành phần.
3. Phương sai hoặc ma trận hiệp phương sai $\Sigma^{(i)}(x)$ (thường giả định là đường chéo).

Một số khó khăn có thể phát sinh như: nếu phương sai quá nhỏ, hàm mất mát có thể tạo ra gradient lớn bất thường. Các kỹ thuật phổ biến để xử lý gồm:

- Cắt gradient (gradient clipping),
- Điều chỉnh gradient tự động (gradient scaling).

6.3 Đơn vị ẩn

Một trong những yếu tố quan trọng nhất khi thiết kế mạng nơ-ron là lựa chọn loại đơn vị ẩn (hidden unit), tức là các nơ-ron trong lớp ẩn. Loại đơn vị ẩn được xác định bởi hàm kích hoạt mà nó sử dụng. Việc chọn đúng hàm kích hoạt ảnh hưởng rất lớn đến khả năng học và hiệu suất của mô hình.

Trong số các lựa chọn hiện nay, ReLU (Rectified Linear Unit) là hàm kích hoạt mặc định nhờ vào sự đơn giản và hiệu quả. Tuy nhiên, trong một số trường hợp, những hàm khác như sigmoid, tanh, hoặc các biến thể của ReLU có thể mang lại kết quả tốt hơn.

Mỗi đơn vị ẩn hoạt động bằng cách áp dụng một hàm phi tuyến $g(z)$ lên đầu ra của một phép biến đổi tuyến tính:

$$z = W^T x + b, \tag{6.49}$$

trong đó x là đầu vào, W là ma trận trọng số và b là hệ số dịch.

Một số hàm kích hoạt không khả vi tại một số điểm. Ví dụ, ReLU được định nghĩa như sau:

$$g(z) = \max(0, z). \tag{6.50}$$

Hàm này không khả vi tại $z = 0$ vì đạo hàm không xác định:

$$g'(z) = \begin{cases} 0, & z < 0, \\ 1, & z > 0. \end{cases} \tag{6.51}$$

Tuy nhiên, trong thực tế, thuật toán như gradient descent vẫn làm việc tốt vì nó không yêu cầu gradient chính xác tuyệt đối. Ngoài ra, các điểm như $z = 0$ rất hiếm gặp do sai số làm tròn số trong máy tính, nên việc chọn một phía đạo hàm là hợp lý.

Dù ReLU rất phổ biến, các hàm khác như sigmoid, tanh, Leaky ReLU hay PReLU vẫn có thể mang lại kết quả tốt hơn trong một số tình huống. Tốt nhất là thử nghiệm trực tiếp trên bài toán cụ thể để chọn lựa phù hợp.

6.3.1 Hàm kích hoạt ReLU và các biến thể

ReLU là một trong những hàm kích hoạt được ưa chuộng nhất hiện nay vì sự đơn giản và hiệu quả của nó:

$$g(z) = \max(0, z). \tag{6.52}$$

Khi đầu vào dương, ReLU giữ nguyên giá trị; còn nếu âm, nó đưa về 0. Điều này giúp mạng học nhanh hơn vì tránh được hiện tượng vanishing gradient mà sigmoid và tanh gặp phải.

Trong thực tế, ReLU thường được sử dụng ngay sau một phép biến đổi tuyến tính:

$$h = g(W^T x + b). \tag{6.53}$$

Một mẹo nhỏ khi khởi tạo mô hình là đặt các phần tử trong b là một giá trị nhỏ dương như 0.1, giúp đảm bảo các nơ-ron không bị "tắt"ngay từ đầu.

Tuy nhiên, ReLU có nhược điểm là gradient bằng 0 với đầu vào âm, gây ra hiện tượng *“dead neurons”*. Để khắc phục, nhiều biến thể đã được đề xuất:

- Leaky ReLU:
$$g(z) = \max(0, z) + \alpha \min(0, z), \tag{6.54}$$
với α nhỏ (ví dụ 0.01), giữ lại một phần gradient cho $z < 0$.
- Parametric ReLU (PReLU): giống Leaky ReLU, nhưng α được học trong quá trình huấn luyện.

- Absolute Value Rectification: một biến thể đặt $\alpha = -1$, dẫn đến:

$$g(z) = |z|, \tag{6.55}$$

  dùng trong các bài toán xử lý ảnh.

- Maxout Units: chia đầu vào thành nhóm, lấy giá trị lớn nhất trong mỗi nhóm:

$$g(z)_i = \max_{j \in G(i)} z_j. \tag{6.56}$$

  Cho phép mô hình học phi tuyến mềm theo từng đoạn, rất linh hoạt.

Các biến thể ReLU đều có chung nguyên lý: giữ cho hàm gần tuyến tính, giúp quá trình tối ưu bằng gradient hiệu quả hơn. Nguyên lý này áp dụng được cả trong mạng RNN, chẳng hạn LSTM dùng tổng tuyến tính để truyền trạng thái theo thời gian nhằm giảm suy giảm gradient.

Việc lựa chọn đúng hàm kích hoạt có thể giúp mô hình học nhanh hơn, hội tụ ổn định hơn và tránh các vấn đề như dead neurons hay mất gradient.

6.3.2 Hàm sigmoid và tanh (hyperbolic tangent)

Trước khi ReLU trở nên phổ biến, hai hàm kích hoạt chính được dùng trong các mạng nơ-ron là sigmoid logistic và tanh:

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.57}$$

$$g(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{6.58}$$

Hai hàm này có mối liên hệ:

$$\tanh(z) = 2\sigma(2z) - 1 \tag{6.59}$$

Điều này cho thấy tanh là phiên bản chuẩn hóa của sigmoid, với đầu ra nằm trong khoảng $[-1, 1]$ thay vì $[0, 1]$.

Hàm sigmoid logistic thường dùng ở lớp đầu ra trong phân loại nhị phân vì đầu ra có thể hiểu là xác suất. Tuy nhiên, khi dùng trong lớp ẩn, nó có hạn chế:

- Dễ bị bão hòa khi z lớn hoặc nhỏ, làm gradient gần như bằng 0, gây vanishing gradient.

- Không đối xứng quanh gốc: $\sigma(0) = 0.5$, khiến việc tối ưu khó khăn hơn.

Hàm tanh cải thiện phần nào so với sigmoid:

- Có tâm đối xứng: $\tanh(0) = 0$, giúp gradient cân bằng hơn.

- Có độ dốc lớn hơn gần gốc, lan truyền gradient tốt hơn.

Với đầu vào nhỏ, mạng nơ-ron dùng tanh có thể được xem gần giống mô hình tuyến tính:

$$\hat{y} = w^T \tanh(U^T \tanh(V^T x)) \tag{6.60}$$

Điều này giúp việc huấn luyện đơn giản hơn.

Mặc dù ReLU hiện nay phổ biến hơn, sigmoid và tanh vẫn có vai trò trong một số ứng dụng đặc biệt:

- RNNs: tanh và sigmoid dùng để kiểm soát thông tin qua thời gian.
- Mô hình xác suất: như Boltzmann Machines, RBMs, cần sigmoid để biểu diễn xác suất.
- Autoencoders: cần kiểm soát đầu ra nằm trong khoảng cố định.

Tóm lại, sigmoid và tanh tuy không còn phổ biến trong lớp ẩn của mạng truyền thẳng, nhưng vẫn hữu ích trong các mạng có tính xác suất hoặc cần ràng buộc đầu ra.

6.3.3 Các loại đơn vị ẩn khác

Ngoài các hàm kích hoạt phổ biến như ReLU, sigmoid hay tanh, mạng nơ-ron còn có thể sử dụng nhiều loại đơn vị ẩn khác, ít phổ biến hơn nhưng hữu ích trong những hoàn cảnh nhất định. Trong phần này, chúng ta sẽ tìm hiểu một số loại đơn vị ẩn đặc biệt và những ứng dụng tiềm năng của chúng.

6.3.3.1 Không dùng hàm kích hoạt

Một cách tiếp cận đơn giản là không sử dụng hàm phi tuyến, tức là sử dụng hàm đồng nhất $g(z) = z$. Đây chính là một đơn vị tuyến tính. Tuy nhiên, nếu toàn bộ mạng chỉ dùng các tầng tuyến tính, thì kết quả cuối cùng cũng chỉ là một phép biến đổi tuyến tính — không có khả năng biểu diễn các quan hệ phi tuyến.

Tuy nhiên, việc chèn thêm các tầng tuyến tính vào giữa các tầng phi tuyến vẫn có thể có lợi. Ví dụ, ta có thể phân rã một ma trận trọng số lớn thành hai ma trận nhỏ hơn:

$$h = g(V^\top U^\top x + b) \tag{6.61}$$

Trong đó, nếu U sinh ra đầu ra có kích thước q thì số lượng tham số sẽ giảm từ np xuống còn $(n + p)q$. Nếu $q \ll p$, điều này giúp giảm đáng kể số lượng tham số, dù phải đánh đổi bằng việc giới hạn mạng vào các phép biến đổi tuyến tính hạng thấp.

6.3.3.2 Dùng hàm softmax trong lớp ẩn

Hàm softmax thường được dùng ở lớp đầu ra để biểu diễn xác suất phân phối trên các lớp rời rạc. Tuy nhiên, trong một số kiến trúc nâng cao như mạng có bộ nhớ (LSTM, Transformer), hàm softmax cũng có thể xuất hiện trong các lớp ẩn. Trong các trường hợp

này, softmax giúp mô hình học cách lựa chọn, chú ý, hoặc tổng hợp thông tin từ nhiều nguồn theo cách có kiểm soát.

Ví dụ, trong mô hình attention của Transformer, softmax được dùng để tính trọng số giữa các vị trí khác nhau trong chuỗi đầu vào, từ đó hướng mô hình tập trung vào các phần quan trọng hơn.

6.3.3.3 Các loại đơn vị ẩn đặc biệt

Ngoài các hàm trên, còn một số hàm kích hoạt đặc biệt khác được sử dụng trong các tình huống cụ thể:

- Radial Basis Function (RBF): Được định nghĩa như sau:

$$h_i = \exp\left(-\frac{1}{\sigma_i^2}\|W_{:,i} - x\|^2\right) \tag{6.62}$$

  RBF có giá trị lớn khi đầu vào x gần với tâm $W_{:,i}$ và giảm nhanh khi x cách xa. Tuy nhiên, do dễ bị bão hòa về 0, các đơn vị RBF thường khó huấn luyện bằng gradient descent.

- Softplus: Là một phiên bản làm mượt của ReLU, có đạo hàm liên tục trên toàn bộ trục số thực:

$$g(a) = \log(1 + e^a) \tag{6.63}$$

  Dù lý thuyết ổn định hơn ReLU, thực nghiệm cho thấy ReLU thường cho kết quả tốt hơn về tốc độ hội tụ và hiệu suất tổng thể.

- Hard Tanh: Là một phiên bản đơn giản hóa của tanh, có giá trị bị cắt trong khoảng $[-1, 1]$:

$$g(a) = \max(-1, \min(1, a)) \tag{6.64}$$

  Hard Tanh có thể hữu ích khi muốn kiểm soát rõ ràng phạm vi đầu ra và tránh bão hòa nghiêm trọng hơn so với tanh.

Nghiên cứu và thiết kế các đơn vị ẩn vẫn là một chủ đề mở trong học sâu. Các hàm kích hoạt mới thường chỉ được áp dụng rộng rãi khi chứng minh được hiệu quả vượt trội rõ rệt trong thực nghiệm.

6.4 Thiết kế Kiến trúc Mạng Nơ-ron

Một thành phần cốt lõi trong xây dựng mạng nơ-ron là quyết định thiết kế kiến trúc của mạng. "Kiến trúc"ở đây đề cập đến tổng thể cấu trúc của mô hình, bao gồm:

- Số lượng tầng (depth),
- Số lượng đơn vị trong mỗi tầng (width),
- Cách các tầng kết nối với nhau (topology).

Phần lớn các mạng nơ-ron hiện đại sử dụng kiến trúc dạng chuỗi (sequential), tức là mỗi tầng là một hàm của đầu ra từ tầng trước. Trong kiến trúc này, mỗi tầng được định

nghĩa như:

$$\mathbf{h}^{(1)} = g^{(1)}\left(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)}\right), \tag{6.65}$$

$$\mathbf{h}^{(2)} = g^{(2)}\left(\mathbf{W}^{(2)\top}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right), \tag{6.66}$$

và tiếp tục như vậy cho đến tầng cuối.

Trong kiến trúc này, hai yếu tố đặc biệt quan trọng là:

- Độ sâu (depth): số lượng tầng trong mạng.
- Độ rộng (width): số lượng đơn vị trong mỗi tầng.

Lý thuyết cho thấy một mạng chỉ cần một tầng ẩn cũng đủ để xấp xỉ mọi hàm liên tục (Universal Approximation Theorem). Tuy nhiên, mạng sâu với nhiều tầng thường đạt hiệu quả tốt hơn — biểu diễn được các hàm phức tạp hơn với ít tham số hơn, đồng thời tổng quát hóa tốt hơn.

Tuy nhiên, mạng càng sâu thì càng khó huấn luyện do các vấn đề như gradient biến mất, bão hòa hay overfitting. Do đó, việc chọn kiến trúc phù hợp cần dựa vào thử nghiệm thực tế, kết hợp với theo dõi lỗi trên tập xác thực (validation set).

Trong các phần tiếp theo, chúng ta sẽ tìm hiểu thêm các kỹ thuật giúp xây dựng kiến trúc hiệu quả như regularization, normalization, dropout, và các kiến trúc mạng chuyên biệt như CNN, RNN hay Transformer.

6.4.1 Khả năng xấp xỉ hàm số và vai trò của độ sâu mạng

Các mô hình tuyến tính hoạt động bằng cách ánh xạ đầu vào đến đầu ra thông qua phép nhân ma trận. Do đó, chúng chỉ có thể biểu diễn các hàm tuyến tính. Ưu điểm của mô hình tuyến tính là rất dễ huấn luyện vì các hàm mất mát thường tạo ra bài toán tối ưu lồi. Tuy nhiên, trong thực tế, nhiều bài toán yêu cầu mô hình phải học được các mối quan hệ phi tuyến phức tạp.

Ban đầu, có thể nghĩ rằng để học một hàm phi tuyến cụ thể, ta phải thiết kế riêng một mô hình cho nó. Tuy nhiên, may mắn thay, mạng nơ-ron lan truyền tiến (feedforward networks) với một hoặc nhiều tầng ẩn đã chứng minh khả năng xấp xỉ phổ quát.

Định lý xấp xỉ phổ quát (Hornik et al., 1989; Cybenko, 1989) chỉ ra rằng: nếu ta có một mạng với ít nhất một tầng ẩn dùng hàm kích hoạt "nén"(ví dụ sigmoid), thì mạng đó có thể xấp xỉ bất kỳ hàm liên tục nào trên một tập hợp hữu hạn chiều với sai số tùy ý nhỏ, miễn là có đủ số lượng đơn vị ẩn. Ngoài ra, đạo hàm của mạng cũng có thể xấp xỉ đạo hàm thật của hàm mục tiêu (Hornik et al., 1990).

Chúng ta không cần hiểu chi tiết về khái niệm "hàm đo được Borel", chỉ cần biết rằng mọi hàm liên tục trên tập đóng và bị chặn trong $\mathbb{R}^n$ đều có thể được xấp xỉ bằng mạng nơ-ron.

Ngoài ra, mạng nơ-ron còn có thể xấp xỉ các ánh xạ từ một không gian rời rạc hữu hạn chiều này sang một không gian rời rạc khác. Về sau, các kết quả tương tự cũng được chứng minh với các hàm kích hoạt không bão hòa, như ReLU (Leshno et al., 1993).

Ý nghĩa thực tiễn: Dù ta có một hàm mục tiêu phức tạp đến đâu, luôn tồn tại một mạng MLP đủ lớn để xấp xỉ nó. Tuy nhiên, điều đó không đảm bảo rằng ta có thể thực sự học được hàm đó trong quá trình huấn luyện.

Có hai lý do khiến việc học có thể thất bại:

- Tối ưu kém: Thuật toán học không tìm ra bộ tham số phù hợp.
- Quá khớp (overfitting): Mạng học quá kỹ tập huấn luyện và không tổng quát hóa được ra dữ liệu mới.

Như đã nói trong Mục 5.2.1, định lý "không có bữa trưa miễn phí"(No Free Lunch) cho thấy: không có thuật toán nào là tốt nhất trong mọi tình huống. Mạng lan truyền tiến là hệ thống biểu diễn hàm rất linh hoạt, nhưng không có công thức chung nào giúp chọn đúng kiến trúc phù hợp cho mọi bài toán.

Mặc dù tồn tại mạng đủ lớn để xấp xỉ bất kỳ hàm nào, định lý không nói rõ cần bao nhiêu đơn vị ẩn. Theo Barron (1993), trong trường hợp xấu, số đơn vị ẩn có thể phải tăng theo cấp số mũ theo số chiều đầu vào n. Ví dụ, số lượng hàm nhị phân khả dĩ trên $\{0,1\}^n$ là 2^{2^n}, nên việc phân biệt hết các cấu hình cần đến $O(2^n)$ bậc tự do.

Tóm lại: Mạng một tầng ẩn có thể xấp xỉ bất kỳ hàm nào, nhưng có thể cần kích thước rất lớn, dẫn đến khó học và dễ overfit. Do đó, sử dụng mạng sâu (nhiều tầng) có thể giúp biểu diễn hiệu quả hơn.

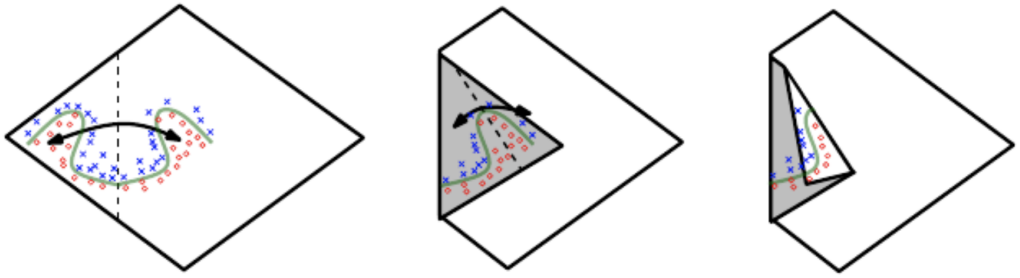
Có nhiều hàm mà chỉ khi dùng mạng sâu thì mới biểu diễn được một cách gọn nhẹ. Nếu dùng mạng nông (ít tầng), cần rất nhiều đơn vị để đạt cùng độ chính xác — thường phải tăng theo cấp số mũ theo số chiều đầu vào.

Những kết quả đầu tiên về việc độ sâu giúp tăng hiệu quả biểu diễn xuất phát từ lý thuyết mạch logic (Håstad, 1986), sau đó mở rộng sang mạng nơ-ron với đơn vị ngưỡng và trọng số không âm (Håstad và Goldmann, 1991; Hajnal et al., 1993), và cả với hàm kích hoạt liên tục (Maass, 1992; Maass et al., 1994).

Ngày nay, ReLU là hàm kích hoạt phổ biến. Leshno et al. (1993) đã chứng minh rằng mạng nông dùng ReLU vẫn có tính chất xấp xỉ phổ quát. Tuy nhiên, điều này không cho ta biết mạng cần rộng bao nhiêu, và không nêu rõ vai trò của độ sâu.

Montufar et al. (2014) đã chỉ ra rằng với các hàm tuyến tính từng đoạn như ReLU hay maxout, số vùng tuyến tính mà mạng tạo ra có thể tăng theo cấp số mũ theo độ sâu. Điều này cho thấy mạng sâu có thể biểu diễn nhiều cấu trúc hơn so với mạng nông có cùng số đơn vị.

Hình 6.5 minh họa cách mạng dùng hàm giá trị tuyệt đối (absolute value) có thể "gấp" không gian đầu vào để tạo ra bản sao phản chiếu của một mẫu. Khi dùng nhiều tầng, số



Hình 6.4: Minh họa trực quan về lợi thế của mạng sâu sử dụng hàm kích hoạt ReLU. (Trái) Một đơn vị giá trị tuyệt đối tạo ra đối xứng trong không gian đầu vào. (Giữa) Một tầng ẩn có thể "gấp" không gian. (Phải) Tầng tiếp theo lặp lại mẫu đã gấp, tạo ra biểu diễn phức tạp hơn. Theo Montufar et al. (2014).

lần "gấp" tăng lên, tạo ra các mẫu phức tạp hơn.

Kết quả chính từ Montufar et al. (2014) chỉ ra rằng: mạng ReLU có d đầu vào, l tầng ẩn và n đơn vị mỗi tầng có thể tạo ra tối đa:

$$O\left(\left(\frac{nd}{d}\right)^{d}(l-1)nd\right)$$

vùng tuyến tính trong không gian đầu vào — tăng theo cấp số mũ của l.

Với mạng maxout có k bộ lọc mỗi đơn vị, số vùng tuyến tính có thể đạt:

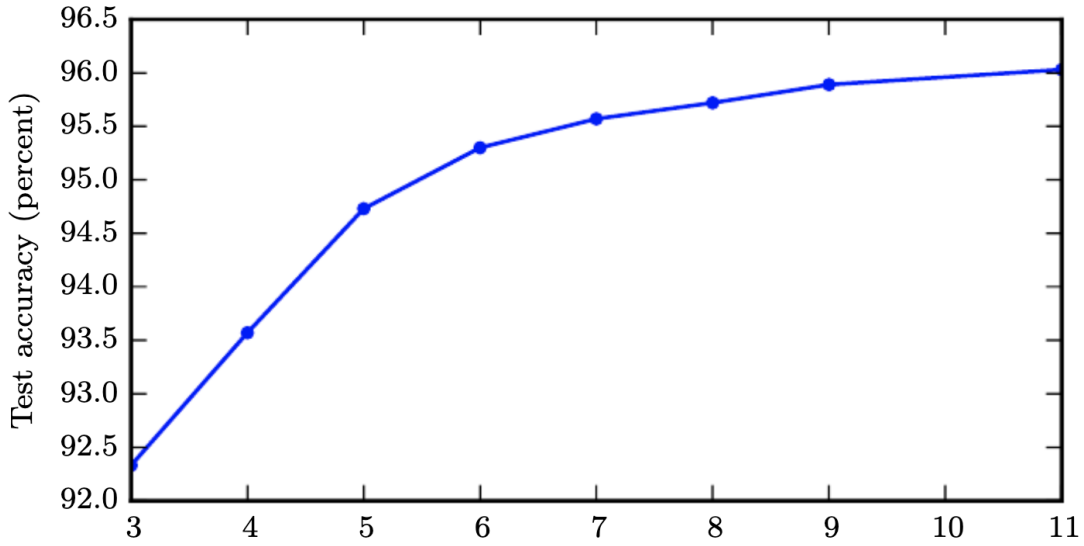
$$O\left(k^{(l-1)+d}\right)$$

Tuy nhiên, không phải mọi hàm thực tế đều phù hợp với cách biểu diễn của mạng sâu. Nhưng từ góc nhìn học biểu diễn (representation learning), mạng sâu rất có ích vì có thể phát hiện và tổ chức lại các yếu tố tiềm ẩn trong dữ liệu theo nhiều lớp trừu tượng.

Thêm vào đó, mạng sâu giống như một chương trình gồm nhiều bước — mỗi tầng xử lý đầu ra của tầng trước, tạo ra một chuỗi các biểu diễn trung gian có thể mang tính trừu tượng cao hơn.

Nhiều nghiên cứu đã chỉ ra rằng mạng sâu thường có khả năng tổng quát hóa tốt hơn trên nhiều tác vụ học máy (Bengio et al., 2007; Erhan et al., 2009; Krizhevsky et al., 2012; Goodfellow et al., 2014d; v.v.).

Hình 6.6 và Hình 6.7 sẽ cho thấy kết quả thực nghiệm chứng minh mạng sâu học hiệu quả hơn, xác nhận rằng việc sử dụng kiến trúc nhiều tầng là một hướng đi hợp lý để cải thiện khả năng học của mô hình.



Hình 6.5: Ảnh hưởng của độ sâu mạng. Kết quả thực nghiệm cho thấy mạng sâu có khả năng tổng quát hóa tốt hơn khi áp dụng cho tác vụ phiên âm các số nhiều chữ số từ hình ảnh địa chỉ. Độ chính xác trên tập kiểm tra tiếp tục tăng khi tăng độ sâu mạng. Dữ liệu từ Goodfellow et al. (2014).

6.4.2 Các cân nhắc về kiến trúc

Cho đến nay, ta đã mô tả mạng nơ-ron như là một chuỗi các tầng đơn giản — trong đó độ sâu và độ rộng là hai yếu tố quan trọng nhất. Tuy nhiên, trong thực tế, các kiến trúc mạng nơ-ron hiện đại thường phức tạp hơn nhiều.

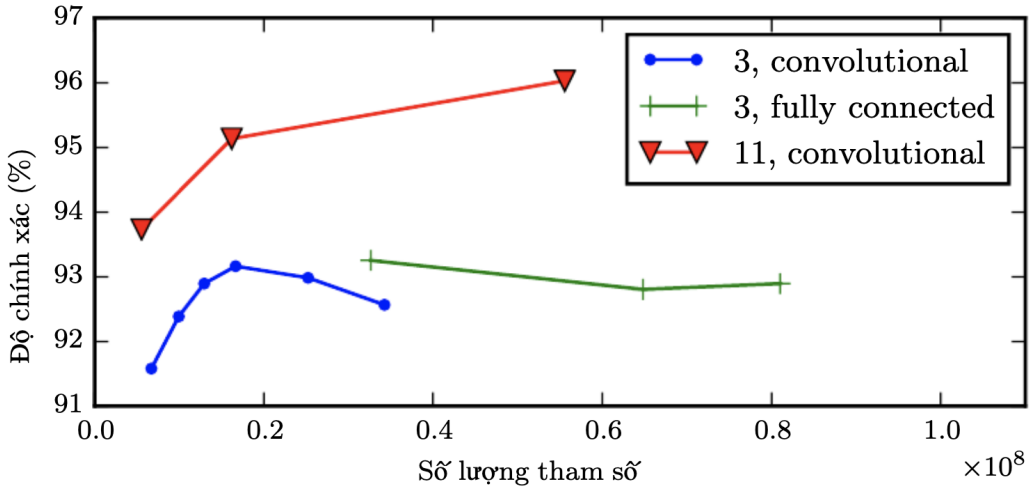
Trong thực tế, nhiều kiến trúc mạng đã được phát triển để giải quyết những loại bài toán cụ thể. Ví dụ:

- Mạng tích chập (Convolutional Networks): Tối ưu cho xử lý ảnh — sẽ được trình bày ở Chương 9.
- Mạng hồi tiếp (Recurrent Neural Networks): Phù hợp với dữ liệu chuỗi — sẽ thảo luận ở Chương 10.

Không phải tất cả các tầng trong mạng đều cần được kết nối theo kiểu chuỗi. Một số kiến trúc hiện đại sử dụng chuỗi chính, nhưng bổ sung thêm kết nối bỏ qua (skip connections), chẳng hạn như từ tầng i đến tầng $i + 2$ hoặc xa hơn. Những kết nối này giúp gradient lan truyền hiệu quả hơn về phía đầu vào trong quá trình huấn luyện.

Một điểm cần quan tâm khác là cách kết nối giữa các tầng. Trong mạng mặc định, tất cả đơn vị đầu vào đều được kết nối với mọi đơn vị đầu ra (fully-connected). Tuy nhiên, các mạng chuyên biệt — như mạng tích chập — thường chỉ kết nối một phần các đơn vị, giúp giảm số tham số và chi phí tính toán.

Chiến lược giảm số kết nối này hiệu quả khi được thiết kế phù hợp với đặc trưng của bài toán. Ví dụ, trong thị giác máy tính, các điểm ảnh gần nhau thường liên quan hơn so với các điểm ở xa, nên chỉ cần kết nối cục bộ (local connections) như trong mạng tích chập.



Hình 6.6: Tác động của số lượng tham số. Hình ảnh cho thấy mô hình sâu có xu hướng hoạt động tốt hơn không chỉ do số tham số lớn mà còn nhờ vào độ sâu. Việc tăng số tham số trong các tầng mà không tăng độ sâu không mang lại cải thiện đáng kể. Dữ liệu từ Goodfellow et al. (2014).

Tóm lại, không có lời khuyên "một công thức cho mọi tình huống"trong thiết kế kiến trúc mạng. Việc chọn kiến trúc phù hợp cần dựa vào kinh nghiệm, thử nghiệm, và hiểu biết về bản chất bài toán — các chương sau sẽ hướng dẫn chi tiết hơn trong từng lĩnh vực ứng dụng.

6.5 Thuật toán lan truyền ngược và các thuật toán vi phân khác

Khi ta đưa đầu vào x vào một mạng nơ-ron truyền thẳng để sinh ra đầu ra $\hat{y}$, dữ liệu sẽ đi theo chiều từ đầu vào đến đầu ra qua các tầng của mạng — quá trình này gọi là lan truyền tiến (forward propagation).

Trong quá trình huấn luyện, đầu ra $\hat{y}$ được đưa vào một hàm chi phí $J(\theta)$ để đánh giá sai số. Mục tiêu là giảm giá trị của J, và để làm được điều đó, ta cần tính gradient của J đối với các tham số θ của mạng.

Thuật toán lan truyền ngược (back-propagation, gọi tắt là backprop) do Rumelhart et al. (1986a) đề xuất, cho phép lan truyền lỗi ngược qua các tầng của mạng để tính toán gradient hiệu quả. Việc tính gradient bằng tay cho mô hình lớn là cực kỳ tốn tài nguyên. Thuật toán backprop tối ưu hoá quy trình này bằng cách sử dụng các kết quả trung gian đã tính trong quá trình lan truyền tiến, từ đó giúp tiết kiệm đáng kể thời gian và bộ nhớ.

Lưu ý: backprop không phải là toàn bộ thuật toán học. Nó chỉ là một phần dùng để tính gradient. Việc sử dụng gradient để cập nhật tham số thường do thuật toán khác đảm nhiệm, chẳng hạn gradient descent hoặc stochastic gradient descent (SGD).

Ngoài ra, backprop không chỉ áp dụng cho mạng nơ-ron nhiều tầng mà có thể áp dụng để tính gradient của bất kỳ hàm số khả vi nào — hoặc thậm chí trả về thông báo nếu hàm không khả vi.

Giả sử ta cần tính gradient $\nabla_x f(x, y)$, trong đó x là tập biến cần đạo hàm, còn y là

biến đầu vào phụ không cần đạo hàm. Trong học máy, gradient phổ biến nhất là:

$$\nabla_\theta J(\theta)$$

Backprop có thể được mở rộng để tính các loại đạo hàm khác, chẳng hạn Jacobian trong trường hợp nhiều đầu ra. Tuy nhiên, trong phần này, ta chỉ tập trung vào trường hợp phổ biến nhất: hàm có một đầu ra vô hướng.

6.5.1 Đồ thị tính toán

Để hiểu chính xác cách hoạt động của thuật toán lan truyền ngược, ta cần biểu diễn mạng nơ-ron dưới dạng một đồ thị tính toán (computation graph).

Trong đồ thị này:

- Mỗi nút (node) biểu diễn một biến — có thể là số vô hướng, véc-tơ, ma trận, tensor, v.v.
- Mỗi cạnh (edge) biểu diễn mối quan hệ phụ thuộc giữa các biến thông qua các phép toán.
- Mỗi phép toán nhận một hoặc nhiều biến làm đầu vào và tạo ra một biến đầu ra.

Mỗi phép toán chỉ sinh ra một biến đầu ra duy nhất (không mất tính tổng quát, vì đầu ra có thể là véc-tơ). Trong thực tế, nhiều framework như TensorFlow hay PyTorch hỗ trợ phép toán với nhiều đầu ra, nhưng để đơn giản hóa trình bày, ta giả định chỉ có một đầu ra.

Ví dụ, nếu biến y được tính từ biến x qua một phép toán, ta biểu diễn bằng một cạnh có hướng từ x đến y. Có thể ghi nhãn phép toán trên cạnh hoặc ngay tại nút y.

Các ví dụ minh hoạ sẽ được trình bày trong Hình 6.8.

6.5.2 Quy tắc chuỗi trong vi phân

Quy tắc chuỗi trong vi phân (Chain Rule of Calculus) là một công cụ toán học quan trọng dùng để tính đạo hàm của các hàm hợp — tức là những hàm được tạo thành từ sự kết hợp của nhiều hàm con đơn giản hơn. Thuật toán lan truyền ngược (backpropagation) chính là một ứng dụng hiệu quả của quy tắc chuỗi, được sắp xếp theo cách tối ưu để giảm chi phí tính toán.

Giả sử $x \in \mathbb{R}$, và hai hàm số f và g ánh xạ từ $\mathbb{R}$ đến $\mathbb{R}$. Nếu định nghĩa:

$$y = g(x), \quad z = f(g(x)) = f(y),$$

thì theo quy tắc chuỗi:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{6.44}$$

Quy tắc chuỗi cũng áp dụng cho trường hợp hàm nhiều biến. Giả sử:

$$x \in \mathbb{R}^m, \quad y \in \mathbb{R}^n,$$

với $g : \mathbb{R}^m \to \mathbb{R}^n$, và $f : \mathbb{R}^n \to \mathbb{R}$, ta có:

$$y = g(x), \quad z = f(y).$$

Lúc này, đạo hàm riêng của z theo từng thành phần x_i được tính như sau:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \tag{6.45}$$

Ta có thể viết gọn dưới dạng vector như sau:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z \tag{6.46}$$

Trong đó:

- $\nabla_x z$ là gradient của z theo x,
- $\frac{\partial y}{\partial x}$ là ma trận Jacobian của $y = g(x)$ có kích thước $n \times m$.

Từ công thức (6.46), ta thấy rằng để tính gradient của z theo x, ta chỉ cần lấy gradient của z theo y, rồi nhân với ma trận Jacobian của y theo x. Thuật toán lan truyền ngược hoạt động bằng cách lặp lại bước này qua từng phép toán trong đồ thị tính toán.

Quy tắc chuỗi cho tensor

Trong thực tế, các mạng nơ-ron thường làm việc với dữ liệu dạng tensor nhiều chiều — chứ không chỉ là vector hoặc ma trận. Tuy nhiên, cách tính gradient vẫn tương tự: ta chỉ cần "làm phẳng"tensor thành vector, tính gradient theo quy tắc chuỗi, rồi đưa kết quả trở lại đúng hình dạng ban đầu.

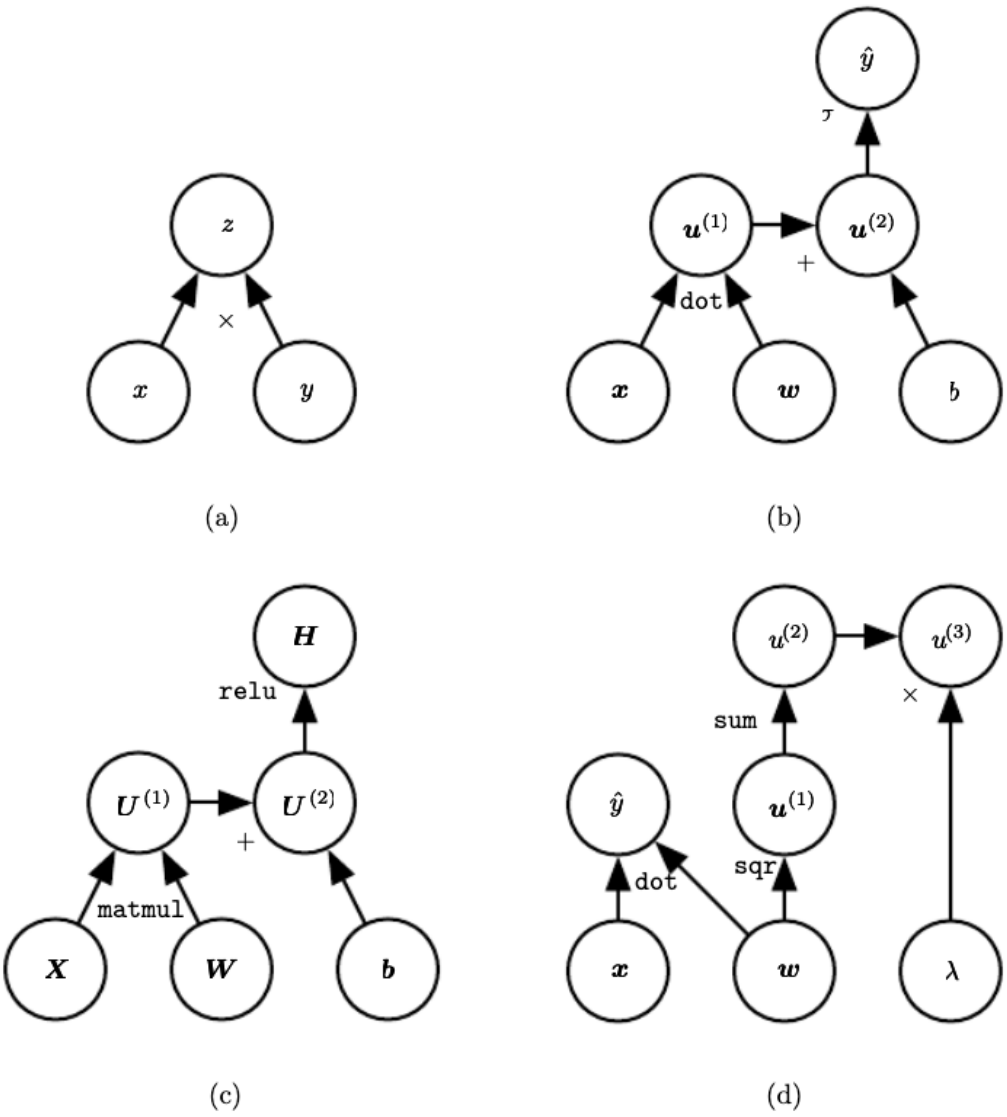
Ký hiệu gradient của một đại lượng vô hướng z theo một tensor X là:

$$\nabla_X z$$

Mỗi phần tử của X được xác định bởi một bộ chỉ số (ví dụ: (i_1, i_2, i_3) nếu X là tensor 3 chiều). Ta có thể ký hiệu tổ hợp chỉ số này bằng một biến duy nhất i, khi đó:

$$(\nabla_X z)_i = \frac{\partial z}{\partial X_i}$$

Tương tự như với véc-tơ:



Hình 6.7: Một số ví dụ về đồ thị tính toán: (a) Biểu diễn phép toán nhân $z = x \cdot y$; (b) Đồ thị của mô hình hồi quy logistic $\hat{y} = \sigma(x^\top w + b)$; (c) Biểu thức $H = \max\{0, XW + b\}$ mô tả các giá trị kích hoạt của tầng ReLU trong một mini-batch; (d) Ví dụ về việc cùng một biến (trọng số w) được sử dụng cho cả dự đoán và điều chuẩn bằng regularization.

$$(\nabla_x z)_i = \frac{\partial z}{\partial x_i}$$

Dưới dạng tổng quát, quy tắc chuỗi cho tensor có thể viết là:

$$\nabla_X z = \sum_j (\nabla_X Y_j) \cdot \frac{\partial z}{\partial Y_j} \tag{6.47}$$

Trong đó:

- $Y = g(X)$ là phép biến đổi từ tensor X,
- $z = f(Y)$ là đại lượng vô hướng cần tối ưu.

Tóm lại, dù ta làm việc với số vô hướng, véc-tơ, ma trận hay tensor, thì cốt lõi của thuật toán lan truyền ngược vẫn dựa trên quy tắc chuỗi — cụ thể là tích giữa Jacobian và gradient được lặp lại tại từng bước của đồ thị tính toán.

6.5.3 Áp dụng quy tắc chuỗi đệ quy để tính lan truyền ngược

Khi áp dụng quy tắc chuỗi, ta có thể dễ dàng viết ra công thức tính gradient của một đại lượng vô hướng theo bất kỳ nút nào trong đồ thị tính toán đã sinh ra giá trị đó. Tuy nhiên, khi thực hiện trên máy tính, ta cần cân nhắc đến hiệu quả thực tế của việc tính toán.

Trong nhiều trường hợp, cùng một phép tính có thể xuất hiện nhiều lần trong quá trình lan truyền ngược. Ta có thể chọn cách lưu trữ kết quả trung gian để tái sử dụng, hoặc tính lại chúng nhiều lần. Điều này được minh họa trong Hình 6.9. Với các đồ thị đơn giản, việc tính lại có thể chấp nhận được, nhưng với đồ thị lớn, việc tính lại nhiều lần sẽ gây lãng phí tài nguyên đáng kể. Ngược lại, lưu trữ lại cũng tiêu tốn bộ nhớ. Tùy vào yêu cầu của hệ thống, ta sẽ có các chiến lược cân bằng giữa thời gian và bộ nhớ.

Để giải thích rõ hơn, ta xét một đồ thị tính toán mà đỉnh cuối cùng là một giá trị vô hướng $u^{(n)}$ (ví dụ như hàm mất mát), và ta cần tính đạo hàm của nó theo các đầu vào $u^{(1)}, \ldots, u^{(ni)}$, tức:

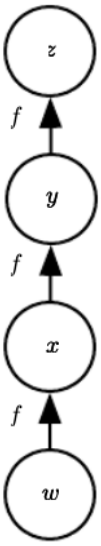
$$\frac{\partial u^{(n)}}{\partial u^{(i)}} \quad \text{với mọi } i \in \{1, 2, \ldots, ni\}.$$

Trong đó, các $u^{(1)}$ đến $u^{(ni)}$ thường là các tham số của mô hình.

Giả sử các nút trong đồ thị được sắp xếp theo thứ tự lan truyền tiến: từ $u^{(ni+1)}$ đến $u^{(n)}$. Mỗi nút $u^{(i)}$ là kết quả của một phép toán $f^{(i)}$, tính trên tập các nút cha $A^{(i)}$:

$$u^{(i)} = f^{(i)}(A^{(i)}). \tag{6.48}$$

Quá trình lan truyền ngược sẽ tạo ra một đồ thị con B, trong đó mỗi nút tương ứng với một nút trong đồ thị gốc G, nhưng tính đạo hàm $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. Quá trình này tuân theo quy tắc



Hình 6.8: Một ví dụ về tính toán lặp lại trong quá trình lan truyền ngược. Giả sử ta có một chuỗi phép toán tuần tự: $x = f(w), \quad y = f(x), \quad z = f(y)$, thì đạo hàm theo quy tắc chuỗi là: $\frac{\partial z}{\partial w} = f'(y) \cdot f'(x) \cdot f'(w)$. Nếu ta không lưu lại giá trị trung gian như $x = f(w)$, $y = f(x)$, thì phải tính lại nhiều lần, gây lãng phí. Thuật toán lan truyền ngược xử lý vấn đề này bằng cách lưu trữ các kết quả trung gian, giúp tiết kiệm thời gian. Tuy nhiên, nếu hạn chế bộ nhớ, ta có thể chọn phương án tính lại.

chuỗi:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in P_a(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \cdot \frac{\partial u^{(i)}}{\partial u^{(j)}}, \tag{6.53}$$

với $P_a(u^{(i)})$ là tập các nút cha của $u^{(i)}$.

Lan truyền ngược đi qua mỗi cạnh của đồ thị đúng một lần, do đó chi phí tính toán tỷ lệ tuyến tính với số cạnh. Điều này giúp tiết kiệm tài nguyên đáng kể so với cách tính thủ công từng đạo hàm từ đầu.

Trong thực tế, ta có thể mở rộng cách làm này cho các nút mang giá trị tensor (thay vì chỉ là vô hướng). Các thuật toán lan truyền ngược được thiết kế để tận dụng lại giá trị trung gian, tránh việc tính toán lặp lại không cần thiết.

Trong các phần sau, ta sẽ cụ thể hóa hơn quá trình lan truyền ngược cho các mạng cụ thể như MLP (mạng nhiều lớp kết nối đầy đủ), và sau đó mở rộng ra phiên bản tổng quát có thể xử lý mọi đồ thị tính toán. Đây cũng là nền tảng của hầu hết các thư viện học sâu hiện đại.

6.5.4 Đạo hàm từ ký hiệu đến ký hiệu

Các biểu thức đại số và đồ thị tính toán đều làm việc với các ký hiệu, tức là các biến chưa mang giá trị cụ thể. Khi huấn luyện hoặc sử dụng mạng nơ-ron, ta sẽ cung cấp các giá trị cụ thể cho các ký hiệu này. Ví dụ, thay vì một biến đầu vào x, ta có thể dùng giá trị

Algorithm 6.1 Lan truyền tiến để tính toán đầu ra $u^{(n)}$ từ các đầu vào $u^{(1)}, \ldots, u^{(n_i)}$.

Input: vector đầu vào x
Output: giá trị đầu ra $u^{(n)}$
for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $A^{(i)} \leftarrow \{u^{(j)} : j \in P_a(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(A^{(i)})$
end for
Return: $u^{(n)}$

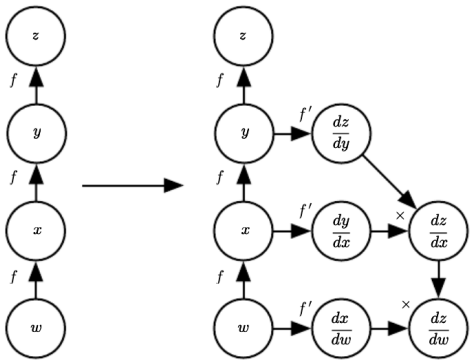
Algorithm 6.2 Phiên bản đơn giản của lan truyền ngược tính đạo hàm của $u^{(n)}$ theo các đầu vào.

Input: Đồ thị tính toán $u^{(1)}, \ldots, u^{(n)}$ (tất cả là vô hướng)
Output: $\left\{ \frac{\partial u^{(n)}}{\partial u^{(i)}} \mid i = 1, \ldots, ni \right\}$

1. Thực hiện lan truyền tiến (theo thuật toán 6.1) để tính các giá trị $u^{(i)}$
2. Tạo bảng `grad_table` lưu $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ cho mọi $u^{(i)}$
3. Gán `grad_table`$[u^{(n)}] \leftarrow 1$
4. For $j = n - 1$ down to 1:
   - Tính:

$$\texttt{grad\_table}[u^{(j)}] \leftarrow \sum_{i:j \in P_a(u^{(i)})} \texttt{grad\_table}[u^{(i)}] \cdot \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

5. Return: `grad_table`$[u^{(1)}], \ldots,$ `grad_table`$[u^{(ni)}]$



Hình 6.9: Phương pháp từ ký hiệu đến ký hiệu: Một đồ thị tính toán mới được sinh ra mô tả cách tính đạo hàm của biểu thức ban đầu. (Trái): Đồ thị ban đầu biểu diễn biểu thức $z = f(f(f(w)))$. (Phải): Đồ thị mới biểu diễn đạo hàm $\frac{dz}{dw}$. Phương pháp này cho phép tiếp tục tính đạo hàm bậc cao hơn.

cụ thể như $[1{,}2,\ 3{,}765,\ -1{,}8]^T$.

Có hai cách tiếp cận phổ biến trong việc tính đạo hàm trong học sâu: đạo hàm từ ký hiệu đến số và đạo hàm từ ký hiệu đến ký hiệu.

Đạo hàm từ ký hiệu đến số (symbol-to-number differentiation) là phương pháp mà hệ thống nhận đầu vào là một đồ thị tính toán kèm theo các giá trị cụ thể tại các nút, và trả về các giá trị số cụ thể của gradient tại các điểm đó. Đây là cách tiếp cận được sử dụng trong các thư viện như `Torch` và `Caffe`.

Đạo hàm từ ký hiệu đến ký hiệu (symbol-to-symbol differentiation) là phương pháp tổng quát hơn: thay vì tính toán ngay giá trị của đạo hàm, hệ thống sẽ xây dựng thêm một phần của đồ thị tính toán mới — phần này mô tả cách tính đạo hàm, dưới dạng ký hiệu. Khi cần, ta có thể cung cấp các giá trị số cụ thể vào đồ thị này để tính ra kết quả.

Ưu điểm của phương pháp từ ký hiệu đến ký hiệu là các đạo hàm cũng được biểu diễn dưới dạng đồ thị, cho phép áp dụng lại lan truyền ngược để tính đạo hàm bậc hai, bậc ba,... Tuỳ thuộc vào yêu cầu, ta có thể chỉ cần xây dựng đồ thị, hoặc vừa xây dựng vừa thực thi với dữ liệu cụ thể.

Trong cuốn sách này, chúng ta sử dụng phương pháp từ ký hiệu đến ký hiệu. Lan truyền ngược sẽ được trình bày như một quá trình xây dựng đồ thị đạo hàm. Đồ thị này sau đó có thể được đánh giá tại bất kỳ thời điểm nào, bằng cách cung cấp các giá trị cụ thể cho các đầu vào.

Thực chất, phương pháp từ ký hiệu đến số là một trường hợp đặc biệt của phương pháp từ ký hiệu đến ký hiệu — chính là quá trình thực thi các phép toán đã được mô tả trong đồ thị đạo hàm. Sự khác biệt chính là ở việc có — hoặc không — giữ lại cấu trúc của đồ thị tính toán trong quá trình tính toán.

6.5.5 Lan truyền ngược tổng quát

Thuật toán lan truyền ngược, xét về bản chất, là một hiện thực hóa hiệu quả của quy tắc chuỗi trong vi phân, được tổ chức một cách hệ thống để tính đạo hàm của một đại lượng vô hướng z theo bất kỳ biến tổ tiên x nào trong đồ thị tính toán.

Khởi đầu, ta gán đạo hàm của z theo chính nó bằng 1:

$$\frac{dz}{dz} = 1$$

Sau đó, gradient sẽ được truyền ngược qua các phép toán trong đồ thị bằng cách nhân gradient hiện tại với ma trận Jacobian của phép toán tương ứng. Khi có nhiều đường dẫn đi từ z đến cùng một nút tổ tiên, ta cộng tất cả các gradient tại nút đó lại.

Mỗi nút trong đồ thị G biểu diễn một biến (tensor), có thể là vô hướng, vector, ma trận hoặc tensor nhiều chiều. Để thực hiện lan truyền ngược tổng quát, mỗi nút V trong đồ thị cần đi kèm các phương thức:

- `get_operation(V)`: trả về phép toán tạo ra V.
- `get_inputs(V, G)`: trả về danh sách các biến đầu vào của phép toán tạo ra V.
- `get_consumers(V, G)`: trả về danh sách các biến sử dụng V làm đầu vào.

Mỗi phép toán `op` cần định nghĩa một hàm `bprop`, chịu trách nhiệm tính gradient truyền ngược qua phép toán. Cụ thể, `bprop` thực hiện tích giữa Jacobian và gradient từ đầu ra:

$$\texttt{op.bprop(inputs, X, G)} = \sum_i (\nabla_X \texttt{op.f(inputs)}_i)\, G_i \tag{6.54}$$

Trong đó:

- `inputs`: danh sách biến đầu vào.
- `op.f`: phép toán toán học.
- X: đầu vào cụ thể mà ta đang tính gradient theo.
- G: gradient của đầu ra.

Ví dụ: Với phép toán $C = AB$ (nhân ma trận), nếu gradient $\frac{\partial z}{\partial C} = G$ đã biết, thì:

$$\frac{\partial z}{\partial A} = GB^\top, \quad \frac{\partial z}{\partial B} = A^\top G$$

Để lan truyền ngược, thuật toán sẽ tuần tự gọi `bprop` cho từng phép toán trong đồ thị.

Algorithm 6.3 Cấu trúc ngoài của lan truyền ngược

Require: T: Tập các biến cần tính gradient
Require: G: Đồ thị tính toán
Require: z: Biến đầu ra vô hướng
1: Cắt tỉa G thành G': chỉ giữ các nút tổ tiên của z và con cháu của các nút trong T
2: Khởi tạo bảng gradient: `grad_table`
3: `grad_table`[z] $\leftarrow 1$
4: **for** mỗi biến $V \in T$ **do**
5: Gọi `build_grad`(V, G, G', `grad_table`)
6: **end forreturn** `grad_table` (chỉ giữ các phần tử trong T)

Algorithm 6.4 Hàm đệ quy `build_grad` tính gradient cho một biến

Require: V: Biến cần tính gradient
Require: G, G': Đồ thị đầy đủ và đồ thị con
Require: `grad_table`: Bảng lưu gradient
1: **if** gradient của V đã có **then return** `grad_table`[V]
2: **end if**
3: $i \leftarrow 1$
4: **for** mỗi $C \in$ `get_consumers`(V, G') **do**
5: `op` $\leftarrow$ `get_operation`(C)
6: $D \leftarrow$ `build_grad`$(C, G, G', \texttt{grad\_table})$
7: $G(i) \leftarrow$ `op.bprop(get_inputs`$(C, G'), V, D)$
8: $i \leftarrow i + 1$
9: **end for**
10: $G \leftarrow \sum_i G(i)$
11: `grad_table`[V] $\leftarrow G$ **return** G

Kết cấu trên giúp thuật toán lan truyền ngược hoạt động với mọi loại đồ thị tính toán bất kỳ, miễn là mỗi phép toán đều định nghĩa đúng phương thức `bprop`. Đây chính là nguyên lý đằng sau các hệ thống autodiff (tự động vi phân) hiện đại như TensorFlow, PyTorch hay JAX.

6.5.6 Ví dụ: Lan truyền ngược trong Huấn luyện MLP

Để minh họa cụ thể cách thuật toán lan truyền ngược hoạt động, ta xét một ví dụ đơn giản về huấn luyện mạng perceptron đa lớp (MLP) với một lớp ẩn duy nhất. Quá trình huấn luyện sử dụng thuật toán giảm dần gradient (gradient descent) theo mini-batch.

Cấu trúc mô hình:

- Đầu vào: một mini-batch các ví dụ được biểu diễn dưới dạng ma trận thiết kế X, mỗi hàng là một ví dụ.
- Nhãn: một vector y chứa các nhãn lớp tương ứng.
- Lớp ẩn: tính toán đặc trưng ẩn $H = \max(0, XW^{(1)})$ sử dụng hàm ReLU.
- Đầu ra: log xác suất chưa chuẩn hoá $Z = HW^{(2)}$.

- Hàm mất mát: entropy chéo giữa y và xác suất được chuẩn hoá từ Z, ký hiệu là J_{MLE}.

Để đơn giản, ta bỏ qua vector độ lệch (biases). Hàm mất mát tổng thể bao gồm entropy chéo và điều chuẩn L2 (weight decay):

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left| W_{i,j}^{(1)} \right|^2 + \sum_{i,j} \left| W_{i,j}^{(2)} \right|^2 \right) \tag{6.56}$$

trong đó:

- J_{MLE}: entropy chéo, đo độ sai khác giữa phân phối thực và phân phối mô hình dự đoán,
- λ: hệ số điều chuẩn, điều chỉnh mức độ phạt vào trọng số lớn để giảm overfitting.

Các bước tính gradient:

1. Lan truyền xuôi: tính toán giá trị tại các nút theo thứ tự từ đầu vào đến đầu ra.
2. Lan truyền ngược: bắt đầu từ hàm chi phí J, tính gradient của J đối với các trọng số $W^{(1)}$ và $W^{(2)}$ thông qua đồ thị.

Mỗi bước lan truyền ngược qua một nút được tính bằng quy tắc chuỗi:

$$\nabla_{W^{(i)}} J = \left(\frac{\partial z}{\partial W^{(i)}} \right)^\top \nabla_z J,$$

trong đó z là đầu ra của phép toán sử dụng $W^{(i)}$.

Ý nghĩa thực tiễn:

Ví dụ trên cho thấy sức mạnh của lan truyền ngược — cho dù mạng có phức tạp đến đâu, chỉ cần mô hình được xây dựng thành một đồ thị tính toán đúng đắn, gradient có thể được tính tự động và chính xác. Điều này giải phóng nhà phát triển khỏi gánh nặng viết đạo hàm thủ công, đặc biệt hữu ích khi mô hình ngày càng lớn và phức tạp hơn.

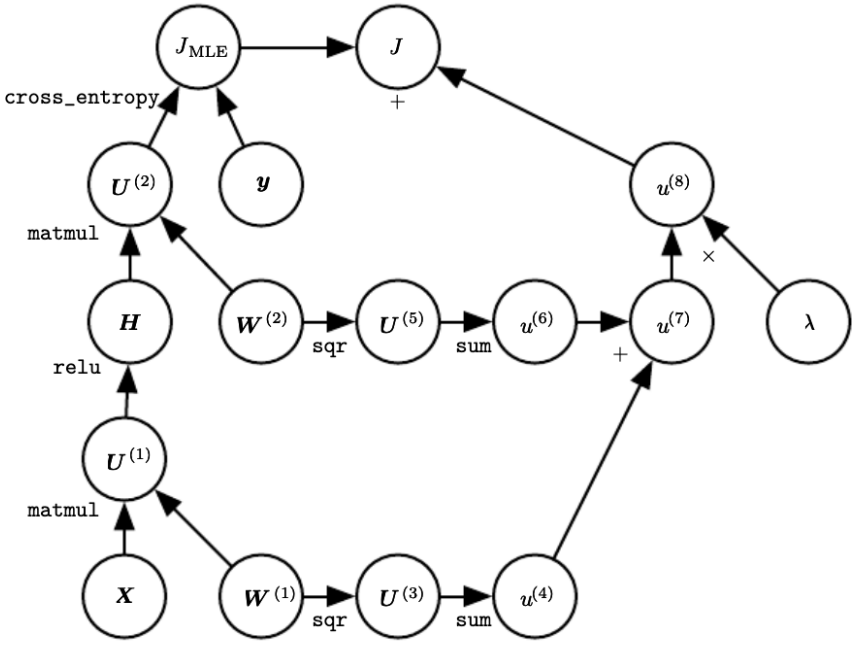
Chúng ta có thể theo dõi sơ bộ cách thức hoạt động của thuật toán lan truyền ngược bằng cách nhìn vào đồ thị lan truyền xuôi trong Hình 6.11. Mục tiêu của quá trình huấn luyện là tính toán gradient của hàm chi phí theo hai tập trọng số: $\nabla_{W^{(1)}} J$ và $\nabla_{W^{(2)}} J$.

Có hai nhánh riêng biệt trong đồ thị dẫn từ chi phí J quay trở lại các trọng số:

- Nhánh 1: đi qua thành phần chi phí điều chuẩn (weight decay), rất đơn giản — đóng góp của nhánh này là $2\lambda W^{(i)}$ đối với từng trọng số $W^{(i)}$.
- Nhánh 2: đi qua hàm mất mát entropy chéo, phức tạp hơn. Gọi G là gradient của log xác suất chưa chuẩn hoá $U^{(2)}$, được cung cấp bởi phép toán `cross_entropy`.

Lan truyền ngược qua nhánh entropy chéo:

1. Từ G, sử dụng quy tắc lan truyền ngược cho tham số thứ hai của phép toán nhân ma



Hình 6.10: Đồ thị tính toán cho một mạng MLP một lớp sử dụng hàm mất mát cross-entropy và điều chuẩn L2. Quá trình lan truyền ngược sẽ tự động tính toán gradient của hàm chi phí đối với từng trọng số trong mô hình.

trận $HW^{(2)}$, ta tính được:

$$\nabla_{W^{(2)}} J = H^\top G$$

2. Đồng thời, lan truyền tiếp gradient theo hướng từ H quay lại mạng:

$$\nabla_H J = GW^{(2)\top}$$

3. Tiếp theo, gradient $\nabla_H J$ được đi qua hàm ReLU. Ta áp dụng quy tắc lan truyền ngược của ReLU: làm bằng 0 các phần tử tương ứng với những phần tử $U^{(1)}$ âm. Gọi kết quả sau bước này là:

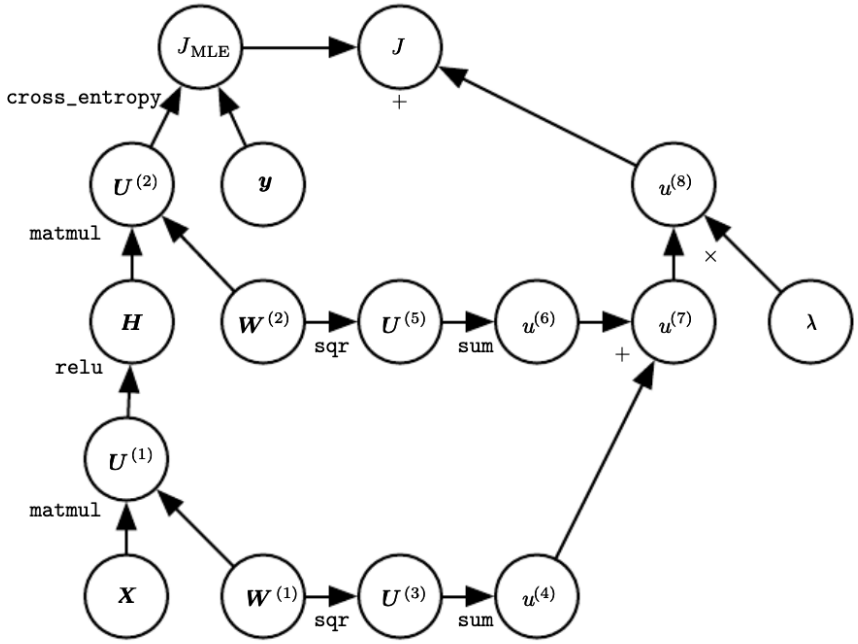
$$G' = \texttt{ReLU\_backward}(U^{(1)}, \nabla_H J)$$

4. Cuối cùng, gradient đối với $W^{(1)}$ được tính bằng quy tắc lan truyền ngược cho phép toán nhân ma trận $XW^{(1)}$:

$$\nabla_{W^{(1)}} J = X^\top G'$$

Sau khi tất cả các gradient được tính toán, ta sử dụng một thuật toán tối ưu như *gradient descent* để cập nhật các tham số $W^{(1)}$ và $W^{(2)}$.

Chi phí tính toán: Với mỗi phép nhân ma trận trong lan truyền xuôi và lan truyền ngược, số lượng phép toán là $O(w)$, với w là tổng số trọng số trong mạng. Điều này bao gồm:



Hình 6.11: Đồ thị tính toán trong quá trình huấn luyện MLP. Mỗi mũi tên cho thấy hướng lan truyền gradient trong quá trình lan truyền ngược. Các nhánh khác nhau từ hàm chi phí tổng quay lại từng trọng số được xử lý riêng biệt và tổng hợp lại.

- $O(mn_h d)$: từ $XW^{(1)}$, với d là số chiều đầu vào và n_h là số đơn vị ẩn,
- $O(mn_h n_o)$: từ $HW^{(2)}$, với n_o là số lớp đầu ra.

Chi phí bộ nhớ: Thuật toán cần lưu trữ giá trị đầu vào của các phép toán phi tuyến (ở đây là ReLU) để sử dụng trong lan truyền ngược. Với m là kích thước mini-batch và n_h là số đơn vị ẩn, bộ nhớ cần thiết là:

$$O(mn_h)$$

Do đó, lan truyền ngược không chỉ hiệu quả về mặt tính toán mà còn hợp lý về yêu cầu bộ nhớ, đặc biệt với các mạng nhỏ hoặc trung bình. Tuy nhiên, với mạng lớn hoặc rất sâu, chiến lược quản lý bộ nhớ (như recomputation hoặc gradient checkpointing) có thể cần được áp dụng.

6.5.7 Những khó khăn thực tiễn

Mô tả về thuật toán lan truyền ngược trong phần này đơn giản hơn nhiều so với các triển khai thực tế. Trong phần trình bày, mỗi phép toán chỉ trả về một tensor duy nhất. Tuy nhiên, trên thực tế, nhiều phép toán có thể trả về nhiều hơn một tensor. Ví dụ, để đồng thời lấy giá trị lớn nhất trong một tensor và chỉ số của nó, ta thường kết hợp cả hai đầu ra trong một phép toán duy nhất nhằm tối ưu hiệu suất.

Ngoài ra, việc kiểm soát tiêu thụ bộ nhớ cũng là một thách thức lớn. Trong các hệ thống thực tế, thay vì cộng dồn các gradient riêng lẻ rồi cộng tổng sau cùng, ta có thể sử dụng một bộ đệm duy nhất và cập nhật nó từng bước — một chiến lược giúp giảm đáng

kể chi phí bộ nhớ.

Các hệ thống thực tế còn phải hỗ trợ nhiều kiểu dữ liệu khác nhau như số thực 32-bit, 64-bit, hoặc số nguyên. Việc xử lý đúng cách từng loại dữ liệu đòi hỏi sự cẩn trọng trong thiết kế.

Một vấn đề khác là không phải mọi phép toán đều có đạo hàm xác định trong toàn bộ miền đầu vào. Hệ thống cần có cơ chế theo dõi và xử lý các trường hợp đạo hàm không xác định hoặc không tồn tại. Tất cả các yếu tố này khiến việc hiện thực hoá thuật toán lan truyền ngược trở nên phức tạp hơn lý thuyết đơn giản.

6.5.8 Sự khác biệt trong và ngoài cộng đồng Deep Learning

Cộng đồng deep learning đã phát triển một cách nhìn riêng về việc tính toán đạo hàm, khác biệt phần nào với lĩnh vực tính đạo hàm tự động nói chung. Thuật toán lan truyền ngược chỉ là một trường hợp đặc biệt của kỹ thuật tích luỹ theo chế độ ngược (reverse mode accumulation), trong khi nhiều kỹ thuật khác đánh giá biểu thức con của quy tắc chuỗi theo những thứ tự khác nhau.

Việc chọn thứ tự đánh giá sao cho chi phí tối thiểu là một bài toán NP-hoàn chỉnh (Naumann, 2008). Vì vậy, thay vì tìm lời giải tối ưu tuyệt đối, các thư viện như Theano hay TensorFlow sử dụng heuristic — chẳng hạn như khớp mẫu biểu thức — để đơn giản hóa đồ thị tính toán của lan truyền ngược.

Ví dụ, xét hàm mất mát cross-entropy được xây dựng từ hàm softmax:

$$q_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{6.67}$$

$$J = -\sum_i p_i \log q_i \tag{6.68}$$

Một nhà toán học biết rằng:

$$\frac{\partial J}{\partial z_i} = q_i - p_i$$

Nhưng thuật toán lan truyền ngược sẽ truyền gradient qua từng phép toán $\exp$, $\sum$, $\log$, nên không tự động đơn giản hóa biểu thức như trên. Một số hệ thống có thể làm điều đó thông qua tối ưu hóa đồ thị.

Chi phí tính toán: Với mỗi đạo hàm riêng phần được tính một lần, tổng chi phí của lan truyền ngược có cùng bậc với lan truyền tiến, cụ thể là $O(\#\text{cạnh})$ của đồ thị. Tuy nhiên, ta có thể giảm chi phí nếu tối ưu hóa đồ thị gradient, dù việc này cũng là một bài toán NP-hoàn chỉnh.

6.5.9 So sánh giữa chế độ ngược và chế độ tiến

Thuật toán lan truyền ngược (reverse mode) được tối ưu cho các hàm có đầu ra vô hướng và nhiều đầu vào. Ngược lại, tích lũy theo chế độ tiến (forward mode) lại phù hợp với các hàm có nhiều đầu ra hơn đầu vào.

Mối quan hệ giữa hai chế độ này có thể minh họa bằng cách nhân chuỗi ma trận:

$$ABCD \tag{6.58}$$

Nếu D là một vectơ cột và A có nhiều hàng, thì nhân từ phải sang trái (backward mode) sẽ chỉ bao gồm các phép nhân ma trận–vectơ, rẻ hơn nhiều so với các phép nhân ma trận–ma trận. Ngược lại, nếu A là ma trận mỏng (nhiều cột, ít hàng), thì forward mode sẽ hiệu quả hơn.

6.5.10 Đạo hàm tự động trên mã nguồn với đồ thị tính toán

Trong cộng đồng học máy, đạo hàm tự động thường được hiện thực thông qua các cấu trúc dữ liệu đồ thị rõ ràng (explicit computation graphs). Các thư viện như TensorFlow, PyTorch, JAX yêu cầu người dùng viết mô hình bằng tập các phép toán có sẵn, mỗi phép toán đi kèm định nghĩa `bprop` riêng.

Cách tiếp cận này có lợi thế là cho phép tuỳ chỉnh quy tắc gradient cho từng phép toán, hỗ trợ tối ưu tốc độ và độ ổn định — điều khó đạt được nếu sử dụng hệ thống đạo hàm tự động hoàn toàn (ví dụ như chuyển đổi mã Python thành mã tính gradient).

Ngược lại, nhiều hệ thống ngoài deep learning như trong kỹ thuật hoặc vật lý — sử dụng đạo hàm tự động bằng cách phân tích mã nguồn viết bằng Python, C, hay Fortran. Phương pháp này linh hoạt hơn nhưng khó đạt hiệu suất tối ưu trong các mạng nơ-ron lớn.

6.5.11 Đạo hàm bậc cao

Một số thư viện deep learning hiện đại hỗ trợ việc tính toán đạo hàm bậc cao hơn, tức là các đạo hàm có bậc lớn hơn 1. Cụ thể, các framework như Theano và TensorFlow cho phép người dùng khai thác khả năng tính toán này. Những thư viện này sử dụng cùng một loại cấu trúc dữ liệu để biểu diễn các biểu thức đạo hàm như chính các hàm ban đầu, nhờ đó đạo hàm của đạo hàm có thể được xử lý bằng cùng một cơ chế lan truyền ngược (backpropagation) hoặc lan truyền tiến (forward-mode differentiation).

Trong bối cảnh deep learning, việc cần đến toàn bộ đạo hàm bậc hai của một hàm vô hướng là khá hiếm. Thay vào đó, điều mà ta thường quan tâm là một số đặc tính cụ thể của ma trận Hessian — chẳng hạn như một vài trị riêng lớn nhất, hoặc việc nhân Hessian với một vectơ bất kỳ.

Giả sử ta có một hàm số:

$$f : \mathbb{R}^n \to \mathbb{R}$$

khi đó ma trận Hessian $H \in \mathbb{R}^{n \times n}$ được định nghĩa là:

$$H = \nabla^2 f(x)$$

Tuy nhiên, trong thực tế các mô hình deep learning thường có đến hàng triệu hoặc hàng tỉ tham số, nên việc lưu trữ hoặc thao tác trực tiếp trên toàn bộ Hessian là không khả thi.

6.5.11.1 Phép nhân Hessian–Vector

Một giải pháp hiệu quả hơn là chỉ tính phép nhân giữa Hessian và một vectơ v:

$$Hv$$

Kỹ thuật phổ biến để thực hiện phép toán này là sử dụng phép đạo hàm bậc hai có hướng, hay cụ thể hơn là kỹ thuật từ Christianson (1992):

$$Hv = \nabla_x \left(\nabla_x f(x)\right)^\top v$$

Trong biểu thức trên: - Đầu tiên, ta tính đạo hàm của $f(x)$ — tạo thành gradient (một véc-tơ), - Sau đó, ta lấy đạo hàm của tích vô hướng giữa gradient đó và véc-tơ v.

Điều này có thể được thực hiện nhờ các thư viện hỗ trợ tự động vi phân — vốn có khả năng phân biệt rõ các hướng đạo hàm và cho phép xây dựng phép toán đạo hàm lồng nhau.

Lưu ý kỹ thuật: Nếu v là một véc-tơ được tạo ra như một phần của đồ thị tính toán ban đầu, phần mềm vi phân cần được hướng dẫn không lan truyền gradient qua quá trình tạo ra v. Điều này nhằm đảm bảo rằng v được xem là một hằng số trong quá trình tính đạo hàm bậc hai.

6.5.11.2 Tái tạo toàn bộ Hessian

Trong một số trường hợp đặc biệt, ta vẫn có thể tính toàn bộ Hessian bằng cách lặp lại phép nhân Hessian–Vector nhiều lần. Cụ thể, nếu $e(i)$ là một véc-tơ one-hot với phần tử thứ i bằng 1 và các phần tử còn lại bằng 0, thì:

$$He(i)$$

là cột thứ i của ma trận Hessian. Bằng cách lặp lại thao tác này với mọi $i = 1, \ldots, n$, ta có thể thu được toàn bộ ma trận Hessian.

Tuy nhiên, do chi phí tính toán và bộ nhớ là quá lớn với các mô hình hiện đại, phương pháp này hầu như không được sử dụng trong thực tế mà chỉ có giá trị trong nghiên cứu lý thuyết hoặc với các mô hình nhỏ.

6.6 Nhìn lại lịch sử

Mạng feedforward có thể được coi là những bộ xấp xỉ hàm phi tuyến hiệu quả, dựa trên việc sử dụng gradient descent để tối thiểu hóa lỗi trong việc xấp xỉ một hàm. Từ góc nhìn này, mạng feedforward hiện đại là sự kết tinh của nhiều thế kỷ tiến bộ trong nhiệm vụ xấp xỉ hàm tổng quát.

Quy tắc chuỗi, nền tảng của thuật toán back-propagation, được phát minh vào thế kỷ 17 (Leibniz, 1676; L'Hôpital, 1696). Tính toán vi phân và đại số đã được sử dụng từ lâu để giải quyết các bài toán tối ưu hóa dưới dạng đóng, nhưng gradient descent chỉ được

giới thiệu như một kỹ thuật để xấp xỉ giải pháp cho các bài toán tối ưu hóa theo từng bước vào thế kỷ 19 (Cauchy, 1847).

Bắt đầu từ những năm 1940, các kỹ thuật xấp xỉ hàm này đã được sử dụng để thúc đẩy các mô hình học máy như perceptron. Tuy nhiên, các mô hình ban đầu chủ yếu dựa trên các mô hình tuyến tính. Các nhà phê bình, như Marvin Minsky, đã chỉ ra những điểm yếu của các mô hình tuyến tính, như việc không thể học hàm XOR. Điều này dẫn đến một thời kỳ hoài nghi đối với mạng nơ-ron. Việc học các hàm phi tuyến yêu cầu phát triển một mạng nhiều lớp (multi-layer perceptron) cùng với phương pháp tính gradient hiệu quả.

Quy tắc chuỗi áp dụng cho mạng nhiều lớp lần đầu tiên được áp dụng hiệu quả trong các ứng dụng điều khiển (Kelley, 1960; Bryson và Denham, 1961; Dreyfus, 1962; Bryson và Ho, 1969; Dreyfus, 1973) và trong phân tích độ nhạy (Linnainmaa, 1976). Werbos (1981) đã đề xuất áp dụng kỹ thuật này để huấn luyện mạng nơ-ron nhân tạo. Sau đó, ý tưởng này được phát triển độc lập bởi LeCun (1985), Parker (1985), và Rumelhart et al. (1986a), với ứng dụng cụ thể trong việc huấn luyện các mạng nhiều lớp thông qua *back-propagation*.

Sự phổ biến của thuật toán lan truyền ngược bắt đầu với cuốn *Parallel Distributed Processing* (Rumelhart et al., 1986b), trong đó trình bày nhiều thí nghiệm thành công với backprop. Các tác giả như Rumelhart và Hinton đã góp phần đáng kể vào việc hình thành phong trào connectionism — nhấn mạnh vai trò kết nối trong nhận thức và học tập, bao gồm cả khái niệm *biểu diễn phân tán* (distributed representation) (Hinton et al., 1986).

Phong trào mạng nơ-ron phát triển mạnh vào đầu thập niên 1990 nhưng sau đó chững lại khi các phương pháp học máy khác nổi lên. Cuộc cách mạng học sâu hiện đại bắt đầu trở lại từ năm 2006, nhờ vào ba yếu tố: dữ liệu lớn hơn, phần cứng mạnh hơn, và phần mềm hiệu quả hơn.

Tuy cốt lõi thuật toán của mạng feedforward không thay đổi nhiều từ những năm 1980, nhiều cải tiến đã giúp cải thiện đáng kể hiệu suất. Một trong những thay đổi then chốt là thay thế hàm mất mát MSE bằng cross-entropy, phù hợp hơn với xác suất và nguyên lý cực đại hợp lý. Điều này đặc biệt quan trọng khi sử dụng sigmoid hoặc softmax ở đầu ra.

Một thay đổi quan trọng khác là chuyển từ đơn vị kích hoạt sigmoid sang ReLU — đơn vị tuyến tính từng phần, giúp tránh hiện tượng bão hòa và cải thiện gradient. ReLU vốn đã được đề xuất trong các mô hình như cognitron (Fukushima, 1975) nhưng bị bỏ quên trong một thời gian dài do lo ngại tính không khả vi. Tuy nhiên, các nghiên cứu từ Jarrett et al. (2009) và Glorot et al. (2011a) đã làm sống lại ReLU và chứng minh lợi ích thực tiễn của nó trong huấn luyện mạng sâu.

Sự hồi sinh của ReLU cũng cho thấy ảnh hưởng từ khoa học thần kinh: (1) nơ-ron sinh học thường không kích hoạt với đầu vào nhỏ, (2) chúng hoạt động tuyến tính với một số đầu vào, và (3) chúng có xu hướng hoạt động thưa (sparse activation).

Trong giai đoạn 2006–2012, mạng feedforward thường được hỗ trợ bởi các mô hình xác

suất. Nhưng sau năm 2012, với sự thành công của các mô hình như AlexNet (Krizhevsky et al., 2012), học dựa trên gradient trong mạng feedforward đã được công nhận là một công nghệ mạnh mẽ, đủ khả năng giải quyết nhiều bài toán mà trước đây đòi hỏi phải có sự hỗ trợ của các kỹ thuật khác.

Ngày nay, mạng feedforward đóng vai trò trung tâm trong nhiều kiến trúc học sâu — từ autoencoder, mạng đối kháng sinh (GAN), cho đến các mô hình sinh như Transformer. Trong tương lai, các mạng này hứa hẹn sẽ tiếp tục mở rộng ứng dụng với sự hỗ trợ của các kỹ thuật tối ưu hóa mới, kiến trúc hiệu quả hơn, và sự tích hợp tốt hơn giữa học giám sát và không giám sát.

Chương tiếp theo sẽ trình bày cách sử dụng và huấn luyện các mô hình mạng nơ-ron này một cách chi tiết.

CHƯƠNG 7. CHÍNH QUY HÓA TRONG HỌC SÂU

7.1 Chính quy hóa (Regularization)

Trong học sâu, chính quy hóa (regularization) là một công cụ quan trọng giúp ngăn chặn mô hình học quá kỹ dữ liệu huấn luyện — hay còn gọi là overfitting — và làm tăng khả năng áp dụng mô hình vào dữ liệu mới. Chính quy hóa hoạt động bằng cách giới hạn khả năng "uốn cong"của mô hình, thường thông qua việc thêm một số ràng buộc hoặc hình phạt vào quá trình huấn luyện.

Có nhiều cách để chính quy hóa mô hình. Một số phương pháp được thiết kế để phản ánh kiến thức sẵn có về bài toán hoặc dữ liệu. Một số khác thì nhằm khuyến khích mô hình đơn giản hơn, vì mô hình càng đơn giản thì càng dễ tổng quát hóa. Đôi khi, các ràng buộc từ regularization còn giúp biến một bài toán không rõ ràng thành bài toán có thể giải được. Một phương pháp chính quy hóa khác, được gọi là ensemble methods, kết hợp nhiều mô hình lại để đưa ra dự đoán ổn định hơn.

Trong học sâu, phần lớn các kỹ thuật regularization tập trung vào việc điều chỉnh cách các ước lượng (estimators) học từ dữ liệu. Mục tiêu của regularization là tạo ra một sự đánh đổi hợp lý giữa độ chệch (bias) và phương sai (variance). Một regularizer tốt sẽ giúp giảm phương sai đáng kể mà không làm tăng độ chệch quá nhiều.

Trong Chương 5, ta đã thảo luận về ba trường hợp của quá trình học:

1. Mô hình quá đơn giản, không học được bản chất dữ liệu (underfitting),
2. Mô hình phù hợp tốt với dữ liệu thật,
3. Mô hình quá phức tạp, khớp cả nhiễu trong dữ liệu huấn luyện (overfitting).

Regularization được dùng để đưa mô hình từ trạng thái thứ ba về trạng thái thứ hai.

Trên thực tế, một mô hình phức tạp không có nghĩa là nó sẽ phù hợp với quá trình sinh dữ liệu thật. Trong học máy, ta hầu như không bao giờ biết rõ quá trình sinh dữ liệu thật là gì. Và vì vậy, không thể đảm bảo rằng mô hình ta dùng là đúng hoặc gần đúng. Tuy nhiên, trong nhiều ứng dụng như xử lý ảnh, âm thanh, hay văn bản, quá trình sinh dữ liệu thực tế rất phức tạp — đến mức gần như không thể mô hình hóa chính xác được. Điều này khiến cho việc kiểm soát độ phức tạp của mô hình trở thành một yếu tố cực kỳ quan trọng.

Vì vậy, việc thiết kế mô hình học tốt không đơn giản là chọn mô hình nhỏ hơn hay ít tham số hơn, mà là chọn mô hình lớn và đủ linh hoạt nhưng được regularize hiệu quả để tránh overfitting.

7.2 Phạt chuẩn tham Số

Chính quy hóa không phải là khái niệm mới. Trước cả thời đại học sâu, các mô hình như hồi quy tuyến tính hay logistic đã sử dụng những phương pháp regularization đơn giản mà hiệu quả.

Một cách tiếp cận phổ biến là thêm một thuật ngữ phạt vào hàm mất mát, ký hiệu là $\Omega(\theta)$, để kiểm soát mức độ phức tạp của mô hình. Khi đó, hàm mục tiêu mới trở thành:

$$\tilde{J}(\theta; \mathbf{X}, \mathbf{y}) = J(\theta; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\theta), \tag{7.1}$$

Ở đây:

- J là hàm mất mát gốc trên dữ liệu huấn luyện,
- $\Omega(\theta)$ là phần phạt thêm vào,
- α là hệ số điều chỉnh, kiểm soát mức độ ảnh hưởng của phần phạt.

Nếu $\alpha = 0$, ta không sử dụng chính quy hóa. Khi α tăng, thuật toán sẽ ưu tiên các mô hình đơn giản hơn bằng cách giới hạn độ lớn của tham số.

Trong quá trình huấn luyện, thuật toán sẽ cố gắng giảm cả lỗi trên dữ liệu lẫn kích thước của tham số θ. Cách lựa chọn $\Omega(\theta)$ sẽ ảnh hưởng trực tiếp đến cách mô hình học.

Một điểm lưu ý quan trọng trong mạng nơ-ron là: chính quy hóa thường chỉ áp dụng lên trọng số (weights) chứ không áp dụng lên bias. Điều này là vì:

- Trọng số quyết định cách các đầu vào tương tác, trong khi bias chỉ ảnh hưởng đến từng nơ-ron riêng lẻ.
- Nếu phạt cả bias, có thể làm mô hình mất khả năng học tốt (underfitting).

Do đó, trong thực hành, ta thường áp dụng thuật ngữ phạt chỉ lên vector trọng số $\mathbf{w}$, không phải toàn bộ θ.

Cũng có thể dùng các hệ số phạt khác nhau cho từng tầng của mạng, nhưng cách làm phổ biến là dùng cùng một hệ số α cho tất cả các tầng — nhằm đơn giản hóa việc điều chỉnh siêu tham số.

7.2.1 Phạt Chuẩn L^2 (Weight Decay)

Một cách phổ biến để điều chỉnh mô hình nhằm tránh hiện tượng quá khớp là sử dụng phương pháp phạt chuẩn L^2, còn được gọi là weight decay (suy giảm trọng số). Phương pháp này kéo các trọng số của mô hình tiến gần về gốc tọa độ[1] bằng cách thêm một thành phần phạt vào hàm mất mát:

$$\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2 \tag{7.2}$$

Từ đó, hàm mất mát được điều chỉnh trở thành:

[1] Thực ra, ta có thể điều chỉnh trọng số về bất kỳ điểm nào trong không gian, không nhất thiết là gốc tọa độ. Nhưng vì ta thường không biết trước trọng số nên là dương hay âm, nên việc đưa trọng số về 0 là lựa chọn hợp lý và phổ biến nhất. Vì vậy, trong phần này ta chỉ tập trung vào trường hợp trọng số được kéo về gốc tọa độ.

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \tag{7.3}$$

Gradient của hàm mất mát sau khi thêm weight decay là:

$$\nabla\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\mathbf{w} + \nabla J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \tag{7.4}$$

Áp dụng quy tắc gradient descent để cập nhật trọng số, ta có:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(\alpha\mathbf{w} + \nabla J(\mathbf{w}; \mathbf{X}, \mathbf{y})). \tag{7.5}$$

Ta có thể viết lại theo cách gộp hệ số như sau:

$$\mathbf{w} \leftarrow (1 - \eta\alpha)\mathbf{w} - \eta\nabla J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \tag{7.6}$$

Dễ thấy, thuật ngữ weight decay làm giảm nhẹ trọng số ở mỗi bước cập nhật trước khi thực hiện bước gradient descent thông thường. Điều này là hiệu ứng tức thì trong một bước. Nhưng điều gì sẽ xảy ra sau nhiều vòng huấn luyện?

Để dễ phân tích, ta dùng xấp xỉ bậc hai quanh điểm tối ưu $\mathbf{w}^*$ (trọng số tối ưu khi không dùng điều chuẩn):

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w}). \tag{7.7}$$

Với các bài toán như hồi quy tuyến tính, hàm mất mát vốn đã là bậc hai, nên xấp xỉ này hoàn toàn chính xác:

$$\hat{J}(\theta) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T H(\mathbf{w} - \mathbf{w}^*), \tag{7.8}$$

trong đó:

- $\mathbf{H}$ là ma trận Hessian tại điểm $\mathbf{w}^*$.
- Vì $\mathbf{w}^*$ là điểm cực tiểu nên $\nabla J(\mathbf{w}^*) = 0$.
- Do đó, hàm chỉ còn thành phần bậc hai.
- Hơn nữa, vì là điểm cực tiểu nên $\mathbf{H}$ là ma trận nửa xác định dương.

Điều kiện để $\hat{J}$ đạt cực tiểu là:

$$\nabla_{\mathbf{w}}\hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \tag{7.7}$$

Giờ ta thêm thành phần weight decay $\alpha\mathbf{w}$ vào gradient, ta có:

$$\alpha\tilde{\mathbf{w}} + \mathbf{H}(\tilde{\mathbf{w}} - \mathbf{w}^*) = 0, \tag{7.9}$$

$$(\mathbf{H} + \alpha\mathbf{I})\tilde{\mathbf{w}} = \mathbf{H}\mathbf{w}^*, \tag{7.10}$$

$$\tilde{\mathbf{w}} = (\mathbf{H} + \alpha\mathbf{I})^{-1}\mathbf{H}\mathbf{w}^*. \tag{7.11}$$

Khi $\alpha \to 0$, thì $\tilde{\mathbf{w}} \to \mathbf{w}^*$. Nhưng khi α lớn thì sao?

Vì $\mathbf{H}$ là đối xứng dương, ta có thể phân rã nó thành:

$$\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T. \tag{7.12}$$

Thay vào biểu thức trên:

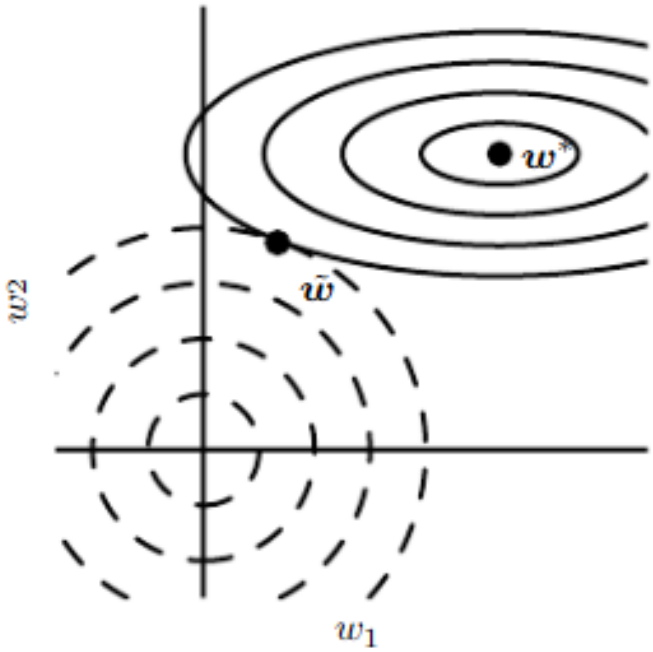
$$\tilde{\mathbf{w}} = (\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T + \alpha\mathbf{I})^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\mathbf{w}^*, \tag{7.13}$$

$$= \mathbf{Q}(\mathbf{\Lambda} + \alpha\mathbf{I})^{-1}\mathbf{\Lambda}\mathbf{Q}^T\mathbf{w}^*. \tag{7.14}$$

Điều này cho thấy weight decay làm giảm các thành phần của $\mathbf{w}^*$ theo từng hướng riêng biệt, tùy vào giá trị trị riêng λ_i:

$$\frac{\lambda_i}{\lambda_i + \alpha}. \tag{7.15}$$

Nếu λ_i lớn hơn nhiều so với α, hiệu ứng điều chuẩn là nhỏ. Nhưng nếu λ_i rất nhỏ, thì thành phần đó gần như bị loại bỏ. Điều này được minh họa trong Hình 7.1.



Hình 7.1: Minh họa tác động của điều chuẩn L^2. Các elip nét liền là đường đồng mức của hàm mất mát gốc, còn các đường tròn nét đứt là đường đồng mức của điều chuẩn. Điểm cân bằng giữa hai mục tiêu là $\hat{\mathbf{w}}$. Theo phương w_1, độ cong nhỏ nên weight decay kéo w_1 mạnh hơn về 0. Trong khi đó, theo phương w_2, hàm mất mát thay đổi mạnh nên weight decay ít ảnh hưởng hơn.

Nói cách khác, chỉ những hướng nào thật sự giúp giảm giá trị hàm mất mát mới được mô hình giữ lại. Những hướng ít ảnh hưởng (có trị riêng nhỏ) sẽ bị "ép"về 0 bởi weight decay.

Đến đây, ta đã xem xét weight decay trong trường hợp tổng quát. Giờ hãy nhìn nó dưới góc độ của bài toán học máy cụ thể hơn, như hồi quy tuyến tính.

Hàm mất mát trong hồi quy tuyến tính là:

$$J(\mathbf{w}) = (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}). \tag{7.16}$$

Thêm điều chuẩn L^2 vào ta có:

$$J(\mathbf{w}) = (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\alpha \mathbf{w}^\top \mathbf{w}. \tag{7.17}$$

Nghiệm không điều chuẩn là:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{7.18}$$

Khi có điều chuẩn, nghiệm chuyển thành:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{7.19}$$

Trong đó, $\mathbf{X}^\top \mathbf{X}$ tỷ lệ với ma trận hiệp phương sai của đầu vào:

$$\frac{1}{m} \mathbf{X}^\top \mathbf{X}. \tag{7.20}$$

Điều chuẩn làm thay đổi nó thành:

$$(\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1}. \tag{7.21}$$

Tức là ta đang giả định dữ liệu đầu vào có phương sai cao hơn. Kết quả là mô hình sẽ bớt tin tưởng vào những đặc trưng không liên quan nhiều đến đầu ra và tự động giảm trọng số của chúng, giúp tránh hiện tượng quá khớp.

7.3 L^1 Regularization

Chuẩn hóa L^1 là một kỹ thuật phổ biến giúp kiểm soát độ lớn của các tham số trong mô hình học sâu. Trong khi chuẩn hóa L^2 (hay còn gọi là weight decay) thường được sử dụng rộng rãi hơn, thì chuẩn hóa L^1 cũng rất hữu ích, đặc biệt khi ta muốn giảm số lượng tham số cần thiết trong mô hình.

Cách hoạt động của chuẩn hóa L^1 là thêm vào hàm mất mát một thành phần mới, phản ánh tổng giá trị tuyệt đối của các trọng số trong vector $\boldsymbol{w}$. Về mặt toán học, ta viết như sau:

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|, \tag{7.22}$$

Trong đó $\|\boldsymbol{w}\|_1$ là chuẩn L^1 của vector trọng số, và mỗi $|w_i|$ là giá trị tuyệt đối của phần tử thứ i trong $\boldsymbol{w}$.

So với L^2, chuẩn hóa L^1 có một đặc điểm đặc biệt: nó khuyến khích nhiều trọng số trở thành đúng bằng 0. Điều này khiến mô hình chỉ giữ lại một số lượng nhỏ trọng số — mô hình trở nên "thưa" (sparse). Đây là một lợi thế lớn trong các bài toán mà chỉ một vài đặc trưng đầu vào là quan trọng, hoặc khi ta muốn mô hình đơn giản và dễ giải thích.

Để áp dụng chuẩn hóa L^1 trong quá trình huấn luyện, ta điều chỉnh hàm mất mát như sau:

$$\tilde{J}(\boldsymbol{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\boldsymbol{w}\|_1 + J(\boldsymbol{w}; \mathbf{X}, \mathbf{y}), \tag{7.23}$$

Trong đó:

- $\alpha > 0$ là một siêu tham số điều chỉnh mức độ ảnh hưởng của chuẩn hóa.
- $J(\boldsymbol{w}; \mathbf{X}, \mathbf{y})$ là hàm mất mát ban đầu, chẳng hạn như MSE hay cross-entropy.

Do hàm trị tuyệt đối không khả vi tại điểm 0, nên khi tính đạo hàm của hàm mất mát có chuẩn hóa L^1, ta phải dùng *đạo hàm phụ* (subgradient):

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \mathbf{X}, \mathbf{y}) = \alpha \, \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \mathbf{X}, \mathbf{y}), \tag{7.24}$$

Trong đó, hàm $\text{sign}(\boldsymbol{w})$ trả về vector gồm các dấu của từng phần tử w_i, tức là -1, 0 hoặc $+1$.

Dễ thấy rằng ảnh hưởng của chuẩn hóa L^1 rất khác với L^2. Với L^2, mỗi thành phần gradient của phần phạt tỉ lệ với chính trọng số đó, nghĩa là càng lớn thì gradient càng mạnh. Còn với L^1, gradient phụ chỉ phụ thuộc vào dấu của trọng số chứ không quan tâm đến độ lớn. Điều này khiến chuẩn hóa L^1 không dễ dàng cho ta nghiệm rõ ràng như trong trường hợp chuẩn hóa L^2.[2]

Nếu ta xét một mô hình đơn giản và dùng khai triển Taylor để xấp xỉ hàm mất mát gần điểm tối ưu $\boldsymbol{w}^*$, ta có thể viết gradient như sau:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.25}$$

với $\boldsymbol{H}$ là ma trận Hessian của hàm mất mát J tại điểm $\boldsymbol{w}^*$.

Tuy nhiên, do bản chất không khả vi của L^1, rất khó tìm nghiệm tổng quát khi Hessian là một ma trận đầy đủ. Vì thế, ta giả sử rằng Hessian là một ma trận chéo: $\boldsymbol{H} = \text{diag}([H_{1,1}, \ldots, H_{n,n}])$ với $H_{i,i} > 0$. Điều này tương đương với giả định rằng các đặc trưng đầu vào đã được xử lý sao cho không còn tương quan với nhau, ví dụ bằng PCA.

Khi đó, hàm mất mát gần điểm tối ưu có thể được viết dưới dạng tổng qua từng thành phần:

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (w_i - w_i^*)^2 + \alpha |w_i| \right]. \tag{7.26}$$

Giải bài toán tối ưu hóa từng chiều i cho ta nghiệm:

$$w_i = \text{sign}(w_i^*) \cdot \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, \, 0 \right\}. \tag{7.27}$$

Ví dụ, nếu $w_i^* > 0$, ta có hai trường hợp:

- Nếu $w_i^* \leq \frac{\alpha}{H_{i,i}}$, thì $w_i = 0$. Nghĩa là trọng số bị kéo về 0 hoàn toàn.

[2] Tương tự như chuẩn hóa L^2, ta cũng có thể dùng chuẩn hóa L^1 để kéo trọng số về một giá trị cụ thể $\boldsymbol{w}^{(0)}$ thay vì về 0. Khi đó, ta dùng $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w} - \boldsymbol{w}^{(0)}\|_1 = \sum_i |w_i - w_i^{(0)}|$.

- Nếu $w_i^* > \frac{\alpha}{H_{i,i}}$, thì trọng số chỉ bị đẩy lùi về gần 0, nhưng không bị triệt tiêu.

Tương tự, nếu $w_i^* < 0$, thì trọng số cũng bị đẩy gần về 0 theo hướng âm.

So sánh với nghiệm của chuẩn hóa L^2:

$$w_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^*,$$

ta thấy rằng nếu w_i^* khác 0, thì w_i cũng sẽ khác 0. Chuẩn hóa L^2 chỉ làm giảm nhẹ trọng số, trong khi L^1 có thể triệt tiêu hoàn toàn trọng số nào đó. Đây là lý do vì sao L^1 giúp tạo ra nghiệm thưa — nhiều trọng số bằng 0.

Khả năng tạo ra nghiệm thưa là một đặc điểm quan trọng của chuẩn hóa L^1, và nó được tận dụng trong bài toán chọn lựa đặc trưng (feature selection). Khi một số trọng số bị đưa về 0, tương ứng là các đặc trưng không còn ảnh hưởng đến đầu ra và có thể loại bỏ.

Một mô hình tiêu biểu dùng chuẩn hóa L^1 là LASSO (Least Absolute Shrinkage and Selection Operator), được giới thiệu bởi Tibshirani năm 1995. Mô hình này dùng hàm mất mát bình phương kết hợp với chuẩn L^1, giúp tự động chọn ra một tập con các đặc trưng đầu vào hiệu quả nhất.

Trong mục 5.6.1, ta đã thấy rằng nhiều kỹ thuật regularization có thể được hiểu là một trường hợp của suy luận MAP (Maximum A Posteriori) trong Bayes. Cụ thể:

- Với chuẩn hóa L^2: giả định rằng mỗi trọng số w_i có phân phối tiên nghiệm Gaussian.
- Với chuẩn hóa L^1: thành phần phạt có dạng

$$\Omega(\boldsymbol{w}) = \alpha \sum_i |w_i| = \alpha \|\boldsymbol{w}\|_1,$$

điều này tương đương với giả định rằng mỗi w_i có phân phối tiên nghiệm Laplace:

$$p(w_i) = \text{Laplace}(w_i; 0, \tfrac{1}{\alpha}).$$

Do đó, khi sử dụng chuẩn hóa L^1, ta thực chất đang tối đa hóa phân phối hậu nghiệm với tiên nghiệm Laplace:

$$\log p(\boldsymbol{w}) = \sum_i \log \text{Laplace}(w_i; 0, \tfrac{1}{\alpha}) = -\alpha \|\boldsymbol{w}\|_1 + n \log \alpha - n \log 2. \tag{7.28}$$

Trong thực tế, các hằng số như $\log \alpha$ hay $\log 2$ thường bị bỏ qua vì chúng không ảnh hưởng đến việc tối ưu $\boldsymbol{w}$.

7.4 Chuẩn hóa dưới dạng bài toán tối ưu có ràng buộc

Ta xét một hàm mất mát được thêm thành phần regularization dựa trên chuẩn của tham số như sau:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\, \Omega(\boldsymbol{\theta}). \tag{7.29}$$

Như đã đề cập trong Mục 4.4, thay vì thêm trực tiếp một thành phần phạt vào hàm mất mát, ta có thể diễn giải regularization như một ràng buộc trên tham số bằng cách sử dụng hàm Lagrangian tổng quát. Hàm này bao gồm hàm mục tiêu gốc cộng với một hoặc nhiều hàm phạt, mỗi hàm phạt là tích của một hệ số Lagrange (còn gọi là hệ số KKT - Karush–Kuhn–Tucker) và một biểu thức thể hiện ràng buộc.

Ví dụ, nếu ta muốn ràng buộc $\Omega(\boldsymbol{\theta}) \leq k$, ta có thể viết hàm Lagrangian:

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\, (\Omega(\boldsymbol{\theta}) - k), \tag{7.30}$$

với $\alpha \geq 0$ là hệ số Lagrange.

Để tìm nghiệm tối ưu của bài toán có ràng buộc, ta giải:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha). \tag{7.31}$$

Như đã thấy trong Mục 4.4, việc giải bài toán này cần cập nhật đồng thời cả $\boldsymbol{\theta}$ và α. Trong một số trường hợp như hồi quy tuyến tính với chuẩn L^2, ta còn có thể giải bài toán này một cách giải tích.

Có nhiều cách tiếp cận để giải các bài toán như vậy:

- Một số dùng gradient descent để cập nhật các tham số.
- Một số khác tận dụng việc đạo hàm triệt tiêu để tìm nghiệm đóng.

Điều quan trọng là nguyên lý hoạt động của α: nếu $\Omega(\boldsymbol{\theta})$ vượt quá giới hạn k, thì α sẽ tăng để đẩy Ω nhỏ lại. Ngược lại, nếu $\Omega(\boldsymbol{\theta}) < k$, thì α có thể giảm. Từ đó ta tìm được α^* hợp lý giúp giữ chuẩn $\Omega(\boldsymbol{\theta})$ gần với giá trị k mà không quá nhỏ hoặc quá lớn.

Nếu ta cố định α^*, bài toán tối ưu chỉ còn phụ thuộc vào $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha^*\, \Omega(\boldsymbol{\theta}), \tag{7.32}$$

Đây chính là bài toán regularization quen thuộc. Có thể hiểu rằng thêm một hàm phạt vào hàm mất mát là tương đương với việc áp đặt một ràng buộc mềm lên trọng số.

Nếu Ω là chuẩn L^2, thì các trọng số sẽ bị giới hạn trong một quả cầu L^2. Còn nếu Ω là chuẩn L^1, thì ta đang ràng buộc các trọng số trong một vùng hình kim tự tháp (diamond

shape).

Trên thực tế, giá trị α không cho ta biết trực tiếp kích thước cụ thể của vùng ràng buộc tương ứng với k. Mối quan hệ giữa α và k phụ thuộc vào dạng cụ thể của hàm mất mát J. Tuy nhiên, ta có thể điều chỉnh vùng ràng buộc bằng cách thay đổi giá trị α: tăng α làm vùng nhỏ lại (giới hạn chặt hơn), còn giảm α làm vùng nới rộng.

Đôi khi, thay vì thêm thành phần phạt vào hàm mất mát, ta lại muốn sử dụng ràng buộc rõ ràng. Cách tiếp cận này được mô tả trong Mục 4.4: ta cập nhật $\boldsymbol{\theta}$ theo hướng giảm hàm J, sau đó chiếu lại $\boldsymbol{\theta}$ vào vùng thỏa mãn ràng buộc $\Omega(\boldsymbol{\theta}) < k$.

Việc sử dụng ràng buộc rõ ràng có một số ưu điểm. Một trong số đó là tránh tạo ra các cực trị địa phương không mong muốn. Khi áp dụng regularization dưới dạng hàm phạt, mô hình có thể hội tụ về những điểm mà trọng số gần như bằng 0 – điều này dẫn đến các đơn vị ẩn không hoạt động (dead units) trong mạng nơ-ron. Trong khi đó, nếu dùng ràng buộc rõ ràng và kỹ thuật chiếu lại, ta có thể ngăn điều này xảy ra, vì ràng buộc chỉ có tác dụng khi trọng số trở nên quá lớn.

Hơn nữa, ràng buộc rõ ràng còn giúp tăng độ ổn định khi tối ưu hóa. Nếu ta sử dụng tốc độ học lớn, các bước cập nhật trọng số có thể trở nên rất lớn do gradient lớn – điều này gây ra hiện tượng vòng lặp phản hồi dương (positive feedback loop). Trọng số ngày càng lớn, gradient cũng lớn dần, và cuối cùng dẫn đến tràn số. Kỹ thuật chiếu lại giúp ngăn chặn điều này bằng cách giữ trọng số trong giới hạn hợp lý.

Hinton2012 khuyến nghị sử dụng ràng buộc rõ ràng khi học với tốc độ cao, vì chúng cho phép khám phá nhanh nhưng vẫn giữ được ổn định. Cụ thể hơn, **Srebro2005** đề xuất ràng buộc chuẩn của từng cột trong ma trận trọng số của mạng nơ-ron, thay vì toàn bộ ma trận. Điều này tránh việc một đơn vị ẩn trở nên quá "mạnh" do có trọng số quá lớn. Nếu viết dưới dạng Lagrangian, thì điều này giống như áp dụng regularization L^2 nhưng với hệ số riêng biệt cho từng đơn vị ẩn. Trong thực tế, cách làm phổ biến là áp dụng ràng buộc rõ ràng và thực hiện chiếu lại từng cột sau mỗi bước cập nhật.

7.5 Regularization và các bài toán thiếu ràng buộc

Trong một số bài toán học máy, regularization không chỉ là lựa chọn để cải thiện mô hình mà còn là yếu tố bắt buộc giúp bài toán trở nên xác định. Nhiều mô hình tuyến tính như hồi quy tuyến tính hay PCA đều yêu cầu nghịch đảo ma trận $X^\top X$. Nhưng khi ma trận này suy biến (singular), tức là không khả nghịch, bài toán sẽ không thể giải bằng cách thông thường.

Ma trận $X^\top X$ có thể suy biến nếu:

- Dữ liệu không có phương sai theo một số hướng.
- Hoặc số mẫu (số hàng của X) ít hơn số đặc trưng (số cột).

Trong những trường hợp đó, regularization giúp ổn định bài toán bằng cách thêm vào một số dương α trên đường chéo:

$$X^\top X + \alpha I.$$

Ma trận điều chỉnh này luôn khả nghịch nếu $\alpha > 0$.

Khi đó, bài toán có nghiệm dạng đóng (closed-form). Nếu không có regularization, bài toán có thể không xác định. Ví dụ, trong hồi quy logistic nếu dữ liệu phân tách tuyến tính, luôn tồn tại một vector $\boldsymbol{w}$ phân loại chính xác toàn bộ dữ liệu. Khi đó, thuật toán như gradient descent sẽ không hội tụ mà tiếp tục tăng độ lớn của $\boldsymbol{w}$ mãi mãi, dẫn đến tràn số.

Regularization như weight decay giúp tránh hiện tượng này. Khi gradient bằng với thành phần weight decay, mô hình sẽ dừng lại, không tiếp tục tăng trọng số nữa.

Ngoài học máy, regularization còn hữu ích trong đại số tuyến tính. Một ví dụ là sử dụng nghịch đảo giả Moore-Penrose để giải hệ tuyến tính không xác định. Một trong các định nghĩa của nghịch đảo giả là:

$$\mathbf{X}^+ = \lim_{\alpha \to 0} (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top. \tag{7.33}$$

Biểu thức này chính là nghiệm của hồi quy tuyến tính có regularization dạng weight decay khi $\alpha \to 0$. Điều này cho thấy nghịch đảo giả có thể được xem như kết quả của việc áp dụng regularization để ổn định hóa bài toán.

7.6 Tăng cường Dữ liệu (Dataset Augmentation)

Một trong những cách hiệu quả nhất để giúp mô hình học sâu tổng quát tốt hơn là cung cấp cho nó nhiều dữ liệu huấn luyện hơn. Tuy nhiên, trong thực tế, dữ liệu sẵn có thường rất hạn chế. Vì vậy, một giải pháp phổ biến là tạo thêm dữ liệu giả để bổ sung vào tập huấn luyện gốc.

Đối với một số bài toán học máy, việc tạo ra dữ liệu giả tương đối dễ thực hiện. Phương pháp này đặc biệt hiệu quả với các bài toán phân loại, nơi mô hình học cần gán nhãn cho đầu vào x — chẳng hạn như hình ảnh — với một nhãn phân loại y.

Một trong những thách thức chính trong huấn luyện mô hình phân loại là làm sao để mô hình học được tính bất biến đối với nhiều loại biến đổi của đầu vào. Vì vậy, ta có thể tạo ra các ví dụ mới bằng cách áp dụng các phép biến đổi lên đầu vào x sao cho không làm thay đổi nhãn y. Những cặp mới (x', y) như vậy vẫn hợp lệ và giúp tăng sự đa dạng của tập dữ liệu.

Tuy nhiên, cách tiếp cận này không phải lúc nào cũng phù hợp. Chẳng hạn, trong bài toán ước lượng mật độ xác suất (density estimation), việc sinh thêm dữ liệu giả là rất khó trừ khi ta đã biết rõ phân phối xác suất — mà đây chính là điều ta đang cố gắng học.

Tăng cường dữ liệu đã được chứng minh là cực kỳ hiệu quả trong các tác vụ phân loại đối tượng, đặc biệt là nhận dạng hình ảnh. Hình ảnh thường có độ phân giải cao và chứa nhiều yếu tố biến đổi — trong đó có nhiều yếu tố có thể được mô phỏng lại dễ dàng bằng

các phép biến đổi hình học hoặc màu sắc.

Ví dụ, đơn giản như việc tịnh tiến ảnh vài pixel theo các hướng khác nhau đã có thể giúp mô hình học được tính bất biến với vị trí đối tượng trong ảnh. Điều này vẫn có ích ngay cả khi mô hình đã được thiết kế với các lớp convolution và pooling (được trình bày trong Chương **??**) vốn đã hỗ trợ tính bất biến này. Ngoài tịnh tiến, những phép biến đổi khác như xoay ảnh, lật ảnh hoặc thay đổi tỉ lệ cũng rất phổ biến và thường mang lại hiệu quả rõ rệt.

Tuy nhiên, khi áp dụng các phép biến đổi, ta cần thận trọng. Một số phép biến đổi có thể làm thay đổi ý nghĩa của nhãn. Ví dụ, trong bài toán nhận diện ký tự, xoay ảnh 180° có thể khiến chữ "p" trở thành chữ "d", hoặc số "6" trở thành "9". Trong những trường hợp này, việc xoay ảnh không còn là một phép tăng cường dữ liệu phù hợp.

Ngoài ra, cũng có những phép biến đổi mà ta muốn mô hình học được tính bất biến, nhưng không dễ để mô phỏng. Chẳng hạn, biến đổi góc nhìn — thay đổi cách ta nhìn đối tượng từ các góc độ khác nhau — không thể thực hiện chỉ bằng thao tác hình học đơn giản trên ảnh gốc.

Tăng cường dữ liệu cũng đã cho thấy hiệu quả rõ rệt trong nhận dạng giọng nói, như được trình bày trong nghiên cứu của **Jaitly2013**.

Thêm nhiễu vào đầu vào cũng có thể xem là một dạng tăng cường dữ liệu. Nghiên cứu của **Sietsma1991** cho thấy việc huấn luyện mạng nơ-ron với dữ liệu bị nhiễu có thể giúp mô hình trở nên bền vững hơn trước nhiễu thực tế. Cách làm này đơn giản là thêm nhiễu ngẫu nhiên vào dữ liệu huấn luyện [**Wendt1994**], hoặc thậm chí ẩn đi một phần đầu vào như đề xuất của **Vincent2008**. Miễn là độ lớn của nhiễu được điều chỉnh phù hợp, những phương pháp này thường mang lại hiệu quả cao.

Một kỹ thuật nổi tiếng có thể xem là dạng đặc biệt của tăng cường dữ liệu là *Dropout*, được trình bày chi tiết trong Mục **??**. Dropout không chỉ giúp tránh quá khớp mà còn khiến mô hình học được cách hoạt động trong điều kiện không đầy đủ — tương tự như khi đầu vào bị nhiễu hay thiếu thông tin.

Hiện nay, việc đánh giá mô hình học máy thường dựa vào các tập benchmark cố định. Kỹ thuật tăng cường dữ liệu, nếu được sử dụng đúng cách, có thể giúp mô hình đạt hiệu suất cao hơn rõ rệt. Vì vậy, để đảm bảo công bằng khi so sánh giữa các mô hình, người ta thường kiểm soát chặt chẽ các chiến lược tăng cường được áp dụng.

Chẳng hạn, giả sử hai mô hình A và B được huấn luyện trên cùng một tập dữ liệu. Nếu A hoạt động tốt hơn khi không có tăng cường dữ liệu, nhưng B lại vượt trội hơn khi có áp dụng kỹ thuật tăng cường mạnh mẽ, thì trong thực tế B có thể là mô hình tốt hơn — vì nó tận dụng tốt hơn dữ liệu được mở rộng. Tuy nhiên, khi đánh giá, cần làm rõ rằng sự cải thiện đến từ bản thân kiến trúc mô hình hay từ kỹ thuật tăng cường.

Do đó, khi một mô hình mới được công bố và cho kết quả vượt trội trên một benchmark, điều quan trọng là phải xác minh xem kết quả đó có đến từ chiến lược tăng cường dữ liệu

đặc biệt tốt (ví dụ như phương pháp cắt ảnh tinh vi hơn, cách thêm nhiễu hợp lý hơn, v.v.) hay không. Việc phân biệt rõ yếu tố nào đóng góp vào sự cải thiện là rất quan trọng trong nghiên cứu học máy.

7.6.1 Khả năng chống nhiễu (Noise Robustness)

Trong Phần **??**, chúng ta đã nói đến việc sử dụng nhiễu như một kỹ thuật tăng cường dữ liệu. Đáng chú ý là với một số mô hình, việc thêm nhiễu có phương sai rất nhỏ vào đầu vào có thể tương đương với việc áp dụng một hình phạt lên chuẩn của trọng số [8], [9]. Đây là một cách nhìn thú vị, cho thấy mối liên hệ giữa tăng cường dữ liệu và regularization.

Tuy nhiên, trong thực tế, việc tiêm nhiễu vào mạng có thể tạo ra ảnh hưởng mạnh hơn nhiều so với chỉ đơn thuần là làm nhỏ trọng số. Điều này đặc biệt đúng nếu nhiễu được thêm vào các tầng ẩn bên trong mạng nơ-ron. Vì vai trò quan trọng này, các phương pháp như vậy đáng được xem xét riêng — ví dụ điển hình là kỹ thuật *dropout* được trình bày trong Phần **??**.

Một hướng tiếp cận khác là thêm nhiễu trực tiếp vào trọng số của mô hình. Phương pháp này rất phổ biến trong các mạng nơ-ron hồi tiếp (RNN), như mô tả trong các nghiên cứu của **jim1996**, [10]. Khi áp dụng cách này, ta không còn coi trọng số là các giá trị cố định, mà xem chúng như những biến ngẫu nhiên — đại diện cho sự không chắc chắn trong mô hình. Việc thêm nhiễu vào trọng số là một cách đơn giản để thể hiện quan điểm này, mang đậm tinh thần xác suất Bayes.

Ngoài ra, việc tiêm nhiễu vào trọng số còn có thể được hiểu là tương đương với một số hình thức regularization truyền thống, nếu ta đưa ra một số giả định. Cách diễn giải này cho thấy rõ cách mà regularization giúp mô hình học một hàm ổn định hơn.

Xét một bài toán hồi quy, nơi ta muốn học một hàm $\hat{y}(\mathbf{x})$ ánh xạ đầu vào $\mathbf{x}$ sang một đầu ra vô hướng y. Hàm mất mát thường dùng là sai số bình phương trung bình:

$$J = \mathbb{E}_{p(\mathbf{x},y)} \left[(\hat{y}(\mathbf{x}) - y)^2 \right]. \tag{7.34}$$

Giả sử tập huấn luyện gồm m mẫu:

$$\left\{ (\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)}) \right\}. \tag{7.35}$$

Bây giờ ta giả định rằng, mỗi khi sử dụng một mẫu huấn luyện, ta sẽ thêm vào trọng số của mạng một nhiễu ngẫu nhiên có phân phối Gaussian:

$$\boldsymbol{\epsilon}_W \sim \mathcal{N}(\mathbf{0}, \eta \mathbf{I}).$$

Với mạng MLP có l tầng, ta ký hiệu mô hình bị nhiễu là $\hat{y}_{\boldsymbol{\epsilon}_W}(\mathbf{x})$. Dù đã thêm nhiễu, ta vẫn muốn tối thiểu hóa sai số giữa đầu ra dự đoán và giá trị thực. Hàm mất mát lúc này

được viết lại như sau:

$$\tilde{J}_W = \mathbb{E}_{p(\mathbf{x},y,\boldsymbol{\epsilon}_W)} \left[(\hat{y}_{\boldsymbol{\epsilon}_W}(\mathbf{x}) - y)^2 \right] \tag{7.36}$$

$$= \mathbb{E}_{p(\mathbf{x},y,\boldsymbol{\epsilon}_W)} \left[\hat{y}^2_{\boldsymbol{\epsilon}_W}(\mathbf{x}) - 2y\hat{y}_{\boldsymbol{\epsilon}_W}(\mathbf{x}) + y^2 \right] \tag{7.37}$$

Trong trường hợp phương sai η đủ nhỏ, việc tối ưu $\tilde{J}_W$ sẽ tương đương với việc tối ưu một hàm mất mát có thêm thành phần regularization như sau:

$$\eta \, \mathbb{E}_{p(\mathbf{x},y)} \left[\|\nabla_{\mathbf{w}} \hat{y}(\mathbf{x})\|^2 \right]$$

Thành phần này khuyến khích mô hình học những hàm đầu ra không quá nhạy cảm với sự thay đổi nhỏ trong trọng số. Nói cách khác, mô hình sẽ ưu tiên các vùng trong không gian tham số mà đầu ra thay đổi chậm khi trọng số bị nhiễu. Điều này đồng nghĩa với việc mô hình có xu hướng đi vào các "vùng phẳng" — những cực tiểu ít nhọn, ổn định hơn. Ý tưởng về những vùng phẳng như vậy được nhấn mạnh trong nghiên cứu của **hochreiter1995**.

Trong trường hợp đơn giản của hồi quy tuyến tính:

$$\hat{y}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

Thành phần regularization phía trên trở thành:

$$\eta \, \mathbb{E}_{p(\mathbf{x})} \left[\|\mathbf{x}\|^2 \right]$$

Đây là một hằng số không phụ thuộc vào tham số mô hình, nên không ảnh hưởng đến gradient của $\tilde{J}_W$ theo trọng số. Tuy vậy, trong các mô hình phi tuyến như mạng MLP, thành phần này có ảnh hưởng rõ rệt và có thể làm thay đổi đáng kể cách mô hình được học.

7.6.2 Tiêm nhiễu vào nhãn đầu ra

Trong các bộ dữ liệu thực tế, không thể tránh khỏi việc tồn tại những nhãn bị gán sai. Nếu mô hình cố gắng cực đại hóa $\log p(y \mid \mathbf{x})$ với các nhãn sai này, hiệu suất tổng thể có thể bị ảnh hưởng nghiêm trọng. Một cách để hạn chế tác động tiêu cực đó là đưa giả định về nhiễu vào quá trình huấn luyện — cụ thể là mô hình hóa trực tiếp khả năng nhãn bị sai.

Ví dụ, ta có thể giả định rằng nhãn y là chính xác với xác suất $1 - \epsilon$, còn tất cả các nhãn sai có tổng xác suất là ϵ. Giả định này có thể được tích hợp vào hàm mất mát thay vì phải sinh thêm dữ liệu nhiễu.

Một kỹ thuật hiệu quả dựa trên giả định này là label smoothing. Đây là một dạng

regularization dành cho các mô hình phân loại dùng softmax với k lớp. Thay vì biểu diễn nhãn theo cách thông thường (nhãn cứng — one-hot) với giá trị $\{0, 1\}$, ta dùng nhãn mềm với phân phối xác suất như sau:

- Đối với nhãn đúng: gán giá trị $1 - \epsilon$
- Đối với mỗi nhãn sai: gán giá trị $\frac{\epsilon}{k-1}$

Khi đó, ta vẫn sử dụng hàm mất mát cross-entropy như bình thường nhưng với các nhãn đã được làm mềm.

Trong huấn luyện truyền thống, việc tối đa hóa likelihood với nhãn cứng có thể dẫn đến việc mô hình học các trọng số rất lớn. Nguyên nhân là vì softmax không thể gán xác suất tuyệt đối 1 hoặc 0 cho các lớp, nên mô hình "cố gắng" bằng cách làm cho phân phối gần với nhãn một cách cực đoan. Label smoothing giúp tránh hiện tượng này bằng cách làm giảm độ sắc nét của nhãn.

Nói cách khác, label smoothing ngăn mô hình rơi vào trạng thái quá tự tin. Mô hình vẫn học để phân loại đúng, nhưng không cố gắng đẩy xác suất của lớp đúng lên tuyệt đối 1, mà chỉ cần cao hơn các lớp còn lại.

Kỹ thuật này được sử dụng từ những năm 1980 và đến nay vẫn rất phổ biến trong các mạng nơ-ron hiện đại, bao gồm cả kiến trúc Inception [11]. Trong thực hành, label smoothing thường được kết hợp với các phương pháp regularization khác như weight decay để tăng độ ổn định trong huấn luyện.

7.7 Học bán giám sát (Semi-Supervised Learning)

Trong học bán giám sát, ta không chỉ sử dụng dữ liệu có nhãn (lấy mẫu từ phân phối $P(\mathbf{x}, \mathbf{y})$) mà còn tận dụng cả dữ liệu không có nhãn (lấy từ $P(\mathbf{x})$) để học mô hình dự đoán $P(\mathbf{y} \mid \mathbf{x})$ hoặc trực tiếp dự đoán nhãn $\mathbf{y}$ từ đầu vào $\mathbf{x}$.

Trong bối cảnh học sâu, học bán giám sát thường liên quan đến việc học biểu diễn đặc trưng $\mathbf{h} = f(\mathbf{x})$. Mục tiêu là học sao cho các điểm thuộc cùng một lớp có biểu diễn gần nhau. Dữ liệu không nhãn đóng vai trò giúp mô hình hiểu được cách tổ chức, cấu trúc của không gian đầu vào — qua đó học ra biểu diễn có ý nghĩa hơn.

Ý tưởng là nếu các điểm gần nhau trong không gian đầu vào thì biểu diễn của chúng cũng nên gần nhau trong không gian đặc trưng. Khi đó, một mô hình phân loại đơn giản, ví dụ như tuyến tính, có thể hoạt động rất tốt trên biểu diễn mới này [12], [13].

Một phương pháp cổ điển theo hướng này là sử dụng phân tích thành phần chính (PCA) để giảm chiều dữ liệu đầu vào trước khi áp dụng thuật toán phân loại.

Thay vì tách riêng phần học có nhãn và không nhãn, ta có thể xây dựng một mô hình kết hợp giữa hai hướng tiếp cận. Cụ thể:

- Một mô hình tạo sinh (generative model) được huấn luyện để ước lượng $P(\mathbf{x})$ hoặc $P(\mathbf{x}, \mathbf{y})$.

- Mô hình này chia sẻ tham số với mô hình phân biệt (discriminative model) ước lượng $P(\mathbf{y} \mid \mathbf{x})$.

Khi đó, hàm mục tiêu sẽ kết hợp cả hai tiêu chí:

$$-\log P(\mathbf{y} \mid \mathbf{x}) \quad \text{và} \quad -\log P(\mathbf{x}) \text{ hoặc } -\log P(\mathbf{x}, \mathbf{y}).$$

Tiêu chí tạo sinh đóng vai trò là một dạng regularization, giúp truyền tải giả định rằng cấu trúc của $P(\mathbf{x})$ phản ánh phần nào cấu trúc của $P(\mathbf{y} \mid \mathbf{x})$, vì chúng được biểu diễn thông qua các tham số chung. Bằng cách điều chỉnh mức độ ảnh hưởng của hai thành phần này, ta có thể đạt được sự cân bằng giữa mục tiêu học có giám sát và không giám sát **lasserre2006**, **larochelle2008**.

Ví dụ: Salakhutdinov và Hinton **salakhutdinov2008** giới thiệu một phương pháp học hàm nhân (kernel machines) cho bài toán hồi quy, trong đó việc tận dụng dữ liệu không nhãn giúp cải thiện đáng kể khả năng dự đoán $P(\mathbf{y} \mid \mathbf{x})$.

Để tìm hiểu kỹ hơn về học bán giám sát, bạn đọc có thể tham khảo thêm tài liệu tổng quan chi tiết trong [14].

7.8 Học đa nhiệm (Multitask Learning)

Học đa nhiệm (*Multitask Learning*, viết tắt là MTL) là một chiến lược nhằm cải thiện khả năng tổng quát hóa của mô hình bằng cách học đồng thời nhiều tác vụ khác nhau. Ý tưởng cốt lõi là khai thác mối liên hệ giữa các tác vụ để hỗ trợ lẫn nhau trong quá trình học. Điều này giống như việc cung cấp thêm các ví dụ huấn luyện bổ sung cho mô hình — ở đây là thông qua những ràng buộc mềm xuất phát từ các tác vụ có liên quan.

Theo **caruana1993multitask**, khi nhiều tác vụ chia sẻ cùng một phần mô hình, phần này buộc phải học ra các biểu diễn trung gian (representations) có tính khái quát cao để phục vụ cho tất cả các tác vụ cùng lúc. Nhờ đó, mô hình học được những đặc trưng có khả năng tổng quát tốt hơn — miễn là sự chia sẻ tham số được thiết kế hợp lý.

Giả sử ta đang giải nhiều bài toán học có giám sát (supervised learning) khác nhau, nhưng tất cả đều sử dụng chung một đầu vào $\mathbf{x}$. Trong học sâu, ta có thể chia mô hình thành hai phần:

- Một phần đầu dùng chung cho tất cả các tác vụ, có thể được xem là biểu diễn trung gian $\mathbf{h}^{(\text{shared})}$,
- Và phần còn lại là các tầng riêng biệt cho từng tác vụ, giúp mô hình điều chỉnh để phù hợp với yêu cầu cụ thể của từng bài toán.

Cấu trúc tham số của mô hình cũng vì thế mà phân thành hai nhóm chính:

1. Tham số riêng cho từng tác vụ: Là các tầng cuối cùng trong mạng nơ-ron, được huấn luyện chỉ với dữ liệu của từng tác vụ, đảm bảo mô hình tối ưu cho từng đầu ra cụ thể.

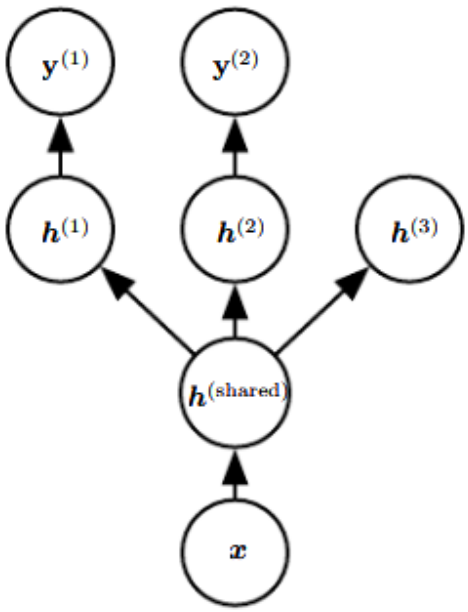
2. Tham số dùng chung: Là các tầng đầu trong mạng nơ-ron, được cập nhật từ dữ liệu tổng hợp của tất cả các tác vụ. Những tham số này có vai trò học các đặc trưng chung, hữu ích cho nhiều bài toán.

Theo **baxter1995learning**, chia sẻ tham số giữa các tác vụ giúp tăng độ chính xác và giảm sai số khái quát. Lý do là vì mô hình nhận được thêm các ví dụ gián tiếp từ các tác vụ liên quan — một dạng mở rộng dữ liệu huấn luyện giúp tăng sức mạnh thống kê (statistical strength) của mô hình. Tuy nhiên, lợi ích này chỉ phát huy khi có mối liên hệ thực sự giữa các tác vụ, nghĩa là chúng chia sẻ một số yếu tố ngầm ẩn nào đó trong dữ liệu.

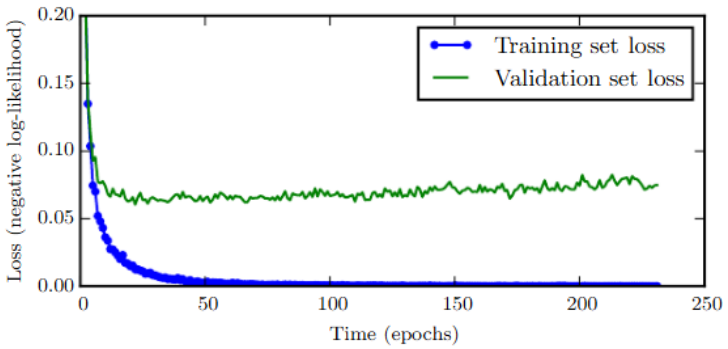
Trong học sâu, một giả định then chốt của học đa nhiệm là: *mặc dù mỗi tác vụ có thể khác nhau về bản chất, nhưng tồn tại một số yếu tố cơ bản trong dữ liệu mà các tác vụ này có thể chia sẻ*. Mục tiêu chính của học đa nhiệm là khám phá và tận dụng những yếu tố chung đó để cải thiện hiệu suất tổng thể trên tất cả các tác vụ.

7.9 Dừng sớm (Early Stopping)

Khi huấn luyện các mô hình học sâu có độ phức tạp cao (tức là có khả năng biểu diễn đủ mạnh để khớp hoàn toàn với dữ liệu huấn luyện), ta thường thấy rằng lỗi trên tập huấn luyện sẽ tiếp tục giảm đều trong suốt quá trình huấn luyện. Tuy nhiên, điều đáng chú ý là lỗi trên tập kiểm tra (validation set) lại không tiếp tục giảm mãi. Sau một thời điểm nào đó, lỗi này bắt đầu tăng lên trở lại, cho thấy mô hình bắt đầu học quá mức (overfit) vào dữ liệu huấn luyện. Hiện tượng này thường xảy ra trong thực nghiệm và được minh họa rõ trong Hình 7.3.



Hình 7.2: Một số cách triển khai học đa nhiệm trong các framework học sâu. Các tầng đầu (chứa biểu diễn $h^{(\text{shared})}$) được chia sẻ giữa các tác vụ, còn các tầng cuối (như $h^{(1)}$, $h^{(2)}$) được chuyên biệt hóa cho từng tác vụ riêng biệt như $y^{(1)}$ và $y^{(2)}$.



Hình 7.3: Biểu đồ minh họa quá trình huấn luyện một mạng *maxout* trên tập dữ liệu MNIST. Trong khi hàm mất mát trên tập huấn luyện tiếp tục giảm đều, thì hàm mất mát trên tập kiểm tra bắt đầu tăng sau một số epoch — cho thấy hiện tượng overfitting.

Từ quan sát trên, một ý tưởng đơn giản nhưng rất hiệu quả là: *thay vì sử dụng bộ tham số cuối cùng sau khi huấn luyện, ta sẽ chọn bộ tham số tại thời điểm mà lỗi kiểm tra đạt mức thấp nhất*. Cụ thể, mỗi khi thấy lỗi kiểm tra giảm xuống thấp hơn kỷ lục trước đó, ta lưu lại một bản sao của tham số mô hình. Sau khi quá trình huấn luyện kết thúc, ta sử dụng bộ tham số tương ứng với thời điểm tốt nhất này.

Thuật toán huấn luyện sẽ dừng lại nếu không có sự cải thiện nào trên tập kiểm tra trong một số vòng lặp nhất định (gọi là *patience* — độ kiên nhẫn), nhằm tránh tiếp tục học quá mức vào tập huấn luyện. Quy trình này được mô tả cụ thể trong *Thuật toán 7.1*.

Kỹ thuật này được gọi là dừng sớm (*early stopping*), và đây có thể xem là một trong những hình thức regularization phổ biến nhất trong học sâu, nhờ sự đơn giản và hiệu quả mà nó mang lại.

Một cách khác để hiểu về dừng sớm là xem đây như một phương pháp lựa chọn siêu tham số (hyperparameter selection). Trong cách nhìn này, số bước huấn luyện (training steps) chính là một siêu tham số — và tương tự như các siêu tham số khác, nó cũng có đường cong hiệu suất hình chữ U trên tập kiểm tra, như minh họa ở Hình 7.3 (và tương tự như Hình 5.3).

Khác với nhiều siêu tham số khác vốn cần được cố định trước khi huấn luyện (ví dụ như kích thước mạng, hệ số regularization), thì với dừng sớm, ta có thể "tự động"chọn giá trị tốt nhất cho số bước huấn luyện chỉ trong một lần chạy. Điều này làm giảm đáng kể chi phí thử nghiệm so với việc huấn luyện riêng biệt với nhiều giá trị khác nhau của siêu tham số.

Chi phí chính khi sử dụng dừng sớm là việc phải đánh giá hiệu suất trên tập kiểm tra sau mỗi vài vòng lặp. Trong điều kiện lý tưởng, ta có thể chạy quá trình đánh giá này song song trên một CPU hoặc GPU riêng biệt để không làm chậm quá trình huấn luyện chính. Nếu tài nguyên hạn chế, ta có thể đánh đổi bằng cách sử dụng tập kiểm tra nhỏ hơn hoặc đánh giá với tần suất thấp hơn — điều này cho phép ta ước lượng hiệu suất một cách tiết kiệm hơn, dù có thể không chính xác hoàn toàn.

Algorithm 7.1 Thuật toán meta dừng sớm để xác định thời điểm huấn luyện tối ưu. Đây là một chiến lược tổng quát có thể áp dụng cho nhiều thuật toán huấn luyện và các tiêu chí đánh giá khác nhau.

Giả sử n là số bước giữa hai lần đánh giá.
Giả sử p là "độ kiên nhẫn" (*patience*) — số lần liên tiếp mà lỗi kiểm tra không cải thiện trước khi dừng.
$\boldsymbol{\theta}_0$ là tham số khởi tạo ban đầu.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
$i \leftarrow 0$ ▷ Số bước huấn luyện đã thực hiện
$j \leftarrow 0$ ▷ Số lần liên tiếp không cải thiện
$v \leftarrow \infty$ ▷ Lỗi kiểm tra tốt nhất hiện tại
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$ ▷ Tham số tốt nhất ghi nhận được
$i^* \leftarrow i$ ▷ Số bước tại thời điểm tốt nhất
while $j < p$ **do**
  Huấn luyện mô hình thêm n bước để cập nhật $\boldsymbol{\theta}$
  $i \leftarrow i + n$
  Tính $v' \leftarrow$ Lỗi kiểm tra($\boldsymbol{\theta}$)
  if $v' < v$ **then**
    $j \leftarrow 0$
    Cập nhật $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Trả về tham số tốt nhất $\boldsymbol{\theta}^*$ và số bước tương ứng i^*

Một chi phí phụ khi sử dụng dừng sớm là phải lưu lại bản sao của các tham số tốt nhất trong suốt quá trình huấn luyện. Tuy nhiên, chi phí này thường không đáng kể vì việc lưu có thể thực hiện vào bộ nhớ chậm (như ổ cứng hoặc bộ nhớ hệ thống), và chỉ thực hiện thỉnh thoảng. Do quá trình ghi này không làm gián đoạn huấn luyện, nó hầu như không ảnh hưởng đến hiệu suất tổng thể.

Điểm mạnh của dừng sớm là tính đơn giản và không xâm lấn. Kỹ thuật này không thay đổi hàm mục tiêu, không thay đổi không gian tham số và không tác động vào quá trình cập nhật trọng số. Điều này giúp dừng sớm dễ dàng tích hợp vào bất kỳ quy trình huấn luyện nào. Trái lại, một số kỹ thuật regularization như weight decay lại yêu cầu điều chỉnh hàm mất mát và có thể khiến mô hình rơi vào cực tiểu cục bộ không mong muốn nếu hệ số decay không được chọn cẩn thận.

Dừng sớm có thể được sử dụng độc lập, hoặc kết hợp với các kỹ thuật regularization khác. Trên thực tế, ngay cả khi sử dụng các phương pháp regularization khác làm thay đổi hàm mất mát, điểm cực tiểu của hàm mất mát vẫn không đảm bảo mang lại khả năng tổng quát tốt nhất — lúc này dừng sớm vẫn có thể giúp cải thiện kết quả.

Một hạn chế của dừng sớm là yêu cầu phải tách riêng một tập kiểm tra (validation set),

nghĩa là không sử dụng toàn bộ dữ liệu cho huấn luyện. Để tận dụng toàn bộ dữ liệu, ta có thể huấn luyện lại một lần nữa sau khi dừng sớm kết thúc. Có hai chiến lược chính:

- Chiến lược 1 (Thuật toán 7.2): Huấn luyện lại từ đầu trên toàn bộ dữ liệu, với số bước bằng đúng số bước tối ưu đã tìm được trước đó.
- Chiến lược 2 (Thuật toán 7.3): Tiếp tục huấn luyện từ các tham số đã tối ưu, nhưng lần này trên toàn bộ dữ liệu. Kết thúc khi hàm mất mát trên tập kiểm tra đạt giá trị thấp hơn ngưỡng ghi nhận trước đó.

Algorithm 7.2 Thuật toán huấn luyện lại sau dừng sớm bằng cách xác định thời gian huấn luyện tối ưu rồi huấn luyện lại từ đầu trên toàn bộ dữ liệu.

Chia tập huấn luyện $(\mathcal{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ thành $(\mathcal{X}^{(\text{subtrain})}, \mathcal{X}^{(\text{valid})})$ và $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$.
Chạy dừng sớm (Thuật toán 7.1) trên tập *subtrain* và *valid*, thu được i^*.
Khởi tạo lại tham số $\boldsymbol{\theta}$ ngẫu nhiên.
Huấn luyện lại mô hình trên toàn bộ tập huấn luyện trong đúng i^* bước.

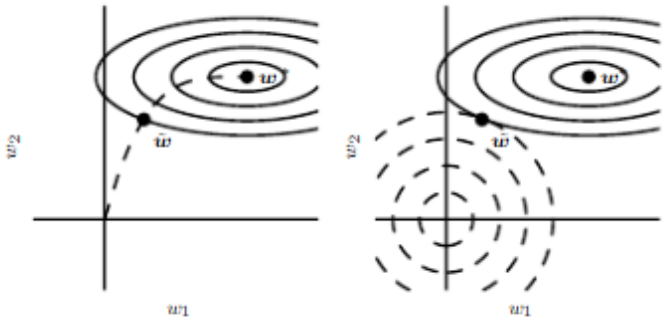
Algorithm 7.3 Thuật toán huấn luyện lại dựa trên giá trị mất mát tại thời điểm dừng sớm, tiếp tục huấn luyện đến khi đạt mất mát tương đương.

Chia tập huấn luyện $(\mathcal{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ thành $(\mathcal{X}^{(\text{subtrain})}, \mathcal{X}^{(\text{valid})})$ và $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$.
Chạy dừng sớm (Thuật toán 7.1) trên tập *subtrain* và *valid*, cập nhật tham số $\boldsymbol{\theta}$.
Ghi lại ngưỡng sai số $\epsilon \leftarrow J(\boldsymbol{\theta}, \mathcal{X}^{(\text{subtrain})}, \mathbf{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \mathcal{X}^{(\text{valid})}, \mathbf{y}^{(\text{valid})}) > \epsilon$ **do**
    Tiếp tục huấn luyện trên toàn bộ tập huấn luyện thêm n bước.
end while

Dừng sớm (*Early stopping*) không chỉ giúp tránh hiện tượng quá khớp (overfitting), mà còn có lợi thế đáng kể về mặt chi phí tính toán. Ngoài việc rõ ràng là giới hạn số vòng lặp huấn luyện, nó còn cung cấp một hình thức regularization hiệu quả mà không cần thêm các hạng tử phạt vào hàm mất mát, và cũng không yêu cầu tính gradient của các thành phần đó.

Cho đến thời điểm này, chúng ta đã chứng minh rằng dừng sớm là một chiến lược regularization, chủ yếu dựa trên việc quan sát các đường cong học: lỗi kiểm tra thường có dạng hình chữ U. Nhưng câu hỏi đặt ra là: *cơ chế nào khiến dừng sớm có tác dụng regularization?*

Bishop [8] và **sjÕêberg1995** đã lập luận rằng: dừng sớm có tác dụng giới hạn quá trình tối ưu hoá trong một vùng tương đối nhỏ quanh điểm khởi tạo ban đầu $\boldsymbol{\theta}_0$, như minh hoạ trong Hình 7.4.



Hình 7.4: Minh hoạ ảnh hưởng của dừng sớm (trái) và regularization chuẩn L^2 (phải). Với dừng sớm, quá trình tối ưu hoá chỉ tiến được một đoạn ngắn từ điểm khởi tạo ban đầu về phía cực tiểu $\mathbf{w}^*$, dừng lại tại $\tilde{\mathbf{w}}$. Với chuẩn L^2, thêm hạng tử phạt làm thay đổi vị trí cực tiểu, khiến nó lệch về gần gốc hơn.

Cụ thể hơn, giả sử ta thực hiện τ bước cập nhật bằng thuật toán gradient descent với tốc độ học (learning rate) là ϵ. Khi đó, tích $\epsilon\tau$ phản ánh một loại "năng lực hiệu quả"(*effective capacity*) — tức là giới hạn phạm vi mà mô hình có thể di chuyển trong không gian tham số. Nếu gradient bị chặn lại trong một giới hạn (ví dụ do clip gradient hoặc do tính chất của hàm mất mát), thì việc giới hạn số vòng lặp và tốc độ học đồng thời sẽ giới hạn thể tích không gian mà mô hình có thể khám phá, tính từ điểm khởi tạo ban đầu.

Theo quan điểm này, ta có thể xem $\epsilon\tau$ hoạt động ngược lại với hệ số weight decay. Nếu weight decay ngăn trọng số trở nên quá lớn, thì dừng sớm giới hạn số bước đi để trọng số không thể tiến quá xa.

Thật thú vị, trong trường hợp đơn giản — ví dụ như mô hình tuyến tính với hàm mất mát bậc hai và thuật toán gradient descent thông thường — ta có thể chứng minh rằng dừng sớm tương đương với regularization theo chuẩn L^2.

Giả sử ta có một mô hình mà tham số duy nhất là vector trọng số $\boldsymbol{\theta} = \boldsymbol{w}$. Ta xấp xỉ hàm mất mát J bằng khai triển Taylor bậc hai quanh nghiệm tối ưu thực nghiệm $\boldsymbol{w}^*$:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.38}$$

với $\boldsymbol{H}$ là ma trận Hessian của J tại điểm $\boldsymbol{w}^*$. Vì $\boldsymbol{w}^*$ là cực tiểu, nên $\boldsymbol{H}$ là nửa xác định dương.

Khi lấy đạo hàm của $\hat{J}$, ta được:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.39}$$

Giả sử ta khởi tạo tại gốc tọa độ:

$$\boldsymbol{w}^{(0)} = \mathbf{0}. \tag{7.40}$$

Với mỗi bước cập nhật theo gradient descent, ta có:

$$\begin{aligned} \boldsymbol{w}^{(\tau)} &= \boldsymbol{w}^{(\tau-1)} - \epsilon \nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}^{(\tau-1)}) && (7.41)\\ &= \boldsymbol{w}^{(\tau-1)} - \epsilon \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) && (7.42)\\ \Rightarrow \boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* &= (\boldsymbol{I} - \epsilon \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). && (7.43) \end{aligned}$$

Chuỗi đệ quy trên cho thấy rằng mỗi bước cập nhật làm thu hẹp khoảng cách giữa $\boldsymbol{w}^{(\tau)}$ và $\boldsymbol{w}^*$, với tốc độ bị điều chỉnh bởi $\epsilon \boldsymbol{H}$. Như vậy, dừng sớm có hiệu ứng làm "giới hạn"việc tiếp cận cực tiểu — mô hình không tiến quá gần $\boldsymbol{w}^*$, đặc biệt nếu hướng gradient theo những chiều mà Hessian có trị riêng nhỏ (tức là mô hình chưa đủ tự tin về hướng đó). Kết quả cuối cùng là một nghiệm mềm hơn, tương tự như khi dùng regularization chuẩn L^2.

Để hiểu sâu hơn cơ chế hoạt động của dừng sớm, ta sẽ chuyển biểu thức cập nhật tham số sang hệ toạ độ của các vector riêng của ma trận Hessian $\boldsymbol{H}$. Bằng cách thực hiện phân tích giá trị riêng (eigendecomposition), ta có:

$$\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top, \tag{7.44}$$

trong đó:

- $\boldsymbol{Q}$ là ma trận trực chuẩn gồm các vector riêng (orthonormal eigenvectors),
- $\boldsymbol{\Lambda}$ là ma trận đường chéo chứa các giá trị riêng (eigenvalues).

Thay vào công thức cập nhật, ta được:

$$\begin{aligned} \boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* &= (\boldsymbol{I} - \epsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*), && (7.45)\\ \Rightarrow \boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*) &= (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). && (7.46) \end{aligned}$$

Giả sử khởi tạo từ gốc $\boldsymbol{w}^{(0)} = \mathbf{0}$, và chọn tốc độ học ϵ đủ nhỏ để đảm bảo $|1 - \epsilon\lambda_i| < 1$, thì sau τ bước, ta có:

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = \left[\boldsymbol{I} - (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})^\tau\right] \boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.47}$$

Giờ hãy so sánh với regularization chuẩn L^2. Từ phương trình (7.13), nghiệm regularized

có thể viết lại dưới dạng:

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \boldsymbol{\Lambda} \boldsymbol{Q}^\top \boldsymbol{w}^*, \tag{7.48}$$

$$\Rightarrow \boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = \left[\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \alpha \boldsymbol{I}\right] \boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.49}$$

So sánh hai biểu thức trên, ta nhận thấy rằng quá trình dừng sớm và regularization L^2 sẽ dẫn đến nghiệm gần giống nhau nếu các siêu tham số ϵ, τ, và α thoả mãn:

$$(\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \alpha. \tag{7.50}$$

Biểu thức này cho thấy rằng trong không gian tham số được xấp xỉ bằng hàm bậc hai, dừng sớm có thể tạo ra hiệu ứng tương đương với regularization chuẩn L^2.

Tiến thêm một bước, nếu lấy logarit hai vế và khai triển Taylor với giả định $\epsilon\lambda_i \ll 1$ và $\lambda_i/\alpha \ll 1$, ta rút ra:

$$\tau \approx \frac{1}{\epsilon \alpha}, \tag{7.51}$$

$$\Rightarrow \alpha \approx \frac{1}{\tau \epsilon}. \tag{7.52}$$

Kết luận quan trọng ở đây là: số vòng lặp huấn luyện τ có vai trò ngược lại với hệ số regularization α. Cụ thể, *huấn luyện càng ngắn thì hiệu ứng regularization càng mạnh*, và ngược lại.

Bên cạnh đó, cả dừng sớm và regularization L^2 đều ảnh hưởng khác nhau đến từng thành phần của trọng số — phụ thuộc vào độ cong (curvature) của hàm mất mát dọc theo từng hướng riêng. Những hướng có trị riêng lớn (độ cong lớn) sẽ được học nhanh hơn và bị regularization ít hơn. Ngược lại, các hướng "phẳng"(trị riêng nhỏ) sẽ bị regularization mạnh hơn.

Tóm lại, quá trình huấn luyện với số vòng lặp giới hạn tương đương với việc tìm nghiệm gần giống như nghiệm của bài toán tối ưu có regularization L^2.

Tuy nhiên, dừng sớm không chỉ đơn giản là giới hạn số bước cập nhật. Quan trọng hơn, dừng sớm dựa vào việc *theo dõi trực tiếp hiệu suất trên tập validation*, và quyết định dừng lại tại thời điểm tốt nhất.

Điều này giúp dừng sớm trở nên linh hoạt và tự điều chỉnh mức độ regularization phù hợp với từng trường hợp cụ thể. Trong khi đó, regularization L^2 yêu cầu thử nghiệm với nhiều giá trị khác nhau của α để tìm ra giá trị tối ưu, điều này tốn thời gian và tài nguyên.

Tóm lại: Dừng sớm là một chiến lược regularization hiệu quả, có thể được xem như

một dạng điều chuẩn tương đương với L^2, nhưng có ưu điểm là dễ áp dụng, ít cần tinh chỉnh và có khả năng tự động điều chỉnh độ mạnh regularization dựa trên tập validation.

7.10 Ràng buộc và chia sẻ tham số (*Parameter Tying and Sharing*)

Trong hầu hết chương này, khi nói đến các kỹ thuật regularization, chúng ta thường tập trung vào việc thêm các ràng buộc hoặc hình phạt vào tham số mô hình nhằm ép các giá trị đó tiến gần về một điểm cụ thể — ví dụ như gốc tọa độ (giá trị 0) trong regularization chuẩn L^2 (còn gọi là weight decay).

Tuy nhiên, trong nhiều tình huống, chúng ta không chỉ muốn ràng buộc giá trị của một tham số đơn lẻ, mà còn muốn biểu diễn kiến thức về mối quan hệ giữa các tham số. Có thể chúng ta không biết chính xác từng tham số nên bằng bao nhiêu, nhưng lại biết rằng một số tham số nên giống nhau vì lý do kiến trúc mô hình hoặc kiến thức chuyên ngành.

Ví dụ, giả sử ta đang làm việc với hai mô hình khác nhau nhưng cùng giải một tác vụ phân loại, chỉ khác nhau về đầu vào. Khi đó, nếu $w^{(A)}$ và $w^{(B)}$ lần lượt là các tham số của hai mô hình, ta có thể kỳ vọng rằng chúng nên giống nhau ở mức độ nào đó.

Cụ thể, giả sử:

- Mô hình A sử dụng tham số $w^{(A)}$, đầu ra được tính bởi $\hat{y}^{(A)} = f(w^{(A)}, x)$,
- Mô hình B sử dụng tham số $w^{(B)}$, đầu ra được tính bởi $\hat{y}^{(B)} = g(w^{(B)}, x)$.

Nếu cả hai mô hình đang giải các bài toán gần giống nhau (ví dụ như cùng loại phân loại nhưng trên hai miền dữ liệu khác nhau), ta có thể thêm vào hàm mất mát một điều kiện regularization để khuyến khích hai bộ tham số giống nhau. Một ví dụ đơn giản là sử dụng:

$$\Omega(w^{(A)}, w^{(B)}) = \|w^{(A)} - w^{(B)}\|_2^2,$$

tức là áp dụng regularization chuẩn L^2 lên hiệu giữa hai bộ tham số. Dĩ nhiên, cũng có thể áp dụng chuẩn L^1 hoặc các loại khác nếu phù hợp với mục tiêu.

Phương pháp này được đề xuất trong công trình của **Lasserre2006**, trong đó mô hình được huấn luyện cùng lúc ở hai chế độ:

- Một chế độ học có giám sát (supervised learning),
- Và một chế độ học không giám sát (unsupervised learning).

Ý tưởng là ràng buộc các tham số của mô hình classifier học từ dữ liệu có nhãn sao cho chúng không quá khác biệt so với các tham số học được từ mô hình không giám sát, vốn được huấn luyện trên cùng một phân phối đầu vào. Cách làm này cho phép các mô hình "chia sẻ thông tin" trong quá trình học, từ đó cải thiện khả năng tổng quát hóa của hệ thống.

Có hai kỹ thuật phổ biến liên quan đến ý tưởng này:

1. Parameter tying (ràng buộc mềm): Ta thêm một điều kiện phạt để khuyến khích các tham số tiến lại gần nhau, nhưng không bắt buộc chúng phải bằng nhau hoàn toàn. Đây là cách làm mềm dẻo và linh hoạt hơn, như đã trình bày ở ví dụ trên.
2. Parameter sharing (ràng buộc cứng): Thay vì chỉ khuyến khích giống nhau, ta áp đặt các tham số phải đúng bằng nhau, tức là dùng chung một tham số trong nhiều phần của mô hình. Đây là một ràng buộc chặt chẽ hơn.

Một ví dụ điển hình của parameter sharing là trong mạng nơ-ron tích chập (CNN). Trong CNN, một kernel (bộ lọc) được áp dụng lặp lại trên toàn bộ ảnh đầu vào. Điều này tương đương với việc tất cả các trọng số của kernel được chia sẻ tại mọi vị trí — tức là một tham số duy nhất được áp dụng cho nhiều vị trí khác nhau.

Lợi ích của chia sẻ tham số trong trường hợp này bao gồm:

- Giảm đáng kể số lượng tham số cần học,
- Tận dụng được cấu trúc không gian của dữ liệu đầu vào (ví dụ như tính dịch chuyển bất biến trong ảnh),
- Tiết kiệm bộ nhớ và tăng hiệu quả tính toán.

Chia sẻ tham số là một kỹ thuật cực kỳ quan trọng và phổ biến trong các mạng nơ-ron hiện đại, đặc biệt là trong xử lý ảnh và chuỗi thời gian. Ngoài CNN, các kiến trúc như RNN hoặc Transformer cũng có thể áp dụng chia sẻ tham số giữa các lớp hoặc các bước thời gian.

7.10.1 Mạng nơ-ron tích chập (Convolutional Neural Networks)

Cho đến nay, ứng dụng phổ biến và thành công nhất của kỹ thuật chia sẻ tham số (*parameter sharing*) là trong các mạng nơ-ron tích chập (CNN — Convolutional Neural Networks), đặc biệt trong lĩnh vực thị giác máy tính.

Một lý do quan trọng khiến CNN hiệu quả với ảnh là vì ảnh tự nhiên thường có nhiều đặc tính thống kê mang tính bất biến theo phép tịnh tiến (*translation invariance*). Nói cách khác, nếu ta dịch một đối tượng trong ảnh (ví dụ như một con mèo) sang trái hoặc phải vài pixel, bản chất hình ảnh vẫn không thay đổi. Điều này có nghĩa là các đặc trưng cần phát hiện không phụ thuộc vào vị trí tuyệt đối trong ảnh mà phụ thuộc vào các mẫu cục bộ.

CNN tận dụng tính chất này bằng cách sử dụng cùng một tập trọng số tại nhiều vị trí khác nhau trong ảnh — thông qua phép tích chập (*convolution*). Cụ thể, một bộ lọc (ví dụ như kernel phát hiện biên hoặc mẫu hình tròn) được áp dụng cho toàn bộ ảnh. Nhờ vậy, nếu có một đặc trưng cần phát hiện, mạng vẫn nhận ra dù nó xuất hiện ở góc trái hay góc phải ảnh.

Lợi ích chính của việc chia sẻ tham số trong CNN bao gồm:

- Giảm số lượng tham số cần học: Thay vì học một trọng số riêng cho từng vị trí ảnh, ta chỉ học một kernel duy nhất áp dụng cho toàn bộ ảnh. Điều này giúp giảm đáng

kể số lượng tham số trong mô hình.

- Cho phép xây dựng mô hình lớn hơn: Với số tham số ít hơn, ta có thể xây dựng các mạng có nhiều tầng hơn, nhiều bộ lọc hơn mà không cần lượng dữ liệu huấn luyện tăng tương ứng.
- Tăng khả năng tổng quát hóa: Khi số lượng tham số giảm, nguy cơ overfitting cũng giảm theo. Mô hình có xu hướng học được những đặc trưng tổng quát hơn thay vì "học thuộc lòng" tập huấn luyện.

Tóm lại, chia sẻ tham số trong CNN là một ví dụ rất điển hình về việc đưa kiến thức chuyên môn (domain knowledge) vào thiết kế kiến trúc mạng. Thay vì để mô hình tự học mọi thứ từ đầu, ta thiết kế kiến trúc sao cho phù hợp với bản chất của dữ liệu — trong trường hợp này là sự bất biến theo tịnh tiến trong ảnh. Điều này không chỉ giúp mô hình học tốt hơn mà còn hiệu quả hơn rất nhiều về mặt tính toán.

Các chi tiết cụ thể hơn về cấu trúc và cơ chế hoạt động của mạng nơ-ron tích chập sẽ được trình bày trong Chương 9.

7.11 Biểu Diễn Thưa (Sparse Representations)

Weight decay (giảm trọng số) hoạt động bằng cách áp đặt một hình phạt trực tiếp lên các tham số của mô hình. Một chiến lược khác là áp dụng hình phạt lên hoạt động (activation) của các đơn vị trong mạng nơ-ron, nhằm khuyến khích các hoạt động này trở nên thưa (sparse). Điều này một cách gián tiếp sẽ áp dụng một hình phạt phức tạp lên các tham số của mô hình.

Chúng ta đã từng thảo luận (ở mục 7.1.2) rằng, việc áp dụng phạt L^1 lên tham số sẽ dẫn đến một mô hình với tham số thưa — nghĩa là nhiều trọng số sẽ có giá trị bằng hoặc gần bằng 0.

Ngược lại, *biểu diễn thưa (representational sparsity)* nói đến việc biểu diễn đầu ra bằng cách sao cho nhiều thành phần trong vector biểu diễn có giá trị bằng 0. Điều này được minh họa rõ hơn qua ví dụ hồi quy tuyến tính bên dưới.

Ví dụ 1: Mô hình tham số thưa

$$\underbrace{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}}_{\mathbf{y}\in\mathbb{R}^m} = \underbrace{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}}_{A\in\mathbb{R}^{m\times n}} \cdot \underbrace{\begin{bmatrix} 2 \\ 3 \\ -2 \\ 1 \\ -5 \\ 4 \end{bmatrix}}_{\mathbf{x}\in\mathbb{R}^n} \tag{7.46}$$

Trong ví dụ trên, chúng ta có một mô hình hồi quy tuyến tính trong đó ma trận A rất

thưa — nghĩa là có nhiều phần tử bằng 0. Đây là ví dụ của một mô hình có tham số thưa.

Ví dụ 2: Biểu diễn thưa

$$\underbrace{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}}_{\mathbf{y}\in\mathbb{R}^m} = \underbrace{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -1 & -3 \\ 3 & 1 & 2 & 0 & 3 & 2 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}}_{B\in\mathbb{R}^{m\times n}} \cdot \underbrace{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}_{\mathbf{h}\in\mathbb{R}^n} \tag{7.47}$$

Còn trong ví dụ thứ hai, ta thấy rằng vector $\mathbf{h}$ là một vector rất thưa — chỉ có hai phần tử khác 0. Mô hình này mô tả quá trình hồi quy tuyến tính nhưng sử dụng một biểu diễn thưa để mô tả đầu vào $\mathbf{x}$ thông qua $\mathbf{h}$.

Việc regularization trên biểu diễn được thực hiện tương tự như cách mà chúng ta áp dụng cho các tham số mô hình. Một cách phổ biến là thêm một hàm phạt theo chuẩn vào hàm mất mát J để điều chỉnh vector biểu diễn h. Hàm mất mát mới sẽ có dạng:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\mathbf{h}), \tag{7.48}$$

trong đó:

- $\Omega(h)$ là hàm phạt chuẩn (norm penalty),
- $\alpha \in [0, \infty)$ điều chỉnh mức độ ảnh hưởng của hàm phạt lên tổng mất mát — α càng lớn thì regularization càng mạnh.

Tương tự như việc sử dụng chuẩn L^1 trên các tham số để thúc đẩy mô hình có trọng số thưa, thì chuẩn L^1 trên biểu diễn h cũng có tác dụng làm cho h trở nên thưa:

$$\Omega(h) = \|h\|_1 = \sum_i |h_i|.$$

Dù chuẩn L^1 là một lựa chọn phổ biến để tạo ra biểu diễn thưa, vẫn còn nhiều lựa chọn khác như:

- Student t prior (Olshausen và Field, 1996; Bergstra, 2011),
- KL divergence penalty (Larochelle và Bengio, 2008),
- Regular hóa trung bình của giá trị kích hoạt: $\frac{1}{m}\sum_i h^{(i)} \approx 0.01$ (Lee et al., 2008; Goodfellow et al., 2009),

Những phương pháp này đặc biệt hữu ích khi muốn biểu diễn h với các thành phần nằm

trên miền giá trị nhỏ (ví dụ, chuẩn hóa về đoạn [0, 1]).

Ngoài ra, ta còn có thể áp dụng biểu diễn thưa bằng cách áp đặt ràng buộc cứng lên số lượng phần tử khác 0 trong vector h. Một phương pháp điển hình là:

Phương pháp này tìm một vector biểu diễn h sao cho:

$$\arg\min_{h, \|h\|_0 < k} \|\mathbf{x} - \mathbf{W}h\|_2^2, \tag{7.49}$$

trong đó:

- $\|h\|_0$ là số phần tử khác 0 trong h,
- $\mathbf{W}$ là ma trận cơ sở (dictionary matrix), thường được ràng buộc là trực giao (orthogonal),
- k là số lượng đặc trưng được chọn (được phép khác 0).

OMP còn được gọi là OMP-k, với k là số lượng đặc trưng mong muốn. Coates và Ng (2011) cho thấy rằng OMP-1 (chỉ chọn một đặc trưng) là một bộ trích đặc trưng rất hiệu quả trong các kiến trúc deep learning.

7.11.1 Bagging và các phương pháp Ensemble khác

Bagging (viết tắt của *bootstrap aggregating*) là một kỹ thuật giúp giảm lỗi khái quát hoá (generalization error) bằng cách kết hợp nhiều mô hình lại với nhau **Breiman1994**. Ý tưởng là huấn luyện nhiều mô hình khác nhau một cách độc lập, sau đó cho tất cả các mô hình này bỏ phiếu (vote) để đưa ra kết quả đầu ra cho tập kiểm tra. Đây là một ví dụ điển hình của chiến lược phổ biến trong học máy gọi là trung bình mô hình (model averaging). Những kỹ thuật theo chiến lược này còn được gọi là ensemble methods.

Lý do mà việc trung bình mô hình hoạt động hiệu quả là vì các mô hình khác nhau thường sẽ mắc các lỗi khác nhau trên tập kiểm tra.

Ví dụ: Xét một tập gồm k mô hình hồi quy. Giả sử mỗi mô hình mắc lỗi ϵ_i trên mỗi ví dụ, với các lỗi này được lấy từ phân phối chuẩn nhiều chiều có trung bình bằng 0, phương sai $\mathbb{E}[\epsilon_i^2] = v$, và hiệp phương sai $\mathbb{E}[\epsilon_i \epsilon_j] = c$. Khi đó lỗi của dự đoán trung bình từ tất cả các mô hình là:

$$\frac{1}{k}\sum_i \epsilon_i.$$

Kỳ vọng của bình phương lỗi (mean squared error) của hệ ensemble là:

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\right)\right], \tag{7.50}$$

$$= \frac{1}{k}v + \frac{k-1}{k}c. \tag{7.51}$$

Phân tích:

- Nếu các lỗi hoàn toàn tương quan ($c = v$), thì:

$$\text{MSE} = v,$$

  tức là trung bình mô hình không cải thiện được gì.

- Nếu các lỗi độc lập ($c = 0$), thì:

$$\text{MSE} = \frac{1}{k}v.$$

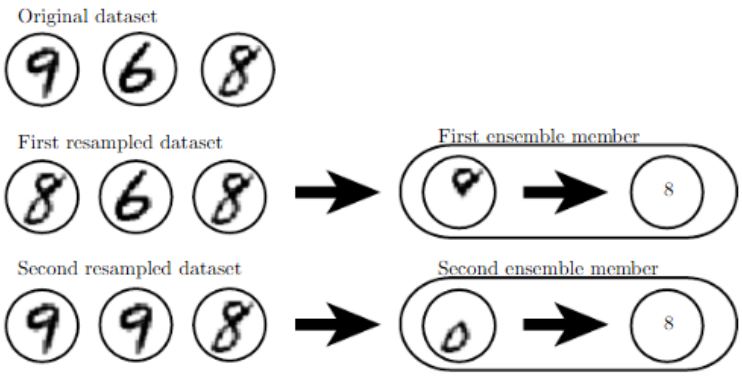
  Tức là lỗi giảm tỉ lệ nghịch với số lượng mô hình trong ensemble.

Ensemble sẽ hoạt động ít nhất cũng bằng một trong các mô hình thành phần, và nếu các mô hình là độc lập, thì hệ ensemble sẽ hoạt động tốt hơn từng mô hình đơn lẻ rất nhiều.

Các phương pháp ensemble khác nhau xây dựng bộ mô hình theo những cách khác nhau. Ví dụ, mỗi thành viên trong bộ mô hình có thể được xây dựng bằng cách huấn luyện một mô hình hoàn toàn khác biệt, sử dụng một thuật toán hoặc hàm mục tiêu khác nhau. Bagging là một phương pháp cho phép sử dụng cùng một loại mô hình, thuật toán huấn luyện và hàm mục tiêu được tái sử dụng nhiều lần.

Cụ thể, bagging liên quan đến việc tạo ra k bộ dữ liệu khác nhau. Mỗi bộ dữ liệu có cùng số lượng ví dụ như bộ dữ liệu ban đầu, nhưng mỗi bộ dữ liệu được tạo ra bằng cách lấy mẫu với thay thế từ bộ dữ liệu ban đầu. Điều này có nghĩa là, với xác suất cao, mỗi bộ dữ liệu sẽ thiếu một số ví dụ từ bộ dữ liệu ban đầu và chứa một số ví dụ bị trùng lặp[3]. Mô hình i sau đó sẽ được huấn luyện trên bộ dữ liệu i. Những sự khác biệt giữa các ví dụ được bao gồm trong mỗi bộ dữ liệu sẽ dẫn đến sự khác biệt giữa các mô hình đã được huấn luyện. Ví dụ Hình 7.5.

[3]Khi cả bộ dữ liệu gốc và bộ dữ liệu được lấy mẫu lại đều chứa m ví dụ, tỉ lệ chính xác của các ví dụ bị thiếu trong bộ dữ liệu mới là $\left(1 - \frac{1}{m}\right)^m$. Đây là khả năng mà một ví dụ cụ thể không được chọn trong tất cả các lần rút mẫu từ bộ dữ liệu nguồn với m lần rút mẫu. Khi m tiến về vô cùng, giá trị này hội tụ về $\frac{1}{e}$, một giá trị lớn hơn một chút so với $\frac{1}{3}$.



Hình 7.5: Ví dụ trực quan về sự khác biệt giữa các bộ dữ liệu được tạo bằng phương pháp bootstrap, từ đó tạo ra sự đa dạng trong các mô hình huấn luyện.

Mạng neural (Neural Networks) thường có đủ sự đa dạng trong các điểm giải pháp để có thể hưởng lợi từ việc trung bình hóa mô hình, ngay cả khi tất cả các mô hình đều được huấn luyện trên cùng một bộ dữ liệu. Sự khác biệt trong việc khởi tạo ngẫu nhiên, việc chọn lựa ngẫu nhiên các minibatches, các siêu tham số, hay các kết quả của việc triển khai không xác định của mạng neural thường đủ để khiến các thành viên khác nhau trong bộ mô hình mắc phải những lỗi độc lập một phần.

Trung bình hóa mô hình là một phương pháp rất mạnh mẽ và đáng tin cậy để giảm lỗi tổng quát. Tuy nhiên, phương pháp này thường không được khuyến khích khi so sánh các thuật toán trong các bài báo khoa học, bởi vì bất kỳ thuật toán học máy nào cũng có thể hưởng lợi đáng kể từ việc trung bình hóa mô hình, tuy nhiên điều này lại đòi hỏi tính toán và bộ nhớ nhiều hơn.

Vì lý do này, việc so sánh các thuật toán thường được thực hiện bằng cách sử dụng một mô hình duy nhất. Các cuộc thi học máy thường được thắng bởi các phương pháp sử dụng việc trung bình hóa mô hình qua hàng chục mô hình. Một ví dụ nổi bật gần đây là Giải thưởng Netflix Grand Prize **Koren2009**.

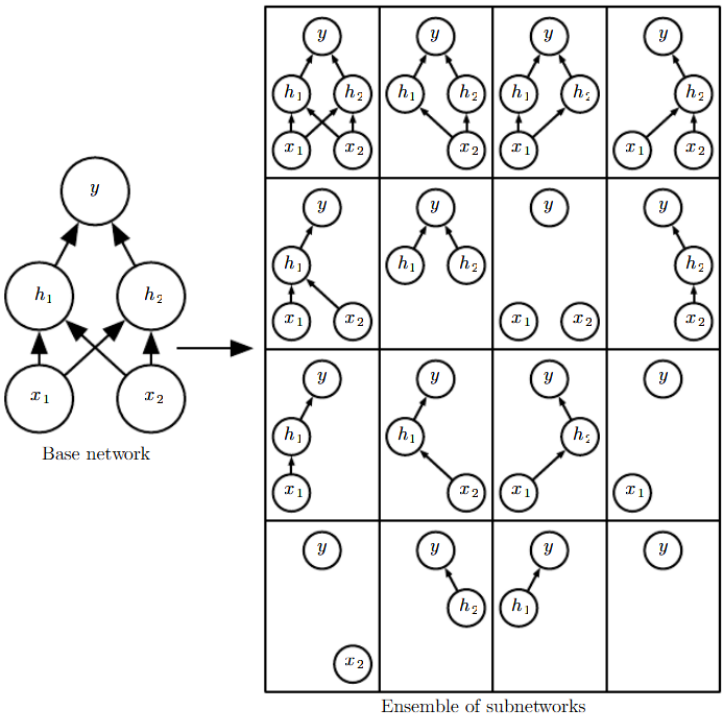
Không phải tất cả các kỹ thuật xây dựng bộ mô hình đều được thiết kế để làm cho bộ mô hình trở nên đều đặn hơn so với các mô hình cá nhân. Ví dụ, một kỹ thuật gọi là *boosting* **freund1996experiments**, **freund1997decision** xây dựng một bộ mô hình có khả năng cao hơn các mô hình cá nhân. Boosting đã được áp dụng để xây dựng các bộ mô hình từ các mạng neural **schwenk1998boosting** bằng cách thêm dần dần các mạng neural vào bộ mô hình. Boosting cũng đã được áp dụng để diễn giải một mạng neural cá nhân như một bộ mô hình **bengio2006convex**, thêm các đơn vị ẩn vào mạng theo từng bước.

7.12 Dropout

Dropout **srivastava2014dropout** cung cấp một phương pháp regularization tính toán rẻ nhưng mạnh mẽ cho một gia đình mô hình rộng lớn. Theo một cách tiếp cận đầu tiên, dropout có thể được coi là một phương pháp làm cho bagging trở nên thực tế đối với các bộ mô hình gồm rất nhiều mạng neural lớn. Bagging bao gồm việc huấn luyện nhiều mô

hình và đánh giá nhiều mô hình trên mỗi ví dụ kiểm tra. Điều này có vẻ không thực tế khi mỗi mô hình là một mạng neural lớn, vì việc huấn luyện và đánh giá các mạng như vậy tốn kém về thời gian và bộ nhớ. Thông thường, người ta sử dụng các bộ mô hình gồm từ năm đến mười mạng neural — Szegedy et al. **szegedy2014going** đã sử dụng sáu mạng để chiến thắng ILSVRC — nhưng khi vượt quá con số này thì việc quản lý trở nên bất tiện. Dropout cung cấp một phép xấp xỉ rẻ tiền để huấn luyện và đánh giá một bộ mô hình bagging gồm rất nhiều mạng neural.

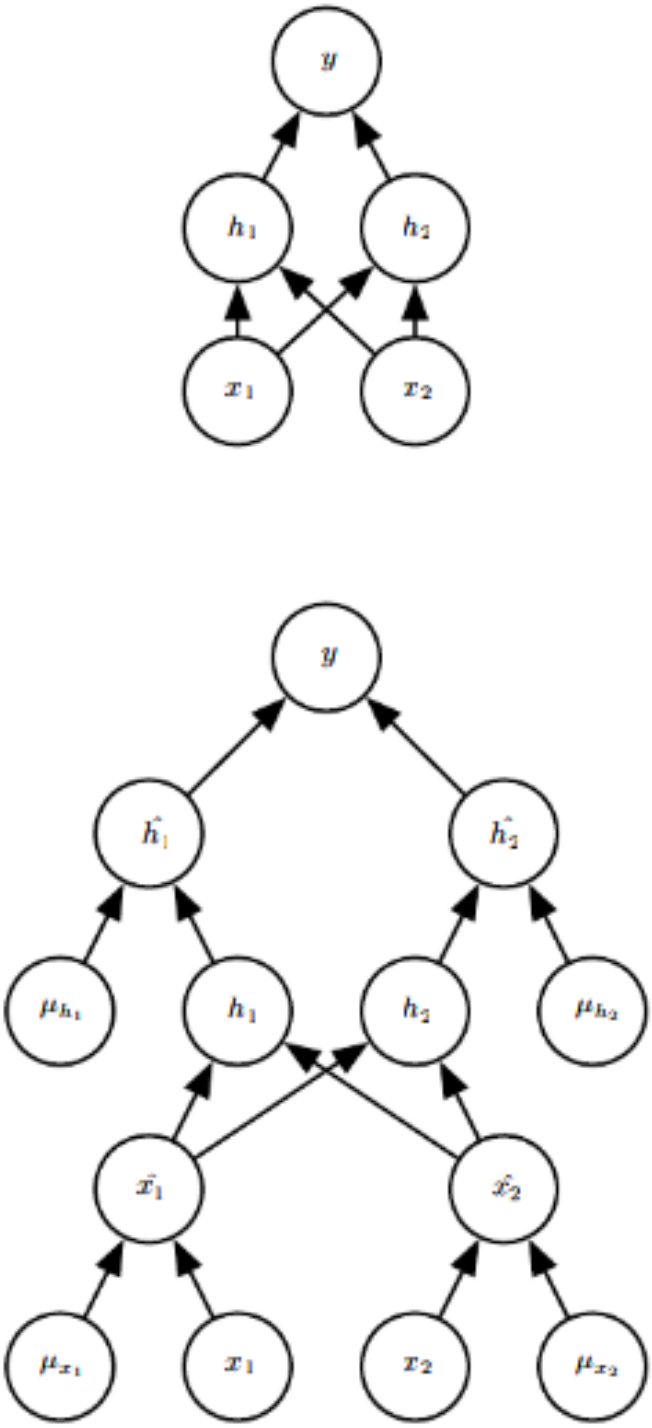
Cụ thể, dropout huấn luyện bộ mô hình gồm tất cả các subnetwork có thể được tạo ra bằng cách loại bỏ các đơn vị không phải là đầu ra từ một mạng cơ sở. Như được minh họa trong Hình 7.6, trong hầu hết các mạng neural hiện đại, dựa trên một loạt các biến đổi affine và các hàm phi tuyến, chúng ta có thể hiệu quả loại bỏ một đơn vị khỏi mạng bằng cách nhân giá trị đầu ra của nó với không. Quy trình này yêu cầu một số sửa đổi nhẹ đối với các mô hình như mạng radial basis function, vốn tính toán sự khác biệt giữa trạng thái của đơn vị và một giá trị tham chiếu. Ở đây, chúng ta trình bày thuật toán dropout dưới dạng nhân với không để đơn giản, nhưng nó có thể dễ dàng được sửa đổi để hoạt động với các phép toán khác loại bỏ một đơn vị khỏi mạng.



Hình 7.6: Dropout huấn luyện một bộ mô hình bao gồm tất cả các subnetwork có thể được tạo ra bằng cách loại bỏ các đơn vị không phải là đầu ra từ một mạng cơ sở. Ở đây, chúng ta bắt đầu với một mạng cơ sở có hai đơn vị đầu vào và hai đơn vị ẩn. Có mười sáu tập con có thể có từ bốn đơn vị này. Chúng ta hiển thị tất cả mười sáu subnetwork có thể được tạo ra bằng cách loại bỏ các tập con khác nhau của các đơn vị từ mạng gốc. Trong ví dụ nhỏ này, một tỷ lệ lớn các mạng kết quả không có đơn vị đầu vào hoặc không có đường nối giữa đầu vào và đầu ra. Vấn đề này trở nên không đáng kể đối với các mạng có các lớp rộng hơn, nơi xác suất loại bỏ tất cả các đường dẫn có thể có từ đầu vào đến đầu ra trở nên nhỏ hơn.

Nhớ lại rằng để học với bagging, chúng ta định nghĩa k mô hình khác nhau, xây dựng k bộ dữ liệu khác nhau bằng cách lấy mẫu từ tập huấn luyện với thay thế, và sau đó huấn luyện mô hình i trên bộ dữ liệu i. Dropout nhằm mục đích xấp xỉ quá trình này, nhưng với một số lượng mạng neural vô cùng lớn. Cụ thể, để huấn luyện với dropout, chúng ta sử dụng một thuật toán học dựa trên minibatch, như stochastic gradient descent. Mỗi khi chúng ta nạp một ví dụ vào minibatch, chúng ta sẽ lấy mẫu ngẫu nhiên một mặt nạ nhị phân khác để áp dụng cho tất cả các đơn vị đầu vào và ẩn trong mạng. Mặt nạ cho mỗi đơn vị được lấy mẫu độc lập với tất cả các đơn vị khác. Xác suất lấy mẫu giá trị mặt nạ bằng một (khiến đơn vị được bao gồm) là một siêu tham số được cố định trước khi bắt đầu huấn luyện. Nó không phải là một hàm của giá trị hiện tại của các tham số mô hình hoặc ví dụ đầu vào. Thông thường, một đơn vị đầu vào được bao gồm với xác suất 0.8, và một đơn vị ẩn được bao gồm với xác suất 0.5. Sau đó, chúng ta thực hiện quá trình lan truyền xuôi, lan truyền ngược và cập nhật học như thường lệ. Hình 7.7 minh họa cách thực hiện lan truyền xuôi với dropout.

x



Hình 7.7: Ví dụ lan truyền tiến qua mạng nơ-ron với kỹ thuật Dropout. (Phía trên): Trong ví dụ này, chúng ta sử dụng một mạng nơ-ron truyền thẳng (feedforward) với hai nút đầu vào, một tầng ẩn có hai nút ẩn và một nút đầu ra. (Phía dưới): Để thực hiện lan truyền tiến với Dropout, chúng ta lấy mẫu ngẫu nhiên một vector mặt nạ $\boldsymbol{\mu}$ với một phần tử tương ứng cho mỗi nút đầu vào hoặc nút ẩn trong mạng. Các phần tử của μ là nhị phân (0 hoặc 1) và được lấy mẫu độc lập với nhau. Xác suất để mỗi phần tử có giá trị bằng 1 (tức là giữ lại nút tương ứng) là một siêu tham số được xác định trước, thông thường là 0.8 đối với các nút đầu vào và 0.5 đối với các nút ẩn. Mỗi nút trong mạng sẽ được nhân với giá trị tương ứng trong mặt nạ μ. Những nút có mặt nạ bằng 0 sẽ bị loại khỏi mạng trong bước lan truyền tiến hiện tại. Sau đó, quá trình lan truyền tiến được tiếp tục như bình thường qua phần còn lại của mạng. Quá trình này tương đương với việc chọn ngẫu nhiên một mạng con trong tất cả các mạng con có thể hình thành từ mạng gốc (như hình 7.6), và thực hiện lan truyền tiến qua mạng con đó.

Một cách chính xác hơn, giả sử ta có một vector mặt nạ μ xác định các đơn vị nào sẽ được giữ lại trong mạng, và $J(\theta, \mu)$ là hàm mất mát của mô hình được xác định bởi tham số θ và mặt nạ μ. Khi đó, huấn luyện bằng dropout tương đương với việc tối thiểu hóa kỳ vọng sau:

$$\mathbb{E}_{\mu} J(\theta, \mu)$$

Biểu thức trên chứa số lượng tổ hợp hàm số mũ các mặt nạ có thể, nhưng ta có thể ước lượng gradient của nó một cách không chệch bằng cách *lấy mẫu ngẫu nhiên các giá trị của* μ.

Khác với bagging, các mô hình trong dropout *chia sẻ tham số*. Mỗi mô hình con là một mạng con (subnetwork) được sinh ra bằng cách loại bỏ một số node ẩn/ngõ vào. Việc chia sẻ tham số này giúp biểu diễn số lượng mô hình hàm số mũ chỉ với một lượng bộ nhớ hữu hạn.

Trong bagging, khi dự đoán, ta lấy trung bình số học các phân phối xác suất đầu ra từ từng mô hình:

$$\frac{1}{k} \sum_{i=1}^{k} p^{(i)}(y \mid x) \tag{7.52}$$

Còn trong dropout, mỗi subnetwork tương ứng với mặt nạ μ sẽ sinh ra một phân phối $p(y \mid x, \mu)$, và dự đoán tổng thể là:

$$\sum_{\mu} p(\mu) p(y \mid x, \mu) \tag{7.53}$$

Do số lượng μ là hàm số mũ, việc tính chính xác là bất khả thi. Thay vào đó, ta *lấy mẫu ngẫu nhiên một số mặt nạ* và trung bình đầu ra. Ngay cả 10–20 mẫu cũng có thể mang lại hiệu suất tốt.

Ngoài trung bình số học, một cách tiếp cận khác để kết hợp các mô hình con trong dropout là sử dụng trung bình hình học các phân phối đầu ra. Phương pháp này có ưu điểm là chỉ cần một lần lan truyền tiến duy nhất, giúp tiết kiệm tài nguyên tính toán đáng kể mà vẫn đạt hiệu quả tương đương Warde-Farley, Goodfellow, Courville **and** Bengio [15].

Tuy nhiên, trung bình hình học của các phân phối xác suất không phải lúc nào cũng tạo thành một phân phối hợp lệ — cụ thể là tổng xác suất có thể khác 1. Để đảm bảo kết quả sau cùng là một phân phối xác suất đúng nghĩa, ta yêu cầu mỗi mô hình con đều phải gán xác suất khác 0 cho mọi nhãn đầu ra, và sau đó ta thực hiện một bước chuẩn hóa.

Phân phối xác suất chưa được chuẩn hóa, thu được từ trung bình hình học trên tất cả

các mô hình con, được định nghĩa bởi:

$$\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x}) = 2^d \sqrt[d]{\prod_{\mu} p(y \mid \mathbf{x}, \mu)}, \tag{7.54}$$

trong đó d là số lượng đơn vị có thể bị dropout. Ta giả định các mặt nạ dropout μ được lấy từ phân phối đều để đơn giản hóa phân tích, nhưng các phân phối không đều cũng có thể áp dụng.

Để đưa ra dự đoán cuối cùng, ta chuẩn hóa lại phân phối:

$$p_{\text{ensemble}}(y \mid \mathbf{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \mathbf{x})}. \tag{7.55}$$

Một phát hiện quan trọng từ công trình của **hinton2012** là ta có thể xấp xỉ rất tốt phân phối tổ hợp $p_{\text{ensemble}}(y \mid \mathbf{x})$ chỉ bằng một mô hình duy nhất. Cụ thể, ta huấn luyện mô hình như bình thường, nhưng khi suy luận (test), ta nhân tất cả trọng số đi ra từ một đơn vị i với xác suất giữ lại của đơn vị đó. Cách tiếp cận này gọi là quy tắc nhân trọng số (weight scaling rule).

Mặc dù chưa có chứng minh lý thuyết đầy đủ, phương pháp này thường cho kết quả rất tốt trên thực tế. Ví dụ, nếu xác suất giữ lại là 0.5, thì trong giai đoạn test, ta có thể nhân trọng số với 0.5 (hoặc nhân đầu ra với 2 trong giai đoạn huấn luyện) để giữ cho kỳ vọng đầu ra không thay đổi.

Trong một số mô hình tuyến tính hoặc không chứa phi tuyến tính mạnh, quy tắc nhân trọng số có thể được chứng minh là chính xác. Xét bài toán phân lớp softmax đơn giản với đầu vào $\mathbf{v}$:

$$P(y = y \mid \mathbf{v}) = \text{softmax}(\mathbf{W}^\top \mathbf{v} + \mathbf{b})_y \tag{7.56}$$

Nếu ta đánh chỉ mục từng mô hình con bằng mặt nạ nhị phân $\mathbf{d}$, quá trình dropout có thể mô phỏng bằng:

$$P(y = y \mid \mathbf{v}; \mathbf{d}) = \text{softmax}(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y \tag{7.57}$$

Dự đoán tổ hợp được xác định bằng cách chuẩn hóa lại trung bình hình học của các phân phối đầu ra từ toàn bộ các mô hình con:

$$P_{\text{ensemble}}(y = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(y' = y' \mid \mathbf{v})} \tag{7.58}$$

với:

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) = \sqrt[2^d]{\prod_{\mathbf{d} \in \{0,1\}^d} P(y = y \mid \mathbf{v}, \mathbf{d})} \tag{7.59}$$

Để chứng minh rằng quy tắc nhân trọng số là chính xác, ta khai triển biểu thức bên trong:

$$= \sqrt[2^d]{\prod_{\mathbf{d} \in \{0,1\}^d} \text{softmax}(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y} \tag{7.61}$$

$$= \sqrt[2^d]{\prod_{\mathbf{d} \in \{0,1\}^d} \frac{\exp(\mathbf{W}_y^\top (\mathbf{d} \odot \mathbf{v}) + b_y)}{\sum_{y'} \exp(\mathbf{W}_{y'}^\top (\mathbf{d} \odot \mathbf{v}) + b_{y'})}} \tag{7.62}$$

$$= \frac{\sqrt[2^d]{\prod_{\mathbf{d}} \exp(\mathbf{W}_y^\top (\mathbf{d} \odot \mathbf{v}) + b_y)}}{\sqrt[2^d]{\prod_{\mathbf{d}} \sum_{y'} \exp(\mathbf{W}_{y'}^\top (\mathbf{d} \odot \mathbf{v}) + b_{y'})}} \tag{7.63}$$

Vì $\tilde{P}$ sẽ được chuẩn hóa trong quá trình tính toán, ta có thể đơn giản hoá tử số:

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + b_y)} \tag{7.53}$$

$$= \exp\left(\frac{1}{2^n} \sum_{\mathbf{d} \in \{0,1\}^n} \mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + b_y \right) \tag{7.54}$$

$$= \exp\left(\frac{1}{2} \mathbf{W}_{y,:}^\top \mathbf{v} + b_y \right) \tag{7.55}$$

Cuối cùng, khi thay biểu thức trên vào phương trình (7.58), ta thu được bộ phân loại softmax với trọng số đã được điều chỉnh theo quy tắc nhân trọng số — cụ thể là trọng số bị nhân với hệ số $\frac{1}{2}$, tương ứng với xác suất giữ lại là 0.5.

Quy tắc chia tỷ trọng (weight scaling rule) vẫn tỏ ra chính xác trong nhiều trường hợp, ví dụ như các mạng hồi quy có đầu ra tuân theo phân phối chuẩn có điều kiện, hoặc các mạng sâu không có phi tuyến trong các lớp ẩn. Tuy nhiên, đối với các mạng nơ-ron sâu có chứa các hàm kích hoạt phi tuyến như ReLU, quy tắc này chỉ là một xấp xỉ. Dù chưa có phân tích lý thuyết chắc chắn, thực nghiệm cho thấy nó hoạt động rất hiệu quả trong nhiều trường hợp.

Goodfellow2013a cho thấy rằng quy tắc chia tỷ trọng thậm chí có thể đạt độ chính xác cao hơn cả việc trung bình Monte Carlo trên các mô hình con — ngay cả khi số lượng mẫu Monte Carlo lên đến 1,000. Ngược lại, **Gal2015** phát hiện rằng trong một số tình huống, việc sử dụng khoảng 20 mẫu Monte Carlo có thể vượt trội hơn. Điều này cho thấy

rằng lựa chọn phương pháp xấp xỉ tối ưu phụ thuộc vào bài toán cụ thể.

Srivastava2014 cũng chứng minh rằng dropout hiệu quả hơn nhiều kỹ thuật regularization truyền thống như: weight decay, ràng buộc chuẩn lọc (filter norm constraints), hay regularization dựa trên độ thưa. Ngoài ra, dropout cũng có thể kết hợp với các phương pháp regularization khác để tăng thêm hiệu quả.

Một trong những điểm mạnh đáng chú ý của dropout là chi phí tính toán thấp. Khi áp dụng trong quá trình huấn luyện, nó chỉ yêu cầu $O(n)$ phép nhân với các số nhị phân ngẫu nhiên và $O(n)$ bộ nhớ để lưu mặt nạ nhị phân phục vụ cho lan truyền ngược. Trong giai đoạn suy diễn (inference), dropout không làm tăng chi phí tính toán so với một mạng thông thường, ngoại trừ việc điều chỉnh trọng số (thường bằng phép nhân đơn giản).

Dropout cũng rất linh hoạt vì không phụ thuộc vào cấu trúc cụ thể hay quy trình huấn luyện. Nó có thể áp dụng hiệu quả cho nhiều loại mô hình như mạng nơ-ron truyền thẳng (feedforward), Restricted Boltzmann Machine (RBM), hay mạng hồi tiếp (RNN). Một số kỹ thuật regularization mạnh khác lại hạn chế kiểu mô hình có thể áp dụng.

Tuy nhiên, cần lưu ý rằng dropout làm giảm khả năng biểu diễn của mạng. Để bù lại, ta cần tăng kích thước mạng và thời gian huấn luyện. Khi làm việc với các tập dữ liệu rất lớn, regularization nói chung ít cần thiết hơn, nên dropout có thể không đem lại nhiều lợi ích.

Khi lượng dữ liệu có nhãn rất ít (dưới 5,000 ví dụ), dropout cũng kém hiệu quả hơn các phương pháp theo quan điểm Bayes. Ví dụ, mạng Bayes của **Neal1996** đã cho kết quả tốt hơn dropout trên tập dữ liệu Alternative Splicing **Xiong2011**. Trong trường hợp này, học không giám sát để trích đặc trưng từ trước có thể hiệu quả hơn so với dropout.

Wager2013 chỉ ra rằng trong hồi quy tuyến tính, dropout tương đương với regularization L^2, nhưng hệ số regularization sẽ thay đổi tùy theo phương sai của từng đặc trưng. Tuy nhiên, đối với các mô hình sâu, điều này không còn đúng nữa.

Một biến thể đáng chú ý của dropout là DropConnect **Wan2013**. Thay vì loại bỏ đơn vị, DropConnect loại bỏ các kết nối riêng lẻ — tức là, mỗi sản phẩm giữa trọng số và trạng thái của đơn vị ẩn được coi như một đơn vị có thể bị drop. Ngoài ra, stochastic pooling cũng là một kỹ thuật ensemble dành cho mạng tích chập, cho phép mạng học cách chú ý vào những vị trí khác nhau trong bản đồ đặc trưng.

Trong số các phương pháp tạo ensemble ẩn (implicit ensemble methods), dropout vẫn là kỹ thuật được sử dụng phổ biến nhất.

Dropout có thể được xem như một hình thức bagging hiệu quả kết hợp với chia sẻ tham số. Mỗi đơn vị ẩn phải học cách hoạt động tốt kể cả khi một số đơn vị khác bị loại bỏ ngẫu nhiên trong mạng. Điều này buộc các đơn vị học được những đặc trưng "mạnh mẽ", không quá phụ thuộc vào sự hỗ trợ của các đơn vị khác. **Hinton2012** đã liên hệ ý tưởng này với cơ chế tiến hóa sinh học — nơi gen cần hoạt động tốt trong nhiều tổ hợp di truyền khác nhau do quá trình sinh sản tái tổ hợp.

Một trong những lý do khiến dropout hoạt động hiệu quả là vì nó phá vỡ thông tin đã học ở các tầng ẩn, thay vì chỉ phá vỡ đầu vào ban đầu. Ví dụ, nếu một đơn vị học cách phát hiện mũi để nhận diện khuôn mặt, việc drop đơn vị đó có nghĩa là mất đi thông tin về chiếc mũi. Mạng sẽ phải học thêm một đơn vị khác để phát hiện miệng hoặc các đặc trưng tổng thể khác, nhờ đó mà trở nên tổng quát hơn và ít phụ thuộc vào một đặc trưng cụ thể.

Khác với các kỹ thuật thêm nhiễu truyền thống (thường là nhiễu cộng vào đầu vào), dropout thực hiện việc loại bỏ thông tin một cách chọn lọc và thích ứng với phân phối dữ liệu đầu vào.

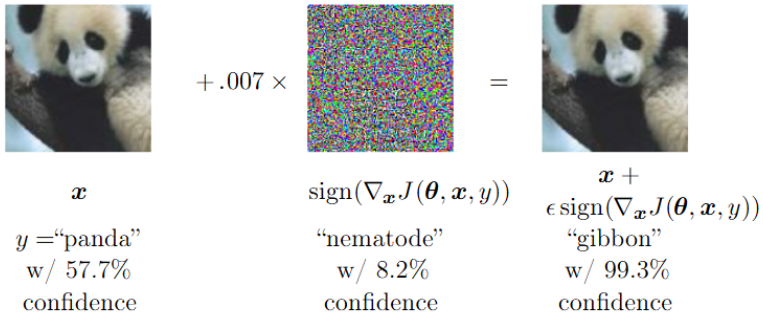
Ngoài ra, dropout áp dụng nhiễu theo cách nhân (multiplicative noise), khiến các đơn vị không thể bù đắp việc dropout bằng cách đơn giản là phóng đại giá trị đầu ra của chúng. Điều này khác với khi thêm nhiễu cộng, trong đó các đơn vị có thể dễ dàng học cách bỏ qua nhiễu.

Một kỹ thuật phổ biến khác là Batch Normalization (BN) cũng áp dụng cả nhiễu cộng và nhiễu nhân lên các tầng ẩn trong quá trình huấn luyện. Dù mục tiêu chính của BN là tối ưu hóa quá trình huấn luyện, nhưng nó cũng có tác dụng regularization rõ rệt. Trong một số trường hợp, BN có thể thay thế dropout mà vẫn duy trì hiệu quả huấn luyện. Chi tiết hơn sẽ được trình bày trong Mục **??**.

7.13 Huấn luyện với Tấn công đối kháng (Adversarial Training)

Trong nhiều bài toán phân loại, các mạng nơ-ron hiện đại đã đạt hiệu suất tương đương hoặc thậm chí vượt qua con người khi đánh giá trên các bộ dữ liệu kiểm tra độc lập và phân phối giống nhau (i.i.d.). Tuy nhiên, điều này dẫn đến câu hỏi: liệu các mô hình này có thực sự "hiểu"bài toán ở mức độ như con người hay không?

Để đánh giá mức độ hiểu biết đó, một phương pháp hiệu quả là tìm kiếm những ví dụ khiến mô hình phân loại sai. **Szegedy2014a** đã phát hiện rằng, ngay cả những mạng có độ chính xác rất cao cũng có thể mắc lỗi 100% trên những ví dụ được tạo ra có chủ đích. Các ví dụ này được tạo bằng cách tối ưu hóa đầu vào ban đầu x, tạo ra một đầu vào mới x' rất gần với x nhưng khiến mô hình dự đoán hoàn toàn sai. Thú vị ở chỗ, x' vẫn rất giống x đến mức con người không nhận ra sự khác biệt, nhưng mô hình lại thay đổi kết quả phân loại.



Hình 7.8: Ví dụ về ví dụ đối kháng tạo ra trên GoogLeNet (Szegedy et al., 2014a) huấn luyện trên ImageNet. Bằng cách cộng vào ảnh gốc một nhiễu rất nhỏ theo hướng của gradient hàm mất mát, mô hình có thể bị đánh lừa hoàn toàn. Hình ảnh được trích từ **Goodfellow2014b**.

Những ví dụ này không chỉ liên quan đến vấn đề bảo mật mà còn mở ra hướng tiếp cận mới cho việc regularization — bằng cách huấn luyện mô hình trên các ví dụ đối kháng để giảm lỗi tổng quát. Cách làm này được gọi là adversarial training, được đề xuất bởi **Goodfellow2014b**.

Các tác giả chỉ ra rằng nguyên nhân chính dẫn đến tính dễ bị tấn công của mạng là vì mô hình có tính tuyến tính quá cao. Mạng nơ-ron hiện đại thường sử dụng nhiều khối tuyến tính (như nhân ma trận), khiến hàm tổng thể gần như tuyến tính. Điều này giúp quá trình tối ưu trở nên dễ dàng, nhưng đồng thời khiến hàm đầu ra rất nhạy với các thay đổi nhỏ trong đầu vào.

Ví dụ, nếu đầu vào là một vector chiều cao, một thay đổi nhỏ ϵ ở mỗi chiều có thể làm đầu ra thay đổi đến $\epsilon \|w\|_1$ — một giá trị rất lớn nếu w có nhiều phần tử. Adversarial training ngăn điều này bằng cách buộc mô hình học các hàm ổn định (gần như hằng số) quanh vùng dữ liệu huấn luyện, tức là đưa thêm một giả định tiên nghiệm (prior) về tính *hằng cục bộ* vào mô hình.

Một điểm thú vị là mô hình tuyến tính hoàn toàn (như hồi quy logistic) không thể kháng cự lại các ví dụ đối kháng vì bị giới hạn bởi chính cấu trúc tuyến tính của nó. Ngược lại, mạng nơ-ron có thể biểu diễn cả các xu hướng tuyến tính và phi tuyến, nhờ đó có thể vừa học từ dữ liệu vừa kháng lại nhiễu nhỏ một cách hiệu quả.

Virtual adversarial training (Miyato et al., 2015) mở rộng ý tưởng này cho bài toán học bán giám sát. Giả sử tại một điểm dữ liệu không gán nhãn x, mô hình gán cho nó nhãn dự đoán $\hat{y}$. Mặc dù nhãn này không chắc chắn là đúng, nhưng vẫn có khả năng cao nếu mô hình đã được huấn luyện tốt.

Ý tưởng là tạo một ví dụ đối kháng x' sao cho x' vẫn gần x, nhưng mô hình lại dự đoán một nhãn khác $y' \neq \hat{y}$. Mục tiêu là buộc mô hình học sao cho cả x và x' cùng đưa ra một nhãn duy nhất, qua đó tăng tính ổn định của mô hình đối với các nhiễu nhỏ trên manifold của dữ liệu đầu vào. Phương pháp này giả định rằng các lớp dữ liệu nằm trên các manifold tách biệt, và một nhiễu nhỏ không nên khiến dữ liệu "nhảy"sang một manifold của lớp khác.

Adversarial training là một kỹ thuật regularization mạnh mẽ, vừa tăng tính tổng quát của mô hình, vừa cải thiện khả năng chống lại các tấn công nhạy cảm — đặc biệt hữu ích trong các ứng dụng an toàn và học bán giám sát.

7.14 Tangent Distance, Tangent Prop và Manifold Tangent Classifier

Một hướng tiếp cận quan trọng để khắc phục "lời nguyền chiều" là giả định rằng dữ liệu không phân bố ngẫu nhiên trong toàn bộ không gian đầu vào, mà tập trung gần một đa tạp (manifold) có chiều thấp hơn. Giả định này đã được giới thiệu ở mục 5.11.3, và một trong những thuật toán đầu tiên khai thác giả định này là Tangent Distance **Simard1993**, **Simard1998**.

Tangent Distance là một thuật toán tìm láng giềng gần không tham số, trong đó không sử dụng khoảng cách Euclid thông thường mà dùng khoảng cách phản ánh hình học của các đa tạp mà điểm dữ liệu thuộc về. Ý tưởng là: nếu hai điểm dữ liệu x_1 và x_2 thuộc cùng một đa tạp, thì chúng thuộc cùng một lớp, và khoảng cách tốt nhất để phân loại là khoảng cách giữa các đa tạp M_1 và M_2 mà chúng nằm trên.

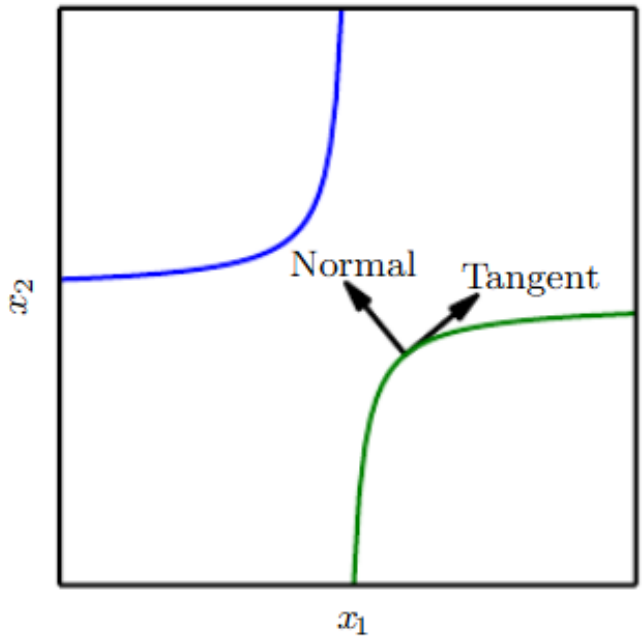
Tuy nhiên, việc tính khoảng cách giữa hai đa tạp là rất phức tạp. Vì vậy, một cách tiếp cận thực tế là xấp xỉ đa tạp bằng mặt phẳng tiếp tuyến tại điểm dữ liệu. Từ đó, ta có thể đo khoảng cách giữa một điểm với mặt phẳng, hoặc giữa hai mặt phẳng tiếp tuyến. Việc này được thực hiện thông qua giải một hệ tuyến tính nhỏ, tương ứng với chiều của đa tạp. Dĩ nhiên, phương pháp này đòi hỏi có thông tin về các vectơ tiếp tuyến.

Cùng với tinh thần đó, Tangent Propagation (Tangent Prop) **Simard1992** là một kỹ thuật regularization cho mạng nơ-ron, trong đó ta bổ sung thêm một hình phạt vào hàm mất mát để khuyến khích mạng không thay đổi đầu ra $f(x)$ khi đầu vào x bị biến đổi theo các hướng biết trước.

Các hướng biến đổi này tương ứng với chuyển động dọc theo đa tạp, ví dụ như các phép biến đổi nhỏ trong ảnh như xoay, phóng to, thu nhỏ, dịch chuyển. Để đạt được tính bất biến cục bộ đó, yêu cầu là đạo hàm của $f(x)$ theo hướng tiếp tuyến $v(i)$ phải nhỏ, tức là:

$$\Omega(f) = \sum_i \left(\nabla_x f(x)^\top v(i)\right)^2 \tag{7.56}$$

Hình phạt $\Omega(f)$ trên sẽ được cộng vào hàm mất mát chính và điều chỉnh bằng một siêu tham số.



Hình 7.9: Minh họa trực quan cho Tangent Propagation và Manifold Tangent Classifier. Mỗi đường cong là một đa tạp đại diện cho một lớp. Tại một điểm trên đường cong, vectơ tiếp tuyến thể hiện hướng mà trong đó đầu vào có thể biến đổi mà không làm thay đổi nhãn lớp. Phương pháp Tangent Prop điều chỉnh mô hình sao cho không nhạy cảm với những biến đổi này. Manifold Tangent Classifier đi xa hơn bằng cách học các hướng tiếp tuyến này từ dữ liệu thay vì chỉ định thủ công.

Tangent Prop từng được sử dụng hiệu quả không chỉ trong học có giám sát mà còn cả trong học củng cố **Thrun1995**. Phương pháp này có mối quan hệ gần với tăng cường dữ liệu (dataset augmentation). Cả hai phương pháp đều sử dụng kiến thức từ domain để làm cho mô hình trở nên bất biến với những biến đổi cụ thể.

Khác biệt ở chỗ:

- Tangent Prop điều chỉnh mô hình bằng đạo hàm, không cần thêm điểm dữ liệu mới.
- Dataset Augmentation tạo các biến thể thực sự của dữ liệu và yêu cầu mô hình dự đoán đúng trên những biến thể đó.

Tuy nhiên, Tangent Prop có hai hạn chế chính:

1. Nó chỉ đảm bảo mô hình ổn định với những thay đổi vô cùng nhỏ (infinitesimal), trong khi dataset augmentation có thể xử lý biến đổi lớn hơn.
2. Tangent Prop khó áp dụng với các mạng dùng hàm kích hoạt ReLU, vì ReLU không có vùng bão hòa để làm nhỏ đạo hàm. Điều này khiến việc giảm độ nhạy của hàm trở nên khó khăn hơn so với sigmoid hoặc tanh.

Tangent Prop cũng liên quan đến:

- Double backpropagation **Drucker1992**: giảm độ lớn của Jacobian bằng regularization.

- Adversarial training **Szegedy2014b**, **Goodfellow2014b**: tạo điểm nhiễu gần với dữ liệu gốc và buộc mô hình phải ổn định với nhiễu đó.

Manifold Tangent Classifier **Rifai2011c** cải tiến Tangent Prop bằng cách loại bỏ nhu cầu phải biết trước vectơ tiếp tuyến. Thay vào đó, phương pháp này sử dụng autoencoder để học đa tạp từ dữ liệu không giám sát và sau đó trích xuất các vectơ tiếp tuyến từ autoencoder. Những vectơ này giúp mô hình trở nên bất biến với các biến đổi phù hợp hơn và phản ánh đúng cấu trúc dữ liệu — bao gồm cả các đặc trưng hình học (như dịch chuyển, xoay) và đặc trưng khái niệm (như cử động của bộ phận cơ thể).

Thuật toán tổng thể là:

1. Huấn luyện một autoencoder trên dữ liệu không giám sát để học đa tạp.
2. Trích xuất các vectơ tiếp tuyến từ autoencoder.
3. Áp dụng hình phạt như Tangent Prop để điều chỉnh bộ phân loại.

Như vậy, chương này đã trình bày nhiều chiến lược điều chỉnh khác nhau cho mạng nơ-ron. Trong các chương tiếp theo, đặc biệt là chương tiếp theo về tối ưu hóa, chúng ta sẽ tiếp tục quay lại và sử dụng những kỹ thuật điều chuẩn này như thành phần thiết yếu của quá trình huấn luyện mô hình học sâu.

CHƯƠNG 8. TỐI ƯU HÓA TRONG HUẤN LUYỆN CÁC MÔ HÌNH SÂU

Các thuật toán trong học sâu thường gắn liền với các bài toán tối ưu hóa dưới nhiều hình thức khác nhau. Chẳng hạn ngay cả các kỹ thuật suy luận trong mô hình tuyến tính như PCA (Phân tích thành phần chính) cũng yêu cầu giải bài toán tối ưu. Ngoài ra, nhiều chứng minh lý thuyết hoặc thiết kế thuật toán cũng dựa trên các công cụ tối ưu hóa phân tích.

Tuy nhiên, trong toàn bộ hệ sinh thái học sâu, bài toán khó khăn và tốn kém nhất chính là tối ưu hóa trong quá trình huấn luyện mạng nơ-ron. Việc dành ra từ vài ngày đến vài tháng trên hàng trăm GPU để huấn luyện một mô hình mạng nơ-ron là hoàn toàn phổ biến trong thực tế. Chính vì lý do đó, một bộ kỹ thuật tối ưu hóa đặc thù đã được phát triển để giải quyết bài toán này một cách hiệu quả.

Chương này sẽ tập trung vào các kỹ thuật tối ưu hóa thực tiễn được sử dụng trong huấn luyện mạng nơ-ron.

Nếu bạn chưa quen với các nguyên lý cơ bản của tối ưu hóa dựa trên gradient, hãy tham khảo lại Chương 4 - giới thiệu tóm tắt về các phương pháp tối ưu số học nói chung.

Mục tiêu cụ thể trong chương này là giải bài toán tối ưu hóa hàm chi phí:

$$\min_{\theta} J(\theta)$$

trong đó θ là tập các tham số của mạng nơ-ron, và $J(\theta)$ là hàm mất mát — thường được tính từ lỗi trên toàn bộ tập huấn luyện và có thể bao gồm thêm các thành phần regularization để kiểm soát độ phức tạp của mô hình.

Nội dung chương này được tổ chức như sau:

- Trước tiên, chúng ta sẽ làm rõ sự khác biệt giữa tối ưu hóa trong học máy và tối ưu hóa thuần túy trong toán học hoặc kỹ thuật.
- Tiếp theo, ta trình bày một loạt các thách thức đặc thù trong việc huấn luyện mạng nơ-ron, bao gồm:
  - độ cong không đều,
  - điểm yên ngựa,
  - cực tiểu cục bộ,
  - sự bất ổn trong gradient.
- Sau đó, chúng ta sẽ thảo luận các thuật toán tối ưu hóa cơ bản và nâng cao, bao gồm:
  - Gradient Descent và các biến thể như Momentum, Nesterov Accelerated Gradient,
  - Các phương pháp thích nghi như Adagrad, RMSprop và Adam,

  - Kỹ thuật khởi tạo tham số hiệu quả.

- Cuối cùng, chương này sẽ giới thiệu các chiến lược tối ưu hóa tổng hợp, tức là kết hợp nhiều thuật toán đơn lẻ thành quy trình tối ưu hóa cấp cao, để tận dụng ưu điểm của từng phương pháp.

Với trọng tâm là huấn luyện hiệu quả các mô hình học sâu, chương này không chỉ trang bị kiến thức thuật toán mà còn truyền tải tư duy chiến lược trong tối ưu hóa thực tiễn.

8.1 Tối ưu hóa trong học máy khác với tối ưu hóa thuần túy

Tối ưu hóa trong học máy, đặc biệt là trong học sâu, có nhiều điểm khác biệt so với tối ưu hóa thuần túy trong toán học hoặc kỹ thuật. Trong học sâu, mục tiêu cuối cùng thường không phải là trực tiếp tối thiểu hóa một hàm chi phí cụ thể. Thay vào đó, chúng ta quan tâm đến một chỉ số hiệu suất P, chẳng hạn như độ chính xác phân loại hoặc lỗi dự đoán trên tập kiểm tra. Tuy nhiên, chỉ số P này thường không thể được tối ưu hóa trực tiếp vì nó không khả vi hoặc không định nghĩa rõ ràng trên toàn bộ không gian tham số. Do đó, chúng ta lựa chọn một hàm chi phí khả vi $J(\theta)$ làm đại diện và tiến hành tối thiểu hóa J với hy vọng rằng điều này sẽ gián tiếp cải thiện P.

Điều này trái ngược với các bài toán tối ưu hóa thuần túy, nơi mà việc tối thiểu hóa một hàm chi phí cụ thể J chính là mục tiêu trung tâm của toàn bộ bài toán.

Một đặc điểm khác biệt quan trọng là trong học máy, đặc biệt với các mô hình có giám sát, hàm chi phí $J(\theta)$ thường có thể được biểu diễn dưới dạng trung bình trên tập dữ liệu huấn luyện. Cụ thể, hàm chi phí thường được định nghĩa là:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}}\left[L(f(x;\theta), y)\right] \tag{8.1}$$

trong đó:

- $L(f(x;\theta), y)$ là hàm mất mát đo sai lệch giữa đầu ra mô hình $f(x;\theta)$ và nhãn đúng y,
- $\hat{p}_{\text{data}}$ là phân phối thực nghiệm (empirical distribution) trên tập dữ liệu huấn luyện.

Trong trường hợp lý tưởng, chúng ta mong muốn mô hình hoạt động tốt trên phân phối dữ liệu thực sự p_{data}, không chỉ trên tập huấn luyện hữu hạn. Vì vậy, mục tiêu thực sự cần tối ưu hóa là:

$$J^*(\theta) = \mathbb{E}_{(x,y)\sim p_{\text{data}}}\left[L(f(x;\theta), y)\right] \tag{8.2}$$

Tuy nhiên, vì p_{data} là không biết rõ, chúng ta buộc phải sử dụng $\hat{p}_{\text{data}}$ như một ước lượng gần đúng và tối ưu hóa $J(\theta)$ thay thế. Đây là lý do tại sao kỹ thuật regularization và các chiến lược tổng quát hóa khác đóng vai trò rất quan trọng trong học máy — chúng giúp khoảng cách giữa J và J^* nhỏ hơn, tức là làm cho việc tối ưu hóa trên dữ liệu huấn luyện

vẫn mang lại hiệu suất tốt trên dữ liệu chưa thấy.

Những khác biệt trên dẫn đến việc phát triển các thuật toán tối ưu hóa riêng biệt cho học sâu — không chỉ tối ưu hoá tốt trên tập huấn luyện, mà còn đảm bảo khả năng tổng quát hóa sang dữ liệu mới.

8.1.1 Giảm thiểu rủi ro thực nghiệm

Mục tiêu chính của một thuật toán học máy là giảm thiểu sai số khái quát kỳ vọng, hay còn gọi là rủi ro (risk), được định nghĩa như trong phương trình (**??**). Rủi ro này là kỳ vọng của hàm mất mát theo phân phối dữ liệu thực sự p_{data}. Nếu ta biết rõ phân phối $p_{\text{data}}(x, y)$, thì việc giảm thiểu rủi ro này trở thành một bài toán tối ưu hóa khả thi và có thể được giải một cách trực tiếp.

Tuy nhiên, trong thực tế, chúng ta không có quyền truy cập vào phân phối thực sự đó mà chỉ có một tập dữ liệu huấn luyện hữu hạn. Do đó, ta phải tìm cách chuyển bài toán học máy thành một bài toán tối ưu hóa gần đúng. Cách đơn giản và phổ biến nhất là thay thế kỳ vọng theo p_{data} bằng kỳ vọng theo phân phối thực nghiệm $\hat{p}_{\text{data}}$, được xác định từ tập dữ liệu huấn luyện. Khi đó, ta tối thiểu hóa hàm mất mát trung bình trên tập huấn luyện:

$$\mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}}\left[L(f(x;\theta),y)\right] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)}), \tag{8.3}$$

trong đó m là số lượng mẫu trong tập huấn luyện.

Phương pháp này được gọi là giảm thiểu rủi ro thực nghiệm (Empirical Risk Minimization — ERM). Trong khuôn khổ này, quá trình học máy trở nên tương tự như một bài toán tối ưu hóa truyền thống: tìm tham số θ sao cho hàm chi phí trung bình trên tập huấn luyện là nhỏ nhất.

Tuy nhiên, việc tối ưu hóa rủi ro thực nghiệm không phải lúc nào cũng mang lại hiệu suất tốt trên dữ liệu chưa thấy. Nguyên nhân là vì ERM dễ dẫn đến hiện tượng overfitting — mô hình có thể khớp hoàn toàn với tập huấn luyện mà không học được các quy luật tổng quát của dữ liệu. Điều này đặc biệt dễ xảy ra khi mô hình có năng lực biểu diễn cao, tức là có nhiều tham số.

Bên cạnh đó, một số hàm mất mát được định nghĩa trên hiệu suất thực sự lại không thân thiện với tối ưu hóa. Ví dụ, *0-1 loss* là một hàm phổ biến trong phân loại, nhưng không khả vi — đạo hàm của nó bằng 0 gần như mọi nơi hoặc không xác định, khiến gradient descent không thể hoạt động hiệu quả.

Vì lý do đó, trong học sâu, chúng ta hiếm khi tối ưu rủi ro thực nghiệm một cách trực tiếp. Thay vào đó, ta thường chọn một hàm mất mát trơn tru hơn (như hàm log loss, cross-entropy) để tối ưu. Những hàm mất mát này có đạo hàm hữu dụng, giúp thuật toán gradient descent hoặc các biến thể của nó hoạt động tốt, dù đôi khi chúng không trực tiếp phản ánh tiêu chí cuối cùng mà ta thực sự muốn tối ưu.

Việc lựa chọn hàm mất mát thay thế này là một phần quan trọng trong thiết kế mô hình học máy, đóng vai trò cầu nối giữa lý thuyết và thực tiễn.

8.1.2 Hàm mất mát thay thế và dừng sớm (Surrogate loss functions and early stopping)

Đôi khi, hàm mất mát mà ta thực sự quan tâm (chẳng hạn như hàm mất mát 0-1 dùng trong phân loại) lại không thể tối ưu hiệu quả. Cụ thể, việc tối thiểu hóa kỳ vọng của hàm mất mát 0-1 là một bài toán khó về mặt tính toán, ngay cả khi áp dụng với các mô hình đơn giản như mô hình tuyến tính **marcotte1992**.

Trong những trường hợp như vậy, ta thường sử dụng một hàm mất mát thay thế (*surrogate loss function*) — tức một hàm mất mát khác có đặc tính gần giống với hàm mất mát ban đầu nhưng dễ tối ưu hơn. Một ví dụ phổ biến là hàm log-likelihood âm của nhãn đúng, được dùng thay cho hàm 0-1 loss. Hàm log-likelihood cho phép mô hình học xác suất điều kiện của các lớp, và nếu mô hình học tốt xác suất này, nó sẽ đưa ra quyết định phân lớp tối ưu về mặt kỳ vọng.

Trong một số trường hợp, việc sử dụng hàm mất mát thay thế còn giúp mô hình học được nhiều hơn. Ví dụ, khi sử dụng log-likelihood để huấn luyện, ta thường thấy rằng sai số 0-1 trên tập kiểm tra vẫn tiếp tục giảm, ngay cả khi sai số 0-1 trên tập huấn luyện đã đạt 0. Nguyên nhân là do mô hình vẫn có thể cải thiện độ tự tin của mình, bằng cách làm tăng khoảng cách giữa các lớp dự đoán — từ đó cải thiện khả năng tổng quát hóa.

Một điểm rất quan trọng cần phân biệt giữa tối ưu hóa trong học máy và tối ưu hóa thuần túy là: trong huấn luyện mô hình, thuật toán thường không chạy đến khi đạt cực tiểu địa phương. Thay vào đó, nó tối ưu một *hàm mất mát thay thế*, nhưng sẽ dừng lại sớm dựa trên một tiêu chí hội tụ được xác định trước — kỹ thuật này gọi là dừng sớm (xem mục **??**).

Thông thường, tiêu chí dừng sớm được xây dựng dựa trên *hàm mất mát thực* như 0-1 loss đo trên tập kiểm tra, nhằm dừng huấn luyện khi mô hình bắt đầu có dấu hiệu overfitting. Điều này dẫn đến việc huấn luyện có thể dừng lại ngay cả khi gradient của hàm mất mát thay thế vẫn còn lớn — một điều rất khác so với tối ưu hóa thuần túy, nơi quá trình tối ưu chỉ kết thúc khi gradient gần như bằng 0.

8.1.3 Giải thuật batch và minibatch

Một điểm khác biệt quan trọng giữa tối ưu hóa trong học máy và tối ưu hóa truyền thống là: trong học máy, hàm mục tiêu thường được tính như tổng các giá trị mất mát trên từng mẫu dữ liệu. Điều này cho phép ta chia nhỏ việc tính toán và chỉ cần dùng một phần nhỏ của dữ liệu để cập nhật mô hình.

Chẳng hạn, trong bài toán ước lượng hợp lý cực đại (maximum likelihood estimation), nếu chuyển sang log space, ta có bài toán tối ưu như sau:

$$\theta_{\text{ML}} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{\text{model}}(x^{(i)}, y^{(i)}; \theta) \tag{8.4}$$

Tối đa hóa tổng trên tương đương với tối đa hóa kỳ vọng theo phân phối thực nghiệm $\hat{p}_{data}$:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{\text{model}}(x, y; \theta) \tag{8.5}$$

Trong thực tế, ta không cần tính chính xác toàn bộ giá trị kỳ vọng này. Thay vào đó, ta thường chỉ cần tính gần đúng gradient của hàm mục tiêu để cập nhật mô hình. Gradient này có thể được viết dưới dạng kỳ vọng:

$$\nabla_\theta J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data}} \nabla_\theta \log p_{\text{model}}(x, y; \theta) \tag{8.6}$$

Tính chính xác kỳ vọng này đòi hỏi phải dùng toàn bộ tập huấn luyện, điều này rất tốn thời gian. Để tiết kiệm, ta có thể chọn ngẫu nhiên một nhóm nhỏ các ví dụ (gọi là minibatch) rồi tính trung bình gradient trên nhóm đó.

Cần lưu ý rằng sai số chuẩn của trung bình (xem phương trình 5.46) với n mẫu là $\sigma/\sqrt{n}$, trong đó σ là độ lệch chuẩn thật sự. Nghĩa là, càng dùng nhiều mẫu thì kết quả càng chính xác, nhưng độ chính xác tăng chậm dần theo quy luật $1/\sqrt{n}$. Vì vậy, nếu dùng minibatch lớn thì gradient gần đúng sẽ chính xác hơn, nhưng tính toán sẽ tốn kém hơn.

Ví dụ, dùng minibatch gồm 100 ví dụ thì gradient có thể kém chính xác hơn so với dùng 10.000 ví dụ, nhưng lại tính nhanh hơn gấp 100 lần. Điều này cho phép thực hiện được nhiều bước cập nhật hơn trong cùng khoảng thời gian, và có thể cải thiện kết quả tổng thể.

Một lý do nữa để dùng minibatch là vì trong nhiều trường hợp, dữ liệu huấn luyện có sự trùng lặp hoặc dư thừa. Giả sử trong trường hợp xấu nhất, tất cả các mẫu đều giống nhau — khi đó, chỉ cần một mẫu duy nhất là đủ để tính gradient chính xác, giúp tiết kiệm nhiều lần tính toán so với việc dùng toàn bộ tập dữ liệu.

Tuy trong thực tế hiếm khi các mẫu giống hệt nhau, nhưng thường có nhiều mẫu đóng góp rất giống nhau vào quá trình cập nhật. Vì vậy, việc lấy mẫu ngẫu nhiên để ước lượng gradient là cách làm hiệu quả và hợp lý.

Nếu một thuật toán sử dụng toàn bộ tập dữ liệu huấn luyện để cập nhật tham số trong mỗi bước, ta gọi đó là thuật toán batch hay còn gọi là thuật toán gradient xác định (deterministic gradient methods), vì nó xử lý toàn bộ dữ liệu như một khối duy nhất trong mỗi lần cập nhật.

Lưu ý: Thuật ngữ "batch" đôi khi dễ gây hiểu nhầm vì cũng được dùng để chỉ "minibatch" trong phương pháp *minibatch stochastic gradient descent*. Thông thường, "batch gradient

descent" ám chỉ việc dùng toàn bộ tập dữ liệu trong mỗi bước cập nhật, còn "batch size" thì chỉ kích thước cụ thể của một minibatch.

Các thuật toán chỉ sử dụng một mẫu dữ liệu tại mỗi bước cập nhật được gọi là stochastic hoặc online methods. Trong đó, "online" thường mô tả trường hợp dữ liệu được cung cấp liên tục theo thời gian thực (streaming), thay vì từ một tập dữ liệu tĩnh mà ta có thể duyệt đi duyệt lại nhiều lần.

Trong deep learning hiện đại, hầu hết các phương pháp tối ưu đều dùng một nhóm nhỏ dữ liệu tại mỗi bước cập nhật — tức là không phải chỉ một mẫu duy nhất, nhưng cũng không phải toàn bộ tập. Nhóm nhỏ này được gọi là minibatch. Trước đây, phương pháp này từng được gọi là *minibatch stochastic methods*, nhưng hiện nay thường gọi tắt là *stochastic methods*.

Ví dụ phổ biến nhất của phương pháp này là Stochastic Gradient Descent (SGD), sẽ được trình bày chi tiết ở mục **??**.

Việc lựa chọn kích thước minibatch (batch size) thường phụ thuộc vào một số yếu tố sau:

- Độ chính xác của gradient: Minibatch lớn giúp ước lượng gradient chính xác hơn, nhưng độ chính xác chỉ tăng chậm theo kích thước — cụ thể, tăng theo tỉ lệ $1/\sqrt{n}$, trong đó n là số mẫu.
- Tận dụng phần cứng: Với các thiết bị có khả năng xử lý song song như GPU hoặc CPU đa nhân, batch quá nhỏ sẽ không sử dụng hết hiệu năng phần cứng. Do đó, cần một batch đủ lớn để đạt hiệu quả tính toán.
- Giới hạn bộ nhớ: Dùng batch lớn yêu cầu nhiều bộ nhớ hơn. Trong nhiều trường hợp, bộ nhớ (đặc biệt là GPU) là giới hạn chính khiến ta không thể chọn batch lớn hơn.
- Tối ưu hiệu năng phần cứng: Một số phần cứng hoạt động hiệu quả hơn với batch có kích thước là lũy thừa của 2 (ví dụ: 32, 64, 128, 256). Batch size nhỏ như 16 đôi khi được dùng cho các mô hình rất lớn.
- Tác dụng regularization: Batch nhỏ giúp thêm nhiễu vào quá trình học, tạo hiệu ứng regularization tốt hơn (Wilson và Martinez, 2003). Trong một số trường hợp, batch size = 1 có thể mang lại kết quả tổng quát hóa tốt nhất, nhưng đòi hỏi phải giảm learning rate để giữ cho quá trình học ổn định. Điều này cũng đồng nghĩa với thời gian huấn luyện lâu hơn.

Mỗi thuật toán tối ưu có cách xử lý minibatch khác nhau. Một số thuật toán rất nhạy cảm với nhiễu từ việc lấy mẫu minibatch, đặc biệt là những thuật toán phức tạp hoặc dùng thêm thông tin bậc hai như đạo hàm cấp hai.

Những phương pháp đơn giản chỉ dùng gradient g để cập nhật tham số thường khá ổn định và có thể hoạt động tốt với batch nhỏ (như 100 mẫu). Ngược lại, các phương pháp bậc hai như sử dụng ma trận Hessian H và cập nhật theo công thức $H^{-1}g$ thường yêu cầu

batch rất lớn (có thể lên đến 10.000 mẫu) để đảm bảo việc ước lượng $H^{-1}g$ không quá nhiễu.

Lưu ý: Ngay cả khi ma trận Hessian H được tính chính xác, nếu nó có điều kiện kém (nghĩa là tỉ số giữa giá trị riêng lớn nhất và nhỏ nhất rất cao), thì phép nhân với H^{-1} cũng có thể khuếch đại lỗi có trong gradient g. Một thay đổi nhỏ trong g cũng có thể gây ra thay đổi lớn trong $H^{-1}g$. Trong thực tế, cả H và g đều chỉ được ước lượng xấp xỉ từ dữ liệu, nên sai số càng dễ bị khuếch đại hơn nữa.

Việc chọn các minibatch một cách ngẫu nhiên là rất quan trọng. Để ước lượng gradient một cách *không chệch* (unbiased), các mẫu trong minibatch cần được lấy độc lập từ phân phối dữ liệu. Ngoài ra, ta cũng muốn các minibatch liên tiếp không phụ thuộc nhau — điều này giúp đảm bảo rằng các bước cập nhật của mô hình không bị thiên lệch do sự trùng lặp dữ liệu giữa các bước.

Trong thực tế, nhiều tập dữ liệu không được tổ chức ngẫu nhiên. Ví dụ, trong dữ liệu y tế, một bệnh nhân có thể có nhiều mẫu xét nghiệm được thu thập theo thời gian. Nếu ta chọn minibatch một cách tuần tự từ dữ liệu gốc (chưa xáo trộn), có khả năng cao một minibatch chỉ chứa dữ liệu từ một vài bệnh nhân cụ thể. Điều này dẫn đến độ lệch lớn trong ước lượng gradient và khiến mô hình khó tổng quát hóa tốt. Vì vậy, cần phải xáo trộn (shuffle) dữ liệu trước khi tạo minibatch.

Lưu ý với dữ liệu rất lớn: Khi tập dữ liệu có hàng tỷ mẫu (như trong trung tâm dữ liệu), việc xáo trộn toàn bộ cho mỗi epoch là không khả thi. Trong trường hợp này, thường ta chỉ cần xáo trộn một lần, sau đó lưu lại thứ tự xáo trộn và sử dụng lại cho các epoch sau. Cách làm này tạo ra một tập hợp minibatch cố định, được dùng lặp lại trong quá trình huấn luyện. Mặc dù không hoàn toàn ngẫu nhiên như lấy mẫu trực tiếp từ phân phối dữ liệu, nhưng thực nghiệm cho thấy cách làm này vẫn rất hiệu quả.

Nhiều bài toán tối ưu trong học máy có cấu trúc cho phép phân tách theo từng mẫu, nên ta có thể cập nhật song song trên nhiều minibatch khác nhau. Điều này mở ra khả năng huấn luyện phân tán theo cách *không đồng bộ* (asynchronous), được trình bày chi tiết ở mục **??**.

Một lý do quan trọng khiến Minibatch Stochastic Gradient Descent (SGD) được ưa chuộng là vì nó có thể ước lượng rất sát gradient của *lỗi tổng quát thực sự* (true generalization error), được định nghĩa ở phương trình (8.2), miễn là các mẫu trong minibatch là độc lập và không trùng lặp.

Trong thực tế, hầu hết các hệ thống SGD hiện nay thường chỉ *xáo trộn dữ liệu một lần*, sau đó lặp lại thứ tự xáo trộn này trong nhiều epoch. Ở epoch đầu tiên, mỗi minibatch cung cấp một ước lượng không chệch của gradient lỗi tổng quát. Tuy nhiên, từ các epoch sau, việc lặp lại các mẫu khiến ước lượng trở nên chệch (biased), vì ta không còn rút mẫu ngẫu nhiên từ phân phối dữ liệu thực nữa.

Trong bối cảnh học online — nơi dữ liệu đến liên tục như một dòng thời gian, chứ

không phải từ một tập dữ liệu cố định — mỗi cặp (x, y) là một ví dụ mới, chưa từng thấy trước đó, được lấy trực tiếp từ phân phối $p_{\text{data}}(x, y)$. Khi đó, mô hình giống như đang học từ trải nghiệm thật, và mỗi cập nhật đều phản ánh trung thực lỗi tổng quát tại thời điểm đó.

Khi cả x và y là biến rời rạc, ta có thể viết lại lỗi tổng quát như sau:

$$J^*(\boldsymbol{\theta}) = \sum_x \sum_y p_{\text{data}}(x, y)\, L(f(x; \boldsymbol{\theta}), y), \tag{8.4}$$

với gradient chính xác tương ứng là:

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_x \sum_y p_{\text{data}}(x, y)\, \nabla_{\boldsymbol{\theta}} L(f(x; \boldsymbol{\theta}), y). \tag{8.5}$$

Biểu thức trên có cấu trúc tương tự như đạo hàm log-likelihood ở phương trình (8.6), nhưng áp dụng được cho nhiều loại hàm mất mát khác nhau, không chỉ riêng log-likelihood, miễn là chúng thỏa các điều kiện nhất định về tính liên tục và đạo hàm.

Ý tưởng then chốt: Chúng ta có thể ước lượng một cách *không chệch* gradient của lỗi tổng quát bằng cách:

- Lấy một minibatch $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$ từ phân phối p_{data},
- Tính gradient trung bình trên minibatch đó:

$$\hat{\mathbf{g}} = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(x^{(i)}; \boldsymbol{\theta}), y^{(i)}).$$

$$\hat{\mathbf{g}} = \frac{1}{m} \sum_{i} \nabla_{\boldsymbol{\theta}} L(f(x^{(i)}; \boldsymbol{\theta}), y^{(i)}). \tag{8.6}$$

Cập nhật tham số $\boldsymbol{\theta}$ theo hướng của $\hat{\mathbf{g}}$ chính là cách mà Stochastic Gradient Descent (SGD) hoạt động để tối ưu lỗi tổng quát.

Cách hiểu này chỉ thực sự đúng khi mỗi ví dụ trong tập huấn luyện được dùng duy nhất một lần, tức là không bị lặp lại. Tuy nhiên, trong thực tế, chúng ta thường huấn luyện qua tập dữ liệu nhiều vòng (nhiều *epoch*) — đặc biệt là khi tập dữ liệu không quá lớn.

Trong kịch bản huấn luyện nhiều epoch, chỉ epoch đầu tiên là nơi minibatch được lấy giống như từ phân phối dữ liệu thực, nên ước lượng gradient tại đây là *không chệch*. Từ epoch thứ hai trở đi, các mẫu bị dùng lại, nên gradient trở thành một *ước lượng chệch* so với gradient thực sự của lỗi tổng quát.

Tuy nhiên, dù có độ lệch, các epoch sau thường vẫn giúp giảm đáng kể lỗi huấn luyện. Và phần lớn thời gian, sự cải thiện trong hiệu suất của mô hình vẫn đủ lớn để bù lại cho sự

lệch trong ước lượng gradient. Nhờ đó, mô hình vẫn học hiệu quả và tổng quát tốt trong thực tế.

Nhưng cần lưu ý: Nếu lặp lại quá nhiều lần trên cùng tập dữ liệu (quá nhiều epoch), mô hình có thể bị *overfitting* — tức là học quá kỹ những đặc điểm riêng của tập huấn luyện và mất khả năng tổng quát hóa với dữ liệu mới. Khi đó, khoảng cách giữa lỗi huấn luyện và lỗi kiểm tra (test error) sẽ tăng lên, phản ánh chất lượng mô hình giảm sút.

Trong một số ứng dụng, dữ liệu đến liên tục và không thể lưu trữ toàn bộ (ví dụ: hệ thống đo thời gian thực, dữ liệu cảm biến, luồng dữ liệu online). Khi đó, chúng ta không thể lặp lại dữ liệu, mà phải xử lý từng mẫu một lần duy nhất. Đây là lúc mà học máy online (online learning) trở nên cần thiết: mỗi ví dụ chỉ được xử lý một lần khi nó xuất hiện, và mô hình liên tục được cập nhật.

Khi dữ liệu đủ lớn, overfitting không còn là vấn đề chính — vì mô hình khó có thể ghi nhớ hết được mọi mẫu. Thay vào đó, mối quan tâm sẽ chuyển sang hai vấn đề quan trọng khác:

- Underfitting: Mô hình chưa học đủ tốt từ dữ liệu, dẫn đến hiệu suất chưa cao.
- Hiệu quả tính toán: Cần đảm bảo rằng mô hình có thể được huấn luyện nhanh chóng và hiệu quả về mặt tài nguyên (thời gian, bộ nhớ, năng lượng).

Tham khảo thêm: **bottou2008tradeoffs** cung cấp một phân tích sâu sắc về mối quan hệ giữa kích thước tập dữ liệu, độ chính xác tính toán, và ảnh hưởng của chúng đến lỗi tổng quát. Tác giả nhấn mạnh rằng trong kỷ nguyên dữ liệu lớn, tắc nghẽn tính toán (computational bottlenecks) mới là yếu tố hạn chế chính, chứ không còn là kích thước tập huấn luyện nhỏ như trước.

8.2 Các thách thức trong việc tối ưu mạng nơ-ron nhân tạo

Tối ưu hóa, nói chung, là một nhiệm vụ rất khó. Trong học máy truyền thống, người ta thường cố gắng tránh những khó khăn của tối ưu hóa tổng quát bằng cách thiết kế hàm mục tiêu và ràng buộc sao cho bài toán trở thành lồi (convex). Với một bài toán lồi, ta có thể đảm bảo rằng bất kỳ điểm cực tiểu cục bộ nào cũng là cực tiểu toàn cục — điều này làm cho bài toán dễ giải hơn rất nhiều.

Tuy nhiên, trong huấn luyện mạng nơ-ron, đặc biệt là các mạng sâu, chúng ta không thể áp dụng cách tiếp cận này. Mạng nơ-ron thường dẫn đến các bài toán tối ưu phi lồi (non-convex), có thể có rất nhiều cực tiểu cục bộ, yên ngựa (saddle points), và các vùng mặt phẳng (flat regions) khiến việc huấn luyện trở nên phức tạp hơn nhiều.

Thậm chí, ngay cả khi bài toán là lồi, việc tìm ra nghiệm tối ưu cũng không hề đơn giản. Các kỹ thuật tối ưu thực tế phải đối mặt với những giới hạn về thời gian, bộ nhớ, và độ chính xác của tính toán. Trong phần này, chúng ta sẽ cùng điểm qua một số thách thức lớn nhất trong việc tối ưu hóa khi huấn luyện các mô hình học sâu.

8.2.1 Tình trạng điều kiện kém (Ill-conditioning)

Một trong những thách thức lớn trong tối ưu, ngay cả với các hàm mục tiêu lồi, là vấn đề tình trạng điều kiện kém của ma trận Hessian $\mathbf{H}$. Đây là một vấn đề phổ biến trong tối ưu số nói chung và đã được đề cập kỹ hơn ở mục 4.3.1.

Trong bối cảnh huấn luyện mạng nơ-ron, tình trạng điều kiện kém xảy ra khá thường xuyên. Khi đó, các thuật toán như SGD có thể bị "kẹt" — nghĩa là, ngay cả khi gradient lớn, chỉ cần bước cập nhật hơi dài cũng có thể làm tăng hàm chi phí thay vì giảm.

Nhớ lại khai triển Taylor bậc hai từ phương trình (4.9): nếu ta thực hiện một bước gradient descent với bước học ε theo hướng $-\mathbf{g}$, thì mức thay đổi xấp xỉ của hàm chi phí là:

$$\Delta J \approx \frac{1}{2}\varepsilon^2 \mathbf{g}^\top \mathbf{H}\mathbf{g} - \varepsilon \mathbf{g}^\top \mathbf{g} \tag{8.10}$$

Nếu điều kiện sau xảy ra:

$$\frac{1}{2}\varepsilon^2 \mathbf{g}^\top \mathbf{H}\mathbf{g} > \varepsilon \mathbf{g}^\top \mathbf{g},$$

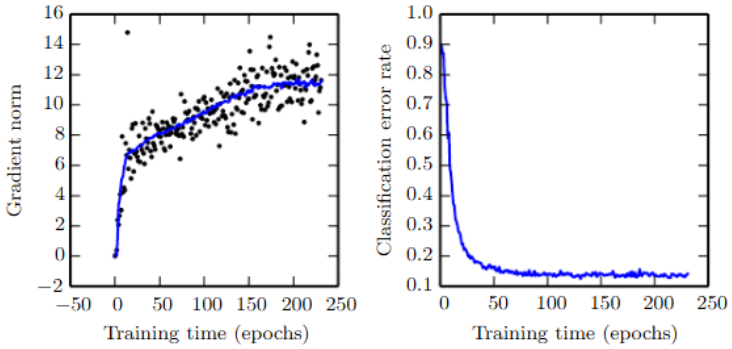
thì bước cập nhật đó sẽ khiến hàm chi phí tăng thay vì giảm.

Để theo dõi tình trạng điều kiện kém trong huấn luyện, ta có thể quan sát hai đại lượng sau:

- Bình phương chuẩn gradient: $\mathbf{g}^\top \mathbf{g}$,
- Độ cong dọc theo hướng gradient: $\mathbf{g}^\top \mathbf{H}\mathbf{g}$.

Trong thực nghiệm, người ta thấy rằng chuẩn gradient có thể không giảm nhiều trong quá trình huấn luyện, nhưng độ cong (curvature) lại tăng lên rất nhanh, khiến phải giảm bước học để tránh nhảy quá xa và làm sai mô hình. Điều này làm cho tốc độ huấn luyện chậm đi rõ rệt dù gradient vẫn lớn.

Hình 8.1 minh họa điều này: chuẩn gradient tăng theo thời gian, nhưng sai số trên tập kiểm định vẫn giảm — cho thấy rằng mô hình vẫn học tốt, dù gradient không giảm như ta thường kỳ vọng trong tối ưu hóa cổ điển.



Hình 8.1: Quá trình huấn luyện mạng nơ-ron tích chập cho bài toán phát hiện đối tượng. (Trái): Chuẩn gradient theo thời gian (mỗi epoch lấy một điểm), đường cong thể hiện trung bình trượt. Thay vì giảm, chuẩn gradient lại tăng. (Phải): Dù vậy, sai số trên tập kiểm định vẫn giảm — cho thấy huấn luyện vẫn hiệu quả dù gradient không nhỏ đi.

Vấn đề điều kiện kém không chỉ xảy ra trong mạng nơ-ron mà còn phổ biến ở nhiều loại bài toán tối ưu khác. Tuy nhiên, một số kỹ thuật xử lý hiệu quả cho các bài toán khác lại không áp dụng được cho mạng nơ-ron. Ví dụ điển hình là phương pháp Newton. Phương pháp này dùng thông tin về ma trận Hessian để cập nhật theo hướng $H^{-1}\mathbf{g}$ và thường rất hiệu quả cho các bài toán lồi. Nhưng với mạng nơ-ron, Hessian thường rất lớn và phi lồi, nên áp dụng trực tiếp phương pháp Newton là không khả thi — trừ khi có sự điều chỉnh đặc biệt, sẽ được trình bày ở các phần sau.

8.2.2 Cực Tiểu Cục Bộ

Một trong những đặc điểm nổi bật của bài toán tối ưu lồi là có thể giảm bài toán về việc tìm một cực tiểu cục bộ. Bất kỳ cực tiểu cục bộ nào cũng đều đảm bảo là cực tiểu toàn cục. Một số hàm lồi có thể có vùng phẳng ở đáy thay vì chỉ có một điểm cực tiểu toàn cục duy nhất, nhưng bất kỳ điểm nào trong vùng phẳng đó cũng là một giải pháp chấp nhận được.

Tuy nhiên, đối với các hàm không lồi, như mạng nơ-ron, có thể xuất hiện nhiều cực tiểu cục bộ. Thực tế, gần như bất kỳ mô hình sâu nào cũng đều có số lượng cực tiểu cục bộ cực kỳ lớn. Tuy nhiên, như chúng ta sẽ thấy, điều này không nhất thiết là một vấn đề lớn.

Mạng nơ-ron và các mô hình có nhiều biến ẩn đều có nhiều cực tiểu cục bộ do vấn đề nhận dạng mô hình. Một mô hình được coi là nhận dạng được nếu một bộ dữ liệu huấn luyện đủ lớn có thể loại trừ tất cả các giá trị tham số của mô hình ngoại trừ một giá trị duy nhất. Các mô hình có biến ẩn thường không nhận dạng được vì ta có thể tạo ra các mô hình tương đương bằng cách hoán đổi các biến ẩn cho nhau.

Ngoài sự đối xứng trong không gian trọng số, nhiều loại mạng nơ-ron còn có các nguyên nhân khác gây ra vấn đề không nhận dạng được. Ví dụ, trong bất kỳ mạng nào sử dụng hàm kích hoạt tuyến tính như ReLU hoặc maxout, ta có thể thay đổi tỷ lệ tất cả các trọng số và độ chệch đầu vào của một đơn vị theo tỷ lệ α nếu ta cũng thay đổi tất cả các

trọng số đầu ra theo tỷ lệ $\frac{1}{\alpha}$.

Các cực tiểu cục bộ có thể trở thành vấn đề nếu chúng có chi phí cao hơn so với cực tiểu toàn cục. Người ta có thể xây dựng các mạng nơ-ron nhỏ, thậm chí không có đơn vị ẩn, mà có các cực tiểu cục bộ có chi phí cao hơn cực tiểu toàn cục.

Tuy nhiên, liệu các mạng nơ-ron có nhiều cực tiểu cục bộ có chi phí cao hay không, và liệu các thuật toán tối ưu có gặp phải chúng hay không vẫn còn là câu hỏi mở. Trong nhiều năm qua, hầu hết các nhà thực hành tin rằng cực tiểu cục bộ là một vấn đề phổ biến gây khó khăn trong tối ưu hóa mạng nơ-ron. Ngày nay, điều này không còn đúng nữa.

8.2.3 Mặt Phẳng, Điểm Yên Ngựa và Các Vùng Phẳng Khác

Đối với nhiều hàm phi lồi trong không gian có chiều cao, các điểm cực tiểu (và cực đại) thực tế hiếm gặp so với một loại điểm khác có đạo hàm bằng không: điểm yên ngựa. Một số điểm xung quanh điểm yên ngựa có giá trị hàm số lớn hơn điểm yên ngựa, trong khi một số điểm khác lại có giá trị thấp hơn. Tại một điểm yên ngựa, ma trận Hessian có cả giá trị trị riêng dương và âm. Các điểm nằm trên các vector riêng liên quan đến giá trị trị riêng dương có giá trị lớn hơn điểm yên ngựa, trong khi các điểm nằm trên các vector riêng liên quan đến giá trị trị riêng âm có giá trị thấp hơn. Chúng ta có thể tưởng tượng một điểm yên ngựa giống như một điểm cực tiểu trong một mặt cắt của hàm chi phí và là điểm cực đại trong một mặt cắt khác. Hình 4.5 minh họa cho điều này.

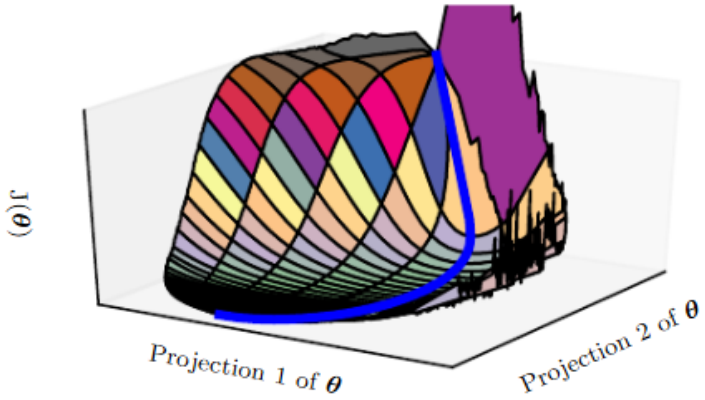
Nhiều loại hàm ngẫu nhiên có hành vi như sau: trong không gian có chiều thấp, các điểm cực tiểu rất phổ biến. Tuy nhiên, trong không gian có chiều cao, các điểm cực tiểu trở nên hiếm hoi và điểm yên ngựa trở nên phổ biến hơn. Đối với một hàm $f : \mathbb{R}^n \to \mathbb{R}$ có đặc điểm này, tỷ lệ kỳ vọng của số điểm yên ngựa so với số điểm cực tiểu tăng theo cấp số mũ với n. Để hiểu rõ hơn về hành vi này, hãy quan sát rằng ma trận Hessian tại một điểm cực tiểu chỉ có các giá trị trị riêng dương. Ngược lại, ma trận Hessian tại điểm yên ngựa có cả giá trị trị riêng dương và âm. Hãy tưởng tượng rằng dấu của mỗi giá trị trị riêng được tạo ra bằng cách tung một đồng xu. Trong không gian một chiều, việc đạt được điểm cực tiểu rất dễ dàng, chỉ cần tung đồng xu và nhận được mặt ngửa. Tuy nhiên, trong không gian n chiều, việc tất cả n lần tung đồng xu đều ra mặt ngửa là một điều cực kỳ không thể xảy ra. Tham khảo Dauphin et al. (2014) để biết thêm lý thuyết về vấn đề này.

Một đặc tính đáng kinh ngạc của nhiều hàm ngẫu nhiên là các giá trị trị riêng của ma trận Hessian trở nên có xu hướng dương khi chúng ta đến gần các vùng có giá trị chi phí thấp. Trong phép ẩn dụ về tung đồng xu, điều này có nghĩa là chúng ta có nhiều khả năng nhận được mặt ngửa n lần khi ở gần một điểm cực tiểu với chi phí thấp. Điều này cũng có nghĩa là các điểm cực tiểu có khả năng có giá trị chi phí thấp hơn là cao. Các điểm cực tiểu có chi phí cao ít gặp hơn và thường là điểm yên ngựa. Các điểm cực tiểu có chi phí cực kỳ cao thường là các điểm cực đại.

Điều này xảy ra đối với nhiều loại hàm ngẫu nhiên. Vậy điều này có xảy ra đối với mạng nơ-ron không? Baldi và Hornik (1989) đã chỉ ra lý thuyết rằng các autoencoder

nông (mạng feedforward được huấn luyện để sao chép đầu vào thành đầu ra, mô tả trong chương 14) không có phi tuyến tính có điểm cực tiểu toàn cục và điểm yên ngựa, nhưng không có điểm cực tiểu với chi phí cao hơn điểm cực tiểu toàn cục. Họ quan sát mà không có bằng chứng rằng kết quả này mở rộng đến các mạng sâu hơn mà không có phi tuyến tính. Đầu ra của những mạng này là một hàm tuyến tính của đầu vào, nhưng chúng vẫn hữu ích để nghiên cứu như là một mô hình cho các mạng nơ-ron phi tuyến tính vì hàm mất mát của chúng là một hàm phi lồi của các tham số. Các mạng này về cơ bản chỉ là nhiều ma trận được ghép lại với nhau. Saxe et al. (2013) đã cung cấp các giải pháp chính xác cho động lực học học tập trong những mạng này và chỉ ra rằng quá trình học trong những mô hình này mô phỏng nhiều đặc điểm chất lượng quan sát được trong quá trình huấn luyện các mô hình sâu với các hàm kích hoạt phi tuyến tính. Dauphin et al. (2014) đã chỉ ra thực nghiệm rằng các mạng nơ-ron thực tế cũng có hàm mất mát chứa rất nhiều điểm yên ngựa có chi phí cao. Choromanska et al. (2014) cũng cung cấp thêm lý luận lý thuyết, cho thấy một lớp các hàm ngẫu nhiên có chiều cao liên quan đến mạng nơ-ron cũng có đặc tính này.

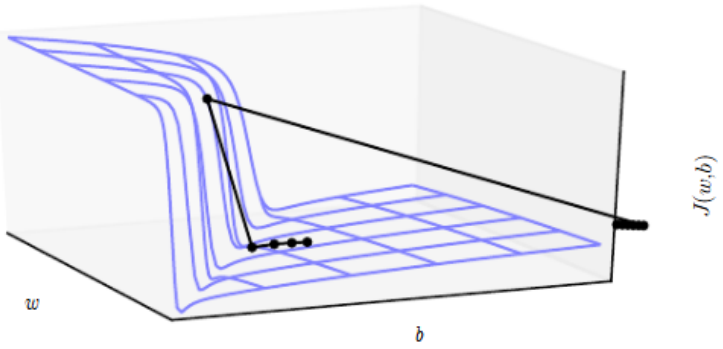
Vậy sự tồn tại của điểm yên ngựa có ý nghĩa gì đối với các thuật toán huấn luyện? Đối với các thuật toán tối ưu hóa bậc nhất, các thuật toán chỉ sử dụng thông tin gradient, tình huống này vẫn chưa rõ ràng. Gradient thường có thể trở nên rất nhỏ gần một điểm yên ngựa. Mặt khác, thực nghiệm cho thấy gradient descent có thể thoát ra khỏi điểm yên ngựa trong nhiều trường hợp. Goodfellow et al. (2015) đã cung cấp các hình ảnh trực quan của một số quỹ đạo học của các mạng nơ-ron hiện đại, với một ví dụ được trình bày trong hình 8.2. Những hình ảnh này cho thấy sự làm phẳng của hàm chi phí gần một điểm yên ngựa nổi bật, nơi tất cả các trọng số đều bằng không, nhưng chúng cũng cho thấy quỹ đạo gradient descent nhanh chóng thoát ra khỏi khu vực này. Goodfellow et al. (2015) cũng lập luận rằng gradient descent trong thời gian liên tục có thể được chứng minh lý thuyết là bị đẩy ra khỏi điểm yên ngựa gần đó, nhưng tình huống có thể khác đối với các ứng dụng thực tế của gradient descent.



Hình 8.2: Minh họa về hàm chi phí của một mạng nơ-ron. Những minh họa này xuất hiện tương tự cho các mạng nơ-ron feedforward, mạng nơ-ron tích chập, và mạng nơ-ron hồi tiếp khi áp dụng cho các tác vụ nhận diện đối tượng thực tế và xử lý ngôn ngữ tự nhiên. Đáng ngạc nhiên, những minh họa này thường không hiển thị nhiều chướng ngại vật rõ ràng. Trước khi phương pháp gradient descent ngẫu nhiên (SGD) thành công trong việc huấn luyện các mô hình rất lớn bắt đầu từ khoảng năm 2012, các bề mặt hàm chi phí của mạng nơ-ron thường được cho là có cấu trúc phi lồi phức tạp hơn so với những gì được tiết lộ qua những hình chiếu này. Chướng ngại vật chính được thể hiện trong hình chiếu này là một điểm yên ngựa có chi phí cao gần nơi các tham số được khởi tạo, nhưng như được chỉ ra bởi con đường màu xanh, quỹ đạo huấn luyện bằng SGD có thể dễ dàng thoát khỏi điểm yên ngựa này. Phần lớn thời gian huấn luyện được dành cho việc di chuyển qua thung lũng khá phẳng của hàm chi phí, có thể là do nhiễu cao trong gradient, điều kiện kém của ma trận Hessian trong khu vực này, hoặc đơn giản là cần phải đi vòng quanh "ngọn núi" cao thấy được trong hình qua một con đường vòng cung gián tiếp. Hình ảnh đã được sửa đổi với sự cho phép từ Goodfellow et al. (2015).

Đối với phương pháp Newton, các điểm yên ngựa rõ ràng là một vấn đề. Gradient descent được thiết kế để di chuyển "xuống dốc"và không được thiết kế để tìm kiếm một điểm cực trị. Tuy nhiên, phương pháp Newton lại được thiết kế để tìm một điểm mà tại đó gradient bằng không. Nếu không có sự điều chỉnh thích hợp, phương pháp Newton có thể nhảy tới một điểm yên ngựa. Sự phổ biến của các điểm yên ngựa trong không gian có chiều cao giải thích lý do tại sao các phương pháp bậc hai không thể thay thế gradient descent trong huấn luyện mạng nơ-ron.

Dauphin et al. (2014) đã giới thiệu một phương pháp Newton không có điểm yên ngựa, nhằm tối ưu hóa bậc hai và chỉ ra rằng phương pháp này cải thiện đáng kể so với phiên bản truyền thống. Tuy nhiên, các phương pháp bậc hai vẫn gặp khó khăn khi mở rộng cho các mạng nơ-ron lớn, nhưng phương pháp Newton không có điểm yên ngựa này vẫn có triển vọng nếu có thể mở rộng quy mô.



Hình 8.3: Hàm mục tiêu của các mạng nơ-ron sâu phi tuyến tính hoặc mạng nơ-ron hồi tiếp thường chứa các phi tuyến tính sắc nét trong không gian tham số, xuất phát từ việc nhân nhiều tham số với nhau. Những phi tuyến tính này gây ra các đạo hàm rất lớn ở một số vị trí. Khi các tham số gần đến những khu vực này, một bước cập nhật của phương pháp gradient descent có thể đẩy các tham số ra rất xa, có thể làm mất hầu hết công sức tối ưu hóa đã thực hiện. Hình ảnh được điều chỉnh với sự cho phép từ Pascanu et al. (2013).

Ngoài cực tiểu và điểm yên ngựa (saddle points), còn có những loại điểm khác có gradient bằng 0. Cực đại cũng tương tự như các điểm yên ngựa từ góc độ tối ưu hóa—nhiều thuật toán không bị thu hút vào chúng, nhưng phương pháp Newton không sửa đổi lại có thể dẫn đến cực đại. Cực đại của nhiều lớp hàm ngẫu nhiên trở nên hiếm một cách theo hàm mũ trong không gian có chiều cao, giống như cực tiểu.

Ngoài ra, cũng có thể tồn tại những vùng rộng và phẳng có giá trị hằng. Ở những vị trí này, gradient và Hessian đều bằng 0. Những vùng suy biến này gây ra các vấn đề lớn cho tất cả các thuật toán tối ưu hóa số học. Trong một bài toán lồi, một vùng rộng và phẳng phải hoàn toàn bao gồm các cực tiểu toàn cục, nhưng trong một bài toán tối ưu hóa tổng quát, một vùng như vậy có thể tương ứng với một giá trị cao của hàm mục tiêu.

8.2.4 Vách Đá và Gradients Bùng Nổ

Các mạng nơ-ron với nhiều lớp thường có những vùng cực kỳ dốc giống như vách đá, như minh họa trong hình 8.3. Những vùng này phát sinh từ việc nhân nhiều trọng số lớn với nhau. Trên bề mặt của một cấu trúc vách đá cực kỳ dốc, bước cập nhật gradient có thể di chuyển các tham số rất xa, thường là nhảy ra ngoài cấu trúc vách đá hoàn toàn.

Vách đá có thể nguy hiểm dù chúng ta tiếp cận nó từ trên hay từ dưới, nhưng may mắn thay, những hệ quả nghiêm trọng nhất có thể tránh được bằng cách sử dụng phương pháp gradient clipping (cắt gradient) được mô tả trong phần 10.11.1. Ý tưởng cơ bản là nhớ rằng gradient chỉ ra không phải kích thước bước tối ưu mà chỉ là hướng tối ưu trong một vùng vô cùng nhỏ. Khi thuật toán gradient descent truyền thống đề xuất bước đi rất lớn, phương pháp gradient clipping can thiệp để giảm kích thước bước, làm cho nó ít có khả năng đi ra ngoài vùng mà gradient chỉ ra hướng dốc nhất. Cấu trúc vách đá thường gặp nhất trong các hàm chi phí của mạng nơ-ron hồi tiếp (recurrent neural networks) vì các mô hình này bao gồm việc nhân nhiều yếu tố, mỗi yếu tố cho mỗi bước thời gian. Do đó, các chuỗi thời gian dài sẽ gặp phải một lượng nhân cực kỳ lớn.

8.2.5 Phụ Thuộc Dài Hạn

Một khó khăn khác mà các thuật toán tối ưu mạng nơ-ron phải vượt qua là khi đồ thị tính toán trở nên cực kỳ sâu. Các mạng feedforward với nhiều lớp có đồ thị tính toán sâu như vậy. Các mạng hồi tiếp, được mô tả trong chương 10, cũng có đồ thị tính toán rất sâu bằng cách áp dụng lặp đi lặp lại cùng một phép toán tại mỗi bước thời gian của một chuỗi thời gian dài. Việc áp dụng lặp lại các tham số giống nhau tạo ra những khó khăn đặc biệt rõ rệt.

Ví dụ, giả sử đồ thị tính toán chứa một đường đi bao gồm việc nhân liên tục với một ma trận W. Sau t bước, điều này tương đương với việc nhân với W^t. Giả sử W có phân tích trị riêng $W = V\text{diag}(\lambda)V^{-1}$. Trong trường hợp đơn giản này, ta có thể dễ dàng thấy rằng:

$$W^t = \left(V\text{diag}(\lambda)V^{-1}\right)^t = V\text{diag}(\lambda)^t V^{-1}.$$

Bất kỳ trị riêng λ_i nào không gần giá trị tuyệt đối là 1 sẽ hoặc bùng nổ nếu chúng lớn hơn 1 về độ lớn, hoặc biến mất nếu chúng nhỏ hơn 1 về độ lớn. Vấn đề gradient biến mất và bùng nổ liên quan đến việc các gradient đi qua đồ thị này cũng bị phóng đại theo $\text{diag}(\lambda)^t$. Gradients biến mất khiến cho việc xác định hướng đi của tham số để cải thiện hàm chi phí trở nên khó khăn, trong khi gradients bùng nổ có thể làm cho quá trình học trở nên không ổn định. Các cấu trúc vách đá được mô tả trước đó, làm động lực cho việc cắt gradient, là một ví dụ của hiện tượng gradient bùng nổ.

Việc nhân liên tục với W tại mỗi bước thời gian như đã mô tả ở đây rất giống với thuật toán phương pháp mạnh (power method) dùng để tìm trị riêng lớn nhất của ma trận W và vector trị riêng tương ứng. Từ góc độ này, không có gì ngạc nhiên khi $x^\top W^t$ sẽ loại bỏ tất cả các thành phần của x vuông góc với vector trị riêng chính của W. Các mạng hồi tiếp sử dụng cùng một ma trận W ở mỗi bước thời gian, nhưng các mạng feedforward thì không, vì vậy ngay cả các mạng feedforward rất sâu cũng có thể tránh được phần lớn vấn đề gradient biến mất và bùng nổ (Sussillo, 2014).

Chúng tôi sẽ hoãn thảo luận thêm về những thử thách trong việc huấn luyện các mạng hồi tiếp cho đến phần 10.7, sau khi các mạng hồi tiếp được mô tả chi tiết hơn.

8.2.6 Gradient Không Chính Xác

Hầu hết các thuật toán tối ưu được thiết kế với giả định rằng chúng ta có thể tiếp cận gradient hoặc ma trận Hessian chính xác. Tuy nhiên, trong thực tế, chúng ta thường chỉ có thể tiếp cận một ước lượng nhiễu hoặc thậm chí sai lệch của các đại lượng này. Gần như mọi thuật toán học sâu đều dựa vào các ước lượng dựa trên mẫu, ít nhất là trong việc sử dụng một minibatch các ví dụ huấn luyện để tính toán gradient.

Trong một số trường hợp, hàm mục tiêu mà chúng ta muốn tối thiểu hóa thực sự không thể giải quyết được. Khi hàm mục tiêu không thể giải quyết, gradient của nó thường cũng không thể tính được. Trong những trường hợp này, chúng ta chỉ có thể xấp xỉ gradient. Những vấn đề này chủ yếu xuất hiện với các mô hình phức tạp hơn mà chúng ta sẽ đề

cập trong phần III. Ví dụ, phương pháp phân kỳ tương phản (contrastive divergence) cung cấp một kỹ thuật để xấp xỉ gradient của log-likelihood không thể giải quyết của một máy Boltzmann.

Các thuật toán tối ưu mạng nơ-ron khác nhau được thiết kế để xử lý các sai sót trong ước lượng gradient. Một cách khác để tránh vấn đề này là chọn một hàm mất mát thay thế dễ xấp xỉ hơn so với hàm mất mát thực sự.

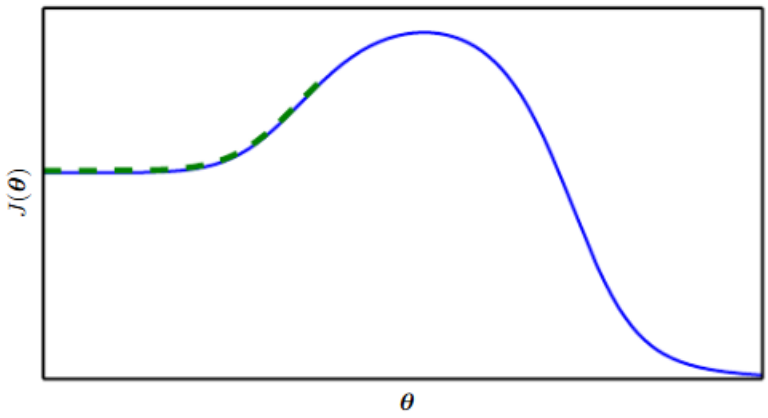
8.2.7 Mối Quan Hệ Kém Giữa Cấu Trúc Cục Bộ và Toàn Cục

Nhiều vấn đề mà chúng ta đã thảo luận cho đến nay liên quan đến các đặc điểm của hàm mất mát tại một điểm duy nhất—có thể rất khó khăn để thực hiện một bước nếu $J(\theta)$ có điều kiện kém tại điểm hiện tại θ, hoặc nếu θ nằm trên một vách đá, hoặc nếu θ là một điểm yên ngựa che giấu cơ hội tiến bộ theo hướng xuống dốc từ gradient.

Tuy nhiên, có thể vượt qua tất cả những vấn đề này tại một điểm duy nhất và vẫn thực hiện kém nếu hướng đi mang lại cải thiện nhiều nhất về mặt cục bộ không chỉ về phía các khu vực xa xôi có chi phí thấp hơn nhiều.

Goodfellow et al. (2015) lập luận rằng phần lớn thời gian chạy của quá trình huấn luyện là do độ dài của quỹ đạo cần thiết để đi đến giải pháp. Hình 8.2 cho thấy rằng quỹ đạo học tập dành phần lớn thời gian của nó để vẽ một cung rộng xung quanh cấu trúc hình núi.

Nhiều nghiên cứu về những khó khăn trong tối ưu hóa đã tập trung vào việc liệu quá trình huấn luyện có đến được điểm cực tiểu toàn cục, điểm cực tiểu cục bộ hay một điểm yên ngựa hay không. Tuy nhiên, trong thực tế, các mạng nơ-ron không đạt được một điểm cực trị nào cả. Hình 8.1 cho thấy các mạng nơ-ron thường không đến được một khu vực có gradient nhỏ. Thực tế, các điểm cực trị như vậy thậm chí không nhất thiết phải tồn tại. Ví dụ, hàm mất mát $-\log p(y|x;\theta)$ có thể không có điểm cực tiểu toàn cục và thay vào đó, tiếp cận một giá trị khi mô hình trở nên tự tin hơn. Đối với một bộ phân loại với y rời rạc và $p(y|x)$ được cung cấp bởi softmax, log-likelihood âm có thể trở nên gần bằng không nếu mô hình có thể phân loại chính xác mọi ví dụ trong bộ dữ liệu huấn luyện, nhưng không thể đạt được giá trị bằng không. Tương tự, một mô hình của các giá trị thực $p(y|x) = N(y; f(\theta), \beta^{-1})$ có thể có log-likelihood âm tiệm cận về vô cùng âm—nếu $f(\theta)$ có thể dự đoán chính xác giá trị của tất cả các mục tiêu y trong bộ huấn luyện, thuật toán học sẽ làm tăng β vô hạn. Xem hình 8.4 để biết ví dụ về sự thất bại của tối ưu hóa cục bộ trong việc tìm giá trị hàm mất mát tốt ngay cả khi không có điểm cực tiểu cục bộ hay điểm yên ngựa.



Hình 8.4: Tối ưu hóa dựa trên các bước đi xuống dốc cục bộ có thể thất bại nếu bề mặt cục bộ không chỉ vào giải pháp toàn cục. Dưới đây là một ví dụ về cách điều này có thể xảy ra, ngay cả khi không có điểm yên ngựa hay cực tiểu cục bộ. Hàm mất mát ví dụ này chỉ có các tiệm cận về phía các giá trị thấp, không phải cực tiểu. Nguyên nhân chính của sự khó khăn trong trường hợp này là việc khởi tạo ở phía sai của "ngọn núi"và không thể di chuyển qua nó. Trong không gian có chiều cao hơn, các thuật toán học có thể thường xuyên đi vòng qua những ngọn núi như vậy, nhưng quỹ đạo liên quan đến việc làm như vậy có thể rất dài và dẫn đến thời gian huấn luyện quá mức, như được minh họa trong Hình 8.2.

Nghiên cứu trong tương lai cần phát triển sự hiểu biết sâu hơn về các yếu tố ảnh hưởng đến độ dài của quỹ đạo học và mô tả chính xác hơn kết quả của quá trình này. Nhiều hướng nghiên cứu hiện tại nhắm đến việc tìm kiếm các điểm khởi tạo tốt cho các bài toán có cấu trúc toàn cục khó khăn, thay vì phát triển các thuật toán sử dụng các bước di chuyển không cục bộ.

Phương pháp gradient descent và gần như tất cả các thuật toán học hiệu quả cho việc huấn luyện mạng nơ-ron đều dựa trên việc thực hiện các bước di chuyển cục bộ nhỏ. Các phần trước đã tập trung chủ yếu vào việc làm thế nào để xác định được hướng chính xác của những bước di chuyển cục bộ này có thể gặp khó khăn. Chúng ta có thể chỉ tính toán một số thuộc tính của hàm mục tiêu, chẳng hạn như gradient của nó, chỉ xấp xỉ, với độ lệch hoặc phương sai trong việc ước lượng hướng chính xác. Trong những trường hợp này, việc di chuyển cục bộ có thể không xác định được một quỹ đạo ngắn hợp lý đến một giải pháp hợp lệ, nhưng chúng ta không thực sự có thể theo đuổi con đường đi xuống cục bộ. Hàm mục tiêu có thể gặp phải các vấn đề, chẳng hạn như điều kiện kém hoặc gradient không liên tục, khiến cho vùng mà tại đó gradient cung cấp một mô hình tốt của hàm mục tiêu trở nên rất nhỏ. Trong những trường hợp này, việc di chuyển cục bộ với các bước có kích thước ϵ có thể xác định một con đường ngắn hợp lý đến giải pháp, nhưng chúng ta chỉ có thể tính toán hướng di chuyển cục bộ với các bước có kích thước $\delta \ll \epsilon$. Trong những trường hợp này, việc di chuyển cục bộ có thể xác định một con đường đến giải pháp, nhưng con đường này chứa nhiều bước, vì vậy việc đi theo nó sẽ tốn chi phí tính toán cao.

Đôi khi, thông tin cục bộ không cung cấp cho chúng ta một hướng dẫn nào, chẳng hạn như khi hàm mục tiêu có một vùng phẳng rộng, hoặc nếu chúng ta vô tình rơi đúng vào

một điểm quan trọng (thường thì kịch bản sau chỉ xảy ra đối với các phương pháp giải quyết trực tiếp các điểm quan trọng, như phương pháp Newton). Trong những trường hợp này, việc di chuyển cục bộ không xác định một con đường đến giải pháp. Trong các trường hợp khác, các bước di chuyển cục bộ có thể quá tham lam và dẫn chúng ta theo một con đường di chuyển xuống dốc nhưng lại đi xa khỏi bất kỳ giải pháp nào, như trong Hình 8.4, hoặc dẫn đến một quỹ đạo không cần thiết dài đến giải pháp, như trong Hình 8.2. Hiện tại, chúng ta chưa hiểu rõ đâu là vấn đề quan trọng nhất khiến cho việc tối ưu hóa mạng nơ-ron trở nên khó khăn, và đây là một lĩnh vực nghiên cứu đang rất tích cực.

Dù vấn đề nào trong số này là quan trọng nhất, tất cả chúng đều có thể được tránh nếu có tồn tại một vùng không gian được kết nối khá trực tiếp với giải pháp bởi một con đường mà việc di chuyển cục bộ có thể theo. Và nếu chúng ta có thể khởi tạo quá trình học trong vùng có hành vi tốt đó. Quan điểm cuối cùng này gợi ý nghiên cứu về việc chọn lựa các điểm khởi tạo tốt để các thuật toán tối ưu hóa truyền thống có thể sử dụng.

8.2.8 Giới Hạn Lý Thuyết của Tối Ưu Hóa

Một số kết quả lý thuyết chỉ ra rằng có những giới hạn đối với hiệu suất của bất kỳ thuật toán tối ưu hóa nào mà chúng ta có thể thiết kế cho mạng nơ-ron (Blum và Rivest, 1992; Judd, 1989; Wolpert và MacReady, 1997). Thông thường, những kết quả này ít ảnh hưởng đến việc sử dụng mạng nơ-ron trong thực tế.

Một số kết quả lý thuyết chỉ áp dụng khi các đơn vị của mạng nơ-ron xuất ra các giá trị rời rạc. Hầu hết các đơn vị mạng nơ-ron xuất ra các giá trị tăng dần mượt mà, điều này làm cho việc tối ưu hóa thông qua tìm kiếm cục bộ trở nên khả thi. Một số kết quả lý thuyết chỉ ra rằng tồn tại các lớp bài toán không thể giải quyết được, nhưng đôi khi rất khó để xác định liệu một bài toán cụ thể có thuộc lớp này hay không. Các kết quả khác cho thấy việc tìm giải pháp cho một mạng có kích thước nhất định là không thể giải quyết được, nhưng trong thực tế, chúng ta có thể dễ dàng tìm được giải pháp bằng cách sử dụng một mạng lớn hơn, trong đó có nhiều cấu hình tham số tương ứng với một giải pháp chấp nhận được.

Hơn nữa, trong bối cảnh huấn luyện mạng nơ-ron, chúng ta thường không quan tâm đến việc tìm kiếm cực tiểu chính xác của một hàm, mà chỉ mong muốn giảm giá trị của nó đủ để có được lỗi tổng quát tốt. Phân tích lý thuyết về việc liệu một thuật toán tối ưu hóa có thể đạt được mục tiêu này hay không là một vấn đề cực kỳ khó khăn. Do đó, phát triển các giới hạn thực tế hơn về hiệu suất của các thuật toán tối ưu hóa vẫn là một mục tiêu quan trọng trong nghiên cứu học máy.

8.3 Thuật Toán Cơ Bản

Trong chương trước, chúng ta đã tìm hiểu thuật toán gradient descent (xem Mục 4.3), trong đó tham số mô hình được cập nhật dựa trên đạo hàm (gradient) tính trên toàn bộ tập huấn luyện. Phương pháp này hiệu quả nhưng rất tốn thời gian khi tập dữ liệu có kích thước lớn.

Một cách tiếp cận nhanh hơn và phổ biến hơn là sử dụng Stochastic Gradient Descent

(SGD) — thuật toán dựa trên việc chọn ngẫu nhiên các minibatch từ tập dữ liệu và cập nhật mô hình theo đạo hàm tính trên minibatch đó. Phương pháp này đã được trình bày trong các mục 5.9 và 8.1.3.

8.3.1 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) và các biến thể là một trong những thuật toán tối ưu phổ biến nhất trong học máy và học sâu. Thay vì tính gradient trên toàn bộ tập dữ liệu (batch gradient), SGD sử dụng một minibatch nhỏ để ước lượng gradient. Điều này giúp giảm chi phí tính toán đáng kể và tăng tốc độ huấn luyện.

Như trình bày trong mục 8.1.3, việc tính gradient trung bình trên minibatch gồm m ví dụ được lấy ngẫu nhiên và độc lập từ phân phối dữ liệu sẽ tạo ra một ước lượng không thiên lệch của gradient thật.

Thuật toán 8.1: Cập nhật tham số bằng SGD

Algorithm 8.1 Stochastic Gradient Descent (SGD)

Require: Lịch tốc độ học $\epsilon_1, \epsilon_2, \ldots$
Require: Tham số khởi tạo θ
1: $k \leftarrow 1$
2: **while** chưa đạt điều kiện dừng **do**
3: Lấy minibatch gồm m ví dụ $\{x^{(1)}, \ldots, x^{(m)}\}$ và nhãn tương ứng $y^{(i)}$
4: Tính gradient trung bình:

$$\hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

5: Cập nhật tham số: $\theta \leftarrow \theta - \epsilon_k \cdot \hat{g}$
6: $k \leftarrow k + 1$
7: **end while**

Ước lượng gradient từ minibatch chứa một thành phần nhiễu (do chọn mẫu ngẫu nhiên). Để SGD hội tụ về điểm cực tiểu của hàm mất mát, cần thỏa mãn điều kiện:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \tag{8.7}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty \tag{8.8}$$

Trong thực tiễn, tốc độ học thường được giảm tuyến tính trong các vòng đầu:

$$\epsilon_k = (1 - \alpha \cdot \tau)\epsilon_0 + \alpha \cdot \tau \cdot \epsilon \tag{8.9}$$

Với $\alpha = \frac{k}{\tau}$. Sau thời điểm τ, tốc độ học được giữ hằng số tại ϵ.

Tốc độ học (*learning rate*) là một trong những siêu tham số quan trọng nhất trong huấn luyện mạng nơ-ron. Việc chọn giá trị phù hợp cho tốc độ học thường mang tính thực nghiệm, nghĩa là cần thử nghiệm nhiều lần và theo dõi kết quả trên *learning curve* (đường cong học tập) – biểu diễn giá trị hàm mất mát theo thời gian huấn luyện. Việc này phần nhiều mang tính nghệ thuật hơn là một khoa học chính xác. Do đó, các lời khuyên về cách chọn tốc độ học nên được xem xét một cách thận trọng.

Khi sử dụng linear schedule – tức tốc độ học giảm tuyến tính theo thời gian – ta cần xác định các tham số: ϵ_0 (tốc độ học ban đầu), ϵ_τ (giá trị tốc độ học sau thời gian τ), và chính τ (thời gian giảm tốc độ học). Trong thực tế, τ thường được đặt tương đương với số vòng lặp cần thiết để quét qua tập huấn luyện vài trăm lần, còn ϵ_τ thường lấy khoảng 1% so với ϵ_0.

Câu hỏi quan trọng là làm thế nào để đặt ϵ_0. Nếu tốc độ học ban đầu quá lớn, đường cong học tập có thể dao động dữ dội, thậm chí làm hàm mất mát tăng lên đáng kể ở một số bước. Một số dao động nhẹ là chấp nhận được, đặc biệt là khi huấn luyện với hàm mất mát ngẫu nhiên như khi dùng *dropout*.

Ngược lại, nếu tốc độ học quá thấp thì quá trình học sẽ chậm chạp. Trong nhiều trường hợp, tốc độ học ban đầu thấp còn có thể khiến mô hình bị "kẹt" ở một giá trị hàm mất mát cao, không học thêm được gì.

Thông thường, tốc độ học tối ưu ban đầu (tính theo tổng thời gian huấn luyện và giá trị hàm mất mát cuối cùng) sẽ cao hơn tốc độ học tốt nhất trong khoảng 100 bước lặp đầu tiên. Vì vậy, chiến lược tốt là quan sát một số bước đầu, sau đó chọn tốc độ học hơi cao hơn giá trị tốt nhất trong giai đoạn này, nhưng không quá cao để gây ra mất ổn định.

Ưu điểm nổi bật nhất của SGD (Stochastic Gradient Descent) và các biến thể như minibatch hay online gradient descent là: *thời gian tính toán mỗi bước cập nhật không phụ thuộc vào số lượng mẫu huấn luyện*. Nhờ đó, ta có thể hội tụ đến một nghiệm gần tối ưu mà không cần phải quét hết toàn bộ dữ liệu nếu tập huấn luyện đủ lớn.

Để phân tích tốc độ hội tụ, người ta thường dùng khái niệm lỗi dư thừa (excess error): $J(\theta) - \min_\theta J(\theta)$, biểu thị khoảng cách giữa hàm mất mát hiện tại và giá trị tối thiểu của nó. Trong bài toán lồi, SGD đạt lỗi dư thừa $O(\frac{1}{\sqrt{k}})$ sau k bước, còn nếu bài toán là strongly convex, tốc độ hội tụ là $O(\frac{1}{k})$. Những giới hạn này là tối ưu, trừ khi có thêm giả định mạnh hơn.

Về lý thuyết, Batch Gradient Descent có tốc độ hội tụ tốt hơn SGD. Tuy nhiên, định lý Cramér-Rao (Cramér, 1946; Rao, 1945) chỉ ra rằng lỗi khái quát (generalization error) không thể giảm nhanh hơn $O(\frac{1}{k})$. Bottou và Bousquet (2008) cho rằng: trong bối cảnh học máy, việc cố gắng thiết kế thuật toán hội tụ nhanh hơn $O(\frac{1}{k})$ là không cần thiết, vì tốc độ đó dễ dẫn đến *overfitting*.

Thêm vào đó, những lợi ích thực tiễn của SGD thường bị che khuất trong phân tích tiệm cận. Cụ thể, với dữ liệu lớn, khả năng cập nhật mô hình nhanh chóng bằng cách tính

gradient từ một số mẫu nhỏ của SGD giúp mô hình tiến bộ vượt trội trong giai đoạn đầu, dù tốc độ hội tụ lâu dài có thể chậm hơn.

Một chiến lược phổ biến là kết hợp ưu điểm của cả hai phương pháp: bắt đầu bằng SGD với minibatch nhỏ để học nhanh lúc đầu, sau đó tăng dần kích thước minibatch trong quá trình huấn luyện nhằm cải thiện độ chính xác và hội tụ ổn định hơn.

Tham khảo thêm: Xem Bottou (1998) để biết chi tiết về thuật toán SGD.

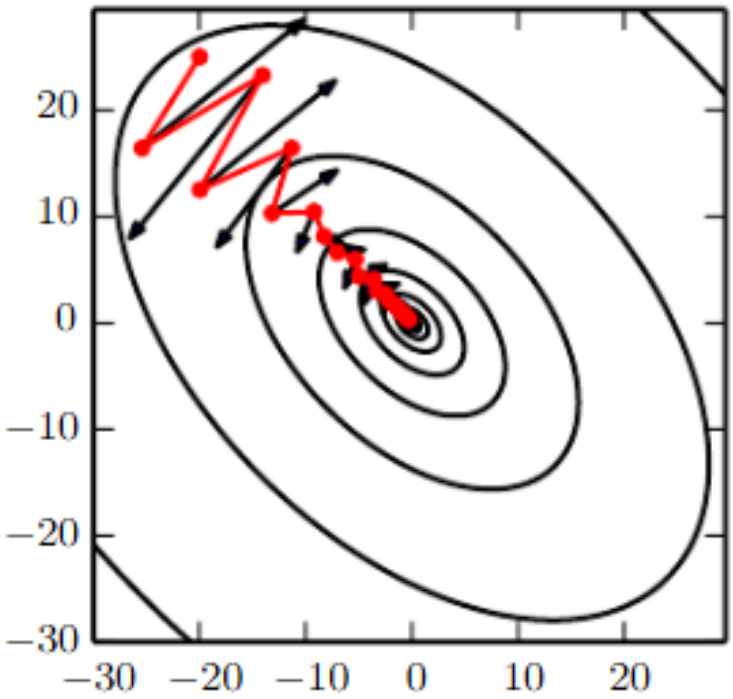
8.3.2 Động lượng (Momentum)

Trong khi phương pháp gradient descent ngẫu nhiên (SGD) vẫn là một chiến lược tối ưu phổ biến, thì việc học bằng SGD đôi khi có thể diễn ra chậm chạp. Phương pháp động lượng (Momentum), được đề xuất bởi Polyak (1964), được thiết kế nhằm tăng tốc quá trình học, đặc biệt là trong các trường hợp:

- Hàm mất mát có độ cong cao (high curvature),
- Gradient nhỏ nhưng ổn định (small but consistent gradients),
- Gradient bị nhiễu (noisy gradients).

Ý tưởng chính của động lượng là tích lũy một trung bình trượt hàm mũ của các gradient trong quá khứ và tiếp tục di chuyển theo hướng đó. Việc này giúp tránh hiện tượng mô hình bị "kẹt" trong các vùng bằng phẳng hoặc dao động quá nhiều do gradient nhiễu.

Tên gọi "động lượng" xuất phát từ phép ẩn dụ vật lý: Gradient âm được coi như là một lực (force) đẩy một hạt (particle) di chuyển trong không gian tham số theo các định luật chuyển động của Newton. Động lượng trong vật lý được định nghĩa là khối lượng nhân với vận tốc. Trong bối cảnh học máy, ta giả định khối lượng là 1, nên vectơ vận tốc v cũng chính là động lượng.



Hình 8.5: Momentum được thiết kế nhằm giải quyết chủ yếu hai vấn đề sau trong quá trình tối ưu hóa: Tình trạng điều kiện kém của ma trận Hessian (*poor conditioning of the Hessian matrix*). Độ biến thiên trong gradient ngẫu nhiên (*variance in the stochastic gradient*). Trong phần này, ta minh họa cách mà momentum giúp vượt qua vấn đề đầu tiên: ma trận Hessian có điều kiện kém. Các đường đồng mức trong hình biểu diễn một hàm mất mát bậc hai có ma trận Hessian điều kiện kém. Trong những bài toán như vậy, không gian tham số có hình dạng giống một thung lũng dài và hẹp hoặc một *hẻm núi* — với các cạnh dốc đứng và đáy rất hẹp. Đường màu đỏ cắt ngang qua các đường đồng mức thể hiện quỹ đạo mà thuật toán học có động lượng (momentum) đi theo khi tối thiểu hóa hàm mất mát này. Tại mỗi điểm trên đường đi, một mũi tên được vẽ ra để chỉ hướng mà thuật toán gradient descent thông thường sẽ chọn tại điểm đó. Ta có thể quan sát rằng trong một bài toán tối ưu với hàm mục tiêu bậc hai có điều kiện kém, gradient descent truyền thống sẽ liên tục dao động qua lại giữa hai bên của hẻm núi do phương gradient bị lệch hướng nghiêm trọng. Quá trình này vừa tốn thời gian vừa không hiệu quả. Ngược lại, momentum tích lũy thông tin từ các bước trước, giúp nó đi đúng hướng dọc theo đáy hẻm núi — chính là trục dài của vùng hội tụ — từ đó rút ngắn thời gian tối ưu đáng kể. Hãy so sánh với hình **??** ở chương 4 để thấy rõ sự khác biệt giữa gradient descent thường và momentum.

Phương pháp động lượng sử dụng một biến phụ trợ v (vectơ vận tốc) để lưu giữ hướng di chuyển tích lũy, và được cập nhật theo quy tắc sau:

$$v \leftarrow \alpha v - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(x^{(i)}; \boldsymbol{\theta}), y^{(i)}) \right) \tag{8.10}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + v \tag{8.11}$$

Trong đó:

- v: vận tốc hiện tại (vectơ tích lũy gradient).

- $\alpha \in [0, 1)$: siêu tham số điều khiển mức độ ảnh hưởng của các gradient cũ (gọi là hệ số động lượng).
- ϵ: tốc độ học (learning rate).
- $\nabla_{\boldsymbol{\theta}}$: gradient theo tham số $\boldsymbol{\theta}$.
- m: kích thước minibatch.

Algorithm 8.2 Stochastic Gradient Descent (SGD) with Momentum

Require: Tốc độ học ϵ, tham số momentum α
Require: Tham số ban đầu θ, vận tốc ban đầu v
1: **while** chưa hội tụ **do**
2: Lấy minibatch gồm m mẫu từ tập huấn luyện
3: Tính gradient:

$$g \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

4: Cập nhật vận tốc: $v \leftarrow \alpha v - \epsilon g$
5: Cập nhật tham số: $\theta \leftarrow \theta + v$
6: **end while**

Thuật toán sử dụng một vector vận tốc v để tích lũy các gradient trong quá khứ theo cách trung bình trượt mũ. Nếu các gradient liên tục có cùng hướng, động lượng sẽ giúp tăng tốc hội tụ theo hướng đó. Ngược lại, nếu gradient thay đổi đột ngột, động lượng sẽ làm trơn quá trình cập nhật.

Khi gradient không đổi theo thời gian, bước cập nhật hội tụ về giá trị:

$$\frac{\epsilon \|g\|}{1 - \alpha} \tag{8.12}$$

Tham số α thường được chọn là 0.9 hoặc 0.99 trong thực nghiệm, giúp tăng tốc tối ưu trong các bài toán có độ cong cao hoặc ma trận Hessian điều kiện kém.

Ta có thể hiểu momentum như một mô phỏng của hạt chịu tác động lực trong không gian tham số. Hạt có vị trí $\theta(t)$, và bị tác động bởi một lực $f(t)$. Theo định luật II Newton, lực này gây ra gia tốc:

$$f(t) = \frac{\partial^2 \theta(t)}{\partial t^2} \tag{8.13}$$

Thay vì xem đây là một phương trình vi phân bậc hai của vị trí $\theta(t)$, ta có thể giới thiệu thêm biến vận tốc $v(t)$ để biểu diễn tốc độ của hạt tại thời điểm t và viết lại động lực học

Newton dưới dạng hệ phương trình vi phân bậc nhất như sau:

$$v(t) = \frac{\partial}{\partial t}\theta(t) \quad (8.19) \tag{8.14}$$

$$f(t) = \frac{\partial}{\partial t}v(t) \quad (8.20) \tag{8.15}$$

Trong đó:

- $\theta(t)$ là vị trí (tham số mô hình) tại thời điểm t.
- $v(t)$ là vận tốc (tốc độ thay đổi tham số).
- $f(t)$ là lực tác động vào hạt.

Thuật toán momentum được xây dựng dựa trên việc giải hệ phương trình vi phân này bằng mô phỏng số. Một phương pháp đơn giản để giải phương trình vi phân là phương pháp Euler, trong đó ta cập nhật các giá trị bằng cách đi từng bước nhỏ theo hướng của đạo hàm.

Câu hỏi đặt ra là: các lực $f(t)$ cụ thể trong mô hình tối ưu là gì?

Một lực chính trong hệ là lực tỉ lệ với âm của đạo hàm hàm mất mát:

$$f_{\text{grad}} = -\nabla_\theta J(\theta)$$

Lực này khiến hạt trượt xuống theo hướng giảm của hàm mất mát. Trong thuật toán gradient descent cơ bản, mỗi bước cập nhật chỉ dựa trên đạo hàm này. Nhưng trong thuật toán momentum, lực này làm thay đổi vận tốc $v(t)$, từ đó ảnh hưởng đến vị trí $\theta(t)$.

Hãy tưởng tượng một *quả bóng khúc côn cầu* đang trượt trên mặt băng. Khi quả bóng đi xuống một dốc đứng (gradient lớn), nó sẽ tăng tốc và tiếp tục trượt theo hướng đó, ngay cả khi dốc bắt đầu thoải ra — đó chính là hiệu ứng momentum.

Nếu chỉ có lực từ gradient, quả bóng sẽ mãi trượt qua lại mà không bao giờ dừng lại, như một vật trên mặt băng hoàn hảo không ma sát. Do đó, ta cần thêm một lực cản tỉ lệ với vận tốc:

$$f_{\text{drag}} = -\gamma v(t)$$

với $\gamma > 0$ là hệ số nhớt.

Trong vật lý, đây là lực cản nhớt — giống như khi hạt chuyển động trong xi-rô. Lực này làm hạt mất dần năng lượng, giúp nó dần dừng lại ở vị trí tối ưu (tối tiểu địa phương).

Chúng ta có thể tưởng tượng nhiều loại lực cản khác nhau:

- Lực cản hỗn loạn (turbulent drag): tỉ lệ với $v(t)^2$, thường gặp khi vật di chuyển trong không khí. Tuy nhiên, khi vận tốc nhỏ, lực này trở nên quá yếu, không đủ để dừng vật lại. Hạt sẽ tiếp tục di chuyển mãi, cách xa điểm ban đầu với khoảng cách tăng theo $O(\log t)$.

- Ma sát khô (dry friction): là lực có độ lớn không đổi, không phụ thuộc vào vận tốc. Nếu gradient nhỏ, lực này có thể khiến hạt dừng lại sớm, chưa kịp đến được điểm cực tiểu.

Lực cản nhớt là lựa chọn hợp lý nhất:

- Khi gradient đủ lớn, nó vẫn có thể khiến hạt tiếp tục di chuyển.
- Khi gradient gần như bằng 0, lực cản sẽ đủ mạnh để dừng hạt lại, giúp thuật toán hội tụ.

Do đó, ta chọn lực cản nhớt $-\gamma v(t)$ vì tính cân bằng giữa sự mềm dẻo và khả năng kiểm soát chuyển động.

Thuật toán momentum là sự kết hợp giữa hai yếu tố:

- Gradient như một lực đẩy hướng về cực tiểu.
- Lực cản nhớt để kiểm soát và làm trơn chuyển động.

Bằng cách mô phỏng vật lý qua hệ phương trình vi phân, momentum giúp quá trình huấn luyện nhanh hơn, mượt mà hơn và dễ dàng vượt qua các hố nhỏ trong hàm mất mát.

8.3.3 Nesterov Momentum

Sutskever et al. (2013) đã giới thiệu một biến thể của thuật toán momentum, lấy cảm hứng từ phương pháp gradient tăng tốc của Nesterov (1983, 2004). Biến thể này được gọi là *Nesterov Accelerated Gradient* (NAG). Quy tắc cập nhật trong trường hợp này được viết như sau:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left[\frac{1}{m} \sum_{i=1}^{m} \mathcal{L} \left(f(x^{(i)}; \theta + \alpha \mathbf{v}), y^{(i)} \right) \right], \tag{8.21}$$

$$\theta \leftarrow \theta + \mathbf{v} \tag{8.22}$$

Trong đó:

- θ là tham số mô hình.
- $\mathbf{v}$ là vận tốc (momentum).
- α là hệ số momentum ($0 < \alpha < 1$).
- ϵ là tốc độ học (learning rate).
- ∇_θ là đạo hàm theo tham số θ.

Sự khác biệt chính giữa Momentum truyền thống và Nesterov Momentum nằm ở điểm mà gradient được tính:

- Với Momentum thường, gradient được tính tại vị trí hiện tại θ.
- Với Nesterov, gradient được tính tại vị trí được "nhìn trước" là $\theta + \alpha \mathbf{v}$, tức là sau khi

áp dụng momentum.

Có thể hiểu rằng Nesterov momentum bổ sung một yếu tố hiệu chỉnh (correction factor) cho phương pháp momentum chuẩn, giúp thuật toán phản ứng trước những thay đổi của hướng di chuyển, từ đó tăng hiệu quả hội tụ.

Trong trường hợp bài toán lồi (convex) và dùng gradient theo mini-batch cố định, Nesterov momentum giúp cải thiện tốc độ hội tụ của sai số dư từ $O(1/k)$ (sau k bước) lên $O(1/k^2)$ như được chứng minh trong công trình của Nesterov (1983).

Tuy nhiên, khi áp dụng cho trường hợp gradient ngẫu nhiên (stochastic gradient descent), cải tiến này không còn giúp tăng tốc độ hội tụ nữa. Điều đó nghĩa là trong thực hành với dữ liệu thực, hiệu quả của Nesterov có thể phụ thuộc vào cách triển khai và điều kiện cụ thể.

Nesterov Momentum là một chiến lược thông minh giúp:

- "Nhìn trước" vị trí tiếp theo dựa trên vận tốc hiện tại.
- Cập nhật gradient ở điểm xa hơn thay vì hiện tại.
- Tăng khả năng hội tụ nhanh hơn trong một số trường hợp lý tưởng.

Thuật toán này đặc biệt hữu ích trong các bài toán lồi và vẫn được áp dụng rộng rãi trong huấn luyện mạng nơ-ron sâu.

8.4 Chiến lược khởi tạo tham số

Trong học sâu, các thuật toán tối ưu thường mang tính lặp và cần một điểm khởi tạo ban đầu. Việc lựa chọn điểm khởi tạo này ảnh hưởng lớn đến quá trình huấn luyện.

- Nếu khởi tạo không tốt, mô hình có thể không hội tụ hoặc gặp lỗi số học nghiêm trọng.
- Khởi tạo ảnh hưởng đến tốc độ hội tụ và chất lượng nghiệm cuối cùng.
- Ngay cả khi đạt chi phí huấn luyện tương tự, nghiệm khác nhau có thể cho sai số khái quát khác nhau.

Vì vậy, lựa chọn chiến lược khởi tạo tham số phù hợp là rất quan trọng để đảm bảo huấn luyện ổn định, nhanh chóng và đạt kết quả tốt.

Algorithm 8.3 Thuật toán 8.3: Stochastic Gradient Descent (SGD) với Nesterov Momentum

Yêu cầu: Tốc độ học ϵ, tham số momentum α
Khởi tạo: Tham số ban đầu θ, vận tốc ban đầu $\mathbf{v}$

1: **while** chưa đạt tiêu chí dừng **do**
2: Lấy một minibatch gồm m ví dụ từ tập huấn luyện $\{x^{(1)}, \ldots, x^{(m)}\}$ với nhãn tương ứng $y^{(i)}$.
3: Áp dụng bước cập nhật tạm thời: $\tilde{\theta} \leftarrow \theta + \alpha\mathbf{v}$.
4: Tính gradient tại điểm tạm thời:

$$\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\tilde{\theta}} \sum_i L\big(f(x^{(i)}; \tilde{\theta}), y^{(i)}\big)$$

5: Cập nhật vận tốc:

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\mathbf{g}$$

6: Cập nhật tham số:

$$\theta \leftarrow \theta + \mathbf{v}$$

7: **end while**

Giải thích:

- Khác với SGD truyền thống, Nesterov momentum "nhìn trước" bằng cách tạm thời tiến thêm một bước theo vận tốc trước khi tính gradient.
- Gradient $\mathbf{g}$ được tính ở vị trí đã cộng trước một phần vận tốc, giúp thuật toán phản ứng nhanh hơn khi sắp đến cực tiểu.
- Vận tốc $\mathbf{v}$ vừa kế thừa một phần vận tốc cũ ($\alpha\mathbf{v}$), vừa được điều chỉnh bởi gradient mới ($-\epsilon\mathbf{g}$).
- Cuối cùng, tham số θ được cập nhật bằng cách cộng thêm vận tốc mới.

Các chiến lược khởi tạo tham số hiện đại thường đơn giản và dựa trên các nguyên lý thực nghiệm. Việc thiết kế những chiến lược khởi tạo tốt hơn là một thách thức lớn bởi quá trình tối ưu hóa mạng nơ-ron hiện nay vẫn chưa được hiểu đầy đủ.

Phần lớn các chiến lược khởi tạo đều nhằm đạt được một số tính chất "đẹp"khi mạng vừa mới khởi tạo. Tuy nhiên, chúng ta chưa có sự hiểu biết tốt về việc những tính chất này được duy trì ra sao trong quá trình huấn luyện. Thêm vào đó, một số điểm khởi tạo có thể giúp tối ưu hóa dễ dàng hơn nhưng lại gây hại cho khả năng tổng quát hóa của mô hình.

Hiện nay, sự hiểu biết về cách điểm khởi tạo ảnh hưởng đến khả năng tổng quát hóa còn rất sơ khai, gần như không có hướng dẫn nào rõ ràng về cách chọn điểm khởi tạo.

Điều duy nhất mà ta biết chắc chắn là: tham số khởi tạo cần phá vỡ tính đối xứng giữa các đơn vị nơ-ron. Nếu hai nơ-ron ẩn có cùng hàm kích hoạt và kết nối cùng một tập input, chúng phải được khởi tạo khác nhau. Nếu không, trong các mô hình và thuật toán huấn

luyện xác định, chúng sẽ luôn được cập nhật giống hệt nhau trong suốt quá trình học, làm giảm tính đa dạng của mạng.

Ngay cả khi thuật toán huấn luyện có tính ngẫu nhiên (ví dụ như sử dụng dropout), tốt nhất vẫn nên khởi tạo để mỗi nơ-ron ban đầu tính toán một hàm khác biệt, nhằm đảm bảo:
- Không mất mẫu đầu vào nào do rơi vào không gian null của lan truyền tiến.
- Không mất mẫu gradient nào trong quá trình lan truyền ngược.

Điều này thúc đẩy việc khởi tạo tham số ngẫu nhiên. Chúng ta có thể tìm kiếm một tập hợp các hàm cơ sở (basis functions) khác biệt, ví dụ như dùng phương pháp trực giao Gram-Schmidt để tạo ma trận trọng số ban đầu, nhưng cách này tốn kém tính toán.

Giải pháp phổ biến hơn là khởi tạo ngẫu nhiên từ phân phối có entropy cao trong không gian chiều cao, rẻ hơn về chi phí tính toán và cũng ít khả năng sinh ra hai đơn vị tính cùng một hàm.

Thông thường:

- Các bias của mỗi đơn vị được đặt thành các hằng số được chọn theo kinh nghiệm.
- Chỉ trọng số được khởi tạo ngẫu nhiên.
- Các tham số phụ khác (ví dụ tham số phương sai có điều kiện) cũng thường được đặt các giá trị hằng.

Hầu hết các trọng số được khởi tạo từ phân phối Gaussian hoặc phân phối đều. Lựa chọn loại phân phối (Gaussian hay Uniform) không ảnh hưởng lớn, nhưng độ lớn của giá trị khởi tạo có ảnh hưởng rất lớn đến quá trình tối ưu hóa và khả năng tổng quát hóa của mạng.

- Trọng số lớn giúp phá vỡ đối xứng mạnh mẽ hơn, tránh tình trạng các đơn vị nơ-ron hoạt động giống nhau.
- Trọng số lớn cũng giúp tín hiệu không bị triệt tiêu khi lan truyền tiến hoặc lan truyền ngược, nhờ vào các phép nhân ma trận cho giá trị lớn hơn.
- Tuy nhiên, trọng số quá lớn có thể dẫn đến bùng nổ giá trị trong lan truyền tiến hoặc ngược.
- Trong mạng hồi tiếp (RNN), trọng số lớn còn có thể gây ra hiện tượng hỗn loạn (chaotic behavior) làm cho mạng trở nên quá nhạy cảm với nhiễu nhỏ.
- Việc gradient bùng nổ có thể giảm nhẹ bằng kỹ thuật gradient clipping (giới hạn giá trị gradient trước bước cập nhật).
- Nếu trọng số quá lớn cũng dễ làm hàm kích hoạt bị bão hòa (ví dụ ReLU âm toàn bộ hoặc sigmoid tiến sát 0/1), gây mất gradient.

Những yếu tố cạnh tranh này quyết định quy mô lý tưởng của trọng số khởi tạo.

Chọn cách khởi tạo phù hợp, nhất là xác định quy mô của giá trị khởi tạo, là yếu tố rất quan trọng, ảnh hưởng sâu sắc đến sự thành công của việc huấn luyện mạng nơ-ron.

Góc nhìn từ tối ưu hóa và từ chính quy hóa (regularization) đôi khi cho ta những khuyến nghị đối lập về cách khởi tạo mạng:

- Tối ưu hóa: trọng số nên đủ lớn để truyền tín hiệu hiệu quả.
- Chính quy hóa: nên giữ trọng số nhỏ để hạn chế overfitting.

Thuật toán tối ưu như gradient descent hoặc stochastic gradient descent thường cập nhật tham số bằng những bước rất nhỏ, và có xu hướng dừng lại gần với tham số khởi tạo (do gradient nhỏ hoặc do dừng sớm để tránh overfitting).

Điều này giống như áp dụng một prior Gaussian với trung bình tại tham số khởi tạo θ_0. Nếu ta đặt θ_0 gần 0, ta ngầm giả định rằng các đơn vị trong mạng không tương tác mạnh trừ khi có lý do đặc biệt.

Nếu khởi tạo θ_0 lớn, tức là ta đã xác định sẵn sự tương tác mạnh giữa các đơn vị.

Một số phương pháp khởi tạo thực nghiệm:

- Khởi tạo trọng số lớp fully connected với m input và n output từ phân phối đều $U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$.
- Glorot và Bengio (2010) đề xuất Xavier initialization:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \tag{8.16}$$

  nhằm cân bằng phương sai của đầu ra và gradient qua các lớp.
- Saxe và cộng sự (2013) đề xuất khởi tạo bằng ma trận trực giao ngẫu nhiên, sau đó nhân với hệ số scale phù hợp với loại hàm kích hoạt.

Dù các giả định lý thuyết (mô hình mạng tuyến tính không có phi tuyến) không hoàn toàn đúng với mạng thực tế, nhưng những phương pháp này thường cho kết quả thực nghiệm tốt.

Chọn cách khởi tạo phù hợp, nhất là xác định quy mô và cấu trúc trọng số ban đầu, có ảnh hưởng sâu sắc đến khả năng học và tổng quát hóa của mạng nơ-ron.

Một quy tắc kinh nghiệm hữu ích khi chọn tỉ lệ khởi tạo ban đầu là xem xét phạm vi hoặc độ lệch chuẩn của các kích hoạt (activation) hoặc gradient trên một minibatch dữ liệu duy nhất.

Nếu trọng số quá nhỏ, phạm vi của các kích hoạt trên minibatch sẽ co lại khi lan truyền tiến về phía trước trong mạng. Bằng cách liên tục xác định lớp đầu tiên có kích hoạt quá nhỏ và tăng trọng số của lớp đó, ta có thể đạt được một mạng có kích hoạt ban đầu hợp lý trên toàn bộ các lớp.

Nếu quá trình học vẫn diễn ra quá chậm, nên xem xét phạm vi hoặc độ lệch chuẩn của các gradient bên cạnh các kích hoạt.

Thủ tục này có thể tự động hóa và thường ít tốn kém hơn tối ưu hóa siêu tham số dựa trên lỗi tập xác thực, bởi vì nó chỉ cần phản hồi từ hành vi của mô hình trên một minibatch duy nhất.

Mishkin và Matas (2015) đã xác định và nghiên cứu quy trình này một cách chính thức.

Cách thiết lập bias cần phối hợp với cách khởi tạo trọng số. Việc đặt bias bằng 0 phù hợp với hầu hết các phương pháp khởi tạo trọng số. Tuy nhiên, có một số trường hợp cần đặt bias khác 0:

- Bias cho đơn vị đầu ra: Khởi tạo bias sao cho phù hợp với thống kê biên của đầu ra. Giả sử trọng số ban đầu đủ nhỏ, ta giải phương trình:

$$\text{softmax}(b) = c$$

  trong đó c là vector biểu diễn xác suất biên của các lớp.

- Tránh saturation khi khởi tạo: Đặt bias của ReLU hidden unit thành 0.1 thay vì 0 để tránh saturation sớm. Cách này không thích hợp cho các phương pháp như random walk initialization (Sussillo, 2014).

- Bias cho các cổng điều khiển: Trong các kiến trúc như LSTM, đơn vị điều khiển đầu ra (như cổng forget) cần khởi tạo bias để giá trị ban đầu của cổng gần 1. Ví dụ, Jozefowicz et al. (2015) khuyến nghị đặt bias của forget gate bằng 1.

Ví dụ, trong mô hình hồi quy tuyến tính với ước lượng phương sai điều kiện:

$$p(y \mid x) = \mathcal{N}(y \mid w^T x + b, 1/\beta)$$

trong đó β là tham số độ chính xác.

Thông thường, có thể khởi tạo β bằng 1 một cách an toàn. Ngoài ra, có thể:

- Giả định trọng số ban đầu đủ nhỏ để bỏ qua ảnh hưởng của trọng số.
- Đặt bias để sinh trung bình biên đúng của đầu ra.
- Đặt tham số phương sai bằng phương sai biên của đầu ra trên tập huấn luyện.

Ngoài các phương pháp ngẫu nhiên hoặc hằng số, có thể sử dụng học máy để khởi tạo:

- Sử dụng tham số học được từ mô hình không giám sát trên cùng tập dữ liệu.
- Huấn luyện có giám sát trên một nhiệm vụ liên quan.
- Thậm chí, huấn luyện trên một nhiệm vụ không liên quan cũng có thể mang lại khởi tạo giúp mô hình hội tụ nhanh hơn khởi tạo ngẫu nhiên.

Một số chiến lược khởi tạo giúp hội tụ nhanh hơn và tổng quát hóa tốt hơn vì mã hóa

thông tin phân phối dữ liệu vào tham số mô hình. Một số khác chỉ đơn giản vì thiết lập đúng tỉ lệ ban đầu hoặc phân biệt chức năng giữa các đơn vị.

8.5 Các thuật toán với tốc độ học thích ứng

Các nhà nghiên cứu mạng nơ-ron từ lâu đã nhận ra rằng tốc độ học (learning rate) là một trong những siêu tham số khó thiết lập nhất, bởi vì nó ảnh hưởng đáng kể đến hiệu suất của mô hình.

Như đã thảo luận trong các mục 4.3 và 8.2, hàm chi phí thường rất nhạy theo một số hướng trong không gian tham số và không nhạy theo những hướng khác.

Thuật toán momentum có thể giúp giảm bớt các vấn đề này phần nào, nhưng đổi lại nó lại giới thiệu thêm một siêu tham số khác.

Trong bối cảnh đó, một câu hỏi tự nhiên được đặt ra: liệu có một cách tiếp cận khác?

Nếu tin rằng các hướng nhạy cảm phần nào thẳng hàng với các trục tọa độ, thì việc sử dụng một tốc độ học riêng cho từng tham số, và tự động điều chỉnh các tốc độ học này trong quá trình huấn luyện, là một ý tưởng hợp lý.

Thuật toán delta-bar-delta (Jacobs, 1988) là một cách tiếp cận kinh nghiệm từ rất sớm để điều chỉnh tốc độ học riêng cho từng tham số mô hình trong quá trình huấn luyện.

Ý tưởng cơ bản của thuật toán:

- Nếu đạo hàm riêng của hàm mất mát theo một tham số giữ nguyên dấu, thì tăng tốc độ học.
- Nếu đạo hàm riêng đó đổi dấu, thì giảm tốc độ học.

Tuy nhiên, quy tắc này chỉ áp dụng được cho tối ưu hóa theo batch đầy đủ.

Gần đây, nhiều phương pháp incremental hoặc mini-batch đã được giới thiệu nhằm thích ứng tốc độ học cho từng tham số trong suốt quá trình học.

Trong phần này, chúng ta sẽ điểm qua một số thuật toán tiêu biểu.

8.5.1 AdaGrad

Thuật toán AdaGrad (trình bày trong Thuật toán 8.4) điều chỉnh tốc độ học riêng cho từng tham số mô hình bằng cách nhân chúng với nghịch đảo của căn bậc hai tổng lũy tích các giá trị bình phương gradient trong suốt quá trình huấn luyện **duchi2011adagrad**.

Các tham số có đạo hàm riêng lớn của hàm mất mát sẽ nhanh chóng bị giảm tốc độ học, trong khi các tham số với đạo hàm riêng nhỏ sẽ có tốc độ học giảm chậm hơn.

Hiệu ứng tổng thể là mô hình tiến bộ nhiều hơn theo các hướng có độ dốc thoải trong không gian tham số.

Trong bối cảnh tối ưu hóa lồi, thuật toán AdaGrad có những tính chất lý thuyết rất hấp dẫn.

Tuy nhiên, trong thực tế huấn luyện mạng nơ-ron sâu, việc tích lũy các bình phương

gradient ngay từ đầu có thể khiến tốc độ học giảm quá nhanh và quá nhiều, dẫn đến quá trình tối ưu bị chậm lại.

AdaGrad hoạt động tốt đối với một số mô hình deep learning, nhưng không phù hợp với tất cả các trường hợp.

8.5.2 RMSProp

Trong quá trình huấn luyện, quỹ đạo học có thể đi qua nhiều cấu trúc khác nhau và cuối cùng đến một vùng có dạng bát lồi (locally convex bowl). Thuật toán AdaGrad sẽ thu nhỏ tốc độ học theo toàn bộ lịch sử bình phương gradient và có thể làm cho tốc độ học quá nhỏ trước khi đi vào một cấu trúc lồi như vậy. Trong khi đó, RMSProp sử dụng một trung bình giảm dần theo thời gian để loại bỏ các giá trị lịch sử từ quá khứ xa xôi, giúp thuật toán có thể hội tụ nhanh chóng sau khi tìm thấy một bát lồi, như thể nó là một phiên bản của thuật toán AdaGrad được khởi tạo trong vùng bát lồi đó.

Algorithm 8.4 Thuật toán AdaGrad

Require: Tốc độ học toàn cục ϵ
Require: Tham số khởi tạo θ
Require: Hằng số nhỏ δ, ví dụ 10^{-7}, để đảm bảo ổn định số học
Khởi tạo biến tích lũy gradient: $r \leftarrow 0$
while chưa đạt điều kiện dừng **do**
    Lấy một minibatch gồm m mẫu từ tập huấn luyện $\{x^{(1)}, \ldots, x^{(m)}\}$ với nhãn tương ứng $y^{(i)}$
    Tính gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    Cộng dồn bình phương gradient: $r \leftarrow r + g \odot g$
    Tính bước cập nhật: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$ ▷ Chia và căn bậc hai từng phần tử
    Cập nhật tham số: $\theta \leftarrow \theta + \Delta\theta$
end while

Algorithm 8.5 Thuật toán RMSProp

Require: Tốc độ học toàn cục ϵ, hệ số suy giảm ρ
Require: Tham số khởi tạo θ
Require: Hằng số nhỏ δ, thường là 10^{-6}, để tránh chia cho số gần 0
Khởi tạo biến tích lũy gradient: $r \leftarrow 0$
while chưa đạt điều kiện dừng **do**
    Lấy một minibatch gồm m mẫu từ tập huấn luyện $\{x^{(1)}, \ldots, x^{(m)}\}$ với nhãn tương ứng $y^{(i)}$
    Tính gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    Cập nhật giá trị tích lũy: $r \leftarrow \rho r + (1 - \rho) g \odot g$
    Tính bước cập nhật: $\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$
    Cập nhật tham số: $\theta \leftarrow \theta + \Delta\theta$
end while

Thuật toán RMSProp được thể hiện ở dạng chuẩn trong Thuật toán 8.5 và kết hợp với momentum Nesterov trong Thuật toán 8.6. So với AdaGrad, việc sử dụng trung bình di động mang đến một siêu tham số mới là ρ, điều khiển độ dài của trung bình di động.

Về mặt thực nghiệm, RMSProp đã được chứng minh là một thuật toán tối ưu hóa hiệu quả và thực tế cho các mạng nơ-ron sâu. Hiện tại, nó là một trong những phương pháp tối ưu được sử dụng rộng rãi trong cộng đồng học sâu.

8.5.3 Thuật toán Adam

Adam (Kingma và Ba, 2014) là một thuật toán tối ưu với tốc độ học thích nghi, kết hợp giữa hai ý tưởng chính: momentum (động lượng) và RMSProp (chia tỷ lệ theo phương sai). Tên gọi "Adam"bắt nguồn từ cụm từ *adaptive moments* – mô hình sử dụng ước lượng bậc nhất và bậc hai của gradient.

- Momentum trong Adam được tính như một trung bình luỹ thừa bậc nhất (first-order moment) của gradient, trong khi RMSProp theo dõi trung bình luỹ thừa bậc hai (second-order moment).
- Adam áp dụng các hệ số hiệu chỉnh sai lệch (bias correction) để điều chỉnh các ước lượng mô-men do ban đầu được khởi tạo tại 0, điều mà RMSProp không làm. Nhờ vậy, Adam khắc phục được độ lệch cao trong giai đoạn đầu huấn luyện.
- Khác với việc áp dụng momentum lên gradient đã được chia tỷ lệ (như cách đơn giản kết hợp RMSProp và momentum), Adam tích hợp trực tiếp momentum và chia tỷ lệ vào quá trình cập nhật, giúp ổn định và hiệu quả hơn.
- Adam thường được đánh giá là khá ổn định với các siêu tham số, tuy nhiên tốc độ học (*learning rate*) đôi khi vẫn cần điều chỉnh so với giá trị mặc định để đạt kết quả tối ưu.

8.5.4 Lựa chọn thuật toán tối ưu phù hợp

Trong các phần trước, chúng ta đã thảo luận nhiều thuật toán tối ưu có khả năng điều chỉnh tốc độ học (learning rate) cho từng tham số riêng biệt. Một câu hỏi tự nhiên đặt ra là: *nên chọn thuật toán nào để huấn luyện mô hình deep learning?*

Thật không may, hiện tại vẫn chưa có một sự đồng thuận rõ ràng nào. Theo nghiên cứu của Schaul et al. (2014), các thuật toán thuộc nhóm learning rate thích nghi như RMSProp và AdaDelta cho kết quả khá ổn định trên nhiều bài toán học khác nhau. Tuy vậy, không có thuật toán nào vượt trội hoàn toàn trong mọi tình huống.

Hiện nay, các thuật toán tối ưu phổ biến nhất gồm:

- SGD (Gradient Descent thuần tuý),
- SGD với Momentum,
- RMSProp,
- RMSProp với Momentum,
- AdaDelta,
- Adam.

Việc lựa chọn thường phụ thuộc vào kinh nghiệm của người dùng với thuật toán cụ thể, bởi điều này ảnh hưởng đến khả năng tinh chỉnh siêu tham số hiệu quả. Một số thuật toán như Adam có xu hướng làm việc tốt với ít điều chỉnh, trong khi SGD có thể cần nhiều thử nghiệm hơn để đạt kết quả tối ưu.

8.6 Phương pháp xấp xỉ bậc hai (Approximate Second-Order Methods)

Trong phần này, chúng ta sẽ tìm hiểu cách áp dụng các phương pháp tối ưu bậc hai để huấn luyện mạng nơ-ron sâu. Bạn đọc có thể tham khảo thêm cách trình bày ban đầu của chủ đề này trong LeCun et al. (1998a).

Để đơn giản hóa, ta chỉ xét hàm mục tiêu là rủi ro thực nghiệm (empirical risk), được định nghĩa như sau:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}(x,y)}\left[L(f(x;\theta),y)\right] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)}) \tag{8.17}$$

trong đó:

- θ là vector tham số của mô hình.
- $f(x;\theta)$ là đầu ra của mô hình với đầu vào x.
- L là hàm mất mát đo độ sai lệch giữa đầu ra mô hình và nhãn thực y.
- m là số mẫu huấn luyện.

Mặc dù chúng ta chỉ trình bày với hàm mục tiêu đơn giản này, các phương pháp tối ưu bậc hai có thể được mở rộng dễ dàng cho các hàm mục tiêu tổng quát hơn, ví dụ như có thêm các thuật toán chính quy hoá tham số đã được thảo luận trong chương 7.

8.7 Phương pháp xấp xỉ bậc hai (Approximate Second-Order Methods)

Trong phần này, chúng ta sẽ tìm hiểu cách áp dụng các phương pháp tối ưu bậc hai để huấn luyện mạng nơ-ron sâu. Bạn đọc có thể tham khảo thêm cách trình bày ban đầu của chủ đề này trong LeCun et al. (1998a).

Để đơn giản hóa, ta chỉ xét hàm mục tiêu là rủi ro thực nghiệm (empirical risk), được định nghĩa như sau:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}(x,y)}\left[L(f(x;\theta),y)\right] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)}) \tag{8.18}$$

trong đó:

- θ là vector tham số của mô hình.
- $f(x;\theta)$ là đầu ra của mô hình với đầu vào x.
- L là hàm mất mát đo độ sai lệch giữa đầu ra mô hình và nhãn thực y.
- m là số mẫu huấn luyện.

Mặc dù chúng ta chỉ trình bày với hàm mục tiêu đơn giản này, các phương pháp tối ưu bậc hai có thể được mở rộng dễ dàng cho các hàm mục tiêu tổng quát hơn, ví dụ như có thêm các thuật toán chính quy hoá tham số đã được thảo luận trong chương 7.

8.7.1 Phương pháp Newton (Newton's Method)

Trong mục 4.3, chúng ta đã giới thiệu các phương pháp tối ưu dựa trên đạo hàm bậc hai. Khác với các phương pháp bậc nhất chỉ sử dụng đạo hàm cấp một (gradient), phương pháp bậc hai tận dụng thêm thông tin từ đạo hàm bậc hai để cải thiện quá trình tối ưu.

Một trong những phương pháp phổ biến nhất là phương pháp Newton. Phương pháp này dựa trên khai triển chuỗi Taylor bậc hai để xấp xỉ hàm mất mát $J(\theta)$ gần một điểm θ_0, bỏ qua các thành phần bậc cao hơn:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H (\theta - \theta_0), \tag{8.19}$$

trong đó:

- $\nabla_\theta J(\theta_0)$ là đạo hàm bậc nhất tại θ_0,
- H là ma trận Hessian — ma trận đạo hàm bậc hai của J tại θ_0.

Để tìm cực trị (minimum), ta lấy đạo hàm của biểu thức trên theo θ và giải phương trình $\nabla_\theta J(\theta) = 0$. Từ đó, ta thu được công thức cập nhật tham số:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0). \tag{8.20}$$

Nếu $J(\theta)$ là hàm lồi và gần như bậc hai (local quadratic), thì phương pháp Newton sẽ nhảy trực tiếp đến điểm cực tiểu bằng cách điều chỉnh gradient với ma trận Hessian.

Trong trường hợp hàm mục tiêu không thuần bậc hai (có các thành phần bậc cao hơn), ta có thể lặp lại bước cập nhật nhiều lần. Khi đó, ta có được thuật toán huấn luyện gọi là Newton lặp (Iterative Newton's Method), như mô tả trong Thuật toán 8.8.

Đối với các bề mặt hàm mục tiêu không thuần bậc hai, phương pháp Newton vẫn có thể được áp dụng lặp lại, miễn là ma trận Hessian vẫn là xác định dương (*positive definite*). Điều này dẫn đến một thủ tục lặp gồm hai bước:

1. Cập nhật hoặc tính toán lại ma trận Hessian (hoặc nghịch đảo của nó),
2. Áp dụng công thức cập nhật theo phương trình Newton:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0).$$

Tuy nhiên, như đã nêu trong mục 8.2.3, phương pháp Newton chỉ phù hợp khi Hessian là xác định dương. Trong học sâu, hàm mất mát thường là phi lồi (nonconvex), chứa nhiều đặc trưng phức tạp như *điểm yên ngựa* (saddle points), khiến phương pháp Newton dễ gặp

lỗi.

Khi Hessian có giá trị riêng âm (không xác định dương), Newton's method có thể cập nhật tham số theo hướng sai. Để khắc phục, ta thường regular hóa Hessian bằng cách cộng thêm một hằng số α vào đường chéo, tức là sử dụng:

$$\theta^* = \theta_0 - [H(f(\theta_0)) + \alpha I]^{-1} \nabla_\theta f(\theta_0), \tag{8.21}$$

trong đó $\alpha > 0$ và I là ma trận đơn vị.

Kỹ thuật này được dùng trong các biến thể gần đúng của phương pháp Newton, ví dụ như thuật toán Levenberg-Marquardt, và hoạt động tốt nếu các giá trị riêng âm của Hessian gần bằng 0. Tuy nhiên, nếu có phương hướng với độ cong âm mạnh (strong negative curvature), thì α phải đủ lớn để loại bỏ ảnh hưởng của các giá trị âm này. Khi đó, biểu thức cập nhật sẽ gần giống như gradient chuẩn chia cho α:

$$\theta^* \approx \theta_0 - \frac{1}{\alpha} \nabla_\theta f(\theta_0),$$

nghĩa là gần giống với gradient descent thông thường.

Bên cạnh vấn đề lý thuyết, phương pháp Newton còn bị hạn chế nghiêm trọng về mặt tính toán. Cụ thể:

- Ma trận Hessian có kích thước $k \times k$, với k là số tham số của mô hình.
- Với mạng nơ-ron hiện đại, k có thể lên đến hàng triệu, khiến việc lưu trữ và nghịch đảo Hessian trở nên phi thực tế.
- Việc tính toán nghịch đảo Hessian có độ phức tạp $O(k^3)$ và phải thực hiện sau mỗi lần cập nhật tham số.

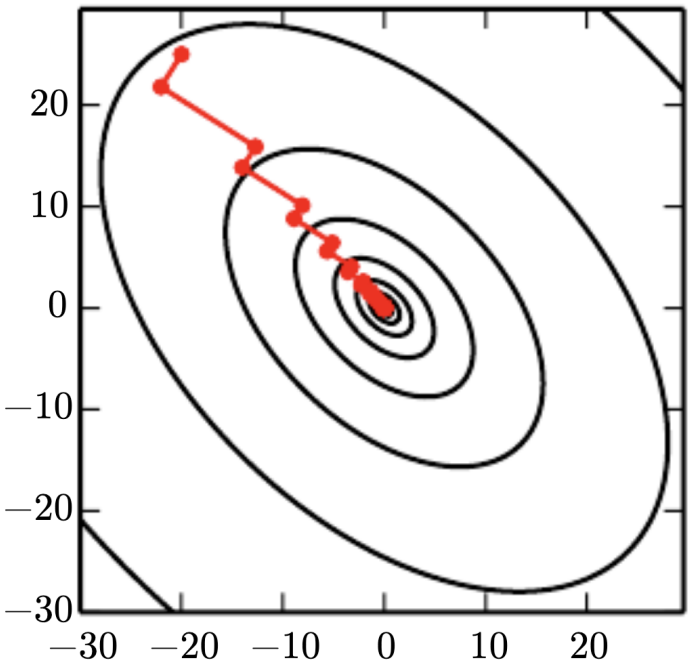
Do đó, phương pháp Newton chỉ khả thi đối với các mạng rất nhỏ. Trong các phần tiếp theo, chúng ta sẽ tìm hiểu các phương pháp xấp xỉ, tận dụng ưu điểm của Newton's method nhưng tránh được những khó khăn về tính toán.

Phương pháp Gradient Conjugate là một phương pháp tối ưu hóa giúp tránh tính toán ma trận nghịch đảo Hessian bằng cách giảm dần theo các hướng conjugate. Ý tưởng cơ bản của phương pháp này xuất phát từ việc nghiên cứu kỹ lưỡng sự yếu kém của phương pháp Gradient Steepest Descent (xem mục 4.3 để biết chi tiết), trong đó các bước tìm kiếm đường đi (line search) được thực hiện theo hướng gradient. Hình 8.6 minh họa cách phương pháp Gradient Steepest Descent, khi áp dụng trong một chảo hình parabol, tiến triển theo một mẫu zig-zag không hiệu quả. Điều này xảy ra vì mỗi hướng tìm kiếm trong phương pháp này, khi được chỉ định bởi gradient, luôn luôn vuông góc với hướng tìm kiếm trước đó.

Giả sử hướng tìm kiếm trước đó là d_{t-1}. Tại điểm cực tiểu, nơi việc tìm kiếm dừng lại, đạo hàm theo hướng này là 0, tức là:

$$\nabla_\theta J(\theta) \cdot d_{t-1} = 0$$

Vì gradient tại điểm này xác định hướng tìm kiếm hiện tại, $d_t = \nabla_\theta J(\theta)$ sẽ không có đóng góp trong hướng của d_{t-1}. Do đó, d_t là một vector vuông góc với d_{t-1}. Mối quan hệ này giữa d_{t-1} và d_t được minh họa trong Hình 8.6 cho nhiều lần lặp của phương pháp Gradient Steepest Descent. Như đã thấy trong hình, việc chọn các hướng giảm gradient vuông góc không giữ được tối thiểu dọc theo các hướng tìm kiếm trước đó. Điều này dẫn đến mẫu zig-zag trong quá trình tối ưu hóa, nơi mà sau khi giảm xuống điểm cực tiểu theo hướng gradient hiện tại, chúng ta phải giảm lại hàm mục tiêu theo hướng gradient trước đó. Do đó, bằng cách theo gradient sau mỗi bước tìm kiếm, chúng ta thực tế là đang "làm lại"tiến trình đã thực hiện theo hướng tìm kiếm trước đó.



Hình 8.6: Phương pháp độ dốc lớn nhất được áp dụng cho bề mặt hàm chi phí bậc hai. Phương pháp này bao gồm việc nhảy đến điểm có chi phí thấp nhất trên đường thẳng được xác định bởi gradient tại điểm hiện tại trong mỗi bước lặp. Cách tiếp cận này giúp khắc phục một số vấn đề gặp phải khi dùng tốc độ học cố định như trong Hình 4.6. Tuy nhiên, ngay cả khi sử dụng bước nhảy tối ưu, thuật toán vẫn tiến tới điểm cực tiểu theo kiểu qua lại (zigzag). Theo định nghĩa, tại điểm cực tiểu của hàm mục tiêu theo một hướng nhất định, gradient tại điểm cuối sẽ vuông góc với hướng đó.

Phương pháp Gradient Conjugate nhằm giải quyết vấn đề này. Trong phương pháp này, chúng ta tìm một hướng tìm kiếm sao cho nó **conjugate** với hướng tìm kiếm trước đó; tức là, nó không "xóa bỏ"sự tiến bộ đã đạt được theo hướng đó. Ở vòng lặp huấn luyện thứ t, hướng tìm kiếm tiếp theo d_t có dạng:

$$d_t = \nabla_\theta J(\theta) + \beta_t d_{t-1}$$

Trong đó, β_t là một hệ số điều chỉnh kiểm soát mức độ mà chúng ta cần thêm lại hướng d_{t-1} vào hướng tìm kiếm hiện tại. Hai hướng d_t và d_{t-1} được gọi là conjugate nếu:

$$d_t^T H d_{t-1} = 0$$

Trong đó H là ma trận Hessian.

Cách tính trực tiếp các hướng conjugate này sẽ liên quan đến việc tính toán các vector riêng của ma trận Hessian để chọn giá trị β_t, nhưng điều này không phù hợp với mục tiêu của chúng ta là phát triển một phương pháp tính toán hiệu quả hơn so với phương pháp Newton trong các bài toán lớn. Vậy liệu chúng ta có thể tính các hướng conjugate mà không cần tính toán này không? Câu trả lời là có, và có hai phương pháp phổ biến để tính β_t:

- **Fletcher-Reeves**:
$$\beta_t = \frac{\nabla_\theta J(\theta_t)^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

- **Polak-Ribière**:
$$\beta_t = \frac{(\nabla_\theta J(\theta_t) - \nabla_\theta J(\theta_{t-1}))^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

Đối với một bề mặt bậc hai, các hướng conjugate đảm bảo rằng gradient theo hướng trước không tăng lên, từ đó giúp giữ được điểm cực tiểu theo các hướng trước đó. Do đó, trong không gian tham số có k chiều, phương pháp conjugate gradient yêu cầu tối đa k bước tìm kiếm để đạt được cực tiểu.

Phương pháp Gradient Conjugate có thể được áp dụng trong các bài toán tối ưu hóa không chỉ với các hàm mục tiêu bậc hai mà còn với các bài toán phi tuyến tính. Trong bối cảnh huấn luyện mạng nơ-ron, hàm mục tiêu có thể không phải là một hàm bậc hai. Mặc dù vậy, phương pháp conjugate gradient vẫn có thể áp dụng, nhưng cần có một số điều chỉnh. Khi không có đảm bảo rằng hàm mục tiêu là bậc hai, các hướng conjugate không còn đảm bảo sẽ giữ cực tiểu theo hướng trước đó. Vì vậy, trong thuật toán conjugate gradient phi tuyến, đôi khi cần phải "khởi động lại"phương pháp bằng cách thực hiện một tìm kiếm theo hướng gradient chưa thay đổi.

Các nhà thực hành đã báo cáo kết quả khả quan khi áp dụng phương pháp conjugate gradient phi tuyến cho việc huấn luyện mạng nơ-ron, mặc dù thường thì phương pháp này sẽ được kết hợp với một số bước của phương pháp gradient ngẫu nhiên (stochastic gradient descent) trước khi bắt đầu sử dụng conjugate gradient phi tuyến. Ngoài ra, mặc dù phương pháp conjugate gradient (phi tuyến) truyền thống được thực hiện theo kiểu batch, nhưng các phiên bản mini-batch cũng đã được sử dụng thành công trong huấn luyện mạng nơ-ron (Le et al., 2011).

Một số biến thể của phương pháp conjugate gradient dành riêng cho mạng nơ-ron cũng

đã được đề xuất, chẳng hạn như thuật toán *scaled conjugate gradients* (Moller, 1993).

8.7.2 Thuật toán BFGS

Thuật toán Broyden–Fletcher–Goldfarb–Shanno (BFGS) cố gắng kết hợp những ưu điểm của phương pháp Newton mà không gặp phải gánh nặng tính toán. Theo cách này, BFGS tương tự như phương pháp gradient liên hợp, nhưng BFGS áp dụng một cách tiếp cận trực tiếp hơn trong việc xấp xỉ cập nhật của Newton.

Nhớ lại rằng cập nhật Newton được đưa ra bởi:

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0), \tag{8.32}$$

với H là ma trận Hessian của J theo θ tại θ_0. Khó khăn tính toán chính trong việc áp dụng cập nhật Newton là việc tính toán nghịch đảo của ma trận Hessian H^{-1}. Phương pháp được áp dụng trong các thuật toán quasi-Newton (trong đó BFGS là thuật toán nổi bật nhất) là xấp xỉ nghịch đảo Hessian bằng một ma trận M_t, ma trận này được cập nhật qua các bước cập nhật bậc thấp để trở thành xấp xỉ tốt hơn cho H^{-1}.

Cách xấp xỉ và phát triển của BFGS được trình bày trong nhiều sách giáo khoa về tối ưu hóa, bao gồm Luenberger (1984). Sau khi xấp xỉ nghịch đảo Hessian M_t được cập nhật, hướng giảm ρ_t được xác định bởi:

$$\rho_t = M_t g_t.$$

Một phép tìm kiếm theo hướng này được thực hiện để xác định kích thước bước ϵ^* được thực hiện theo hướng đó. Cập nhật cuối cùng cho các tham số là:

$$\theta_{t+1} = \theta_t + \epsilon^* \rho_t. \tag{8.33}$$

Giống như phương pháp gradient liên hợp, thuật toán BFGS lặp đi lặp lại một loạt các tìm kiếm theo hướng với thông tin bậc hai. Tuy nhiên, khác với gradient liên hợp, thành công của phương pháp này không phụ thuộc quá nhiều vào việc tìm kiếm theo hướng có thể đi đến gần điểm tối thiểu thực tế. Do đó, so với phương pháp gradient liên hợp, BFGS có ưu điểm là có thể dành ít thời gian hơn trong việc tinh chỉnh mỗi bước tìm kiếm.

Tuy nhiên, một nhược điểm của thuật toán BFGS là nó phải lưu trữ ma trận nghịch đảo Hessian M, điều này yêu cầu bộ nhớ $O(n^2)$, khiến BFGS không thực tế đối với hầu hết các mô hình học sâu hiện đại, thường có hàng triệu tham số.

Chi phí bộ nhớ của thuật toán BFGS có thể giảm đáng kể bằng cách tránh lưu trữ hoàn toàn ma trận xấp xỉ nghịch đảo Hessian M. Thuật toán L-BFGS tính toán xấp xỉ M sử dụng cùng phương pháp với thuật toán BFGS nhưng giả định rằng M_{t-1} là ma trận đơn vị, thay vì lưu trữ xấp xỉ từ bước này sang bước tiếp theo.

Nếu được sử dụng với tìm kiếm chính xác theo hướng, các hướng xác định bởi L-BFGS là liên hợp với nhau. Tuy nhiên, khác với phương pháp gradient liên hợp, quy trình này

vẫn hoạt động tốt ngay cả khi tối thiểu của phép tìm kiếm theo hướng chỉ được tiếp cận một cách xấp xỉ. Chiến lược L-BFGS này, không lưu trữ, có thể được tổng quát để bao gồm nhiều thông tin hơn về ma trận Hessian bằng cách lưu trữ một số vector được sử dụng để cập nhật M tại mỗi bước thời gian, điều này chỉ tốn $O(n)$ bộ nhớ mỗi bước.

8.8 Các Chiến Lược Tối Ưu Hóa và Meta-Thuật Toán

Nhiều kỹ thuật tối ưu hóa không phải là các thuật toán cụ thể mà là các mẫu tổng quát có thể được chuyên biệt hóa để tạo ra các thuật toán, hoặc các tiểu chương trình có thể được tích hợp vào nhiều thuật toán khác nhau.

8.8.1 8.7.1 Chuẩn Hóa Theo Lô (Batch Normalization)

Chuẩn hóa theo lô (Batch Normalization) (Ioffe và Szegedy, 2015) là một trong những đổi mới thú vị gần đây trong việc tối ưu hóa các mạng nơ-ron sâu, và thực tế, nó không phải là một thuật toán tối ưu hóa. Thay vào đó, nó là một phương pháp tái tham số hóa thích ứng, được thúc đẩy bởi sự khó khăn trong việc huấn luyện các mô hình rất sâu.

Các mô hình rất sâu bao gồm việc kết hợp nhiều hàm hoặc lớp. Đạo hàm cho biết cách cập nhật mỗi tham số, giả định rằng các lớp khác không thay đổi. Trong thực tế, chúng ta cập nhật tất cả các lớp cùng một lúc. Khi thực hiện cập nhật, kết quả không mong muốn có thể xảy ra vì nhiều hàm được kết hợp với nhau và thay đổi đồng thời, sử dụng các cập nhật được tính toán dưới giả định rằng các hàm khác vẫn không thay đổi.

Lấy một ví dụ đơn giản, giả sử chúng ta có một mạng nơ-ron sâu với chỉ một đơn vị trên mỗi lớp và không sử dụng hàm kích hoạt tại mỗi lớp ẩn:

$$\hat{y} = xw_1w_2w_3 \dots w_l.$$

Ở đây, w_i là trọng số được sử dụng bởi lớp i. Đầu ra của lớp i là $h_i = h_{i-1}w_i$. Đầu ra $\hat{y}$ là một hàm tuyến tính của đầu vào x nhưng là một hàm phi tuyến tính của các trọng số w_i. Giả sử hàm chi phí đã đưa ra một gradient là 1 đối với $\hat{y}$, vì vậy chúng ta muốn giảm $\hat{y}$ một chút. Thuật toán lan truyền ngược sau đó có thể tính toán một gradient $g = \nabla_w \hat{y}$.

Xem xét những gì xảy ra khi chúng ta thực hiện một cập nhật $w \leftarrow w - \epsilon g$. Phương trình chuỗi Taylor bậc nhất của $\hat{y}$ dự đoán rằng giá trị của $\hat{y}$ sẽ giảm theo $\epsilon g^\top g$. Nếu chúng ta muốn giảm $\hat{y}$ đi 0.1, thông tin bậc nhất có sẵn trong gradient cho thấy ta có thể đặt tỷ lệ học là $\epsilon = 0.1g^\top g$. Tuy nhiên, cập nhật thực tế sẽ bao gồm các hiệu ứng bậc hai và bậc ba, kéo dài đến các hiệu ứng bậc l. Giá trị mới của $\hat{y}$ được cho bởi:

$$\hat{y} = x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l).$$

Một ví dụ về một hạng tử bậc hai phát sinh từ cập nhật này là $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$. Hạng tử này có thể không đáng kể nếu $\prod_{i=3}^{l} w_i$ nhỏ, hoặc có thể lớn theo hàm mũ nếu các trọng số trên các lớp 3 đến l lớn hơn 1. Điều này khiến việc chọn tỷ lệ học phù hợp rất khó khăn, vì các hiệu ứng của việc cập nhật các tham số của một lớp phụ thuộc mạnh mẽ vào tất cả

các lớp khác. Các thuật toán tối ưu bậc hai giải quyết vấn đề này bằng cách tính toán một cập nhật mà tính đến các tương tác bậc hai này, nhưng chúng ta có thể thấy rằng trong các mạng rất sâu, ngay cả các tương tác bậc cao hơn cũng có thể có ý nghĩa. Ngay cả các thuật toán tối ưu bậc hai cũng đắt đỏ và thường yêu cầu rất nhiều xấp xỉ khiến chúng không thể tính toán tất cả các tương tác bậc hai quan trọng. Việc xây dựng một thuật toán tối ưu bậc n với $n > 2$ do đó có vẻ như là không thể. Vậy chúng ta có thể làm gì thay thế?

Chuẩn hóa theo lô cung cấp một cách tinh tế để tái tham số hóa gần như bất kỳ mạng sâu nào. Phương pháp tái tham số hóa này giảm đáng kể vấn đề điều phối các cập nhật giữa nhiều lớp. Chuẩn hóa theo lô có thể được áp dụng cho bất kỳ lớp đầu vào hoặc ẩn nào trong một mạng. Gọi H là một minibatch các kích hoạt của lớp cần chuẩn hóa, được sắp xếp thành một ma trận thiết kế, với các kích hoạt cho mỗi ví dụ xuất hiện ở một hàng của ma trận. Để chuẩn hóa H, chúng ta thay thế nó bằng:

$$H' = \frac{H - \mu}{\sigma},$$

trong đó μ là một vector chứa trung bình của mỗi đơn vị và σ là một vector chứa độ lệch chuẩn của mỗi đơn vị. Phép toán này áp dụng broadcasting vector μ và σ vào mỗi hàng của ma trận H. Trong mỗi hàng, phép toán được thực hiện theo từng phần tử, do đó, $H_{i,j}$ sẽ được chuẩn hóa bằng cách trừ μ_j và chia cho σ_j. Các phần còn lại của mạng sẽ hoạt động trên H' theo cách giống như trước khi sử dụng H.

Trong quá trình huấn luyện, ta tính toán:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} H_{i,:},$$

và

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (H_{i,:} - \mu)^2 + \delta,$$

trong đó δ là một giá trị nhỏ dương như 10^{-8}, được áp dụng để tránh gặp phải gradient không xác định khi $\sqrt{z}$ tại $z = 0$. Quan trọng là ta sẽ thực hiện lan truyền ngược (backpropagation) qua các phép toán này để tính toán trung bình và độ lệch chuẩn, và áp dụng chúng để chuẩn hóa H. Điều này có nghĩa là gradient sẽ không bao giờ đề xuất một phép toán làm tăng độ lệch chuẩn hoặc trung bình của h_i; các phép toán chuẩn hóa sẽ loại bỏ tác động của những hành động như vậy và làm mất thành phần của nó trong gradient. Đây là một sáng chế quan trọng trong phương pháp batch normalization. Các phương pháp trước đây đã bao gồm việc thêm hình phạt vào hàm chi phí để khuyến khích các đơn vị có thống kê hoạt hóa chuẩn hóa hoặc can thiệp để chuẩn hóa lại thống kê của các đơn vị sau mỗi bước gradient descent. Phương pháp trước thường dẫn đến việc chuẩn hóa không hoàn hảo, trong khi phương pháp sau thường dẫn đến việc tốn thời gian vì thuật toán học liên tục đề xuất thay đổi trung bình và phương sai, và bước chuẩn hóa lại vô hiệu

hóa sự thay đổi này. Batch normalization tái tham số hóa mô hình để một số đơn vị luôn được chuẩn hóa theo định nghĩa, giúp tránh được cả hai vấn đề trên một cách khéo léo.

Tại thời gian kiểm tra, μ và σ có thể được thay thế bằng các giá trị trung bình và độ lệch chuẩn chạy được tính toán trong suốt quá trình huấn luyện. Điều này cho phép mô hình được đánh giá trên một ví dụ đơn lẻ mà không cần sử dụng các định nghĩa của μ và σ phụ thuộc vào một minibatch toàn bộ.

Lấy ví dụ về $\hat{y} = xw_1w_2 \dots w_l$, ta nhận thấy rằng chúng ta có thể giải quyết phần lớn những khó khăn trong việc học mô hình này bằng cách chuẩn hóa h_{l-1}. Giả sử rằng x được lấy từ một phân phối Gaussian chuẩn. Khi đó, h_{l-1} cũng sẽ đến từ một phân phối Gaussian, vì phép biến đổi từ x sang h_l là tuyến tính. Tuy nhiên, h_{l-1} sẽ không còn có trung bình bằng 0 và phương sai bằng 1. Sau khi áp dụng batch normalization, ta thu được $\hat{h}_{l-1}$ chuẩn hóa, phục hồi lại các thuộc tính trung bình bằng 0 và phương sai bằng 1. Hầu như mọi cập nhật đối với các lớp phía dưới sẽ không thay đổi phân phối của $\hat{h}_{l-1}$ vì nó luôn duy trì là một Gaussian chuẩn. Lúc này, đầu ra $\hat{y}$ có thể được học như một hàm tuyến tính đơn giản $\hat{y} = w_l\hat{h}_{l-1}$. Việc học trong mô hình này trở nên rất đơn giản vì các tham số ở các lớp dưới không còn ảnh hưởng trong hầu hết các trường hợp; đầu ra của chúng luôn được chuẩn hóa lại thành một Gaussian chuẩn. Trong một số trường hợp đặc biệt, các lớp dưới có thể có tác động. Việc thay đổi một trọng số của lớp dưới về 0 có thể khiến đầu ra trở nên suy thoái, và thay đổi dấu của một trọng số lớp dưới có thể làm đảo ngược mối quan hệ giữa $\hat{h}_{l-1}$ và y. Tuy nhiên, những tình huống này rất hiếm gặp.

Nếu không có chuẩn hóa, gần như mọi cập nhật sẽ có tác động cực kỳ mạnh mẽ đến các thống kê của h_{l-1}. Batch normalization đã làm cho việc học mô hình này dễ dàng hơn rất nhiều. Tuy nhiên, trong ví dụ này, sự dễ dàng trong việc học cũng đến với cái giá là làm cho các lớp dưới không còn có ích. Trong ví dụ tuyến tính của chúng ta, các lớp dưới không còn có bất kỳ tác động có hại nào, nhưng chúng cũng không còn có bất kỳ tác dụng hữu ích nào. Điều này là do chúng ta đã chuẩn hóa các thống kê bậc nhất và bậc hai, điều mà

8.8.2 Phương Pháp Xuống Tọa Độ (Coordinate Descent)

Trong một số trường hợp, có thể giải quyết nhanh chóng một bài toán tối ưu bằng cách phân tách nó thành các phần riêng biệt. Nếu ta tối ưu hàm mục tiêu $f(x)$ theo một biến x_i, sau đó tối ưu tiếp theo một biến khác x_j, và cứ tiếp tục như vậy theo từng biến, ta sẽ đảm bảo đạt được điểm cực tiểu (hoặc cực đại) của hàm mục tiêu. Phương pháp này được gọi là xuống tọa độ (Coordinate Descent), vì ta tối ưu từng tọa độ (biến) một. Một cách tổng quát hơn, xuống tọa độ khối (Block Coordinate Descent) là việc tối ưu một nhóm các biến cùng lúc, thay vì tối ưu từng biến riêng biệt. Thuật ngữ "xuống tọa độ"thường được dùng để chỉ cả phương pháp tối ưu hóa từng biến một và phương pháp tối ưu hóa một nhóm các biến.

Phương pháp này phù hợp nhất khi các biến trong bài toán tối ưu có thể được phân chia rõ ràng thành các nhóm độc lập, hoặc khi việc tối ưu một nhóm biến nào đó hiệu quả hơn

nhiều so với việc tối ưu tất cả các biến cùng một lúc. Ví dụ, xét hàm mục tiêu

$$J(H, W) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (X - WH)^2_{i,j} \tag{8.38}$$

Hàm này mô tả một bài toán học gọi là *mã hóa thưa thớt* (Sparse Coding), trong đó mục tiêu là tìm một ma trận trọng số W có thể giải mã một ma trận kích hoạt H để tái tạo lại bộ dữ liệu huấn luyện X. Trong hầu hết các ứng dụng của mã hóa thưa thớt, cũng thường sử dụng việc *ràng buộc trọng số* (weight decay) hoặc ràng buộc về chuẩn của các cột trong W để ngăn ngừa những giải pháp không hợp lý với H rất nhỏ và W rất lớn.

Hàm J không phải là một hàm lồi (convex). Tuy nhiên, ta có thể phân tách các tham số của bài toán tối ưu thành hai tập: các tham số từ điển W và các đại diện mã hóa H. Việc tối ưu hàm mục tiêu theo từng nhóm biến này là một bài toán lồi. Do đó, phương pháp xuống tọa độ khối cho phép ta sử dụng các thuật toán tối ưu lồi hiệu quả, bằng cách luân phiên giữa việc tối ưu W với H cố định, và tối ưu H với W cố định.

Tuy nhiên, phương pháp xuống tọa độ không phải là chiến lược tối ưu khi giá trị của một biến ảnh hưởng mạnh đến giá trị tối ưu của một biến khác. Chẳng hạn như trong hàm

$$f(x) = (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$$

Trong đó α là một hằng số dương. Mẫu thức đầu tiên khuyến khích hai biến x_1 và x_2 có giá trị tương đồng với nhau, trong khi mẫu thức thứ hai khuyến khích chúng gần bằng không. Giải pháp tối ưu là $x_1 = x_2 = 0$. Phương pháp Newton có thể giải quyết bài toán này trong một bước duy nhất, vì đây là một bài toán bậc hai xác định dương. Tuy nhiên, nếu α nhỏ, phương pháp xuống tọa độ sẽ tiến triển rất chậm, vì mẫu thức đầu tiên không cho phép thay đổi một biến một cách đáng kể mà không ảnh hưởng đến biến còn lại.

8.8.3 Phương Pháp Trung Bình Polyak (Polyak Averaging)

Phương pháp trung bình Polyak (Polyak và Juditsky, 1992) bao gồm việc tính trung bình của nhiều điểm trong quỹ đạo đi qua không gian tham số mà thuật toán tối ưu đã đi qua. Nếu qua t bước của thuật toán gradient descent, các điểm tham số là $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(t)}$, thì kết quả của thuật toán trung bình Polyak là

$$\hat{\theta}^{(t)} = \frac{1}{t} \sum_{i=1}^{t} \theta^{(i)}.$$

Với một số lớp bài toán nhất định, như gradient descent áp dụng cho các bài toán lồi (convex problems), phương pháp này có những đảm bảo mạnh mẽ về hội tụ. Tuy nhiên, khi áp dụng vào các mạng nơ-ron (neural networks), lý do biện minh cho phương pháp này chủ yếu mang tính chất trực giác hơn, nhưng thực tế nó cho kết quả rất tốt.

Ý tưởng cơ bản của phương pháp này là thuật toán tối ưu có thể nhảy qua lại qua một

thung lũng nhiều lần mà không bao giờ ghé qua một điểm gần đáy của thung lũng. Tuy nhiên, trung bình của tất cả các điểm ở hai bên của thung lũng sẽ gần với đáy của thung lũng.

Trong các bài toán không lồi (nonconvex problems), quỹ đạo tối ưu có thể rất phức tạp và đi qua nhiều vùng khác nhau. Việc đưa vào các điểm trong không gian tham số từ quá khứ xa xôi, mà có thể bị ngăn cách bởi các rào cản lớn trong hàm chi phí, có vẻ không phải là hành vi hữu ích. Do đó, khi áp dụng phương pháp trung bình Polyak vào các bài toán không lồi, thường sử dụng một phương pháp trung bình động có sự suy giảm theo cấp số mũ (exponentially decaying running average):

$$\hat{\theta}^{(t)} = \alpha\hat{\theta}^{(t-1)} + (1-\alpha)\theta^{(t)} \tag{8.39}$$

Trong đó α là một tham số điều chỉnh tốc độ suy giảm. Phương pháp trung bình động này được sử dụng rộng rãi trong nhiều ứng dụng. Ví dụ gần đây, Szegedy et al. (2015) đã sử dụng phương pháp này trong các bài toán học sâu.

8.8.4 Đào Tạo Trước Giám Sát (Supervised Pretraining)

Đôi khi, việc huấn luyện trực tiếp một mô hình để giải quyết một nhiệm vụ cụ thể có thể quá tham vọng nếu mô hình quá phức tạp và khó tối ưu hóa, hoặc nếu nhiệm vụ đó rất khó khăn. Trong những trường hợp này, việc huấn luyện một mô hình đơn giản hơn để giải quyết nhiệm vụ trước, sau đó làm cho mô hình trở nên phức tạp hơn có thể là một chiến lược hiệu quả hơn. Một chiến lược khác là huấn luyện mô hình giải quyết một nhiệm vụ đơn giản trước, sau đó chuyển sang giải quyết nhiệm vụ cuối cùng. Những chiến lược này, bao gồm việc huấn luyện các mô hình đơn giản cho các nhiệm vụ đơn giản trước khi đối mặt với thách thức huấn luyện mô hình mong muốn để thực hiện nhiệm vụ cuối cùng, được gọi chung là đào tạo trước.

Các thuật toán tham lam (*greedy algorithms*) chia một vấn đề thành nhiều thành phần và sau đó giải quyết từng thành phần tối ưu nhất trong điều kiện tách biệt. Thật không may, việc kết hợp các thành phần tối ưu riêng biệt không đảm bảo sẽ mang lại giải pháp tối ưu hoàn chỉnh. Tuy nhiên, các thuật toán tham lam thường rẻ hơn về mặt tính toán so với các thuật toán tìm kiếm giải pháp tối ưu chung, và chất lượng của giải pháp tham lam thường có thể chấp nhận được, nếu không phải là tối ưu. Các thuật toán tham lam cũng có thể được theo sau bởi một giai đoạn tinh chỉnh (fine-tuning), trong đó một thuật toán tối ưu hóa chung sẽ tìm kiếm giải pháp tối ưu cho toàn bộ vấn đề. Việc khởi tạo thuật toán tối ưu hóa chung bằng một giải pháp tham lam có thể giúp tăng tốc quá trình và cải thiện chất lượng của giải pháp tìm được.

Đào tạo trước, đặc biệt là đào tạo trước tham lam, rất phổ biến trong học sâu (*deep learning*). Trong phần này, chúng ta sẽ mô tả cụ thể các thuật toán đào tạo trước chia bài toán học giám sát thành những bài toán học giám sát đơn giản hơn. Cách tiếp cận này được gọi là đào tạo trước giám sát tham lam.

Trong phiên bản ban đầu của đào tạo trước giám sát tham lam (Bengio et al., 2007), mỗi giai đoạn bao gồm một nhiệm vụ huấn luyện học giám sát chỉ liên quan đến một phần của các lớp trong mạng nơ-ron cuối cùng. Một ví dụ về đào tạo trước giám sát tham lam được minh họa trong hình 8.7, trong đó mỗi lớp ẩn được thêm vào sẽ được đào tạo trước như một MLP giám sát nông, sử dụng đầu ra của lớp ẩn đã được huấn luyện trước đó làm đầu vào. Thay vì đào tạo một lớp tại một thời điểm, Simonyan và Zisserman (2015) đã huấn luyện trước một mạng nơ-ron tích chập sâu (eleven weight layers) và sau đó sử dụng bốn lớp đầu tiên và ba lớp cuối cùng của mạng này để khởi tạo các mạng sâu hơn (với tới mười chín lớp trọng số). Các lớp giữa của mạng mới, rất sâu, sẽ được khởi tạo ngẫu nhiên. Sau đó, mạng mới này được huấn luyện chung.

Một tùy chọn khác được Yu et al. (2010) khám phá là sử dụng các đầu ra của các MLP đã được huấn luyện trước, cũng như đầu vào thô, làm đầu vào cho mỗi giai đoạn tiếp theo.

Tại sao đào tạo trước giám sát tham lam lại có ích? Giả thuyết ban đầu được đề xuất bởi Bengio et al. (2007) là phương pháp này giúp cung cấp sự hướng dẫn tốt hơn cho các mức độ trung gian của một hệ thống phân cấp sâu. Nói chung, đào tạo trước có thể giúp cả trong việc tối ưu hóa và trong việc tổng quát hóa mô hình.

Một phương pháp liên quan đến đào tạo trước giám sát mở rộng ý tưởng này trong bối cảnh của học chuyển giao (transfer learning): Yosinski et al. (2014) đã huấn luyện trước một mạng nơ-ron tích chập sâu với tám lớp trọng số trên một tập hợp các nhiệm vụ (một tập con của 1.000 danh mục đối tượng ImageNet) và sau đó khởi tạo một mạng cùng kích thước với các lớp đầu tiên của mạng đầu tiên. Tất cả các lớp của mạng thứ hai (với các lớp trên được khởi tạo ngẫu nhiên) sau đó được huấn luyện chung để thực hiện một tập hợp các nhiệm vụ khác (một tập con khác của các danh mục đối tượng ImageNet), với ít ví dụ huấn luyện hơn so với tập nhiệm vụ đầu tiên. Các phương pháp khác về học chuyển giao với các mạng nơ-ron được thảo luận trong phần 15.2.

Một hướng nghiên cứu liên quan khác là phương pháp FitNets (Romero et al., 2015). Phương pháp này bắt đầu bằng cách huấn luyện một mạng nơ-ron có độ sâu đủ thấp và chiều rộng đủ lớn (số lượng đơn vị mỗi lớp) để dễ dàng huấn luyện. Mạng này sau đó trở thành một *giáo viên* cho một mạng thứ hai, được gọi là *học sinh*. Mạng học sinh sâu hơn và mỏng hơn nhiều (từ mười một đến mười chín lớp) và sẽ khó huấn luyện với SGD trong điều kiện bình thường. Việc huấn luyện mạng học sinh trở nên dễ dàng hơn bằng cách huấn luyện mạng học sinh không chỉ để dự đoán đầu ra cho nhiệm vụ ban đầu mà còn để dự đoán giá trị của lớp trung gian trong mạng giáo viên. Nhiệm vụ bổ sung này cung cấp một bộ gợi ý về cách sử dụng các lớp ẩn và có thể đơn giản hóa vấn đề tối ưu hóa.

Các tham số bổ sung được giới thiệu để hồi quy lớp trung gian của mạng giáo viên năm lớp từ lớp trung gian của mạng học sinh sâu hơn. Tuy nhiên, thay vì dự đoán mục tiêu phân loại cuối cùng, mục tiêu là dự đoán lớp ẩn trung gian của mạng giáo viên. Các lớp thấp hơn của mạng học sinh do đó có hai mục tiêu: giúp các đầu ra của mạng học sinh hoàn thành nhiệm vụ của chúng, cũng như dự đoán lớp trung gian của mạng giáo viên.

Mặc dù một mạng mỏng và sâu có vẻ khó huấn luyện hơn một mạng rộng và nông, mạng mỏng và sâu có thể tổng quát tốt hơn và chắc chắn có chi phí tính toán thấp hơn nếu nó đủ mỏng để có ít tham số hơn.

Không có các gợi ý về lớp ẩn, mạng học sinh sẽ hoạt động rất kém trong các thí nghiệm, cả trong tập huấn luyện và tập kiểm tra. Các gợi ý về lớp trung gian có thể là một trong những công cụ giúp huấn luyện các mạng nơ-ron mà nếu không có chúng sẽ khó huấn luyện, nhưng các kỹ thuật tối ưu hóa khác hoặc thay đổi trong kiến trúc cũng có thể giải quyết vấn đề.

8.8.5 Thiết kế mô hjình để hỗ trợ tối ưu hóa

Một chiến lược tốt nhất để cải thiện tối ưu hóa không phải lúc nào cũng là cải thiện thuật toán tối ưu hóa. Thực tế, nhiều cải tiến trong tối ưu hóa các mô hình học sâu đã đến từ việc thiết kế mô hình sao cho dễ tối ưu hơn.

Về lý thuyết, chúng ta có thể sử dụng các hàm kích hoạt có tính chất biến đổi theo các mẫu không đơn điệu, dao động lên xuống. Tuy nhiên, điều này sẽ làm cho việc tối ưu hóa trở nên cực kỳ khó khăn. Trên thực tế, điều quan trọng hơn là chọn một gia đình mô hình sao cho dễ tối ưu hơn là sử dụng một thuật toán tối ưu hóa mạnh mẽ. Hầu hết các tiến bộ trong học mạng nơ-ron suốt ba mươi năm qua đã được đạt được bằng cách thay đổi gia đình mô hình thay vì thay đổi quy trình tối ưu hóa. Gradient stochasitc với moment, thuật toán đã được sử dụng để huấn luyện mạng nơ-ron từ những năm 1980, vẫn đang được sử dụng trong các ứng dụng mạng nơ-ron hiện đại.

Cụ thể, các mạng nơ-ron hiện đại thể hiện một sự lựa chọn thiết kế khi sử dụng các phép biến đổi tuyến tính giữa các lớp và các hàm kích hoạt có thể phân biệt gần như mọi nơi, với độ dốc đáng kể trong phần lớn miền của chúng. Một số đổi mới trong mô hình như LSTM, các đơn vị tuyến tính khôi phục (ReLU) và đơn vị maxout đều chuyển hướng về việc sử dụng các hàm tuyến tính hơn so với các mô hình trước đây như mạng sâu với các đơn vị sigmoid. Những mô hình này có những đặc tính tốt giúp tối ưu hóa dễ dàng hơn. Gradient có thể lan truyền qua nhiều lớp miễn là ma trận Jacobian của phép biến đổi tuyến tính có các giá trị đặc trưng hợp lý. Hơn nữa, các hàm tuyến tính luôn tăng theo một hướng duy nhất, vì vậy ngay cả khi đầu ra của mô hình rất xa với giá trị đúng, chỉ cần tính gradient là ta có thể biết được hướng di chuyển để giảm hàm mất mát.

Nói cách khác, các mạng nơ-ron hiện đại đã được thiết kế sao cho thông tin gradient tại địa phương của chúng tương ứng khá tốt với việc di chuyển về phía một giải pháp xa xôi.

Các chiến lược thiết kế mô hình khác có thể giúp tối ưu hóa dễ dàng hơn. Ví dụ, các đường dẫn tuyến tính hoặc các kết nối bỏ qua (skip connections) giữa các lớp sẽ giảm chiều dài của đường đi ngắn nhất từ các tham số của lớp dưới cùng đến đầu ra, từ đó giảm thiểu vấn đề gradient biến mất (vanishing gradient problem) (Srivastava et al., 2015). Một ý tưởng liên quan đến các kết nối bỏ qua là thêm các bản sao bổ sung của đầu ra được kết nối với các lớp ẩn trung gian trong mạng, như trong GoogLeNet (Szegedy et al., 2014a)

và các mạng được giám sát sâu (deeply supervised nets) (Lee et al., 2014). Những "đầu ra phụ"này được huấn luyện để thực hiện cùng một nhiệm vụ như đầu ra chính ở phía trên cùng của mạng, nhằm đảm bảo rằng các lớp dưới nhận được một gradient lớn. Khi quá trình huấn luyện hoàn thành, các đầu ra phụ này có thể bị loại bỏ. Đây là một phương pháp thay thế cho các chiến lược tiền huấn luyện đã được giới thiệu trong phần trước.

Theo cách này, ta có thể huấn luyện tất cả các lớp cùng một lúc trong một giai đoạn duy nhất, nhưng thay đổi kiến trúc sao cho các lớp trung gian (đặc biệt là các lớp dưới) có thể nhận được một số gợi ý về những gì chúng nên làm thông qua một đường dẫn ngắn hơn. Những gợi ý này cung cấp tín hiệu lỗi cho các lớp dưới cùng.

8.8.6 Tối Ưu Hóa Cho Huấn Luyện Các Mô Hình Học Sâu

Một phương pháp tiếp cận có thể giúp tìm được cực tiểu toàn cục là sử dụng thông tin về vị trí của cực tiểu toàn cục để giải quyết các phiên bản mờ dần của bài toán. Phương pháp này có thể gặp phải ba vấn đề chính. Thứ nhất, nó có thể định nghĩa thành công một chuỗi các hàm chi phí, trong đó hàm đầu tiên là lồi và cực tiểu của nó theo dõi từ hàm này sang hàm kế tiếp, cuối cùng đi đến cực tiểu toàn cục. Tuy nhiên, nó có thể yêu cầu quá nhiều hàm chi phí gia tăng đến mức chi phí của toàn bộ quy trình vẫn rất cao. Các bài toán tối ưu hóa NP-hard vẫn là NP-hard, ngay cả khi các phương pháp tiếp tục (continuation methods) có thể áp dụng. Hai vấn đề còn lại khiến phương pháp tiếp tục không hiệu quả là khi phương pháp này không thể áp dụng được. Đầu tiên, hàm chi phí có thể không trở thành lồi, dù có làm mờ như thế nào đi nữa. Ví dụ, hàm $J(\theta) = -\theta^\top\theta$ không trở thành lồi. Thứ hai, hàm có thể trở thành lồi khi bị mờ đi, nhưng cực tiểu của hàm mờ này lại theo dõi đến một cực tiểu cục bộ thay vì cực tiểu toàn cục của hàm chi phí gốc.

Mặc dù phương pháp tiếp tục chủ yếu được thiết kế để giải quyết vấn đề cực tiểu cục bộ, nhưng hiện nay người ta không còn coi cực tiểu cục bộ là vấn đề chính trong tối ưu hóa mạng nơ-ron. May mắn thay, các phương pháp tiếp tục vẫn có thể hỗ trợ. Các hàm mục tiêu dễ dàng hơn do phương pháp tiếp tục giới thiệu có thể loại bỏ các vùng phẳng, giảm phương sai trong các ước lượng gradient, cải thiện độ điều chỉnh của ma trận Hessian, hoặc làm bất kỳ điều gì giúp việc tính toán các cập nhật cục bộ dễ dàng hơn hoặc cải thiện sự tương ứng giữa các hướng cập nhật cục bộ và tiến trình hướng tới giải pháp toàn cục.

Bengio et al. (2009) đã nhận thấy rằng một phương pháp có tên là *curriculum learning*, hay *shaping*, có thể được hiểu như một phương pháp tiếp tục. *Curriculum learning* dựa trên ý tưởng lên kế hoạch cho quá trình học, bắt đầu bằng việc học các khái niệm đơn giản và tiến dần đến học các khái niệm phức tạp hơn dựa trên những khái niệm đơn giản đó. Chiến lược cơ bản này trước đây đã được biết đến là thúc đẩy tiến bộ trong huấn luyện động vật (Skinner, 1958; Peterson, 2004; Krueger và Dayan, 2009) và trong học máy (Solomonoff, 1989; Elman, 1993; Sanger, 1994). Bengio et al. (2009) đã lý giải chiến lược này như một phương pháp tiếp tục, trong đó các hàm $J(i)$ đầu tiên được làm dễ dàng hơn bằng cách tăng ảnh hưởng của các ví dụ đơn giản (hoặc bằng cách gán các hệ số lớn hơn cho đóng góp của chúng vào hàm chi phí, hoặc bằng cách lấy mẫu chúng thường xuyên

hơn), và thực nghiệm đã chỉ ra rằng kết quả tốt hơn có thể đạt được khi theo một chương trình học trong một nhiệm vụ mô hình ngôn ngữ mạng nơ-ron quy mô lớn. *Curriculum learning* đã thành công trong nhiều bài toán ngôn ngữ tự nhiên (Spitkovsky et al., 2010; Collobert et al., 2011a; Mikolov et al., 2011b; Tu và Honavar, 2011) và thị giác máy tính (Kumar et al., 2010; Lee và Grauman, 2011; Supancic và Ramanan, 2013). *Curriculum learning* cũng được xác nhận là phù hợp với cách thức con người dạy học (Khan et al., 2011): các giáo viên bắt đầu bằng cách cho học sinh thấy các ví dụ dễ dàng và điển hình hơn, sau đó giúp học sinh tinh chỉnh bề mặt quyết định với các trường hợp ít rõ ràng hơn. Các chiến lược học theo chương trình học hiệu quả hơn trong việc dạy học con người so với các chiến lược lấy mẫu đồng đều các ví dụ và cũng có thể tăng hiệu quả của các chiến lược dạy học khác (Basu và Christensen, 2013).

Một đóng góp quan trọng khác trong nghiên cứu về *curriculum learning* đã xuất hiện trong bối cảnh huấn luyện mạng nơ-ron hồi quy để nắm bắt các phụ thuộc dài hạn: Zaremba và Sutskever (2014) đã phát hiện ra rằng kết quả tốt hơn nhiều có thể đạt được với một chương trình học ngẫu nhiên, trong đó một hỗn hợp ngẫu nhiên các ví dụ dễ và khó luôn được trình bày cho người học, nhưng tỷ lệ trung bình của các ví dụ khó (ở đây là những ví dụ có phụ thuộc dài hạn hơn) dần dần được tăng lên. Với một chương trình học xác định, không có cải thiện nào so với cơ sở (huấn luyện thông thường từ tập huấn luyện đầy đủ) được quan sát.

Chúng ta đã mô tả gia đình cơ bản của các mô hình mạng nơ-ron và cách để điều chỉnh và tối ưu hóa chúng. Trong các chương tiếp theo, chúng ta sẽ chuyển sang các chuyên biệt hóa của gia đình mạng nơ-ron cho phép các mạng nơ-ron mở rộng quy mô lên rất lớn và xử lý dữ liệu đầu vào có cấu trúc đặc biệt. Các phương pháp tối ưu hóa được thảo luận trong chương này thường có thể áp dụng trực tiếp cho những kiến trúc chuyên biệt này mà không cần thay đổi nhiều hoặc thay đổi hoàn toàn.

CHƯƠNG 9. MẠNG NƠ-RON TÍCH CHẬP

Mạng nơ-ron tích chập (LeCun, 1989), hay còn gọi là *convolutional neural networks* hoặc CNNs, là một loại mạng nơ-ron được thiết kế đặc biệt để xử lý các loại dữ liệu có cấu trúc dạng lưới. Dạng phổ biến nhất của dữ liệu lưới là hình ảnh, được biểu diễn dưới dạng ma trận hai chiều của các điểm ảnh cho mỗi kênh màu. Ngoài ra, dữ liệu chuỗi thời gian cũng có thể được coi là một dạng lưới một chiều, với mỗi điểm dữ liệu là một mẫu đo được tại một mốc thời gian xác định.

CNN đã đạt được những thành công vượt trội trong nhiều ứng dụng thực tế. Cái tên "mạng nơ-ron tích chập" bắt nguồn từ việc trong mạng có sử dụng một phép toán gọi là tích chập (convolution) — một dạng đặc biệt của phép toán tuyến tính. Thay vì sử dụng phép nhân ma trận thông thường như trong mạng fully connected, CNN sử dụng phép tích chập tại ít nhất một lớp trong kiến trúc của mình.

Trong chương này, chúng ta sẽ bắt đầu bằng cách tìm hiểu khái niệm tích chập là gì. Tiếp theo, chúng ta sẽ giải thích lý do tại sao tích chập lại hữu ích trong mạng nơ-ron. Sau đó, chúng ta sẽ làm quen với một phép toán khác thường được dùng chung với tích chập, gọi là pooling.

Cần lưu ý rằng trong mạng nơ-ron, phép tích chập thường không hoàn toàn giống với cách định nghĩa trong toán học hoặc kỹ thuật. Vì vậy, chúng ta sẽ trình bày một số biến thể thường gặp của phép tích chập trong học máy, và cách áp dụng nó trên dữ liệu có số chiều khác nhau.

Sau đó, chúng ta sẽ thảo luận về các kỹ thuật giúp tăng hiệu quả tính toán của tích chập. Vì CNN được lấy cảm hứng từ thần kinh học, chúng ta cũng sẽ trình bày một số nguyên lý sinh học liên quan và kết thúc bằng việc tổng kết vai trò quan trọng của CNN trong sự phát triển của học sâu.

Một điều quan trọng là chương này không nhằm hướng dẫn cách lựa chọn kiến trúc mạng CNN cho một ứng dụng cụ thể. Mục tiêu của chương là giới thiệu các công cụ cơ bản mà CNN cung cấp. Việc lựa chọn và thiết kế kiến trúc phù hợp sẽ được thảo luận trong chương 11.

Lĩnh vực nghiên cứu kiến trúc CNN hiện đang phát triển rất nhanh. Chỉ trong vài tuần hoặc vài tháng, có thể xuất hiện một kiến trúc mới vượt trội hơn trên một tập kiểm chuẩn nào đó. Do đó, không thể mô tả đầy đủ tất cả các kiến trúc hiện đại trong một cuốn sách in. Tuy nhiên, những kiến trúc tiên tiến nhất đều được xây dựng từ các khối cơ bản mà chúng ta sẽ trình bày trong chương này.

9.1 Phép toán tích chập

Nói một cách đơn giản, tích chập (convolution) là cách kết hợp hai hàm để tạo ra một hàm mới. Để hiểu rõ tích chập dùng làm gì, hãy xem một ví dụ cụ thể.

Giả sử chúng ta đang theo dõi vị trí của một tàu vũ trụ bằng cảm biến laser. Cảm biến này trả về một giá trị $x(t)$ tại mỗi thời điểm t, đại diện cho vị trí của tàu. Cảm biến có thể bị nhiễu, nên để có được ước lượng chính xác hơn, ta muốn làm trơn dữ liệu bằng cách lấy trung bình các giá trị gần thời điểm hiện tại, nhưng ưu tiên nhiều hơn cho các giá trị gần thời điểm đó.

Giả sử có một hàm trọng số $w(a)$, với a là khoảng thời gian cách thời điểm hiện tại. Khi đó, ước lượng làm trơn tại thời điểm t là:

$$s(t) = \int x(a)w(t-a)\,da \tag{9.1}$$

Đây chính là phép tích chập (convolution). Ký hiệu thường dùng là dấu sao ($*$), nên có thể viết lại:

$$s(t) = (x * w)(t) \tag{9.2}$$

Trong ví dụ này:

- $x(t)$ là dữ liệu đầu vào (input signal).
- $w(t)$ là hạt nhân tích chập (convolution kernel).
- $s(t)$ là đầu ra (output), còn gọi là *bản đồ đặc trưng (feature map)*.

Để $s(t)$ thực sự là trung bình có trọng số, w cần:

- Là hàm mật độ xác suất (tổng tích phân bằng 1).
- Có giá trị 0 với $a < 0$, tức là không sử dụng thông tin từ tương lai.

Dù vậy, trong toán học nói chung, tích chập có thể áp dụng cho bất kỳ hai hàm nào mà tích phân của tích của chúng tồn tại.

Trong máy tính, dữ liệu thường là rời rạc. Ví dụ, cảm biến đo một giá trị mỗi giây. Khi đó, t là số nguyên, và ta có:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{9.3}$$

Trong học máy, dữ liệu đầu vào và kernel thường là mảng đa chiều, hay còn gọi là tensor. Tổng vô hạn thường được thay thế bằng một tổng hữu hạn để phù hợp với tính toán thực tế.

Ví dụ, khi đầu vào là ảnh hai chiều I và kernel là K, tích chập được tính như sau:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{9.4}$$

Do tính chất giao hoán, có thể viết lại:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{9.5}$$

Cách viết sau thường dễ lập trình hơn, vì chỉ số m, n có thể cố định kích thước.

Tính chất giao hoán là do kernel bị lật so với đầu vào. Trong toán học, điều này quan trọng để đảm bảo tính chất lý thuyết, nhưng trong thực tế thì không. Nhiều thư viện học sâu không thực hiện việc lật kernel, và thay vào đó dùng phép tương quan chéo (cross-correlation):

$$S(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{9.6}$$

Ở đây, kernel được giữ nguyên và không bị lật. Dù phép này không đúng theo định nghĩa tích chập toán học, các thư viện vẫn gọi nó là "convolution".

Trong học sâu, kernel là tham số được học. Nếu mạng có lật kernel, thì nó sẽ học một kernel lật ngược so với khi không lật. Về bản chất, kết quả đầu ra vẫn có thể tương đương.

Cần lưu ý rằng trong mạng nơ-ron, tích chập thường không được dùng riêng lẻ. Nó luôn đi kèm với các thành phần khác như hàm kích hoạt, chuẩn hóa, và pooling. Khi kết hợp như vậy, tính giao hoán của tích chập không còn giữ được vai trò trung tâm. Hình 9.1 minh họa một ví dụ về tích chập hai chiều (không lật kernel) theo kiểu *valid* — nghĩa là kernel chỉ áp dụng tại những vị trí mà toàn bộ nó nằm gọn trong ảnh đầu vào.

Phép tích chập rời rạc cũng có thể được xem như một phép nhân ma trận đặc biệt — nơi nhiều phần tử trong ma trận bị ràng buộc phải giống nhau.

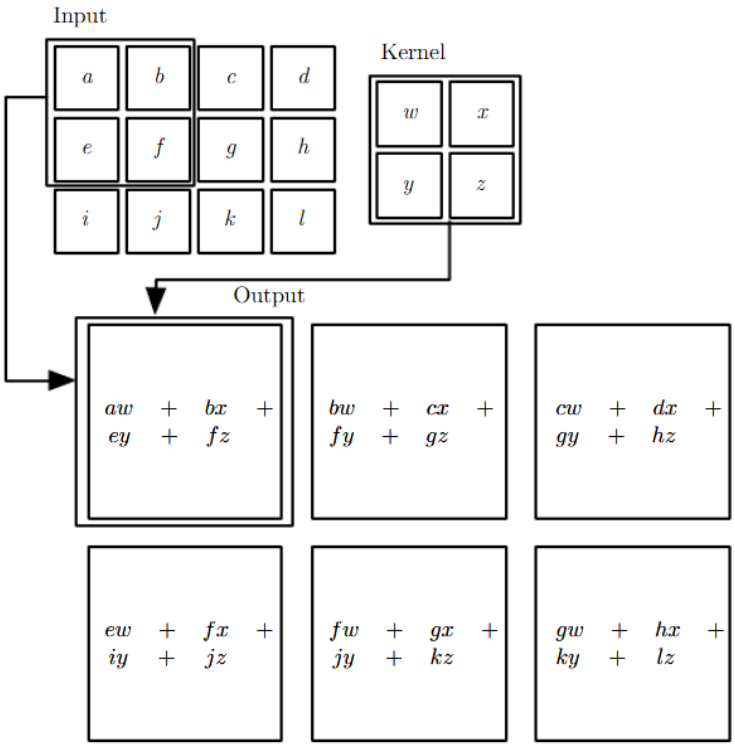
- Với tích chập một chiều, ta có ma trận Toeplitz, trong đó mỗi hàng là bản dịch (dịch trái/phải) của hàng phía trên.
- Với tích chập hai chiều, ma trận tương ứng là ma trận vòng lặp khối đôi (*doubly block circulant*), một mở rộng hai chiều của ma trận Toeplitz.

Những ma trận này không chỉ có nhiều phần tử bằng nhau, mà còn thường rất thưa (rất nhiều giá trị bằng 0). Điều này là vì kernel thường nhỏ hơn ảnh nhiều, nên nó chỉ ảnh hưởng đến một vùng cục bộ.

Điều quan trọng là bất kỳ thuật toán nào có thể xử lý nhân ma trận cũng có thể sử dụng tích chập, vì về bản chất, tích chập là một phép toán tuyến tính. Tuy nhiên, trong thực tế, các mạng CNN hiện đại khai thác đặc tính thưa và lặp lại của tích chập để tối ưu tốc độ xử lý và tiết kiệm bộ nhớ — nhất là với dữ liệu lớn như ảnh có độ phân giải cao.

9.2 Mục đích sử dụng tích chập

Tích chập trong mạng nơ-ron sâu được dùng vì nó mang lại ba lợi ích quan trọng:



Hình 9.1: Ví dụ tích chập 2D không lật kernel với kiểu "valid". Các mũi tên thể hiện cách kernel áp dụng lên vùng tương ứng để tạo ra phần tử phía trên bên trái của tensor đầu ra.

- Tương tác thưa (sparse interactions)
- Chia sẻ tham số (parameter sharing)
- Biến đổi tương đương (equivariant representations)

Ngoài ra, tích chập còn giúp mạng xử lý tốt các đầu vào có kích thước thay đổi.

Trong mạng truyền thống, mỗi đơn vị đầu ra được kết nối đến tất cả các đầu vào — dẫn đến rất nhiều tham số và chi phí tính toán lớn. Trong khi đó, mạng CNN dùng kết nối thưa — mỗi đầu ra chỉ phụ thuộc vào một vùng nhỏ đầu vào thông qua một kernel nhỏ.

Ví dụ:

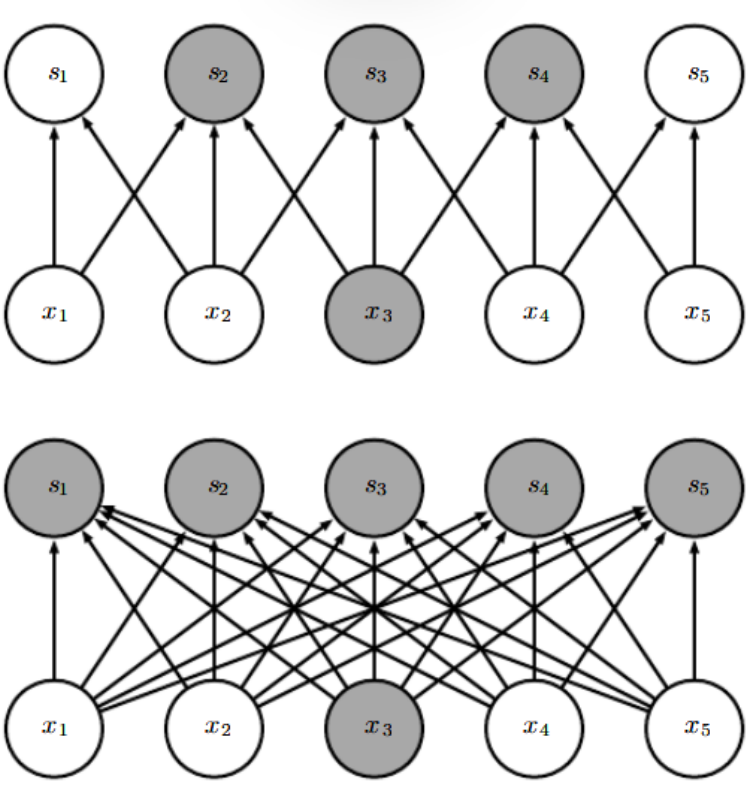
- Với mạng fully connected có m đầu vào và n đầu ra: cần $m \times n$ tham số, thời gian $O(m \times n)$.
- Với mạng tích chập, mỗi đầu ra chỉ phụ thuộc vào k đầu vào (với $k \ll m$): chỉ cần $k \times n$ tham số, thời gian $O(k \times n)$.

Điều này giúp:

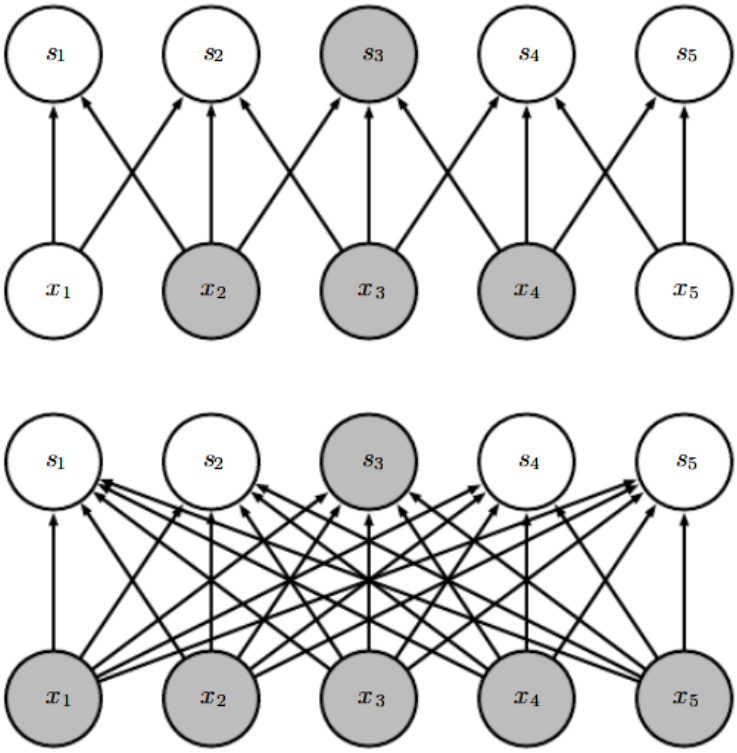
- Tiết kiệm bộ nhớ.
- Dễ học hơn vì ít tham số hơn.
- Tăng tốc độ tính toán.

Mặc dù các lớp tích chập chỉ có thể mô hình hóa tương tác cục bộ, nhưng khi chồng

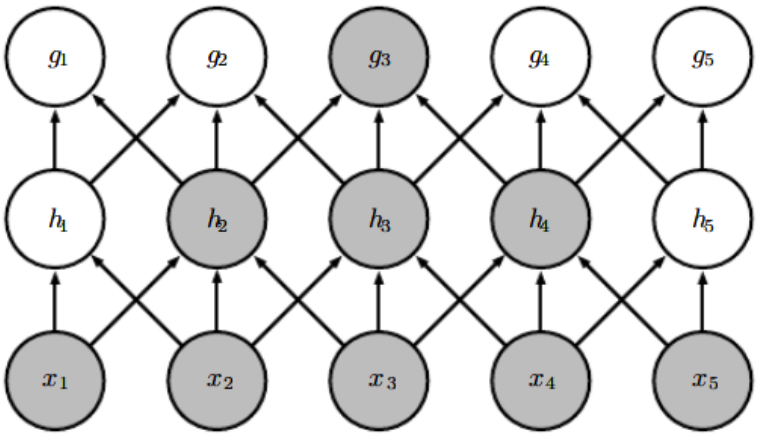
nhiều lớp lên nhau, các đơn vị ở tầng sâu có thể gián tiếp "nhìn thấy"phần lớn hoặc toàn bộ ảnh đầu vào — vì mỗi lớp mở rộng vùng tiếp nhận (receptive field).



Hình 9.2: Kết nối thưa từ góc nhìn đầu vào. Một đơn vị đầu vào x_3 chỉ ảnh hưởng đến ba đầu ra nếu dùng tích chập (trên), nhưng ảnh hưởng đến tất cả đầu ra nếu dùng nhân ma trận đầy đủ (dưới).



Hình 9.3: Kết nối thưa từ góc nhìn đầu ra. Đơn vị đầu ra s_3 chỉ phụ thuộc vào ba đầu vào gần nó nếu dùng tích chập (trên), nhưng phụ thuộc vào toàn bộ đầu vào nếu dùng nhân ma trận (dưới).



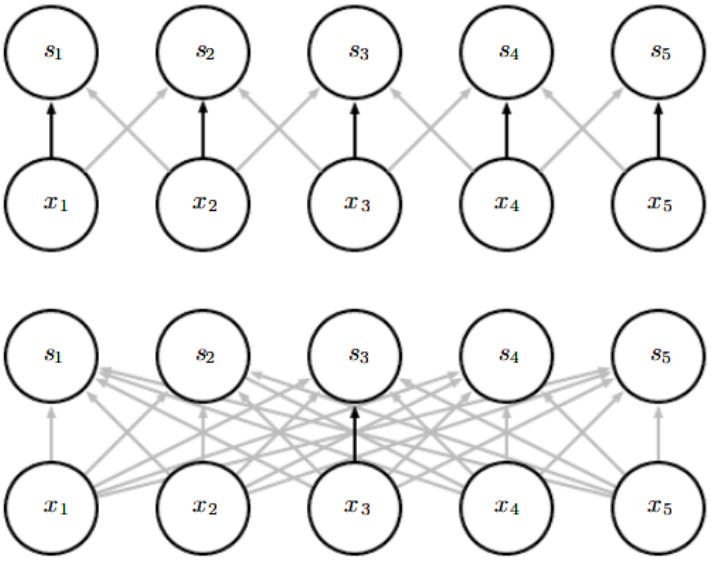
Hình 9.4: Ở các tầng sâu, vùng tiếp nhận trở nên lớn hơn — đặc biệt nếu có pooling hoặc stride — cho phép các đơn vị ở tầng cao nhìn thấy vùng rộng hơn của ảnh gốc.

Một điểm quan trọng khác là chia sẻ tham số. Trong mạng truyền thống, mỗi kết nối có trọng số riêng, và trọng số đó chỉ dùng một lần. Trong khi đó, mạng CNN sử dụng cùng một kernel cho mọi vị trí trong ảnh — nghĩa là mỗi trọng số được dùng nhiều lần, ở nhiều vị trí khác nhau.

Điều này giúp:

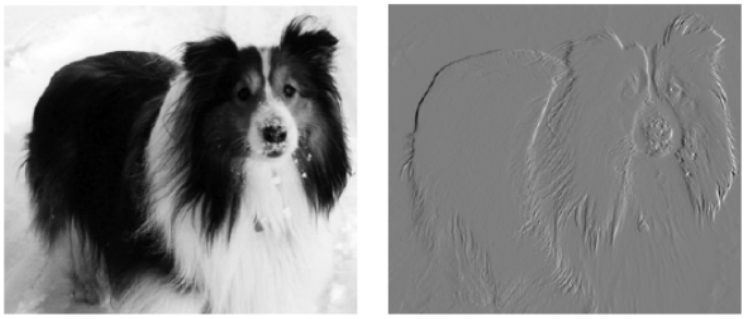
- Giảm mạnh số lượng tham số phải học (chỉ còn k thay vì $m \times n$).

- Tăng hiệu quả thống kê (học được từ ít dữ liệu hơn).
- Giữ nguyên tốc độ tính toán (vẫn là $O(k \times n)$).



Hình 9.5: Chia sẻ tham số: (Trên) Một trọng số trong kernel được dùng ở nhiều vị trí đầu vào. (Dưới) Trọng số chỉ dùng một lần trong mạng fully connected.

Ví dụ trong Hình 9.6 minh họa cách tích chập có thể được dùng để phát hiện biên trong ảnh một cách hiệu quả hơn rất nhiều so với nhân ma trận — không chỉ về tốc độ mà cả về số lượng tham số cần thiết.



Hình 9.6: Tích chập giúp phát hiện cạnh trong ảnh với số phép tính ít hơn rất nhiều so với nhân ma trận đầy đủ. Tích chập sử dụng lại trọng số, ít phép nhân, và tận dụng tính chất cục bộ của đặc trưng hình ảnh.

Trong mạng nơ-ron, lớp tích chập có một tính chất đặc biệt nhờ vào việc chia sẻ tham số, gọi là tính đồng biến với phép tịnh tiến (equivariance to translation). Nói đơn giản, nếu ta dịch đầu vào, thì đầu ra cũng sẽ dịch theo một cách tương ứng.

Cụ thể, nếu một hàm $f(x)$ là đồng biến với một phép biến đổi g, thì ta có:

$$g(f(x)) = f(g(x))$$

Trong trường hợp của tích chập, nếu g là phép dịch chuyển đầu vào, thì tích chập sẽ đồng biến với phép đó.

Ví dụ, giả sử I là hàm biểu diễn độ sáng của ảnh tại mỗi điểm nguyên (x, y). Ta định nghĩa phép dịch chuyển ảnh sang phải một đơn vị là $I'(x, y) = I(x - 1, y)$. Khi đó, nếu ta dịch ảnh trước rồi tích chập, hoặc tích chập rồi dịch kết quả, thì kết quả cuối cùng vẫn như nhau.

Điều này rất hữu ích khi xử lý dữ liệu thời gian hoặc hình ảnh. Với dữ liệu thời gian (như âm thanh, cảm biến...), điều này nghĩa là nếu một đặc trưng xuất hiện muộn hơn trong dòng thời gian, thì đầu ra cũng sẽ thể hiện điều đó tương ứng. Với ảnh, nếu một đối tượng dịch chuyển trong ảnh, thì biểu diễn của nó trong đầu ra cũng dịch chuyển tương ứng.

Nhờ tính chất này, mạng có thể phát hiện các đặc trưng như cạnh, góc, hoặc đường cong ở bất kỳ đâu trong ảnh mà không cần học lại từ đầu. Điều này chính là kết quả trực tiếp của việc chia sẻ tham số trong tích chập.

Tuy nhiên, trong một số bài toán, chia sẻ tham số trên toàn bộ không gian ảnh có thể không phù hợp. Ví dụ, nếu ảnh luôn chứa một khuôn mặt ở trung tâm, thì ta có thể muốn học các tham số riêng biệt cho từng vùng: phần trên học nhận diện lông mày, phần dưới học nhận diện cằm.

Ngoài ra, tính đồng biến này chỉ áp dụng cho phép dịch chuyển. Với các phép biến đổi như xoay hoặc thay đổi tỷ lệ, tích chập không tự động có tính đồng biến. Để xử lý những biến đổi này, ta cần bổ sung thêm cơ chế khác — chẳng hạn như data augmentation, kiến trúc xoay bất biến, hoặc mạng capsule.

Cuối cùng, một lợi ích khác của tích chập là giúp xử lý hiệu quả các loại dữ liệu có kích thước thay đổi — điều mà các mạng fully connected không làm được. Chúng ta sẽ bàn kỹ hơn về điều này trong phần 9.7.

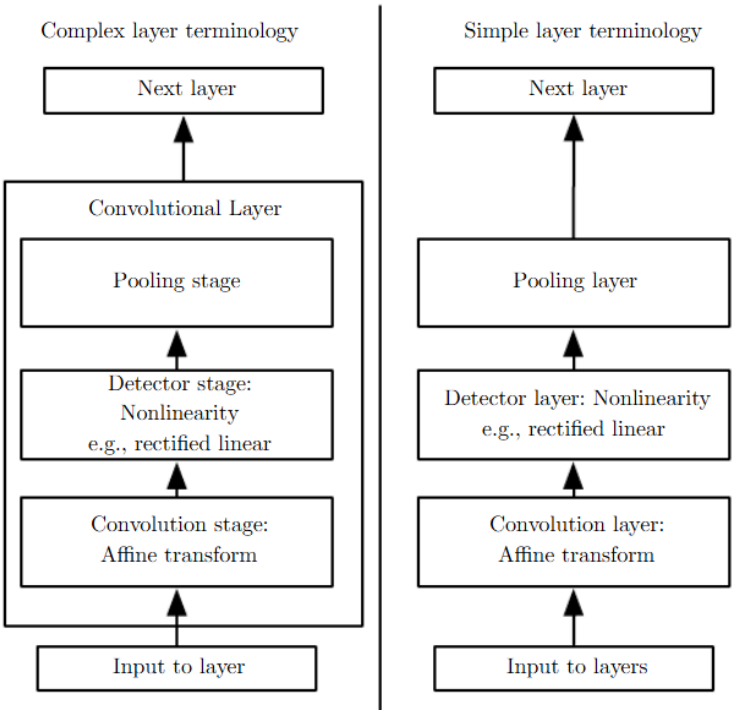
9.3 Pooling

Một lớp cơ bản trong mạng nơ-ron tích chập thường gồm ba bước chính (xem Hình 9.7):

1. Tích chập: nhiều kernel được áp dụng song song để quét qua đầu vào, trích xuất các đặc trưng cục bộ như cạnh, góc, hoặc kết cấu.
2. Phi tuyến: thường dùng hàm kích hoạt ReLU để tăng tính biểu diễn phi tuyến cho mạng.
3. Pooling: tóm tắt thông tin trong một vùng nhỏ, giúp giảm số chiều và tăng tính ổn định của mạng.

Phép pooling lấy nhiều giá trị trong một vùng và rút gọn thành một giá trị duy nhất, đại diện cho vùng đó. Một số phương pháp phổ biến:

- Max pooling: lấy giá trị lớn nhất trong vùng.



Hình 9.7: Cấu trúc điển hình của một lớp trong mạng CNN. (Trái) Gom nhóm các bước (tích chập, phi tuyến, pooling) thành một lớp duy nhất. (Phải) Mỗi bước là một lớp riêng biệt. Trong sách này, chúng ta dùng cách nhóm ở bên trái.

- Average pooling: lấy trung bình các giá trị.
- L2 pooling: tính chuẩn L2 của các giá trị.
- Pooling có trọng số: các giá trị gần trung tâm được ưu tiên hơn.

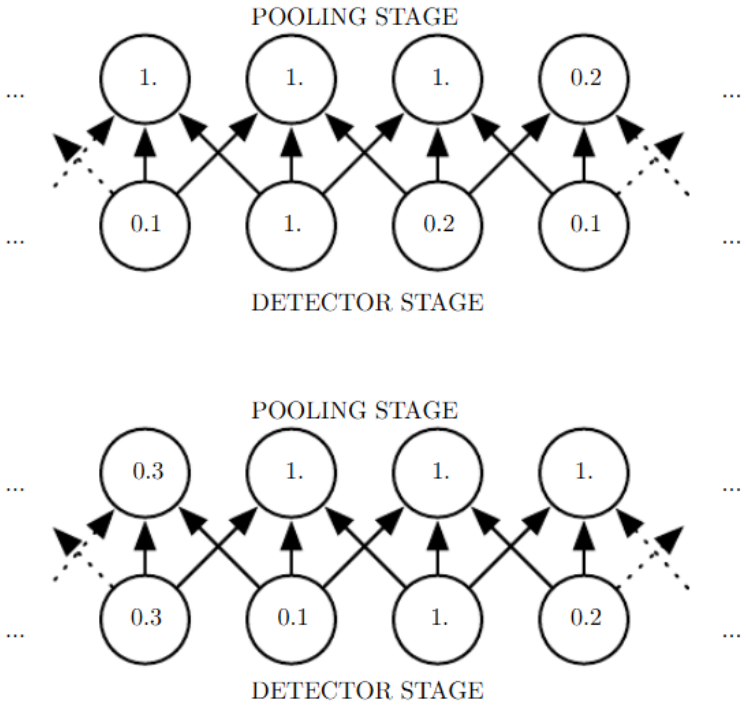
Pooling giúp:

- Giảm số chiều dữ liệu (downsampling), tiết kiệm bộ nhớ và tăng tốc tính toán.
- Tăng khả năng bất biến với dịch chuyển nhỏ — tức là nếu ảnh dịch nhẹ, đầu ra vẫn gần như không thay đổi.
- Giảm nguy cơ overfitting bằng cách bỏ qua các chi tiết nhỏ không quan trọng.

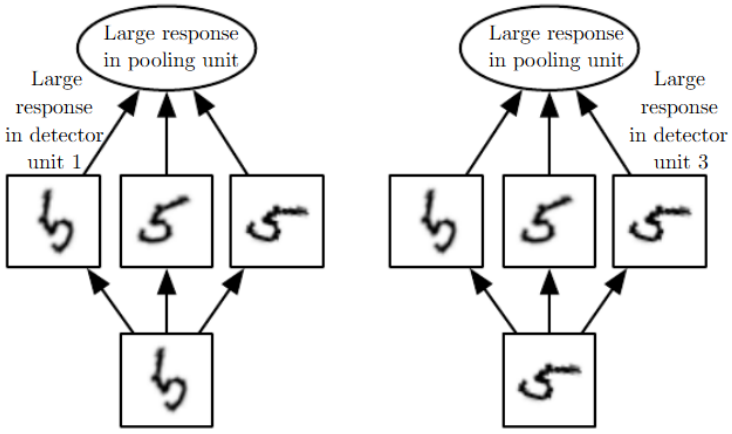
Pooling thêm vào mạng một giả định rằng các đặc trưng quan trọng không bị ảnh hưởng bởi dịch chuyển nhỏ — một giả định phù hợp trong nhiều bài toán thị giác, ví dụ như nhận diện đối tượng, nơi ta không cần biết chính xác mắt hay miệng nằm ở pixel nào, chỉ cần biết có xuất hiện là đủ.

Tuy nhiên, nếu bài toán yêu cầu nhận diện cấu trúc hình học chính xác (ví dụ nhận diện góc tạo bởi hai cạnh), thì pooling có thể gây mất thông tin quan trọng. Trong những trường hợp đó, cần sử dụng pooling cẩn thận.

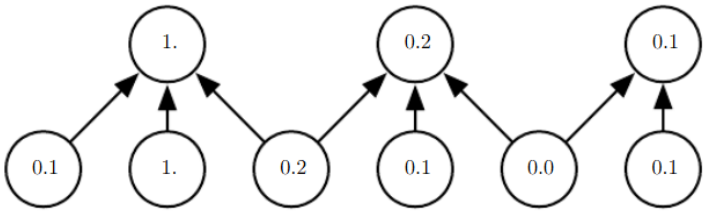
Một lợi ích khác là pooling giúp giảm số lượng đặc trưng cần xử lý ở các lớp sau, nhất là khi ta chỉ lấy mẫu đầu ra mỗi k pixel (stride pooling). Điều này giúp giảm đáng kể số phép tính, tiết kiệm bộ nhớ, và đôi khi còn giúp mạng học hiệu quả hơn. Ví dụ minh họa sẽ được trình bày ở Hình 9.10.



Hình 9.8: Hiệu ứng của max pooling đối với dịch chuyển nhỏ. (Trên) Trước và sau khi áp dụng pooling. (Dưới) Khi đầu vào bị dịch nhẹ, đầu ra của lớp phi tuyến thay đổi đáng kể, nhưng đầu ra của max pooling gần như không đổi.



Hình 9.9: Ví dụ về cách mạng học được tính bất biến. Một đơn vị pooling có thể học cách bỏ qua những thay đổi nhỏ trong đầu vào nếu nó kết hợp đúng các đặc trưng. Trong hình, ba bộ lọc học phát hiện chữ số 5 viết tay ở các góc xoay khác nhau. Bất kỳ biến thể nào của số 5 cũng làm một bộ lọc phản ứng mạnh, và max pooling sẽ chọn phản ứng mạnh nhất, giúp mạng nhận ra số 5 bất kể xoay thế nào. Phương pháp này được dùng trong mạng như maxout (Goodfellow et al., 2013a) và nhiều kiến trúc CNN khác. Max pooling giúp chống dịch chuyển, còn nhiều kênh lọc giúp mạng chống lại biến đổi phức tạp hơn.



Hình 9.10: Pooling kết hợp với giảm kích thước. Ở đây, ta dùng max pooling với vùng lọc rộng 3 và bước nhảy là 2. Điều này giúp giảm kích thước đầu ra mà vẫn giữ được thông tin quan trọng. Vùng cuối dù nhỏ vẫn được giữ lại để tránh bỏ sót dữ liệu.

Trong nhiều bài toán xử lý ảnh, pooling đóng vai trò quan trọng khi ảnh đầu vào có kích thước không cố định. Để có đầu ra chuẩn dùng cho lớp phân loại, một cách làm phổ biến là điều chỉnh vùng và bước nhảy trong pooling để đầu ra luôn có kích thước cố định.

Ví dụ, lớp pooling cuối có thể luôn tóm tắt ảnh thành bốn vùng tương ứng với bốn góc — bất kể ảnh gốc lớn hay nhỏ.

Một số nghiên cứu như Boureau et al. (2010) đã đề xuất các tiêu chí để chọn kiểu pooling phù hợp với từng tác vụ. Các biến thể nâng cao hơn gồm:

- Pooling động theo từng ảnh, dùng phân cụm để xác định vùng pooling (Boureau et al., 2011).
- Học một cấu trúc pooling cố định từ dữ liệu, rồi áp dụng cho tất cả ảnh (Jia et al., 2012).

Dù pooling rất hiệu quả, nhưng nó cũng có thể gây khó khăn với các kiến trúc cần lan truyền thông tin từ đầu ra ngược về đầu vào, như Boltzmann machine hoặc autoencoder. Những vấn đề này sẽ được bàn chi tiết hơn ở phần III. Mục 20.6 trình bày pooling trong mạng Boltzmann tích chập, và mục 20.10.6 đề cập đến các kỹ thuật "gần nghịch đảo"để phục hồi thông tin sau pooling.

Cuối cùng, Hình **??** sẽ giới thiệu một số kiến trúc CNN hoàn chỉnh dùng pooling để phân loại ảnh.

9.4 Convolution và Pooling như một loại prior rất mạnh

Trong học máy, prior là giả định ban đầu mà ta có về mô hình hoặc tham số trước khi xem dữ liệu. Đây là một khái niệm quen thuộc trong xác suất — bạn có thể hình dung như một “niềm tin ban đầu” trước khi được cung cấp thông tin thực tế.

Ví dụ: nếu bạn tin rằng ngày mai trời có thể mưa, thì đó là một prior. Khi bạn xem dự báo thời tiết (tức là dữ liệu), bạn có thể thay đổi suy đoán này. Khi đó, prior được cập nhật thành posterior.

Trong mô hình hóa xác suất, ta thường mô tả prior bằng một phân phối xác suất tiên nghiệm — như đã đề cập ở mục 5.6.

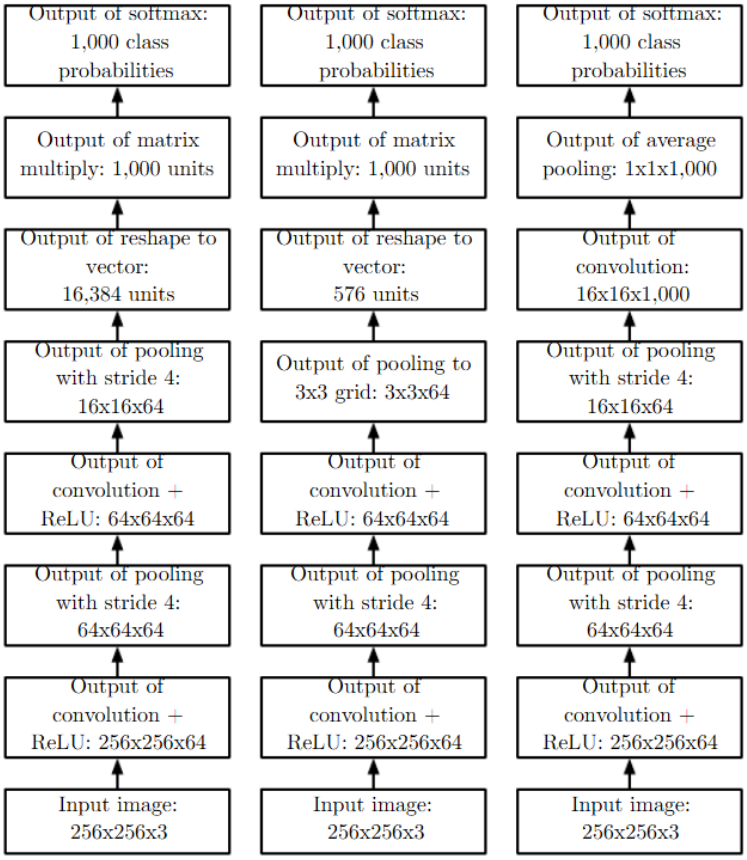
Mức độ mạnh hay yếu của một prior phụ thuộc vào mức độ hạn chế mà nó áp đặt:

- Prior yếu: như Gaussian có phương sai lớn — cho phép mô hình linh hoạt, ít giả định.
- Prior mạnh: như Gaussian có phương sai nhỏ — hạn chế nhiều, buộc mô hình học những gì phù hợp với giả định ban đầu.
- Prior cực mạnh (vô cùng mạnh): loại trừ hoàn toàn khả năng học một số hàm — dù dữ liệu có hỗ trợ.

Trong CNN, việc sử dụng các lớp convolution và pooling có thể được xem là đang áp đặt một prior cực mạnh lên hàm mà mô hình có thể học. Cụ thể, ta giả định rằng:

- Mạng nên học các đặc trưng mang tính cục bộ — tức là chỉ liên quan đến một vùng nhỏ trong ảnh.
- Các đặc trưng này nên được phát hiện tại mọi vị trí trong ảnh (chia sẻ tham số).
- Mạng nên ổn định trước các dịch chuyển nhỏ (bất biến qua pooling).

Những giả định này hạn chế không gian hàm mà mô hình có thể học, nhưng đổi lại, chúng cực kỳ phù hợp với dữ liệu dạng lưới như ảnh.



Hình 9.11: Ba kiểu kiến trúc CNN cho phân loại ảnh. (Trái): Dùng ảnh kích thước cố định, kết hợp convolution, pooling và fully connected. (Giữa): Cho phép ảnh có kích thước thay đổi, vẫn dùng fully connected bằng cách pooling về đầu vào có kích thước cố định. (Phải): Không dùng fully connected; mỗi bản đồ đặc trưng được trung bình lại và đưa thẳng vào softmax.

Từ góc độ xác suất, CNN giống như một mạng fully connected nhưng có thêm các prior cực mạnh:

- Trọng số của neuron ở các vị trí khác nhau phải giống nhau — chỉ khác vị trí không gian (dịch chuyển).
- Trọng số ngoài vùng quan sát phải bằng 0.

Hay nói cách khác:

- Hàm cần học chỉ nên có tương tác cục bộ.
- Hàm cần có tính dịch biến — phản ứng của mạng sẽ dịch chuyển nếu đầu vào dịch chuyển.

Pooling tiếp tục bổ sung thêm một prior mạnh nữa: kết quả đầu ra nên bất biến với dịch chuyển nhỏ.

Tất nhiên, nếu cố gắng áp đặt những prior này bằng cách viết mạng fully connected và thêm ràng buộc thủ công thì sẽ rất tốn kém và kém hiệu quả. CNN giúp ta "nhúng"các giả định này trực tiếp vào cấu trúc mạng.

- Nhược điểm — dễ gây underfitting: Nếu giả định của prior không phù hợp, chẳng hạn bài toán cần giữ nguyên vị trí chi tiết trong ảnh, thì pooling có thể làm mất thông tin quan trọng, gây lỗi huấn luyện. Một số kiến trúc như Inception (Szegedy et al., 2014a) xử lý bằng cách chỉ pooling ở một số kênh.
- Không nên so sánh trực tiếp CNN với mô hình khác: Vì CNN được xây dựng với hiểu biết trước về cấu trúc dữ liệu (topology), còn các mô hình khác phải tự học. Do đó, nên đánh giá chúng trên các bộ benchmark khác nhau để đảm bảo công bằng.

9.5 Mạng Convolutional

Trong deep learning, khi nói đến "convolution", ta không dùng đúng phép toán convolution rời rạc như trong toán học mà sử dụng một biến thể đơn giản hơn và hiệu quả hơn. Chúng ta sẽ làm rõ sự khác biệt và đặc điểm hữu ích của phiên bản thực tế này.

Thường thì mỗi lớp trong mạng sẽ thực hiện nhiều phép toán convolution cùng lúc—mỗi cái dùng một kernel khác nhau để trích xuất các loại đặc trưng khác nhau từ ảnh, dù ở các vị trí khác nhau.

Đầu vào của mạng không chỉ là một ma trận giá trị đơn giản, mà thường là nhiều kênh dữ liệu. Ví dụ, ảnh màu có 3 kênh: đỏ, xanh lá, xanh dương. Sau một lớp convolution, đầu ra sẽ là nhiều kênh đặc trưng, mỗi kênh thể hiện một loại đặc trưng tại nhiều vị trí không gian khác nhau. Do đó, đầu vào và đầu ra là các tensor 3 chiều: (kênh, chiều cao, chiều rộng). Nếu làm việc với nhiều ảnh cùng lúc (batch), thì là tensor 4 chiều—nhưng ta sẽ bỏ qua trục batch để đơn giản hóa.

Vì convolution dùng nhiều kênh, phép toán tuyến tính này không còn đảm bảo tính giao hoán như convolution truyền thống, trừ khi số kênh vào và ra bằng nhau.

Giả sử ta có tensor kernel 4 chiều K, trong đó $K_{i,j,k,l}$ biểu thị mức kết nối từ đầu vào kênh j tới đầu ra kênh i, lệch k hàng và l cột. Đầu vào V là tensor 3 chiều với $V_{i,j,k}$ là giá trị tại kênh i, hàng j, cột k. Khi đó, đầu ra Z sẽ là:

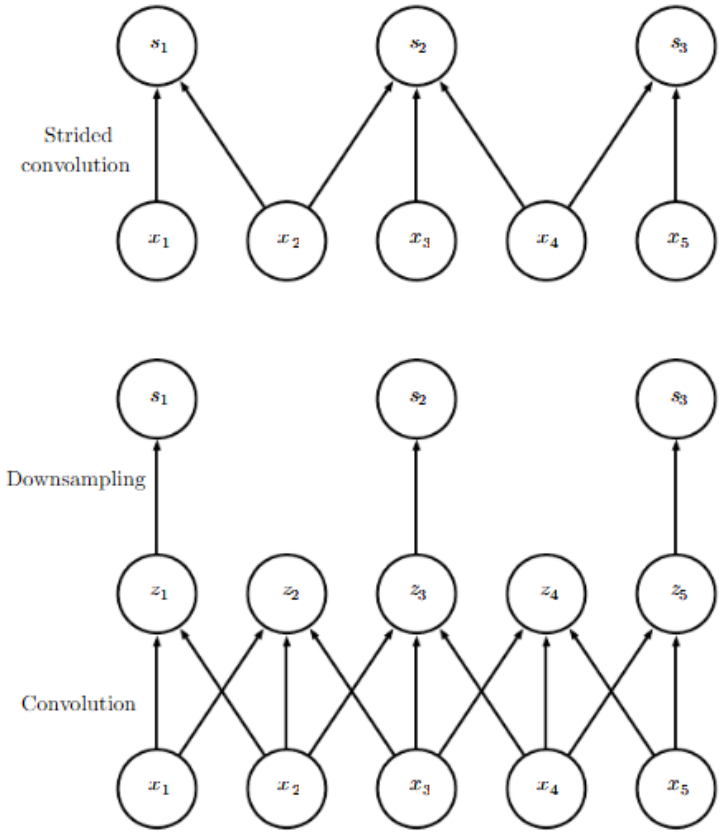
$$Z_{i,j,k} = \sum_{l,m,n} V_{i,j+m-1,k+n-1} K_{i,l,m,n}$$

Tổng chỉ thực hiện ở những vị trí hợp lệ.

Nếu muốn giảm chi phí tính toán, ta có thể giảm số vị trí áp dụng kernel, tức là giảm tần suất lấy mẫu đầu ra (downsampling). Cụ thể, nếu chỉ lấy mỗi s pixel, ta định nghĩa:

$$Z_{i,j,k} = c(K, V, s)_{i,j,k} = \sum_{l,m,n} V_{i,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}$$

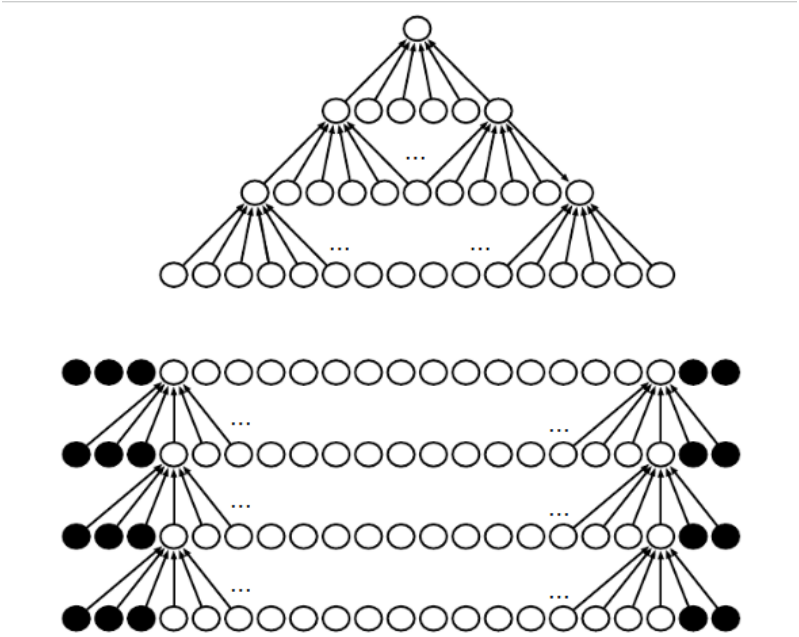
Ở đây, s là stride—bước nhảy của phép convolution. Có thể dùng stride khác nhau theo từng chiều. Xem ví dụ Hình 9.12.



Hình 9.12: Convolution với stride. Trên: thực hiện convolution với stride = 2. Dưới: tương đương toán học với thực hiện convolution stride = 1 rồi downsample. Cách làm phía dưới tốn nhiều tính toán vì tạo ra giá trị rồi loại bỏ.

Một yếu tố quan trọng khi xây dựng mạng convolution là khả năng tự động padding đầu vào bằng 0. Nếu không pad, đầu ra sẽ nhỏ lại sau mỗi lớp, vì kernel chiếm không

gian. Padding giúp bạn giữ nguyên kích thước đầu ra như ý muốn, thay vì buộc phải dùng kernel nhỏ hoặc chấp nhận mất diện tích biểu diễn. Xem ví dụ Hình 9.13.



Hình 9.13: Tác động của việc thêm zero padding đến kích thước mạng. Xem ví dụ mạng CNN sử dụng bộ lọc có chiều rộng bằng 6 ở mỗi lớp. (Trên) Không dùng zero padding, nên mỗi lớp làm giảm chiều rộng biểu diễn đi 5 pixel. Với ảnh đầu vào 16 pixel, chỉ có thể thêm 3 lớp chập, trong đó lớp cuối không còn di chuyển được. (Dưới) Thêm 5 pixel zero padding ở mỗi lớp giúp giữ nguyên kích thước ảnh đầu ra, cho phép xây dựng mạng sâu tùy ý.

Khi thực hiện tích chập trên ảnh, việc thêm "zero padding"(đệm bằng giá trị 0 ở rìa ảnh) là rất quan trọng. Có ba cách dùng zero padding mà bạn cần lưu ý:

- Trường hợp 1: Không padding (valid convolution) Khi kernel chỉ được áp vào các vị trí mà nó nằm hoàn toàn trong ảnh, đầu ra sẽ nhỏ hơn đầu vào. Nếu ảnh đầu vào rộng m, kernel rộng k, thì ảnh đầu ra sẽ rộng:

$$m - k + 1$$

Nếu dùng kernel lớn hoặc nhiều lớp chập, kích thước ảnh sẽ giảm nhanh chóng, cuối cùng còn 1×1, và các lớp chập sau sẽ vô nghĩa.

- Trường hợp 2: Padding để giữ nguyên kích thước (same convolution) Thêm đủ số pixel 0 để giữ kích thước ảnh không đổi. Cách này cho phép thêm nhiều lớp chập tùy ý. Tuy nhiên, các pixel ở rìa sẽ ảnh hưởng ít hơn đến đầu ra, vì chúng có ít pixel lân cận hơn để kernel "nhìn thấy".

- Trường hợp 3: Padding làm tăng kích thước (full convolution) Padding nhiều đến mức mỗi pixel đầu vào góp phần vào k vị trí đầu ra trong mỗi chiều:

$$m + k - 1$$

Cách này tạo đầu ra lớn hơn đầu vào, nhưng vẫn gặp vấn đề là pixel ở biên ảnh hưởng ít hơn pixel ở giữa, khiến mô hình khó học các đặc trưng đồng nhất khắp ảnh.

Trong thực tế, cách padding hiệu quả nhất thường nằm giữa "valid"và "same", tùy bài toán. Việc chọn đúng mức padding giúp mạng học tốt mà không làm giảm độ chính xác do ảnh bị thu nhỏ quá mức.

Trong một số trường hợp, thay vì dùng tích chập có chia sẻ tham số, ta dùng lớp kết nối cục bộ (locally connected layers) — tức là mỗi vị trí có bộ trọng số riêng biệt. Đây là ý tưởng do LeCun đề xuất từ những năm 1980.

Các kết nối ở lớp này vẫn giữ nguyên như mạng MLP, nhưng mỗi kết nối mang trọng số riêng, được biểu diễn bằng tensor 6 chiều W, với các chỉ số:

$$i : \text{Kênh đầu ra}, \quad j : \text{Hàng đầu ra}, \quad k : \text{Cột đầu ra},$$

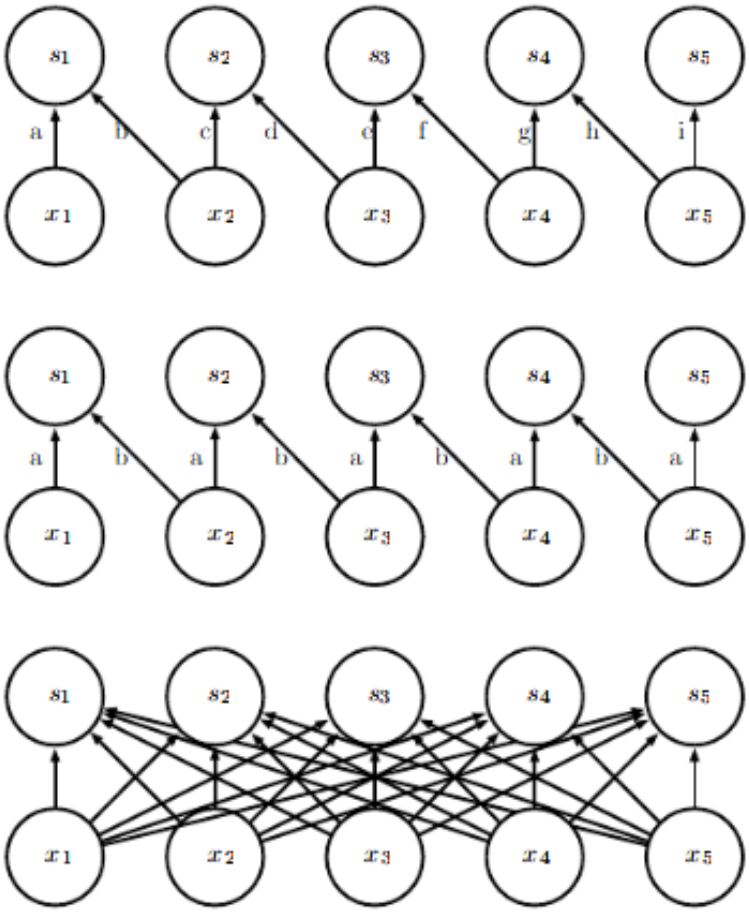
$$l : \text{Kênh đầu vào}, \quad m : \text{Độ lệch hàng}, \quad n : \text{Độ lệch cột}.$$

Phép toán tuyến tính tại mỗi vị trí là:

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]$$

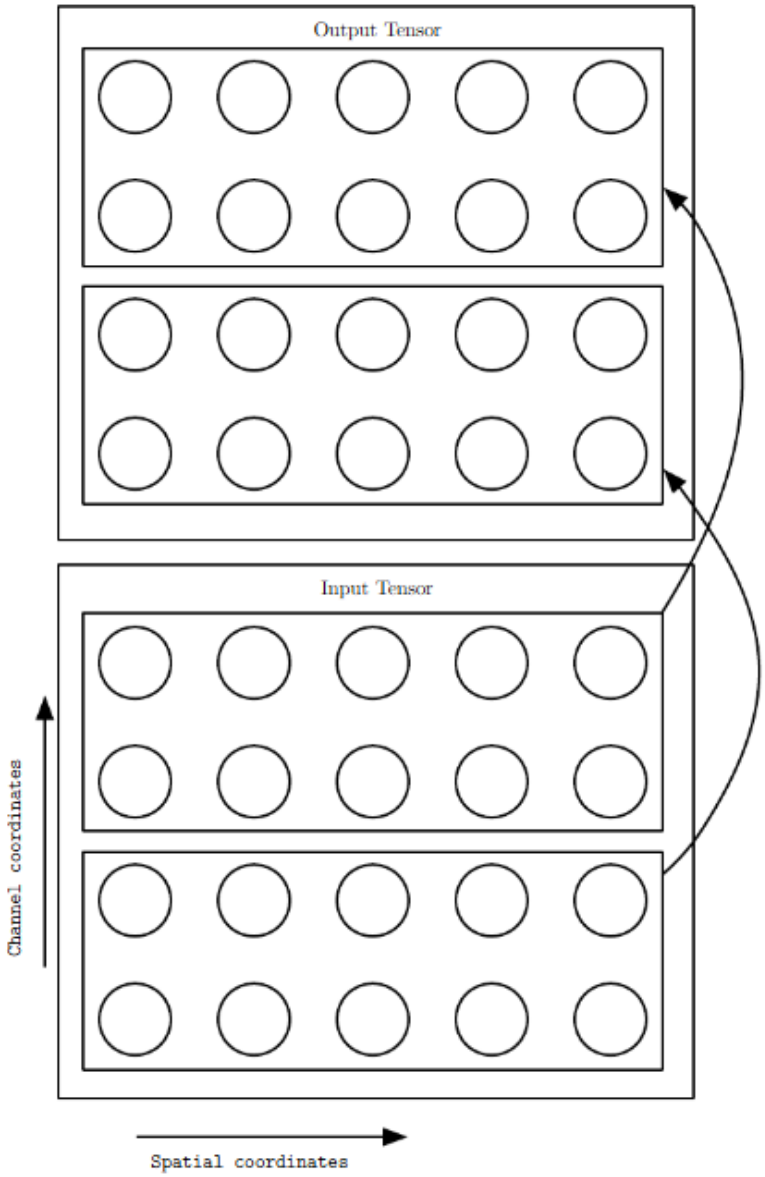
(Phương trình 9.9)

Phép toán này còn được gọi là "unshared convolution", tức là tích chập không chia sẻ tham số. Về mặt cấu trúc kết nối thì giống tích chập, nhưng mỗi vị trí có bộ trọng số khác nhau.



Hình 9.14: So sánh ba loại lớp: kết nối cục bộ, tích chập, và fully connected. Lớp kết nối cục bộ: mỗi kết nối có trọng số riêng (không chia sẻ). Lớp tích chập: kết nối giống như kết nối cục bộ, nhưng chia sẻ trọng số giữa các vị trí. Lớp fully connected: mỗi đơn vị kết nối với mọi đơn vị đầu vào, mỗi kết nối cũng có trọng số riêng.

Lớp kết nối cục bộ rất phù hợp khi đặc trưng chỉ phụ thuộc vào vùng lân cận nhỏ, và không cần giống nhau ở mọi nơi trong ảnh.

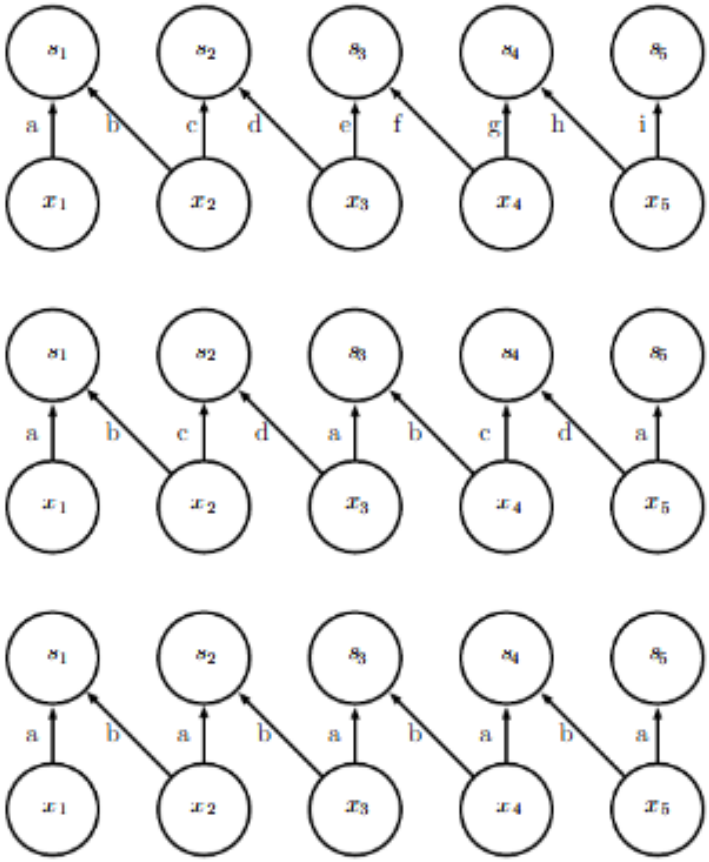


Hình 9.15: Một cách khác để giảm số lượng tham số: giới hạn kết nối giữa các kênh đầu vào và đầu ra. Ví dụ, 2 kênh đầu ra đầu tiên chỉ nhận thông tin từ 2 kênh đầu vào đầu tiên, 2 kênh sau từ 2 kênh tiếp theo. Cách này giúp giảm chi phí tính toán mà vẫn giữ hiệu suất.

—

Một biến thể nữa là tích chập lắp ghép (tiled convolution)(Gregor và LeCun, 2010; Le et al., 2010). Đây là bước trung gian giữa lớp kết nối cục bộ và lớp tích chập chuẩn. Thay vì học trọng số riêng cho từng vị trí, ta học một số lượng kernel cố định, rồi áp dụng các kernel này luân phiên khi quét qua ảnh.

Nói cách khác, mỗi vị trí gần nhau có thể dùng các kernel khác nhau, giống như lớp cục bộ, nhưng số lượng kernel bị giới hạn và dùng lại theo chu kỳ—giúp giảm lượng tham số cần học.



Hình 9.16: So sánh giữa lớp kết nối cục bộ, tích chập lắp ghép và tích chập chuẩn. Dùng kernel có chiều rộng 2 pixel. (Trên): Lớp kết nối cục bộ—mỗi kết nối có trọng số riêng (gắn nhãn a, b, c, d...). (Giữa): Tiled convolution—sử dụng luân phiên các kernel khác nhau (như a–b, rồi c–d), lặp lại sau mỗi t bước. (Dưới): Tích chập chuẩn—dùng một kernel duy nhất cho mọi vị trí (nhãn a–b áp dụng toàn ảnh).

Tiled convolution (hay còn gọi là phép chập phân tán) là một khái niệm quan trọng trong mạng nơ-ron tích chập. Để định nghĩa toán học, giả sử K là một tensor 6 chiều, trong đó hai chiều đầu tiên đại diện cho các vị trí trong bản đồ đầu ra. Thay vì có bộ lọc riêng cho từng vị trí, các vị trí này sẽ tuần tự sử dụng luân phiên một tập hợp gồm t bộ lọc theo mỗi chiều.

Khi t bằng đúng chiều rộng đầu ra, thì tiled convolution trở thành một lớp kết nối cục bộ (locally connected layer) chính xác.

Đầu ra tại vị trí $Z_{i,j,k}$ được tính theo công thức:

$$Z_{i,j,k} = \prod_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j\%t+1,k\%t+1}$$

Trong đó: - $\%$ là phép chia lấy dư (modulo), - Ví dụ: $t\%t = 0$, $(t+1)\%t = 1$, v.v.

Một điểm thú vị là các lớp kết nối cục bộ và tiled convolution có thể kết hợp hiệu quả với **max pooling**. Nếu các bộ lọc học được cách nhận diện các biến thể khác nhau của cùng một đặc trưng, thì phép max pooling giúp làm cho đầu ra không nhạy cảm với

những biến thể đó. Điều này khác với lớp tích chập chuẩn, vốn được thiết kế để có tính **dịch biến**, tức là phản ứng nhất quán với các dịch chuyển nhỏ.

Để huấn luyện mạng CNN, ta cần tính đạo hàm (gradient) của hàm mất mát theo các tham số kernel, dựa trên gradient của đầu ra. Trong một số trường hợp đơn giản, ta có thể tính gradient bằng một phép tích chập khác, nhưng khi dùng stride > 1, việc này trở nên phức tạp hơn và không thể đơn giản dùng tích chập.

Về bản chất, tích chập là một phép toán tuyến tính, và có thể biểu diễn bằng phép nhân ma trận nếu ta "duỗi thẳng"tensor đầu vào thành vector. Khi đó, phép tích chập tương đương với nhân một ma trận thưa (sparse matrix), trong đó mỗi phần tử của kernel xuất hiện nhiều lần tại các vị trí khác nhau.

Khi lan truyền ngược (backpropagation), ta cần nhân với chuyển vị của ma trận tích chập để tính gradient. Đây là bước thiết yếu để huấn luyện mạng sâu hoặc các mô hình như autoencoders, RBM hay sparse coding.

9.6 Đầu ra có cấu trúc trong mạng nơ-ron tích chập

Mạng CNN không chỉ dùng để phân loại hoặc hồi quy một nhãn đơn, mà còn có thể tạo ra đầu ra có cấu trúc phức tạp, ví dụ như bản đồ xác suất cho từng pixel.

Chẳng hạn, mô hình có thể xuất ra một tensor S, trong đó mỗi phần tử $S_{i,j,k}$ là xác suất rằng pixel tại vị trí (j, k) thuộc về lớp i. Điều này rất hữu ích cho các tác vụ như phân đoạn ảnh (image segmentation).

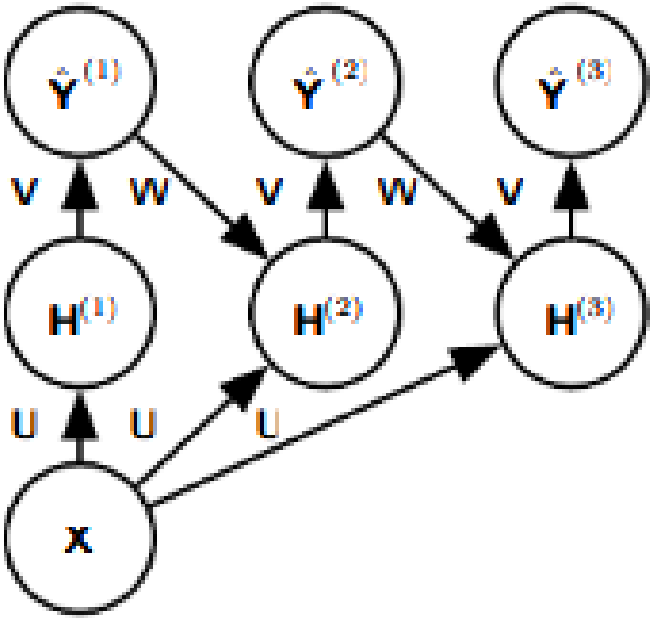
Một vấn đề là bản đồ đầu ra thường nhỏ hơn đầu vào, do các lớp pooling làm giảm kích thước. Có ba cách xử lý:

1. Không dùng pooling (Jain et al., 2007) để giữ kích thước ảnh.

2. Chấp nhận bản đồ nhãn độ phân giải thấp hơn (Pinheiro & Collobert, 2014, 2015).

3. Dùng pooling stride = 1 để giảm mất mát thông tin không gian.

Một cách hiệu quả để gán nhãn pixel là lặp lại việc dự đoán nhiều lần, mỗi lần cải thiện kết quả của lần trước. Mỗi bước sử dụng cùng trọng số—giống như một mạng hồi tiếp.



Hình 9.17: Một ví dụ mạng CNN hồi tiếp dùng để gán nhãn pixel. Ảnh đầu vào là tensor X, đầu ra là $\hat{Y}$ — bản đồ xác suất theo từng pixel và lớp. Ở mỗi bước, ước lượng $\hat{Y}$ cũ được dùng làm đầu vào cho bước tiếp theo. Bộ lọc U dùng để trích đặc trưng từ ảnh, V để tạo ra dự đoán, và W dùng để xử lý lại $\hat{Y}$ cũ (trừ bước đầu, thay bằng 0). Mô hình này dùng cùng trọng số cho tất cả bước lặp.

Sau khi có được bản đồ nhãn cho từng pixel, ta có thể xử lý thêm để phân đoạn ảnh thành các vùng. Các kỹ thuật xử lý sau như mô hình đồ thị hoặc tối ưu hóa phân vùng giúp gom nhóm các pixel gần nhau về cùng nhãn (Briggman et al., 2009; Turaga et al., 2010; Farabet et al., 2013). Một số mạng còn được huấn luyện để tối đa hóa mục tiêu tương đương với mô hình đồ thị (Ning et al., 2005; Thompson et al., 2014).

9.7 Các Loại Dữ Liệu

Dữ liệu dùng trong CNN thường có nhiều kênh—mỗi kênh là một loại quan sát khác nhau tại mỗi điểm không gian hoặc thời gian. Xem bảng 9.1 để thấy ví dụ với số chiều và số kênh khác nhau.

Cho đến giờ, chúng ta mới chỉ xét các dữ liệu đầu vào có cùng kích thước. Nhưng một lợi thế lớn của CNN là xử lý được dữ liệu đầu vào có kích thước thay đổi—điều mà mạng truyền thống không làm được vì cần một ma trận trọng số cố định.

Ví dụ: với ảnh có kích thước thay đổi, mạng fully connected không biết làm sao để áp dụng cùng một ma trận cho mọi ảnh. Nhưng với CNN, bạn chỉ cần chạy bộ lọc nhiều hoặc ít lần tùy ảnh lớn hay nhỏ. Ngay cả khi đầu ra cũng cần thay đổi theo đầu vào, như trong bài toán phân đoạn ảnh, CNN vẫn xử lý tốt. Trường hợp khác—khi cần đầu ra có kích thước cố định, như phân loại ảnh—ta có thể dùng thêm bước pooling với vùng thay đổi theo kích thước ảnh đầu vào, sao cho số đầu ra vẫn giữ cố định. Một ví dụ minh họa trong hình **??**.

Tuy nhiên, cần lưu ý rằng CNN chỉ nên dùng cho dữ liệu có kích thước thay đổi nhưng cùng kiểu thông tin—ví dụ, chuỗi thời gian dài ngắn khác nhau hoặc ảnh to nhỏ khác nhau. Nếu mỗi dữ liệu có các loại đặc trưng hoàn toàn khác nhau (như hồ sơ thí sinh mà người này có điểm thi chuẩn hóa, người kia không có), thì không thể dùng cùng một bộ lọc CNN áp lên các thuộc tính đó một cách hợp lý.

Loại dữ liệu	Một kênh (Single channel)	Đa kênh (Multichannel)
1 chiều (1-D)	Dạng sóng âm thanh: Trục được tích chập là trục thời gian. Âm thanh được rời rạc theo thời gian, và mỗi thời điểm có một giá trị biên độ.	Hoạt hình khung xương: Nhân vật 3D chuyển động theo thời gian bằng cách thay đổi góc các khớp xương. Mỗi kênh là một góc quay quanh một trục tại một khớp.
2 chiều (2-D)	Âm thanh đã biến đổi Fourier: Dữ liệu được biểu diễn dưới dạng ma trận 2 chiều, với trục là tần số và thời gian. Tích chập theo thời gian giúp mô hình dịch biến theo thời gian; tích chập theo tần số giúp mô hình không bị ảnh hưởng bởi cao độ.	Ảnh màu: Mỗi kênh lưu giá trị màu đỏ, xanh lá hoặc xanh dương. Tích chập theo cả chiều ngang và dọc giúp học các đặc trưng không gian bất biến với dịch chuyển.
3 chiều (3-D)	Dữ liệu thể tích: Thường xuất hiện trong hình ảnh y tế như ảnh chụp CT hoặc MRI, có ba chiều không gian.	Video màu: Một trục là thời gian, hai trục còn lại là chiều cao và chiều rộng khung hình.

Bảng 9.1: Các ví dụ về định dạng dữ liệu 1D, 2D, và 3D có thể dùng trong mạng nơ-ron tích chập, với cả trường hợp một kênh và đa kênh.

9.8 Các thuật toán tích chập hiệu quả

Trong các ứng dụng hiện đại, mạng nơ-ron tích chập thường có quy mô rất lớn, với hàng triệu đơn vị. Để xử lý được lượng tính toán này, cần tận dụng các tài nguyên tính toán song song, như đã trình bày trong mục 12.1. Tuy nhiên, không phải lúc nào phần cứng mạnh cũng sẵn có, nên việc tối ưu hóa chính thuật toán tích chập cũng có thể giúp tăng tốc đáng kể.

Một cách để tăng tốc là chuyển đầu vào và bộ lọc sang miền tần số bằng biến đổi Fourier, sau đó thực hiện phép nhân điểm, rồi biến đổi ngược trở lại miền không gian. Với một số kích thước nhất định, phương pháp này có thể nhanh hơn so với tính tích chập trực tiếp trong miền không gian.

Một cách khác để tăng tốc là khai thác tính chất phân tách của kernel. Nếu một kernel d chiều có thể viết thành tích ngoài của d vector, mỗi vector tương ứng với một chiều, thì kernel đó được gọi là *separable* (có thể phân tách). Khi đó, ta có thể thay thế tích chập nhiều chiều bằng việc thực hiện liên tiếp các phép tích chập một chiều với từng vector. Cách này nhanh hơn rất nhiều so với tích chập toàn phần.

Cụ thể, nếu kernel có độ rộng w ở mỗi chiều, thì tích chập thông thường cần thời gian và bộ nhớ là $O(w^d)$. Ngược lại, với kernel phân tách, chỉ cần $O(w \times d)$. Tuy nhiên, không

phải kernel nào cũng có thể biểu diễn theo dạng này.

Hiện nay, việc tìm ra các phương pháp tích chập nhanh hoặc xấp xỉ mà vẫn đảm bảo độ chính xác mô hình là một chủ đề nghiên cứu được quan tâm. Các kỹ thuật chỉ tối ưu quá trình truyền xuôi cũng đặc biệt hữu ích, vì trong môi trường ứng dụng thực tế, việc triển khai nhanh và hiệu quả mô hình thường được ưu tiên hơn quá trình huấn luyện.

9.9 Tính năng ngẫu nhiên hoặc học không giám sát

Phần tốn kém nhất trong huấn luyện mạng CNN thường nằm ở việc học các tính năng (feature). Trong khi đó, lớp đầu ra thường nhẹ hơn vì số lượng đầu vào cho lớp này đã được giảm đi nhiều nhờ các lớp pooling. Nếu sử dụng huấn luyện có giám sát với gradient descent, thì mỗi bước cần chạy đầy đủ cả truyền xuôi và truyền ngược qua toàn mạng, gây tốn tài nguyên. Một cách để giảm chi phí này là sử dụng các tính năng không học theo cách có giám sát. Có ba chiến lược cơ bản để thu được các kernel mà không cần dùng đến dữ liệu gán nhãn:

- Khởi tạo ngẫu nhiên các kernel.
- Thiết kế thủ công các kernel để phát hiện cạnh, đường viền hoặc các mẫu đơn giản.
- Học kernel bằng tiêu chí không giám sát.

Thật bất ngờ là các kernel ngẫu nhiên vẫn có thể cho kết quả tốt trong thực tế (Jarrett et al., 2009; Saxe et al., 2011; Pinto et al., 2011; Cox và Pinto, 2011). Ví dụ, Saxe et al. (2011) cho thấy rằng khi ta kết hợp các lớp tích chập với pooling và khởi tạo trọng số ngẫu nhiên, mạng vẫn có thể học được các bộ lọc tần số và có tính bất biến với dịch chuyển. Đây là một cách đơn giản và rẻ tiền để xây dựng kiến trúc mạng.

Một cách tiếp cận ở giữa là học các tính năng theo phương pháp không yêu cầu truyền xuôi và truyền ngược đầy đủ mỗi lần cập nhật. Tương tự như cách huấn luyện tham lam từng lớp trong mạng perceptron đa lớp, ta huấn luyện lớp đầu tiên riêng biệt, sau đó trích xuất đặc trưng từ lớp đó, rồi huấn luyện lớp kế tiếp cũng một cách riêng biệt, và tiếp tục như vậy.

Việc học không giám sát có thể giúp điều chỉnh mô hình tốt hơn (regularization) so với huấn luyện có giám sát, hoặc đơn giản là giúp huấn luyện các mạng lớn hơn nhờ chi phí tính toán thấp hơn.

9.10 Cơ sở thần kinh học cho Mạng nơ-ron tích chập

Mạng nơ-ron tích chập là một trong những thành tựu nổi bật nhất của trí tuệ nhân tạo lấy cảm hứng từ sinh học. Mặc dù CNN được xây dựng từ nhiều lĩnh vực khác nhau, nhưng một số nguyên lý cốt lõi trong thiết kế mạng đã xuất phát từ nghiên cứu về thần kinh học.

Nguồn gốc của CNN có thể được truy về các thí nghiệm thần kinh học từ nhiều năm trước khi mô hình tính toán được hình thành. Các nhà khoa học thần kinh David Hubel và Torsten Wiesel đã tiến hành các nghiên cứu kéo dài nhằm khám phá cơ chế hoạt động của

hệ thống thị giác ở động vật có vú (Hubel và Wiesel, 1959, 1962, 1968). Họ đã giành giải Nobel nhờ những đóng góp nền tảng này. Bằng cách ghi nhận hoạt động của các tế bào thần kinh đơn lẻ trong não mèo, họ phát hiện rằng một số tế bào phản ứng rất mạnh với các mẫu ánh sáng cụ thể, ví dụ như các đường thẳng nghiêng theo hướng nhất định, trong khi hoàn toàn không phản ứng với những mẫu ánh sáng khác.

Phát hiện này là nền tảng cho rất nhiều khái niệm trong học sâu. Từ góc độ học sâu, ta có thể tập trung vào vùng V1 — vỏ não thị giác chính — nơi đầu tiên trong não xử lý thông tin thị giác.

Một cách đơn giản để hình dung là khi mắt nhìn thấy một đối tượng, thông tin từ võng mạc sẽ truyền qua các vùng thị giác và đến V1. Tại đây, các tế bào thần kinh phản ứng với các đặc trưng cơ bản như cạnh và đường thẳng trong ảnh. Sau đó, các tín hiệu này được chuyển tiếp đến các vùng não cao hơn để xử lý thêm — giống như cách các tầng sau trong CNN xử lý các đặc trưng ngày càng phức tạp hơn.

Phương pháp *reverse correlation* chỉ ra rằng phần lớn các tế bào V1 có thể được mô tả bằng các *hàm Gabor*. Hàm Gabor xác định trọng số tại mỗi điểm ảnh 2D trong ảnh đầu vào. Ta có thể coi ảnh là một hàm $I(x, y)$ theo toạ độ không gian, và tế bào thần kinh đơn giản sẽ áp dụng một hàm trọng số $w(x, y)$ lên ảnh tại các vị trí khác nhau. Đáp ứng của tế bào được tính bằng:

$$s(I) = \sum_{x \in X} \sum_{y \in Y} w(x, y) I(x, y).$$

Trong đó, hàm trọng số $w(x, y)$ có dạng:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(f x' + \phi),$$

với:

$$x' = (x - x_0) \cos(\tau) + (y - y_0) \sin(\tau),$$

và:

$$y' = -(x - x_0) \sin(\tau) + (y - y_0) \cos(\tau).$$

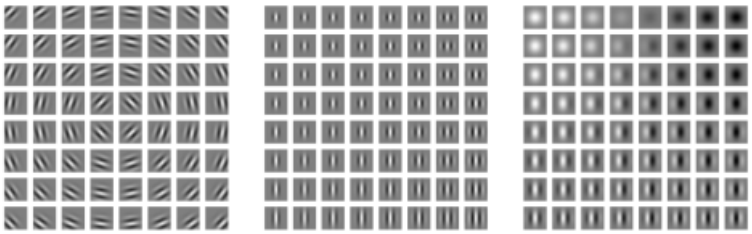
Các tham số $\alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau$ điều chỉnh hình dạng và vị trí của hàm Gabor. Ví dụ, x_0, y_0 xác định vị trí trung tâm của vùng cảm nhận, còn τ xác định hướng (góc xoay) mà tế bào phản ứng mạnh nhất.

Hàm Gabor có hai thành phần chính: một hàm Gaussian và một hàm cosine. Thành

phần Gaussian:

$$\alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right)$$

giúp tập trung phản ứng của tế bào vào các điểm gần trung tâm vùng cảm nhận (nơi $x' = y' = 0$). Tham số α điều chỉnh cường độ phản ứng tổng thể, còn β_x và β_y điều chỉnh độ rộng của vùng cảm nhận theo hai chiều.



Hình 9.18: Các hàm Gabor với các tham số khác nhau. Màu trắng: trọng số dương lớn; màu đen: trọng số âm lớn; màu xám: trọng số gần bằng 0. Mặt Trái: Các tham số tọa độ x_0, y_0 và góc xoay τ thay đổi. Mặt Giữa: Các tham số điều chỉnh độ rộng Gaussian β_x, β_y thay đổi. Mặt Phải: Các tham số tần số và pha trong hàm cosine, f và ϕ, được thay đổi.

Yếu tố cosine $\cos(fx' + \phi)$ điều khiển cách mà tế bào đơn phản ứng với sự thay đổi độ sáng theo phương x'. Tham số f xác định tần số không gian, trong khi ϕ điều chỉnh độ lệch pha của tín hiệu.

Tổng thể, một tế bào đơn sẽ phản ứng mạnh nhất với các mẫu ánh sáng có tần số không gian và hướng cụ thể tại một vị trí xác định. Phản ứng đạt cực đại khi sóng độ sáng trong ảnh trùng pha với trọng số của tế bào — tức là ảnh sáng tại các vùng có trọng số dương và tối tại các vùng có trọng số âm. Ngược lại, khi hình ảnh lệch pha hoàn toàn so với trọng số — tức là ảnh tối tại các vùng có trọng số dương và ngược lại — tế bào sẽ bị ức chế mạnh.

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(fx' + \phi)$$

Trong khi tế bào đơn phản ứng với một hướng cụ thể và một pha nhất định, tế bào phức tạp có thể phản ứng với cùng một hướng mà không nhạy với pha. Điều này đạt được bằng cách tính chuẩn L2 từ hai tế bào đơn lệch pha nhau một phần tư chu kỳ:

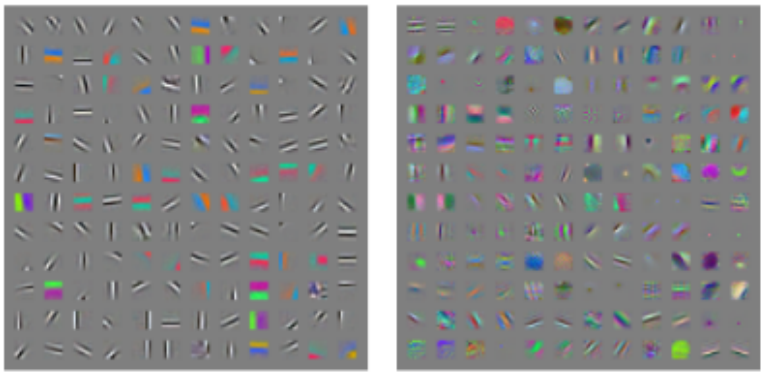
$$c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$$

Một trường hợp đặc biệt là khi hai phản ứng s_0 và s_1 chỉ khác nhau ở pha ϕ, và lệch pha một góc $\pi/2$, tạo thành một cặp quadrature.

Một điểm tương đồng đáng chú ý giữa học máy và thần kinh học là các đặc trưng mà các mô hình học được ở lớp đầu thường giống với các đặc trưng trong vùng V1. Olshausen và Field (1996) đã chứng minh rằng một thuật toán học không giám sát đơn giản — sparse

coding — có thể học được các đặc trưng gần giống với tế bào đơn.

Nhiều mô hình học sâu hiện đại cũng học ra những đặc trưng tương tự ở các lớp đầu tiên. Tuy nhiên, việc một mô hình học được các đặc trưng giống sinh học không có nghĩa là nó thực sự mô phỏng lại cách hoạt động của não người.



Hình 9.19: Nhiều thuật toán học máy học được các bộ lọc phát hiện cạnh hoặc màu sắc tương tự các hàm Gabor có trong vỏ não thị giác V1. Bên trái: Trọng số học được bởi một thuật toán học không giám sát từ các ảnh nhỏ. Bên phải: Các bộ lọc của lớp đầu tiên trong một mạng convolutional maxout huấn luyện đầy đủ. Các bộ lọc liền kề điều khiển cùng một đơn vị maxout.

9.11 Mạng tích chập trong lịch sử phát triển của học sâu

Mạng tích chập đã có một vai trò quan trọng trong lịch sử hình thành và phát triển của học sâu. Đây là một ví dụ điển hình về việc vận dụng kiến thức từ thần kinh học vào thiết kế mô hình học máy. CNN cũng là một trong những mô hình đầu tiên thành công trong các bài toán thị giác máy tính, và là mô hình học sâu đầu tiên được ứng dụng thương mại rộng rãi — và vẫn đóng vai trò quan trọng cho đến ngày nay.

Ví dụ, trong những năm 1990, nhóm nghiên cứu tại AT&T đã xây dựng một mạng tích chập để đọc chữ viết tay trên séc ngân hàng **LeCun1998b**. Cuối thập kỷ, hệ thống này được triển khai bởi công ty NCR và được sử dụng để đọc hơn 10% số lượng séc ở Mỹ. Sau đó, Microsoft cũng áp dụng mạng tích chập trong các hệ thống nhận dạng chữ viết tay và OCR **Simard2003**. Thêm chi tiết về các ứng dụng này có thể tham khảo ở chương 12 hoặc LeCun et al. (2010) để biết lịch sử chi tiết đến năm 2010.

Mạng tích chập cũng giành chiến thắng trong nhiều cuộc thi học máy, nổi bật nhất là chiến thắng trong cuộc thi ImageNet năm 2012 (Krizhevsky et al., 2012), được xem như khởi đầu cho làn sóng thương mại hóa học sâu. Tuy nhiên, CNN đã được sử dụng thành công từ trước đó để chiến thắng các cuộc thi về học máy và thị giác.

Mạng tích chập cũng là một trong những mạng đầu tiên được huấn luyện thành công bằng thuật toán lan truyền ngược (back-propagation). Thật thú vị là trong khi các mạng fully connected tổng quát khi đó bị xem là khó huấn luyện, thì CNN lại hoạt động tốt. Có thể lý do là CNN sử dụng ít tham số hơn nhờ chia sẻ trọng số, nên dễ thử nghiệm và tinh chỉnh hơn. Thêm vào đó, CNN cũng dễ mở rộng khi có phần cứng mạnh.

Ngày nay, với phần cứng hiện đại và kỹ thuật huấn luyện tiên tiến, cả các mạng fully connected cũng có thể hoạt động tốt. Điều này khiến nhiều người tin rằng những rào cản trước đây phần lớn là do định kiến — nhiều người không tin vào mạng nơ-ron nên không đầu tư nghiêm túc để khai thác chúng. Nhưng nhờ vào hiệu quả ổn định của CNN trong nhiều thập kỷ, chúng đã mở đường cho sự phát triển của học sâu và trí tuệ nhân tạo hiện đại.

CNN mang đến một cách chuyên biệt hóa cho mạng nơ-ron xử lý dữ liệu có cấu trúc dạng lưới — ví dụ như ảnh số. Ảnh số thường được biểu diễn dưới dạng ma trận số, mỗi phần tử là độ sáng của một pixel. CNN cho phép xử lý hiệu quả và mở rộng đến những mạng có kích thước rất lớn.

Phương pháp này đặc biệt hiệu quả với dữ liệu không gian như ảnh. Còn với dữ liệu tuần tự một chiều (như văn bản hay âm thanh), một kiến trúc mạng khác — mạng nơ-ron hồi tiếp (Recurrent Neural Networks — RNN) — được thiết kế riêng để xử lý dạng dữ liệu này. RNN sẽ được giới thiệu chi tiết trong chương tiếp theo.

CHƯƠNG 10. MẠNG NƠ-RON HỒI TIẾP

Mạng nơ-ron hồi tiếp (Recurrent Neural Networks - RNNs), được giới thiệu bởi Rumelhart và cộng sự vào năm 1986 [16], là một loại mạng nơ-ron được thiết kế riêng cho việc xử lý dữ liệu theo chuỗi. Nếu như mạng nơ-ron tích chập (CNN) phù hợp với dữ liệu dạng lưới như hình ảnh, thì RNN lại rất hiệu quả với các dữ liệu tuần tự theo thời gian như văn bản, âm thanh hay tín hiệu từ cảm biến.

Tương tự như việc CNN có thể mở rộng để xử lý hình ảnh có kích thước lớn hoặc thay đổi, RNN cũng có khả năng xử lý các chuỗi có độ dài khác nhau, thậm chí rất dài – điều mà các mạng truyền thẳng thông thường khó làm được.

Một yếu tố quan trọng giúp phân biệt RNN với mạng nhiều lớp thông thường chính là cơ chế chia sẻ tham số (parameter sharing). Ý tưởng này đã xuất hiện từ những năm 1980 trong học máy, và là nền tảng để RNN có thể hoạt động hiệu quả. Thay vì gán một bộ tham số riêng cho từng thời điểm trong chuỗi, RNN sử dụng cùng một tập tham số cho tất cả các bước thời gian. Điều này không chỉ giúp mô hình có khả năng tổng quát tốt với các chuỗi dài chưa từng gặp, mà còn tận dụng được sự lặp lại về mặt thống kê giữa các vị trí trong chuỗi.

Việc chia sẻ tham số là đặc biệt quan trọng trong những trường hợp thông tin quan trọng có thể xuất hiện ở bất kỳ đâu trong chuỗi. Ví dụ, hãy xem xét hai câu sau:

- "Tôi đã đến Nepal vào năm 2009."
- "Vào năm 2009, tôi đã đến Nepal."

Cả hai câu đều chứa thông tin chính là "năm 2009". Một mô hình tốt cần nhận ra được thông tin này, dù nó nằm ở đầu hay cuối câu. Nếu dùng mạng truyền thẳng với đầu vào có chiều dài cố định, mỗi vị trí trong câu sẽ có một bộ tham số riêng, khiến mô hình phải học cách xử lý ngôn ngữ cho từng vị trí cụ thể – điều này rất kém hiệu quả. Ngược lại, RNN với cơ chế chia sẻ tham số sẽ học một cách duy nhất để xử lý mọi bước thời gian, giúp mô hình linh hoạt và gọn nhẹ hơn nhiều.

Một hướng tiếp cận có liên quan đến RNN là áp dụng phép tích chập trên chuỗi thời gian một chiều, và đây cũng chính là nền tảng cho các mạng nơ-ron trễ thời gian (Time-Delay Neural Networks – TDNNs) [17][18][19].

Trong phương pháp này, ta chia sẻ tham số bằng cách áp dụng cùng một bộ lọc (kernel) tại mỗi thời điểm trong chuỗi – tương tự như cách CNN hoạt động trên hình ảnh. Tuy nhiên, cách làm này vẫn là một mô hình *nông*, bởi vì đầu ra tại mỗi thời điểm chỉ bị ảnh hưởng bởi một vùng lân cận của đầu vào.

Trong khi đó, mạng nơ-ron hồi tiếp (RNN) chia sẻ tham số theo một cách khác. Mỗi đầu ra tại thời điểm hiện tại không chỉ phụ thuộc vào đầu vào tại thời điểm đó, mà còn phụ thuộc vào trạng thái (đầu ra) của các thời điểm trước, thông qua cùng một công thức

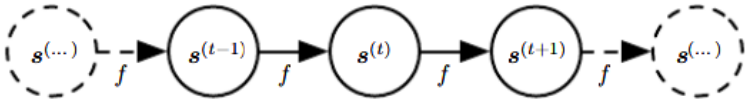
cập nhật lặp lại. Cách tính này tạo ra một đồ thị tính toán sâu, trong đó các giá trị được cập nhật nối tiếp nhau qua thời gian, với mỗi bước đều liên kết chặt chẽ với bước trước đó.

Để hình dung dễ hơn, giả sử RNN đang xử lý một chuỗi các vector $\mathbf{x}(t)$, với chỉ số thời gian $t \in \{1, 2, \ldots, \tau\}$. Trong thực tế, ta thường xử lý đồng thời nhiều chuỗi trong một minibatch, mỗi chuỗi có thể có độ dài khác nhau. Nhưng để đơn giản hóa trình bày, phần này sẽ bỏ qua chỉ số của minibatch. Hơn nữa, chỉ số thời gian t trong RNN không nhất thiết phải thể hiện thời gian thực — đôi khi nó đơn giản chỉ là vị trí của phần tử trong chuỗi. RNN cũng có thể được mở rộng để áp dụng cho dữ liệu hai chiều như hình ảnh, hoặc trong các trường hợp mà toàn bộ chuỗi đầu vào đã có sẵn từ trước, ta còn có thể thêm vào các liên kết ngược theo thời gian để mô hình có thể khai thác thông tin từ cả hai chiều thời gian.

Trong chương này, chúng ta sẽ mở rộng khái niệm về đồ thị tính toán để bao gồm cả các chu trình (cycles) — tức là trường hợp một biến tại thời điểm hiện tại có thể ảnh hưởng đến chính nó trong tương lai. Việc này là cơ sở để định nghĩa các mạng nơ-ron hồi tiếp, nơi thông tin được lưu giữ và lan truyền theo thời gian thông qua các kết nối lặp lại.

Ở phần cuối chương, chúng ta sẽ cùng nhau tìm hiểu nhiều cách khác nhau để xây dựng, huấn luyện và ứng dụng mạng RNN trong thực tế.

Đối với bạn đọc muốn đi sâu hơn vào chi tiết về mạng nơ-ron hồi tiếp, có thể tham khảo thêm tài liệu của Graves [20], một trong những nguồn kinh điển trong lĩnh vực này.



Hình 10.1: Một hệ động lực cổ điển được mô tả bởi phương trình (10.1), minh họa dưới dạng đồ thị tính toán được trải phẳng theo thời gian. Mỗi nút trong hình biểu diễn trạng thái tại một thời điểm t, và hàm f dùng để tính trạng thái tại thời điểm $t + 1$ dựa trên trạng thái ở thời điểm t. Bộ tham số θ được giữ nguyên và sử dụng cho tất cả các bước thời gian.

10.1 Triển khai đồ thị tính toán

Đồ thị tính toán (computational graph) là một cách biểu diễn trực quan quá trình tính toán, từ đầu vào và tham số đến đầu ra và hàm mất mát. Khái niệm này đã được giới thiệu tổng quát trong Mục 6.5.1.

Trong phần này, chúng ta sẽ tìm hiểu khái niệm triển khai (*unfolding*) — tức là mở rộng một phép tính mang tính đệ quy hoặc hồi quy thành một chuỗi các bước cụ thể, thể hiện dưới dạng đồ thị tính toán lặp lại. Mỗi bước trong chuỗi này thường tương ứng với một thời điểm cụ thể trong chuỗi dữ liệu. Việc triển khai như vậy giúp chia sẻ tham số qua thời gian và tạo ra một mạng sâu có cấu trúc lặp.

Giả sử ta có một hệ thống động học được mô tả bằng phương trình:

$$s(t) = f(s(t-1); \theta), \tag{10.1}$$

trong đó:

- $s(t)$ là trạng thái của hệ tại thời điểm t,
- θ là bộ tham số của hàm f.

Đây là một phương trình hồi quy: trạng thái hiện tại $s(t)$ phụ thuộc vào trạng thái ngay trước đó $s(t-1)$.

Nếu chỉ xét một số hữu hạn bước thời gian — giả sử là τ bước — ta có thể triển khai quá trình này bằng cách lặp lại công thức trên $\tau - 1$ lần. Chẳng hạn, nếu $\tau = 3$, ta có:

$$s(3) = f(s(2); \theta) \tag{10.2}$$

$$= f(f(s(1); \theta); \theta). \tag{10.3}$$

Khi triển khai theo cách này, quá trình tính toán sẽ tạo thành một đồ thị có hướng không chu trình (DAG), vì không còn vòng lặp trong biểu diễn nữa.

Hình 10.1 minh họa việc triển khai đồ thị cho phương trình trên.

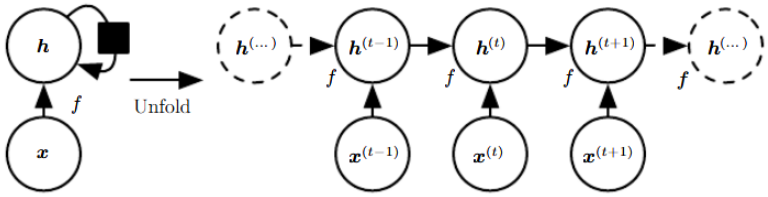
Tiếp theo, nếu hệ thống chịu ảnh hưởng của một tín hiệu đầu vào $x(t)$ tại mỗi bước, thì công thức trở thành:

$$s(t) = f(s(t-1), x(t); \theta), \tag{10.4}$$

trong đó:

- Trạng thái $s(t)$ phụ thuộc không chỉ vào trạng thái $s(t-1)$ mà còn vào tín hiệu đầu vào $x(t)$.

Điều này có nghĩa là $s(t)$ đang tích lũy và lưu trữ thông tin từ toàn bộ chuỗi đầu vào đã quan sát trước đó.



Hình 10.2: Một mạng hồi quy không sinh đầu ra, chỉ có tác dụng truyền và xử lý thông tin từ chuỗi đầu vào x, thông qua một trạng thái ẩn h lan truyền theo thời gian. (Trái): Biểu diễn dưới dạng mạch, mô phỏng như một hệ thống sinh học hoặc vật lý. Các hình vuông đen biểu diễn độ trễ một bước thời gian. (Phải): Biểu diễn dạng đồ thị tính toán đã được trải thẳng theo thời gian, giúp hình dung rõ cách trạng thái h thay đổi tại từng thời điểm.

Mạng nơ-ron hồi tiếp (RNN) là một kiến trúc học sâu rất quan trọng, đặc biệt hiệu quả với dữ liệu tuần tự như văn bản, âm thanh hoặc chuỗi thời gian. Khác với mạng truyền thẳng (feedforward), RNN có khả năng ghi nhớ thông tin quá khứ thông qua trạng thái ẩn, được cập nhật qua từng bước thời gian.

RNN có thể được định nghĩa bởi công thức:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{10.5}$$

Trong đó:

- $h^{(t)}$ là trạng thái ẩn tại thời điểm t,
- $x^{(t)}$ là đầu vào tại thời điểm t,
- θ là tập tham số của mô hình,
- f là hàm phi tuyến, như tanh hoặc ReLU.

Mỗi trạng thái $h^{(t)}$ là một bản tóm tắt (summary) của toàn bộ chuỗi dữ liệu từ thời điểm đầu tiên đến hiện tại. Tuy nhiên, vì tóm tắt này được nén vào một vector duy nhất nên chắc chắn sẽ mất mát một phần thông tin.

Tùy thuộc vào nhiệm vụ, RNN có thể học cách tập trung vào những phần quan trọng của chuỗi và bỏ qua phần không cần thiết. Ví dụ:

- Trong dự đoán từ tiếp theo (mô hình ngôn ngữ), chỉ cần nhớ thông tin đủ để dự đoán từ sau.
- Trong autoencoder tuần tự (sẽ trình bày ở chương 14), $h^{(t)}$ phải đủ thông tin để tái tạo toàn bộ chuỗi.

Công thức hồi tiếp của RNN có thể biểu diễn theo hai cách:

1. Biểu diễn mạch hồi tiếp: Sử dụng một nút duy nhất với kết nối vòng, biểu diễn dòng chảy thông tin qua thời gian.
2. Biểu diễn trải thẳng: Tách mỗi bước thời gian thành một bản sao riêng biệt của hàm f, giúp minh họa rõ luồng thông tin từ quá khứ đến hiện tại. Cách này còn gọi là *unfolding*.

Khi unfolding đến thời điểm t, ta có:

$$h^{(t)} = g^{(t)}(x^{(1)}, x^{(2)}, \ldots, x^{(t)}) \tag{10.6}$$

$$= f(h^{(t-1)}, x^{(t)}; \theta) \tag{10.7}$$

Trong đó, $g^{(t)}$ là một hàm tổng hợp chuỗi, được xây dựng từ các lần áp dụng lặp lại của f.

Triển khai RNN theo thời gian có hai lợi ích lớn:

1. Mỗi bước xử lý một đầu vào cố định, bất kể độ dài chuỗi.
2. Cùng một hàm f và tham số θ dùng cho mọi bước — giúp chia sẻ và giảm số lượng tham số cần học.

Nhờ đó, RNN có khả năng tổng quát hóa với chuỗi dài hơn dữ liệu huấn luyện và yêu cầu ít dữ liệu hơn để học hiệu quả.

- Biểu diễn hồi tiếp: ngắn gọn, trực quan về cấu trúc.
- Biểu diễn trải thẳng: rõ ràng về quá trình tính toán, đặc biệt hữu ích khi tính gradient trong huấn luyện.

10.2 Mạng nơ-ron hồi tiếp (Recurrent Neural Networks)

Dựa trên hai khái niệm đã học là triển khai qua thời gian và chia sẻ tham số, ta có thể xây dựng nhiều dạng mạng RNN khác nhau tùy theo bài toán.

Một số cấu trúc RNN phổ biến gồm:

- Sinh đầu ra ở mỗi bước, có kết nối hồi tiếp giữa các trạng thái ẩn — thường thấy trong mô hình ngôn ngữ, dịch máy, nhận dạng giọng nói... (Hình **??**).
- Sinh đầu ra ở mỗi bước, nhưng hồi tiếp từ đầu ra trước tới trạng thái ẩn hiện tại — đơn giản hóa cấu trúc, đôi khi phù hợp hơn cho các bài toán đơn giản (Hình **??**).
- Chỉ sinh một đầu ra sau khi đọc toàn bộ chuỗi, phù hợp cho bài toán phân loại chuỗi như phân tích cảm xúc (Hình **??**).

Dạng RNN trong Hình **??** và công thức (10.8) có khả năng tính toán phổ quát — về lý thuyết, có thể tính được bất kỳ hàm nào mà một máy Turing có thể tính, với một mạng RNN hữu hạn kích thước:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{10.8}$$

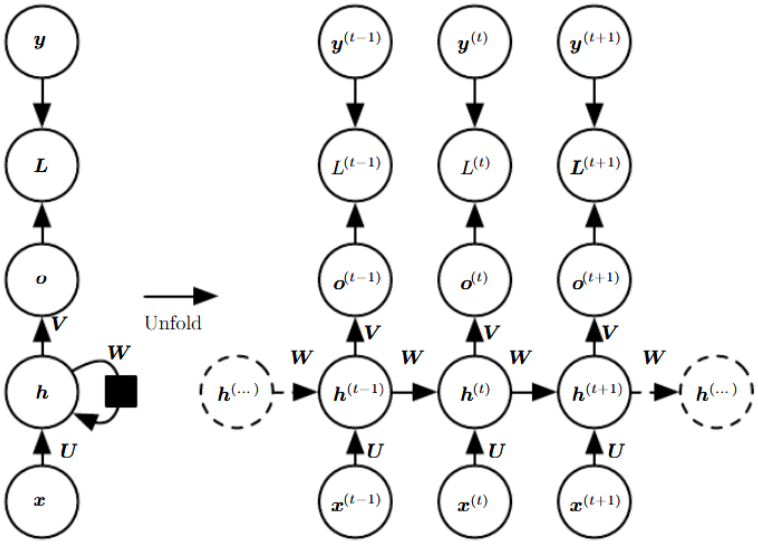
Dù đây là kết quả mang tính lý thuyết, nó cho thấy sức mạnh tiềm năng của RNN — miễn là có đủ bước thời gian và cơ chế lượng tử hóa đầu ra phù hợp (theo Siegelmann và Sontag, 1991, 1995).

Trong thiết lập mô phỏng máy Turing, ta có thể sử dụng một mạng RNN để thực hiện bất kỳ phép tính nào mà một máy Turing có thể làm được, với một số giả định:

- Mạng RNN nhận chuỗi đầu vào dưới dạng nhị phân.
- Đầu ra của mạng cũng được lượng tử hóa thành nhị phân, bằng cách rời rạc hóa xác suất hoặc dùng ngưỡng.
- Với một mạng RNN duy nhất có kích thước cố định, ta có thể mô phỏng bất kỳ máy Turing nào — nghĩa là mạng này có thể được dùng để giải bất kỳ bài toán nào, chỉ

cần thay đổi chuỗi nhị phân đầu vào để mô tả từng bài toán cụ thể.

Ví dụ, trong nghiên cứu nổi tiếng của Siegelmann và Sontag (1995), một mạng RNN với chỉ 886 đơn vị ẩn đã đủ khả năng mô phỏng mọi máy Turing. Mạng này dùng số hữu tỉ với độ chính xác tùy ý để biểu diễn các trạng thái và trọng số, từ đó mô phỏng cả những cấu trúc như ngăn xếp không giới hạn của máy Turing.



Hình 10.3: Đồ thị tính toán cho quá trình huấn luyện một mạng RNN. Mạng ánh xạ chuỗi đầu vào x sang chuỗi đầu ra o, sau đó tính hàm mất mát L bằng cách so sánh đầu ra o với nhãn mục tiêu y. Nếu dùng softmax cho đầu ra, thì o là các log xác suất chưa chuẩn hóa, và L sẽ tính $\hat{y} = \text{softmax}(o)$ để so sánh với y. Ba loại kết nối được tham số hóa bằng các ma trận: U cho kết nối từ đầu vào đến ẩn, W cho kết nối ẩn-ẩn, và V cho kết nối từ ẩn đến đầu ra. (Trái) là sơ đồ mạch có hồi tiếp, (phải) là đồ thị tính toán đã trải thẳng theo thời gian.

Trong phần này, chúng ta phát triển các phương trình truyền tiến (forward propagation) cho mạng RNN như trong Hình 10.3. Hình này không mô tả rõ hàm kích hoạt, nhưng ta có thể giả sử sử dụng hàm `tanh`. Đầu ra cũng không được xác định rõ, nên giả sử rằng ta đang làm việc với các biến rời rạc như từ ngữ hoặc ký tự.

Trong trường hợp này, một cách hợp lý để biểu diễn đầu ra là dùng $o(t)$ như các log xác suất chưa chuẩn hóa, sau đó áp dụng hàm `softmax` để tính xác suất chuẩn hóa:

$$a(t) = b + Wh(t-1) + Ux(t), \tag{10.8}$$

$$h(t) = \tanh(a(t)), \tag{10.9}$$

$$o(t) = c + Vh(t), \tag{10.10}$$

$$\hat{y}(t) = \text{softmax}(o(t)). \tag{10.11}$$

Trong đó:

- b, c: vector bias cho lớp ẩn và lớp đầu ra.

- W: ma trận trọng số cho kết nối hồi tiếp giữa các trạng thái ẩn.
- U: ma trận trọng số từ đầu vào đến lớp ẩn.
- V: ma trận trọng số từ lớp ẩn đến lớp đầu ra.

Mạng này ánh xạ một chuỗi đầu vào sang một chuỗi đầu ra có cùng độ dài. Tổng hàm mất mát trên toàn bộ chuỗi là tổng các mất mát tại từng bước thời gian:

$$L\left(\{x(1),\ldots,x(\tau)\},\{y(1),\ldots,y(\tau)\}\right)=\sum_{t}L(t) \tag{10.13}$$

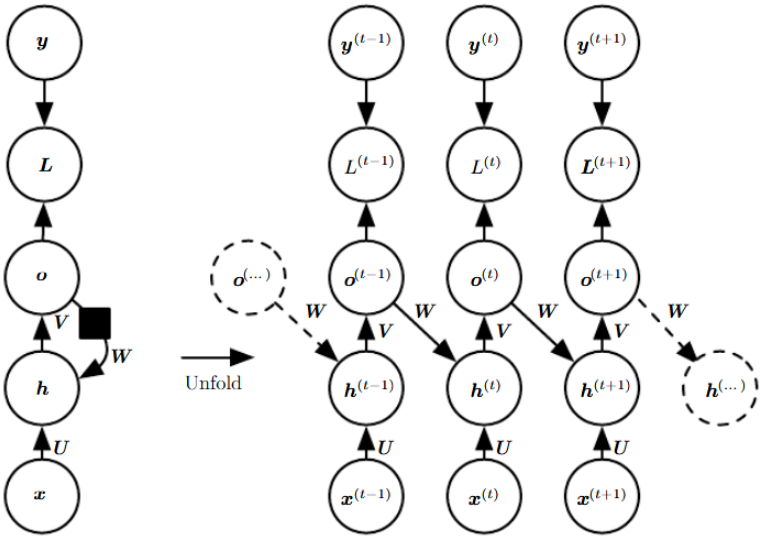
Trong đó:

$$L(t)=-\log p_{\text{model}}\left(y(t)\mid\{x(1),\ldots,x(t)\}\right). \tag{10.14}$$

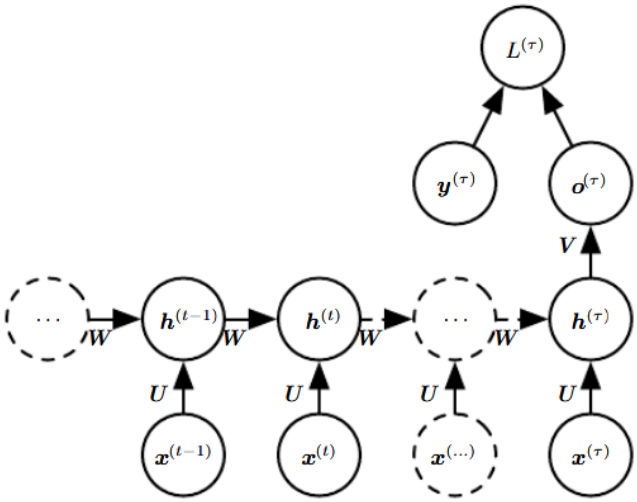
Xác suất này được trích xuất từ thành phần tương ứng trong vector $\hat{y}(t)$ đầu ra.

Để huấn luyện mạng, ta cần tính gradient của hàm mất mát này với các tham số. Quá trình gồm hai bước: truyền tiến từ trái sang phải, sau đó lan truyền ngược (backpropagation) từ phải sang trái. Vì mỗi bước phụ thuộc vào bước trước đó, nên cả quá trình cần $O(\tau)$ thời gian và không thể song song hóa. Ngoài ra, ta cũng cần lưu lại tất cả trạng thái $h(t)$ để phục vụ quá trình lan truyền ngược, do đó chi phí bộ nhớ cũng là $O(\tau)$.

Phương pháp lan truyền ngược này, áp dụng cho mạng đã được trải thẳng theo thời gian, gọi là lan truyền ngược qua thời gian (Backpropagation Through Time – BPTT). Kỹ thuật này sẽ được trình bày cụ thể hơn trong Mục 10.2.2. Dù RNN với kết nối hồi tiếp mạnh mẽ trong khả năng mô hình hóa, chúng lại có chi phí huấn luyện khá lớn.



Hình 10.4: Một biến thể của mạng RNN, trong đó hồi tiếp chỉ xảy ra từ đầu ra trở lại lớp ẩn. (Trái) là sơ đồ mạch, (phải) là biểu diễn đồ thị trải thẳng. Mạng này kém mạnh mẽ hơn mạng trong Hình 10.3, vì không có hồi tiếp trực tiếp giữa các trạng thái ẩn. Do đó, thông tin từ quá khứ chỉ được truyền gián tiếp thông qua các đầu ra. Nếu đầu ra không đủ phong phú, mạng sẽ mất thông tin quan trọng từ quá khứ. Tuy nhiên, ưu điểm là các bước thời gian có thể huấn luyện gần như độc lập, từ đó cho phép song song hóa tốt hơn (chi tiết trong Mục 10.2.1).



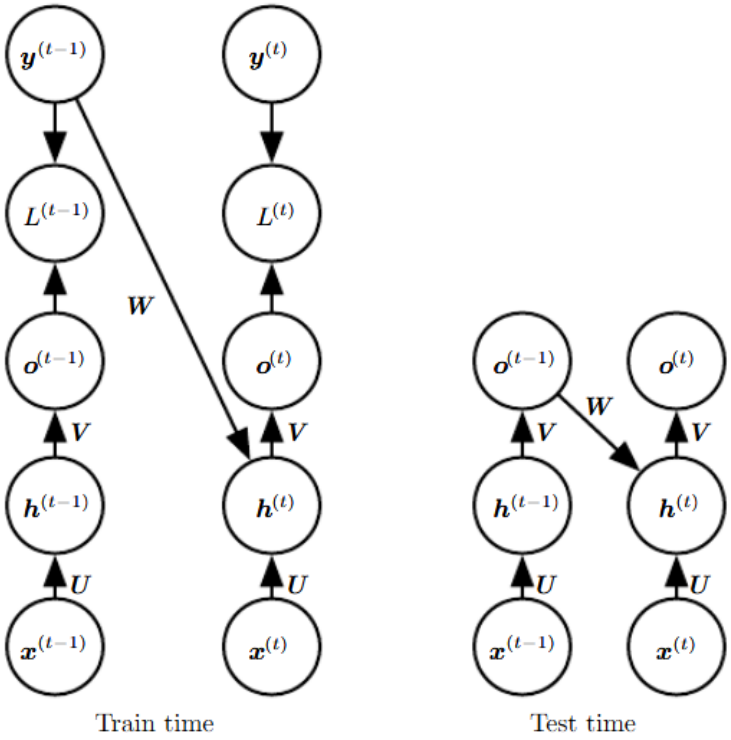
Hình 10.5: Biến thể khác của RNN, chỉ sinh một đầu ra duy nhất ở cuối chuỗi. Mạng như vậy thường dùng để tóm tắt toàn bộ chuỗi thành một vector cố định, sau đó sử dụng cho các tác vụ khác. Có thể có mục tiêu tại bước cuối cùng, hoặc đầu ra $o(t)$ sẽ được lan truyền gradient ngược từ các mô-đun phía sau trong mô hình.

10.2.1 Teacher forcing và mạng nơ-ron với sự tái liên kết đầu ra

Mạng nơ-ron có kết nối hồi tiếp chỉ từ đầu ra ở bước hiện tại đến trạng thái ẩn ở bước tiếp theo (như Hình 10.4) thường có năng lực mô hình hóa yếu hơn so với các mạng có hồi tiếp giữa các đơn vị ẩn. Điều này có nghĩa là loại mạng này không thể mô phỏng máy Turing phổ quát.

Lý do là vì mạng không thể lưu trữ trạng thái quá khứ thông qua các đơn vị ẩn — thông tin quá khứ bắt buộc phải được mã hóa hoàn toàn trong các đầu ra. Tuy nhiên, trong quá trình huấn luyện, đầu ra được học để khớp với các giá trị mục tiêu, chứ không phải để lưu giữ lịch sử đầu vào. Do đó, trừ khi các giá trị mục tiêu trong tập huấn luyện được thiết kế để chứa đầy đủ trạng thái hệ thống, mạng sẽ khó có thể tận dụng tốt thông tin từ quá khứ.

Tuy nhiên, việc loại bỏ hồi tiếp giữa các đơn vị ẩn lại mang đến một lợi thế: nếu hàm mất mát tại mỗi bước thời gian chỉ phụ thuộc vào dự đoán tại bước đó, thì tất cả các bước thời gian có thể được xử lý độc lập. Điều này cho phép quá trình huấn luyện được song song hóa, bởi vì ta không cần phải chờ kết quả của bước trước đó — đầu ra đúng đã có sẵn trong tập huấn luyện.



Hình 10.6: Minh họa kỹ thuật *teacher forcing*. (Trái) Trong quá trình huấn luyện, đầu ra đúng $y(t)$ từ tập huấn luyện được dùng làm đầu vào cho $h(t+1)$. (Phải) Khi triển khai mô hình, ta không có $y(t)$, nên phải dùng đầu ra mô hình $o(t)$ thay thế.

Đối với các mô hình có kết nối hồi tiếp từ đầu ra quay lại, một kỹ thuật huấn luyện hiệu quả được sử dụng là teacher forcing. Đây là một phương pháp phổ biến trong huấn luyện RNN, bắt nguồn từ tiêu chuẩn tối đa hóa xác suất (maximum likelihood).

Trong teacher forcing, thay vì sử dụng đầu ra mô hình ở thời điểm t làm đầu vào cho thời điểm $t+1$, ta sử dụng chính nhãn đúng $y(t)$ từ dữ liệu huấn luyện. Ví dụ:

$$\log p(y(1), y(2)|x(1), x(2)) = \log p(y(2)|y(1), x(1), x(2)) + \log p(y(1)|x(1), x(2))$$

Ở đây, tại thời điểm $t = 2$, mô hình được huấn luyện để dự đoán $y(2)$ dựa trên cả chuỗi đầu vào và đầu ra đúng từ bước trước. Như vậy, trong quá trình huấn luyện, đầu vào của mạng ở bước tiếp theo sẽ là nhãn thực tế chứ không phải là đầu ra dự đoán.

Phương pháp teacher forcing cũng có thể áp dụng cho các mạng có hồi tiếp giữa các trạng thái ẩn, miễn là mô hình có kết nối từ đầu ra quay lại các tính toán trong tương lai. Khi các trạng thái ẩn phụ thuộc vào các thời điểm trước đó, ta cần sử dụng thuật toán lan truyền ngược qua thời gian (Backpropagation Through Time – BPTT).

Tuy nhiên, teacher forcing cũng có một điểm yếu. Trong giai đoạn suy luận (inference), mô hình phải tự sinh ra đầu ra và dùng chính đầu ra đó làm đầu vào cho bước kế tiếp — khác với lúc huấn luyện khi đầu ra đúng luôn sẵn có. Điều này dẫn đến sự khác biệt giữa dữ liệu huấn luyện và dữ liệu mô hình thấy khi triển khai.

Một cách giảm thiểu sự khác biệt này là kết hợp giữa huấn luyện với *teacher forcing* và chế độ *tự do chạy* (free-running mode) — trong đó mô hình phải tự sử dụng đầu ra dự đoán để tiếp tục. Chẳng hạn, ta có thể yêu cầu mô hình dự đoán chuỗi nhiều bước về tương lai, buộc nó học cách khôi phục trạng thái nếu đã trôi lệch so với chuỗi đúng.

Một kỹ thuật khác do Bengio et al. (2015b) đề xuất là trộn ngẫu nhiên giữa đầu vào từ dữ liệu thật và đầu vào do mô hình sinh ra. Chiến lược này theo phong cách *curriculum learning*, tức là dần dần tăng tỷ lệ đầu vào sinh ra — giúp mô hình thích nghi tốt hơn với chế độ vận hành thực tế, nơi không có sẵn đầu ra đúng.

10.2.2 Tính toán gradient trong mạng nơ-ron hồi tiếp (RNN)

Việc tính toán gradient trong mạng RNN có thể thực hiện thông qua thuật toán lan truyền ngược (backpropagation) như với các mạng nơ-ron truyền thống. Ta chỉ cần áp dụng thuật toán này lên đồ thị tính toán đã được trải thẳng theo thời gian, mà không cần đến thuật toán chuyên biệt nào. Sau khi có gradient, ta có thể sử dụng chúng với bất kỳ phương pháp tối ưu hóa dựa trên gradient nào.

Để hiểu rõ cách hoạt động của thuật toán Backpropagation Through Time (BPTT), ta xét một ví dụ tính gradient trên mạng RNN cơ bản. Các nút trong đồ thị bao gồm: tham số U, V, W, b, c, và các nút theo thời gian $x(t), h(t), o(t), L(t)$. Với mỗi nút N, gradient $\nabla_N L$ được tính đệ quy từ các nút phía sau.

Trước hết, vì hàm mất mát tổng là tổng của các mất mát tại từng thời điểm, nên gradient tại mỗi thời điểm bắt đầu từ:

$$\frac{\partial L}{\partial L(t)} = 1$$

Gradient đối với đầu ra tại thời điểm t:

$$(\nabla o(t) L)_i = \hat{y}(t)_i - y(t)_i$$

Gradient tại trạng thái ẩn $h(t)$ được lan truyền ngược qua thời gian như sau:

$$\nabla h(t)L = W^\top \text{diag}\left(1 - h(t+1)^2\right) \nabla h(t+1)L + V^\top \nabla o(t)L$$

Gradient đối với các tham số chính được tính như sau:

$$\nabla cL = \sum_t \nabla o(t)L$$

$$\nabla bL = \sum_t \text{diag}\left(1 - h(t)^2\right) \nabla h(t)L$$

$$\nabla VL = \sum_t \nabla o(t)Lh(t)^\top$$

$$\nabla WL = \sum_t \text{diag}\left(1 - h(t)^2\right) \nabla h(t)Lh(t-1)^\top$$

$$\nabla UL = \sum_t \text{diag}\left(1 - h(t)^2\right) \nabla h(t)Lx(t)^\top$$

Những công thức này minh họa rõ ràng cách gradient được lan truyền ngược trong mạng RNN, từ thời điểm cuối về thời điểm đầu.

10.2.3 Mạng hồi tiếp như các mô hình đồ thị có hướng

Trong các ví dụ đã trình bày, ta giả định rằng hàm mất mát $L(t)$ là hàm chênh lệch chéo (cross-entropy) giữa đầu ra mô hình $o(t)$ và nhãn mục tiêu $y(t)$. Tuy nhiên, giống như trong mạng lan truyền tiến, ta có thể sử dụng nhiều loại hàm mất mát khác nhau tùy vào bài toán cụ thể.

Thông thường, ta mong muốn diễn giải đầu ra của mạng RNN như một phân phối xác suất, và chọn hàm mất mát sao cho phù hợp. Ví dụ, nếu đầu ra là biến liên tục và phân phối đầu ra giả định là Gaussian đơn vị, thì sai số bình phương trung bình (mean squared error) sẽ tương ứng với hàm chênh lệch chéo trong trường hợp đó.

Trong các bài toán dự đoán tuần tự, nếu ta dùng log-likelihood làm mục tiêu huấn luyện, ví dụ như phương trình (10.12), thì mạng được huấn luyện để ước lượng xác suất có điều kiện của phần tử tiếp theo dựa trên quá khứ. Ta có thể tối đa hóa:

$$\log p(y(t)|x(1), \ldots, x(t)), \tag{10.29}$$

hoặc nếu mô hình có hồi tiếp từ đầu ra:

$$\log p(y(t)|x(1),\ldots,x(t),y(1),\ldots,y(t-1)). \tag{10.30}$$

Quá trình này thực chất là phân rã phân phối xác suất chung của toàn bộ chuỗi y thành chuỗi các xác suất điều kiện tại từng bước. Nếu trong mô hình ta không đưa các giá trị y quá khứ vào làm điều kiện, thì các giá trị $y(t)$ được giả định là độc lập có điều kiện cho trước chuỗi x. Khi đó, mô hình đồ thị có hướng sẽ không có cung nối từ các $y(i)$ trước đó đến $y(t)$.

Ngược lại, nếu ta cung cấp các giá trị y thực (không phải giá trị mô hình dự đoán) làm đầu vào, thì mô hình đồ thị có hướng sẽ có các cạnh từ $y(1),\ldots,y(t-1)$ đến $y(t)$. Điều này phản ánh mối quan hệ phụ thuộc theo thời gian trong chuỗi đầu ra khi huấn luyện. Để minh họa đơn giản, giả sử một mạng RNN được sử dụng để mô hình hóa một chuỗi các biến ngẫu nhiên vô hướng $Y = \{y(1),\ldots,y(\tau)\}$, không có đầu vào bổ sung x. Trong trường hợp này, đầu vào tại thời điểm t chỉ đơn giản là giá trị đầu ra tại thời điểm $t-1$. Mạng RNN khi đó xác định một mô hình đồ thị có hướng trên tập các biến y.

Phân phối chung của chuỗi này có thể được biểu diễn bằng quy tắc chuỗi (chain rule) cho các xác suất có điều kiện:

$$P(Y) = P(y(1),\ldots,y(\tau)) = \prod_{t=1}^{\tau} P(y(t)|y(t-1),y(t-2),\ldots,y(1)), \tag{10.31}$$

trong đó, ở thời điểm $t=1$, biểu thức điều kiện là rỗng.

Log-likelihood âm của một chuỗi quan sát cụ thể là:

$$L = \sum_t L(t), \tag{10.32}$$

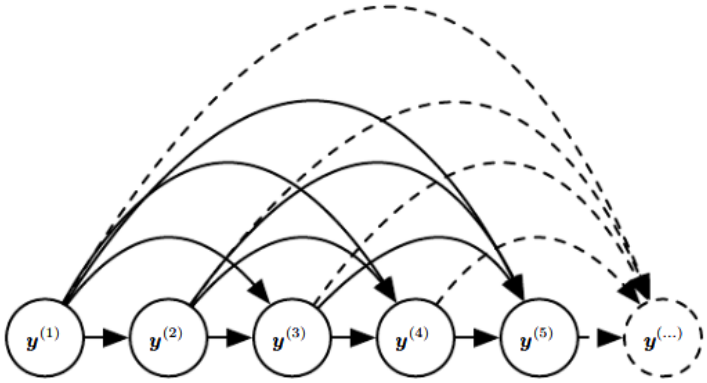
với mỗi thành phần được định nghĩa như sau:

$$L(t) = -\log P(y(t) = y(t)|y(t-1),y(t-2),\ldots,y(1)). \tag{10.33}$$

Trong mô hình đồ thị, các cạnh đại diện cho các mối quan hệ phụ thuộc trực tiếp giữa các biến. Thông thường, để đơn giản hóa tính toán và tăng hiệu quả thống kê, ta sẽ loại bỏ các cạnh tương ứng với mối liên hệ yếu. Ví dụ, trong nhiều ứng dụng, ta giả định tính chất Markov — tức là $y(t)$ chỉ phụ thuộc vào vài bước trước đó $\{y(t-k),\ldots,y(t-1)\}$, chứ không phải toàn bộ lịch sử.

Tuy nhiên, trong các tình huống mà các thông tin từ quá khứ xa vẫn ảnh hưởng đến hiện tại, giả định Markov là không đủ. RNN đặc biệt hữu ích trong những trường hợp như vậy — khi một giá trị $y(i)$ trong quá khứ xa có thể ảnh hưởng trực tiếp đến $y(t)$, mà không

cần phải đi qua tất cả các bước trung gian.

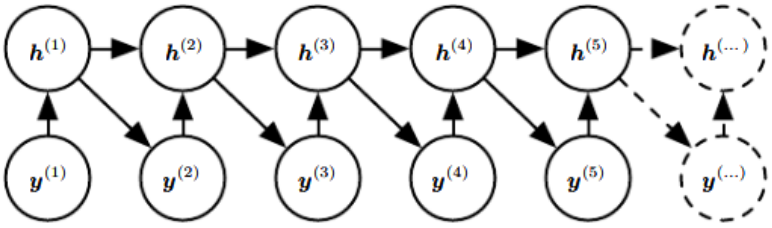


Hình 10.7: Mô hình đồ thị có hướng đầy đủ cho chuỗi $y(1), y(2), \dots$. Mỗi biến trong quá khứ $y(i)$ có thể ảnh hưởng đến bất kỳ biến tương lai nào $y(t)$ với $t > i$. Việc tham số hóa trực tiếp từ đồ thị này sẽ rất tốn kém vì số lượng tham số tăng theo thời gian. RNN giải quyết vấn đề này bằng cách dùng cấu trúc chia sẻ tham số hiệu quả, như minh họa ở Hình 10.8.

Một cách diễn giải mạng RNN là xem nó như một mô hình đồ thị có hướng với cấu trúc đầy đủ. Tức là, về lý thuyết, mọi giá trị trong quá khứ có thể ảnh hưởng trực tiếp đến mọi giá trị tương lai. Tuy nhiên, thay vì biểu diễn các cạnh này rõ ràng, RNN sử dụng trạng thái ẩn $h(t)$ như một đại diện nén để tổng hợp thông tin từ quá khứ.

Nếu coi các trạng thái ẩn $h(t)$ là các biến trong mô hình đồ thị (dù về mặt kỹ thuật, chúng là các hàm xác định của đầu vào), ta có thể xây dựng một mô hình đồ thị có cấu trúc hiệu quả hơn. So với việc biểu diễn đầy đủ phân phối xác suất của chuỗi — đòi hỏi đến $O(k^\tau)$ tham số nếu mỗi biến nhận k giá trị khác nhau — mạng RNN chỉ cần $O(1)$ tham số, nhờ vào cơ chế chia sẻ qua thời gian.

Như được mô tả trong phương trình (10.5), RNN sử dụng cùng một hàm f với tham số θ lặp đi lặp lại tại mỗi bước thời gian, giúp mô hình hóa hiệu quả các mối quan hệ dài hạn. Hình 10.8 minh họa cách đưa các nút ẩn $h(t)$ vào mô hình đồ thị, đóng vai trò là "cầu nối"giữa quá khứ và tương lai.



Hình 10.8: Việc đưa biến trạng thái $h(t)$ vào mô hình đồ thị của RNN giúp đạt được tham số hóa hiệu quả, dựa trên phương trình (10.5). Mỗi bước thời gian có cấu trúc giống nhau và chia sẻ cùng một bộ tham số.

Dù việc tham số hóa đã được đơn giản hóa đáng kể, một số phép toán như dự đoán các giá trị bị thiếu giữa chuỗi vẫn còn phức tạp.

RNN phải đánh đổi: giảm số lượng tham số bằng cách chia sẻ qua thời gian, nhưng điều này cũng làm quá trình tối ưu hóa trở nên khó khăn hơn. Giả định cơ bản ở đây là quan hệ giữa bước hiện tại và bước kế tiếp là không đổi theo thời gian. Nếu muốn mô hình học được các thay đổi theo thời gian, ta có thể đưa thêm chỉ số thời gian t như một đầu vào phụ tại mỗi bước.

Lấy mẫu từ mạng RNN

Để hoàn thiện cái nhìn tổng thể về RNN dưới dạng mô hình đồ thị, ta cần mô tả quá trình lấy mẫu từ mô hình.

Về cơ bản, lấy mẫu từ RNN là lấy mẫu tuần tự từng bước từ phân phối có điều kiện:

- Bắt đầu từ một đầu vào khởi tạo, - Ở mỗi bước thời gian, sử dụng đầu ra được sinh ra làm đầu vào cho bước tiếp theo, - Quá trình tiếp tục cho đến khi đạt đến điều kiện dừng.

Một câu hỏi quan trọng là: *Làm sao để mạng biết khi nào nên dừng lại?* Có vài cách phổ biến:

- Dùng ký hiệu đặc biệt kết thúc chuỗi: Nếu đầu ra là chuỗi ký tự, ta có thể thêm một ký tự "kết thúc" đặc biệt. Khi ký tự này được sinh ra, quá trình lấy mẫu dừng lại (Schmidhuber, 2012).
- Dùng đơn vị đầu ra Bernoulli: Mỗi bước thời gian, mô hình sinh ra thêm một giá trị Bernoulli (thường dùng hàm sigmoid), dự đoán xem có nên tiếp tục hay dừng lại. Phương pháp này linh hoạt hơn và có thể áp dụng cho các chuỗi không phải ký tự, ví dụ như chuỗi số thực.
- Dự đoán trực tiếp độ dài chuỗi τ: Mô hình sẽ dự đoán một giá trị τ, rồi sinh chuỗi với đúng số bước như vậy. Ta có thể bổ sung thêm đầu vào về số bước còn lại $(\tau - t)$ để giúp mô hình kiểm soát việc kết thúc hợp lý.

Trong trường hợp này, phân phối chung được phân tách thành:

$$P(x(1), \ldots, x(\tau)) = P(\tau)P(x(1), \ldots, x(\tau)|\tau)$$

Chiến lược dự đoán độ dài chuỗi trước đã được áp dụng, ví dụ trong nghiên cứu của Goodfellow et al. (2014d).

10.2.4 Mô hình hóa chuỗi có điều kiện theo ngữ cảnh với mạng RNN

Ở các phần trước, ta đã thấy cách mạng RNN có thể được hiểu như một mô hình đồ thị có hướng trên chuỗi các biến ngẫu nhiên $y(t)$, trong trường hợp không có đầu vào x. Tuy nhiên, trong thực tế, RNN thường được sử dụng để xử lý các chuỗi có đầu vào $x(1), x(2), \ldots, x(\tau)$, như mô tả trong phương trình (10.8). Khi đó, mạng RNN không chỉ mô hình hóa phân phối chung $P(y)$, mà còn mô hình hóa phân phối có điều kiện $P(y|x)$, tức là chuỗi đầu ra phụ thuộc vào chuỗi đầu vào.

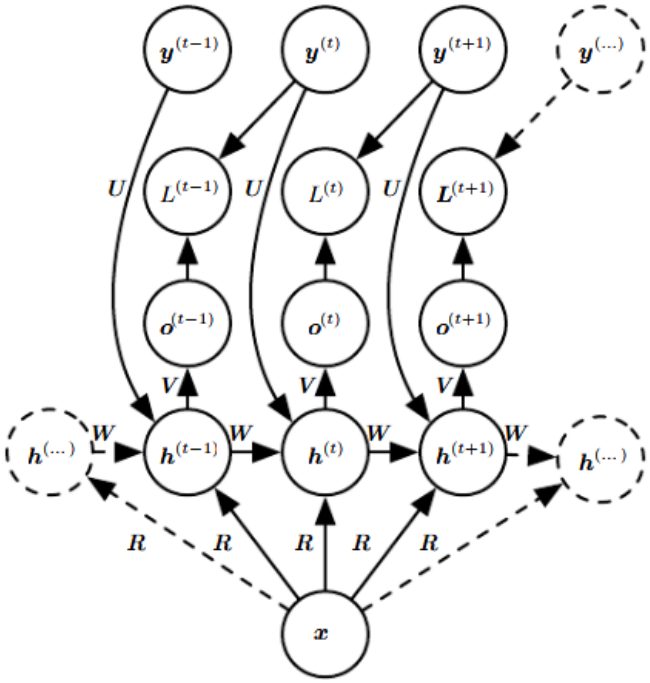
Tương tự như đã trình bày với mạng truyền thẳng trong Mục 6.2.1.1, bất kỳ mô hình

nào mô tả phân phối $P(y;\theta)$ cũng có thể được chuyển thành một mô hình có điều kiện $P(y|x)$, bằng cách cho phép các tham số θ trở thành một hàm của x. Trong mạng RNN, có nhiều cách để làm điều này. Dưới đây là một số lựa chọn phổ biến nhất.

Trước tiên, giả sử đầu vào là một vector duy nhất x. Ta có thể cung cấp x cho RNN theo các cách sau:

1. Thêm x vào như đầu vào tại mọi bước thời gian.
2. Dùng x để khởi tạo trạng thái ban đầu $h(0)$.
3. Kết hợp cả hai phương án trên.

Phương pháp phổ biến nhất là cách (1), được minh họa trong Hình 10.9. Ở đây, một ma trận trọng số mới R được sử dụng để ánh xạ x sang không gian trạng thái ẩn, tạo thành $x^\top R$. Giá trị này sau đó được cộng vào đầu vào tại mỗi bước thời gian như một dạng bias bổ sung. Ta có thể xem x như tham số hóa một phần của mạng — biến tham số tĩnh θ trong mô hình không điều kiện thành tham số hóa động $\omega = \omega(x)$.



Hình 10.9: Mạng RNN nhận một vector đầu vào cố định x và sinh ra một chuỗi đầu ra Y. Mô hình này phù hợp với các bài toán như mô tả hình ảnh, nơi một ảnh tĩnh được mã hóa thành vector x và được sử dụng để sinh ra mô tả văn bản. Trong quá trình huấn luyện, mỗi từ $y(t)$ được dùng vừa làm đầu vào, vừa làm mục tiêu để mô hình học dự đoán.

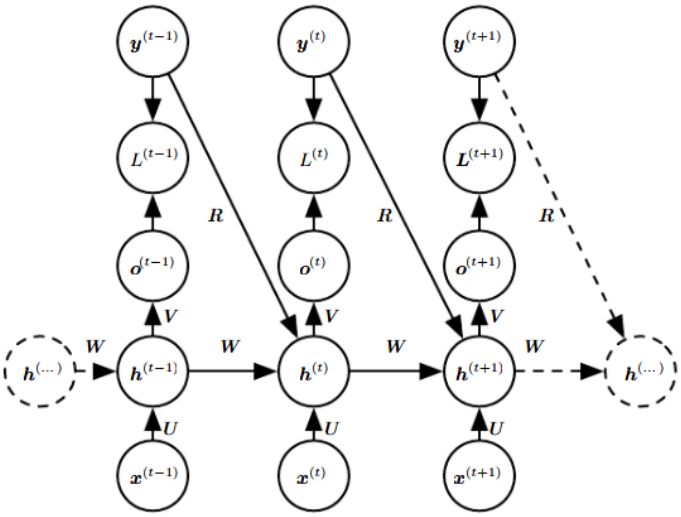
Ngoài việc nhận một vector cố định, mạng RNN cũng có thể nhận một chuỗi các vector $x(t)$ làm đầu vào. Mô hình RNN như trong phương trình (10.8) mô tả phân phối có điều kiện:

$$P(y(1), \ldots, y(\tau)|x(1), \ldots, x(\tau))$$

Với giả định rằng mỗi $y(t)$ phụ thuộc vào tất cả các $x(1), \ldots, x(t)$, ta có thể viết:

$$P(y(t)|x(1), \ldots, x(t)) \tag{10.35}$$

Giả định này là hạn chế, vì nó khiến các giá trị y trở nên độc lập có điều kiện với nhau khi biết chuỗi đầu vào. Nếu ta muốn cho phép các giá trị $y(t)$ phụ thuộc vào các y trước đó, ta cần thêm các kết nối hồi tiếp từ đầu ra đến trạng thái ẩn, như trong Hình 10.10. Nhờ đó, mô hình có thể học các mối quan hệ phức tạp hơn giữa các phần tử trong chuỗi đầu ra.



Hình 10.10: Mạng RNN có điều kiện nhận một chuỗi đầu vào x và sinh ra chuỗi đầu ra y có cùng độ dài. Khác với Hình 10.3, ở đây có thêm kết nối từ đầu ra trước đó đến trạng thái hiện tại, giúp mô hình hóa các mối quan hệ phụ thuộc phức tạp hơn giữa các phần tử $y(t)$.

Tuy nhiên, mô hình này vẫn có một hạn chế: chuỗi đầu vào và chuỗi đầu ra bắt buộc phải có độ dài bằng nhau. Cách vượt qua giới hạn này sẽ được trình bày trong Mục 10.4 tiếp theo.

10.3 Bidirectional RNNs

Các mạng RNN mà chúng ta đã xét trước đây đều là mạng có nguyên nhân (causal), nghĩa là trạng thái tại thời điểm t chỉ chứa thông tin từ quá khứ $x(1), \ldots, x(t-1)$ và đầu vào hiện tại $x(t)$. Một số mô hình còn có thể sử dụng cả thông tin từ các giá trị đầu ra trước đó y, nếu chúng có sẵn.

Tuy nhiên, trong nhiều ứng dụng thực tế, ta cần dự đoán $y(t)$ dựa trên toàn bộ chuỗi đầu vào — cả quá khứ và tương lai. Ví dụ như trong nhận dạng giọng nói, một âm thanh hiện tại có thể chỉ được giải nghĩa chính xác nếu biết những âm vị tiếp theo. Tương tự, trong nhận dạng chữ viết tay hoặc dịch máy, việc hiểu đúng tại một thời điểm thường cần thông tin từ cả hai phía của chuỗi.

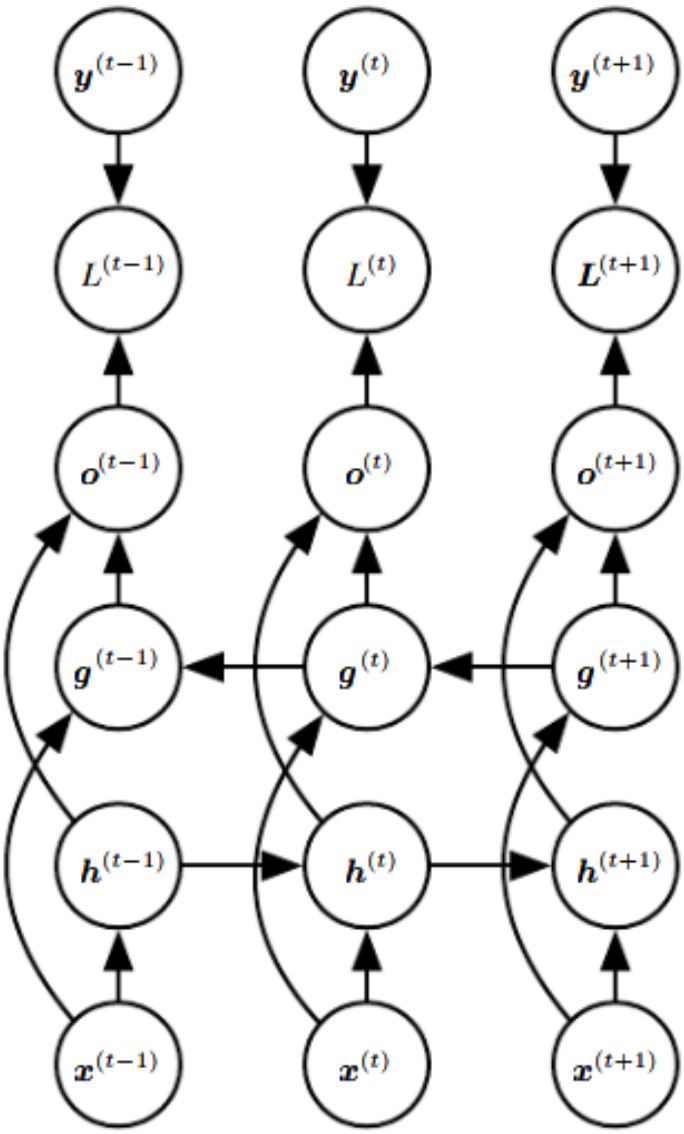
Mạng RNN hai chiều (Bidirectional RNNs) được thiết kế để xử lý những trường hợp như vậy (Schuster và Paliwal, 1997), và đã cho thấy hiệu quả cao trong các bài toán như

nhận dạng chữ viết tay (Graves et al., 2008), giọng nói (Graves và Schmidhuber, 2005), hay sinh học tính toán (Baldí et al., 1999).

Ý tưởng chính là dùng hai mạng RNN chạy song song:

- Một mạng chạy xuôi thời gian từ $t = 1$ đến $t = T$
- Một mạng chạy ngược từ $t = T$ về $t = 1$

Tại mỗi thời điểm t, đầu ra $o(t)$ được tính từ cả hai trạng thái ẩn: $h(t)$ (tiến) và $g(t)$ (lùi), như minh họa trong Hình 10.11.



Hình 10.11: Mô hình RNN hai chiều tính toán chuỗi đầu ra y từ chuỗi đầu vào x. Trạng thái $h(t)$ lan truyền thông tin từ quá khứ, trong khi $g(t)$ lan truyền thông tin từ tương lai. Do đó, đầu ra $o(t)$ có thể tổng hợp cả hai chiều thông tin.

Cách làm này cũng có thể mở rộng cho dữ liệu 2 chiều như hình ảnh. Bằng cách sử dụng bốn RNN chạy theo các hướng lên, xuống, trái, phải, ta có thể tính toán biểu diễn tại mỗi điểm ảnh (i, j) phụ thuộc vào toàn bộ ảnh. So với mạng tích chập, RNN trên ảnh

thường tốn kém hơn nhưng có thể học được các tương tác dài hạn theo chiều ngang hoặc dọc.

10.4 Kiến trúc encoder-decoder cho chuỗi đến chuỗi (sequence-to-sequence)

Trước đây, ta đã thấy rằng một RNN có thể:

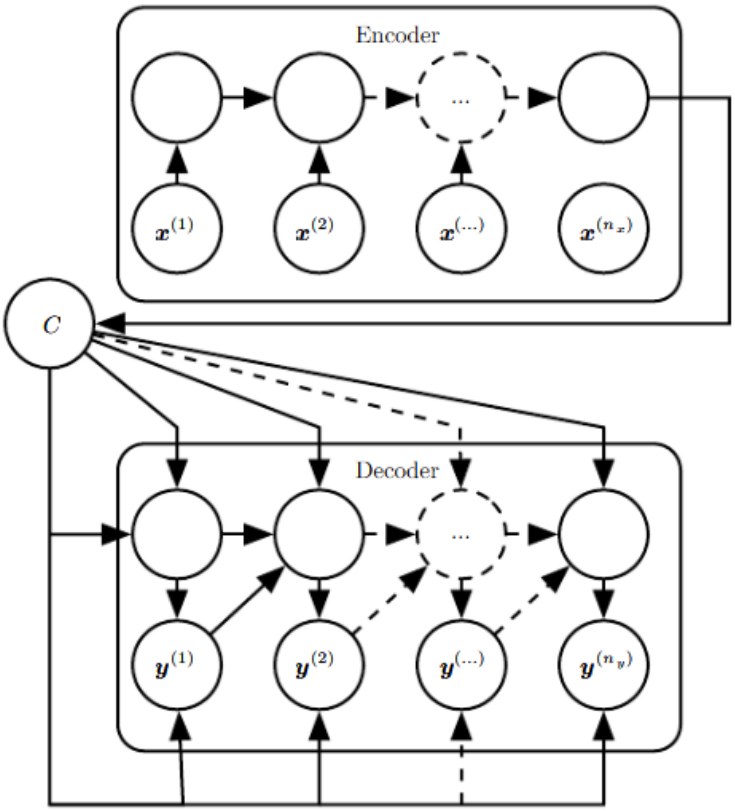
- ánh xạ một chuỗi thành một vector cố định (Hình 10.5),
- ánh xạ một vector thành một chuỗi (Hình 10.9),
- ánh xạ một chuỗi đầu vào thành một chuỗi đầu ra cùng độ dài (Hình 10.3, 10.4, 10.10, 10.11).

Tuy nhiên, trong nhiều ứng dụng thực tế như dịch máy, nhận dạng giọng nói hay trả lời câu hỏi, đầu vào và đầu ra thường có độ dài khác nhau. Vì vậy, ta cần một mô hình linh hoạt hơn: encoder-decoder hay sequence-to-sequence.

Kiến trúc này được đề xuất độc lập bởi Cho et al. (2014a) và Sutskever et al. (2014). Họ huấn luyện mô hình để ánh xạ chuỗi đầu vào $X = (x(1), \ldots, x(n_x))$ thành chuỗi đầu ra $Y = (y(1), \ldots, y(n_y))$, với $n_x \neq n_y$ là điều bình thường.

- Encoder (RNN mã hóa): xử lý chuỗi đầu vào và tạo ra một ngữ cảnh C, thường là trạng thái ẩn cuối cùng.
- Decoder (RNN giải mã): nhận ngữ cảnh C làm đầu vào và sinh ra chuỗi đầu ra.

Quá trình này được minh họa trong Hình 10.12.



Hình 10.12: Kiến trúc encoder-decoder RNN, ánh xạ một chuỗi đầu vào $(x(1), \ldots, x(n_x))$ thành một chuỗi đầu ra $(y(1), \ldots, y(n_y))$ thông qua một biến ngữ cảnh C.

Trong kiến trúc cơ bản, ngữ cảnh C là một vector. Tuy nhiên, điều này gây ra giới hạn khi cần tóm tắt các chuỗi dài. Bahdanau et al. (2015) đã đề xuất cơ chế chú ý (attention) để khắc phục vấn đề này bằng cách cho phép decoder tham chiếu trực tiếp đến các bước cụ thể của chuỗi đầu vào. Chi tiết về cơ chế chú ý sẽ được trình bày ở Mục 12.4.5.1.

10.5 Mạng nơ-ron hồi tiếp sâu (Deep Recurrent Networks)

Trong một RNN đơn giản như ở Hình 10.3, các phép tính có thể được chia làm ba phần:

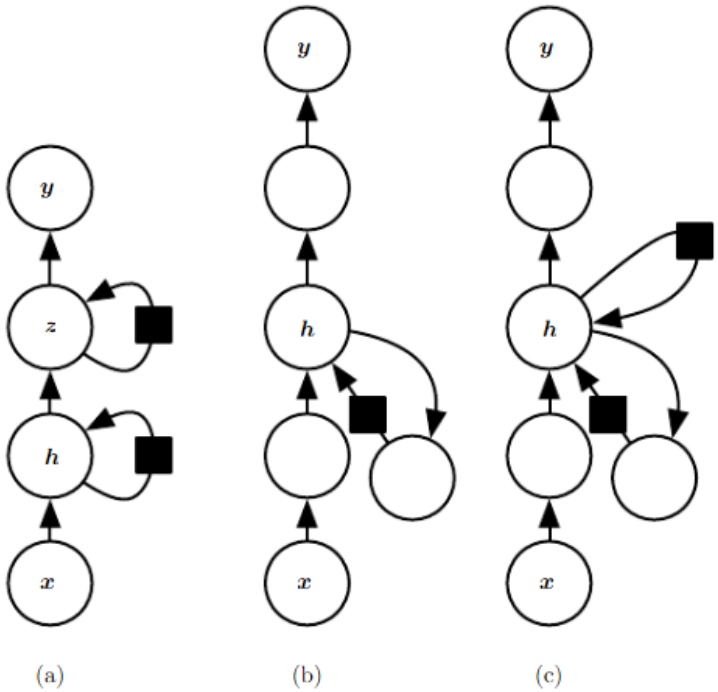
1. Từ đầu vào đến trạng thái ẩn,
2. Từ trạng thái ẩn trước đến trạng thái ẩn tiếp theo,
3. Từ trạng thái ẩn đến đầu ra.

Mỗi phần này thường chỉ là một lớp nông (linear + activation). Câu hỏi đặt ra là: *Liệu có lợi không nếu ta làm các phép biến đổi này sâu hơn?* — tức là thêm nhiều lớp trong mỗi khối. Các nghiên cứu (Graves et al., 2013; Pascanu et al., 2014a) cho thấy điều này rất có ích cho khả năng biểu diễn.

Có nhiều cách để "làm sâu"RNN:

- (a): Chia trạng thái ẩn thành nhiều lớp, xếp chồng theo chiều sâu — giống như mạng MLP nhiều lớp (Hình 10.13a).

- (b): Thay vì dùng ma trận tuyến tính đơn giản, thay bằng các MLP nhiều lớp cho từng khối (input-to-hidden, hidden-to-hidden, hidden-to-output).
- (c): Dùng kết nối bỏ qua (skip connection) giữa các tầng để tránh việc gradient bị suy giảm khi chiều sâu tăng.



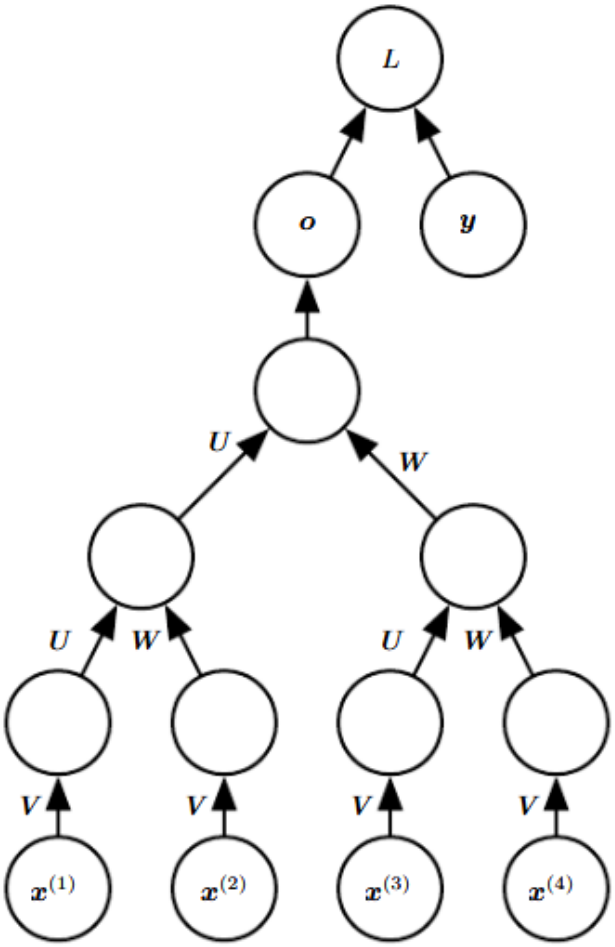
Hình 10.13: Các cách làm sâu mạng RNN: (a) dùng nhiều tầng trạng thái ẩn, (b) dùng MLP cho các khối biến đổi, (c) thêm kết nối bỏ qua để duy trì đường đi ngắn giữa các bước thời gian.

Thêm chiều sâu giúp tăng khả năng biểu diễn, nhưng cũng làm tối ưu hóa phức tạp hơn. Kết nối bỏ qua là một kỹ thuật quan trọng giúp giảm bớt khó khăn này, cho phép gradient lan truyền tốt hơn qua thời gian và các lớp sâu.

10.6 Mạng nơ-ron đệ quy (Recursive Neural Networks)

Mạng nơ-ron đệ quy (Recursive Neural Networks – RecNN)[1] là một biến thể mở rộng của mạng RNN, trong đó đồ thị tính toán không còn là chuỗi tuyến tính mà có cấu trúc dạng cây sâu (deep tree). Hình 10.14 minh họa cách tổ chức này.

[1] Chúng tôi không viết tắt RecNN là "RNN" để tránh nhầm với "mạng nơ-ron hồi tiếp" (Recurrent Neural Network).



Hình 10.14: Mạng nơ-ron đệ quy với đồ thị tính toán dạng cây thay vì chuỗi. Một chuỗi đầu vào có độ dài biến đổi $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$ được ánh xạ thành một biểu diễn cố định o thông qua cùng một bộ tham số U, V, W. Đây là ví dụ cho học có giám sát, trong đó đầu ra được gán nhãn mục tiêu y.

Mạng đệ quy được giới thiệu bởi Pollack (1990), và được Bottou (2011) phân tích như một công cụ tiềm năng để học suy luận. Các ứng dụng thực tế của RecNN đã xuất hiện trong xử lý dữ liệu có cấu trúc cây như:

- Xử lý cấu trúc dữ liệu tổng quát (Frasconi et al., 1997, 1998),
- Ngôn ngữ tự nhiên — như phân tích cú pháp và phân loại cảm xúc câu (Socher et al., 2011a, 2013a),
- Thị giác máy tính — như phân đoạn ảnh (Socher et al., 2011b).

Một điểm mạnh nổi bật của mạng đệ quy so với RNN là khả năng rút ngắn độ sâu phi tuyến trong tính toán. Với chuỗi có độ dài τ, độ sâu của RecNN chỉ là $\mathcal{O}(\log \tau)$, thay vì τ như trong RNN. Điều này giúp mô hình xử lý tốt hơn các quan hệ phụ thuộc dài trong chuỗi.

Tuy nhiên, câu hỏi đặt ra là: *nên xây dựng cây như thế nào*? Một số lựa chọn phổ biến:

- Dùng cây cố định, chẳng hạn như cây nhị phân cân bằng,

- Dựa vào kiến thức ngoài, như cây cú pháp trong xử lý ngôn ngữ tự nhiên,
- Tốt nhất là để mô hình tự học ra cấu trúc cây phù hợp cho từng đầu vào cụ thể (Bottou, 2011).

Mạng đệ quy cũng có nhiều biến thể. Ví dụ, Frasconi et al. (1997, 1998) đề xuất mô hình mà đầu vào và đầu ra có thể được gắn tại các nút khác nhau trong cây, không cần tất cả nút đều có đầu vào hoặc đầu ra.

Ngoài ra, phép tính tại mỗi nút không nhất thiết phải là phép nhân tuyến tính + phi tuyến như trong nơ-ron truyền thống. Chẳng hạn, Socher et al. (2013a) sử dụng các phép toán tensor và song tuyến (bilinear forms), vốn được chứng minh là rất hiệu quả trong việc mô hình hóa quan hệ giữa các khái niệm được biểu diễn bằng vector liên tục (embedding), như trong các công trình của Weston et al. (2010) và Bordes et al. (2012).

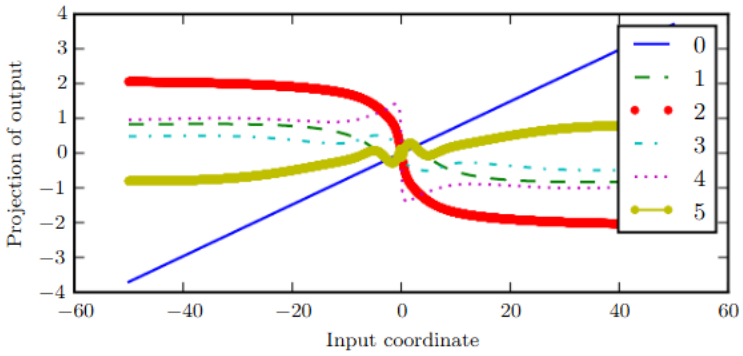
10.7 Thách thức về phụ thuộc dài hạn

Chúng ta đã đề cập sơ lược đến vấn đề toán học khi học các phụ thuộc dài hạn trong mạng hồi tiếp (RNN) ở Mục 8.2.5. Vấn đề cốt lõi nằm ở chỗ: khi gradient được lan truyền qua nhiều bước thời gian, chúng có xu hướng hoặc là tiêu biến (vanishing) — xảy ra thường xuyên và làm gradient gần như bằng 0, hoặc là bùng nổ (exploding) — ít xảy ra hơn nhưng gây ảnh hưởng nghiêm trọng đến quá trình tối ưu.

Ngay cả khi giả định rằng các tham số của mạng RNN được tối ưu tốt để tránh gradient bùng nổ, thì việc học được các mối quan hệ dài hạn vẫn rất khó khăn. Lý do là vì thông tin cần được truyền qua nhiều bước, và mỗi bước đều liên quan đến việc nhân với Jacobian — khiến ảnh hưởng của các bước xa giảm nhanh theo cấp số nhân so với những bước gần.

Nhiều công trình nghiên cứu đã chỉ ra và phân tích sâu vấn đề này (Hochreiter, 1991; Doya, 1993; Bengio et al., 1994; Pascanu et al., 2013). Trong phần này, chúng ta sẽ mô tả chi tiết hơn cơ chế gây ra vấn đề, và các phần tiếp theo sẽ trình bày những phương pháp để khắc phục.

RNN thực hiện việc lặp lại một hàm qua thời gian — mỗi bước thời gian tương ứng với một lần áp dụng hàm đó. Việc ghép nhiều hàm phi tuyến này có thể tạo nên một hàm tổng thể rất phức tạp, như được minh họa trong Hình 10.15.



Hình 10.15: Sự phức tạp tăng lên khi lặp nhiều lần hàm phi tuyến. Mỗi đường cong biểu diễn một mặt cắt tuyến tính của hàm phi tuyến sau một số bước lặp. Ta thấy rằng độ dốc thay đổi rất mạnh giữa các vùng, khiến gradient tại hầu hết các vị trí gần như bằng 0, chỉ vài vị trí có gradient lớn, và những vùng này thay đổi liên tục.

Một cách dễ hiểu hơn là xem RNN như đang nhân ma trận lặp lại. Ví dụ, với RNN đơn giản không có đầu vào và không có phi tuyến:

$$h(t) = W^\top h(t-1)$$

Khi đó:

$$h(t) = (W^\top)^t h(0)$$

Nếu ma trận W có phân tích riêng:

$$W = Q\Lambda Q^\top$$

thì:

$$h(t) = Q\Lambda^t Q^\top h(0)$$

Ở đây, các giá trị riêng λ sẽ được lũy thừa t. Nếu $|\lambda| < 1$, chúng nhanh chóng tiến về 0 (tiêu biến). Nếu $|\lambda| > 1$, chúng phát triển rất nhanh (bùng nổ). Dần dần, chỉ còn lại thành phần theo hướng vector riêng lớn nhất, còn mọi thông tin khác trong trạng thái ban đầu đều bị loại bỏ.

Để hiểu rõ hơn, ta có thể xét ví dụ đơn giản với một trọng số vô hướng w. Sau t bước, giá trị là:

$$h(t) = w^t$$

Tùy vào độ lớn của w, kết quả sẽ bùng nổ (nếu $|w| > 1$) hoặc tiêu biến (nếu $|w| < 1$). Nhưng nếu dùng mạng phi hồi tiếp với trọng số khác nhau tại mỗi bước $w^{(t)}$, vấn đề lại khác.

Ví dụ:

$$h(t) = \prod_{t=1}^{\tau} w^{(t)}$$

Giả sử các $w^{(t)}$ là biến ngẫu nhiên độc lập, trung bình bằng 0, phương sai bằng v. Khi đó, phương sai của tích trên là $\mathcal{O}(v^\tau)$. Nếu muốn phương sai tổng thể là v^*, chỉ cần chọn $v = \sqrt[\tau]{v^*}$. Điều này cho thấy các mạng feedforward sâu có thể tránh gradient tiêu biến nếu khởi tạo trọng số đúng cách (Sussillo, 2014).

Tuy nhiên, RNN lại không dễ như vậy. Các nghiên cứu (Hochreiter, 1991; Bengio et al., 1994) đã chỉ ra rằng: để mạng có thể ghi nhớ thông tin dài hạn và ổn định trước các nhiễu nhỏ, nó phải nằm trong vùng hoạt động mà gradient tự nhiên bị tiêu biến.

Cụ thể, nếu mạng có thể mô hình hóa được phụ thuộc dài hạn, thì gradient tương ứng phải đi qua rất nhiều bước — và sẽ bị thu nhỏ (hoặc phóng đại) theo cấp số nhân so với gradient của các phụ thuộc ngắn hạn. Điều này tạo ra một thế khó khi tối ưu: gradient cho phụ thuộc ngắn hạn sẽ lớn hơn nhiều, chi phối quá trình học, trong khi gradient cho phụ thuộc dài hạn lại quá nhỏ để cập nhật hiệu quả.

Trong phần tiếp theo, chúng ta sẽ khám phá các phương pháp được đề xuất để đối phó với vấn đề này, đặc biệt là các kiến trúc như LSTM và GRU.

10.8 Mạng Echo State (Echo State Networks - ESNs)

Một trong những khó khăn lớn nhất khi huấn luyện mạng nơ-ron hồi tiếp (RNN) là học được trọng số hồi tiếp từ $h(t-1)$ đến $h(t)$, và trọng số đầu vào từ $x(t)$ đến $h(t)$. Để tránh khó khăn này, một ý tưởng đơn giản nhưng hiệu quả đã được đề xuất (Jaeger, 2003; Maass et al., 2002; Jaeger and Haas, 2004):

> Chúng ta sẽ thiết lập trước các trọng số hồi tiếp sao cho các đơn vị ẩn có khả năng ghi nhớ được lịch sử của chuỗi đầu vào, và chỉ huấn luyện lớp đầu ra.

Đây chính là nguyên lý hoạt động của Echo State Networks (ESNs) và Liquid State Machines (LSMs). Trong đó:

- ESNs sử dụng các đơn vị ẩn với giá trị liên tục.
- LSMs sử dụng các nơ-ron dạng xung (spiking neurons) với đầu ra nhị phân.

Cả hai đều thuộc nhóm phương pháp gọi là Reservoir Computing — tạm dịch là "tính toán bằng hồ chứa". Ở đây, lớp ẩn hoạt động giống như một "hồ" đặc trưng thời gian, chứa các biến thể phức tạp của lịch sử chuỗi đầu vào.

Một cách hình dung khác: Reservoir Computing tương tự như các mô hình hạt nhân (kernel machines), với điểm khác biệt là:

- Một chuỗi đầu vào với độ dài bất kỳ sẽ được ánh xạ sang một vector trạng thái cố định $h(t)$.
- Sau đó, ta huấn luyện một mô hình dự đoán tuyến tính (ví dụ hồi quy tuyến tính) trên $h(t)$.

Vì chỉ huấn luyện tầng đầu ra, hàm mất mát như MSE sẽ là hàm lồi, giúp việc tối ưu trở nên nhanh và ổn định với các thuật toán đơn giản.

Câu hỏi then chốt đặt ra là: *Làm sao thiết lập được trọng số đầu vào và trọng số hồi tiếp để mạng có thể lưu trữ thông tin quá khứ một cách hiệu quả?*

Câu trả lời từ cộng đồng reservoir computing là: Hãy xem mạng như một hệ động lực, và thiết lập các trọng số sao cho mạng hoạt động ở vùng "gần biên ổn định" (edge of stability).

Một khái niệm quan trọng trong thiết lập này là bán kính phổ (spectral radius), được định nghĩa là giá trị lớn nhất trong mô-đun (tức trị tuyệt đối) của các trị riêng của ma trận Jacobian:

$$J(t) = \frac{\partial h(t)}{\partial h(t-1)}$$

- Nếu $|\lambda| > 1$: nhiễu ban đầu sẽ được khuếch đại nhanh chóng (gây nổ gradient).
- Nếu $|\lambda| < 1$: nhiễu sẽ bị làm mờ dần (gây tiêu biến gradient).

Vì vậy, để giữ cho tín hiệu từ quá khứ có thể truyền đến tương lai mà không biến mất hoặc phát nổ, ta cần đặt bán kính phổ gần 1 — hoặc hơi lớn hơn 1 một chút.

Xét ví dụ đơn giản về một mạng tuyến tính không phi tuyến:

$$h(t+1) = W^\top h(t)$$

Khi ma trận W có bán kính phổ nhỏ hơn 1 (gọi là "co lại" hay *contractive*), tín hiệu sẽ dần biến mất theo thời gian $\rightarrow$ mạng sẽ nhanh chóng "quên" quá khứ. Ngược lại, nếu bán kính phổ lớn hơn 1, tín hiệu sẽ bị khuếch đại quá mức $\rightarrow$ mạng trở nên không ổn định.

Trong thực tế, chúng ta thường dùng hàm kích hoạt như $\tanh$ để giới hạn đầu ra trong khoảng $(-1, 1)$, giúp ngăn mạng bị bùng nổ tín hiệu. Tuy nhiên:

- Dù có hàm phi tuyến, nếu trọng số đủ lớn và đầu vào nằm trong vùng tuyến tính của $\tanh$, mạng vẫn có thể khuếch đại tín hiệu.
- Nhưng điều này ít khi xảy ra cho toàn bộ chuỗi đầu vào.

Chiến lược tiêu chuẩn trong ESN là: giữ bán kính phổ ở một giá trị cố định (ví dụ 3), sau đó dùng phi tuyến $\tanh$ để giới hạn sự khuếch đại.

Đáng chú ý là các kỹ thuật của ESN không chỉ áp dụng cho mạng có trọng số cố định. Chúng còn có thể được dùng để khởi tạo RNN thông thường trước khi huấn luyện bằng phương pháp lan truyền ngược qua thời gian (BPTT).

Theo Sutskever (2012, 2013), nếu khởi tạo mạng với bán kính phổ khoảng 1.2 và dùng cách khởi tạo trọng số thưa thớt (sparse init), thì mạng RNN có khả năng học tốt hơn các phụ thuộc dài hạn trong chuỗi.

10.9 Các đơn vị rò rỉ và những chiến lược khác để xử lý đa thang thời gian

Một trong những hướng tiếp cận để giải quyết bài toán học các phụ thuộc dài hạn là thiết kế mô hình có thể hoạt động trên nhiều thang thời gian. Cụ thể, một số phần của mô hình sẽ xử lý tín hiệu ở thang thời gian chi tiết để nắm bắt các đặc trưng ngắn hạn, trong khi các phần khác sẽ hoạt động ở thang thời gian thô hơn để truyền tải thông tin từ quá khứ xa đến hiện tại một cách hiệu quả hơn.

Có nhiều chiến lược để xây dựng mô hình đa thang thời gian, tiêu biểu bao gồm:

- Thêm các kết nối bỏ qua theo thời gian (*skip connections*),
- Sử dụng các đơn vị rò rỉ (*leaky units*) có hằng số thời gian khác nhau,
- Loại bỏ các kết nối gần để buộc mô hình tập trung vào thông tin dài hạn.

10.9.1 Thêm kết nối bỏ qua theo thời gian (Skip Connections through Time)

Một chiến lược đơn giản để giúp mô hình tiếp cận thông tin từ quá khứ xa là thêm các kết nối trực tiếp từ các bước thời gian trước đó đến bước thời gian hiện tại, với độ trễ $d > 1$. Đây là một ý tưởng được đề xuất từ sớm bởi Lin et al. (1996), và có liên hệ với khái niệm mạng nơ-ron có độ trễ (delay neural networks) được Lang và Hinton (1988) nhắc đến.

Trong một mạng hồi tiếp thông thường, các kết nối thời gian thường chỉ từ t đến $t + 1$. Việc thêm kết nối từ t đến $t + d$ (với $d > 1$) giúp tạo ra các đường dẫn trực tiếp từ quá khứ xa hơn.

Như đã nêu trong Mục 8.2.5, gradient lan truyền qua nhiều bước thời gian có thể suy giảm theo cấp số mũ $\exp(-\tau)$. Nếu ta chèn thêm kết nối với độ trễ d, tốc độ tiêu biến có thể giảm xuống còn $\exp(-\tau/d)$ — tức gradient suy giảm chậm hơn, và nhờ đó mô hình có khả năng học tốt hơn các phụ thuộc dài hạn.

- Các kết nối trễ d đóng vai trò như "cầu nối" từ quá khứ xa đến hiện tại.
- Tuy nhiên, do vẫn tồn tại các kết nối ngắn hạn, gradient vẫn có nguy cơ bùng nổ nếu không được điều chỉnh đúng cách.

10.9.2 Đơn vị rò rỉ (Leaky Units) và phổ các thang thời gian khác nhau

Một hướng tiếp cận khác là dùng các đơn vị có kết nối tự hồi tuyến tính với trọng số gần 1, cho phép duy trì trạng thái lâu dài hơn mà không làm gradient bùng nổ hay tiêu biến nhanh chóng.

Giả sử ta muốn duy trì một trung bình động $\mu(t)$ của một đại lượng $v(t)$:

$$\mu(t) \leftarrow \alpha\mu(t-1) + (1-\alpha)v(t)$$

Ở đây, α điều chỉnh mức độ ảnh hưởng của quá khứ:

- $\alpha \to 1$: thông tin quá khứ được giữ lại lâu hơn.

- $\alpha \to 0$: trung bình động phản ứng nhanh, ít lưu giữ quá khứ.

Những đơn vị ẩn có cập nhật theo công thức trên được gọi là leaky units. Chúng tương đương với việc áp dụng bộ lọc thông thấp (low-pass filter) cho tín hiệu thời gian.

So với việc thêm kết nối bỏ qua với độ trễ d (chỉ ảnh hưởng ở các mốc thời gian rời rạc), leaky units cho phép trạng thái hiện tại được cập nhật liên tục với mức độ "rò rỉ" điều chỉnh linh hoạt bởi α.

Có hai cách để xác định giá trị α:

1. Gán cố định: Ví dụ chọn ngẫu nhiên tại lúc khởi tạo và giữ nguyên trong quá trình huấn luyện.
2. Học được: Cho phép α là tham số huấn luyện, được tối ưu cùng với các trọng số khác.

Cách dùng leaky units với nhiều giá trị α đã được Mozer (1992) và Pascanu et al. (2013) chứng minh là hiệu quả. Cách này cũng đóng vai trò quan trọng trong kiến trúc Echo State Networks (Jaeger et al., 2007).

10.9.3 Loại bỏ kết nối (Removing Connections)

Chiến lược cuối cùng là loại bỏ các kết nối giữa các bước thời gian liên tiếp, buộc mô hình chỉ cập nhật trạng thái sau mỗi khoảng thời gian cố định. Cách này được gọi là tổ chức trạng thái theo nhiều thang thời gian (multi-timescale).

Khác với việc thêm kết nối bỏ qua, nơi đơn vị có thể "chọn" học phụ thuộc dài hay ngắn, thì ở đây, đơn vị bị ép buộc phải hoạt động theo chu kỳ dài hơn.

Có hai cách phổ biến để thực hiện:

- Sử dụng nhiều nhóm leaky units: Mỗi nhóm được gán một giá trị α khác nhau, tương ứng với thang thời gian khác nhau. Nhóm có α lớn sẽ lưu giữ thông tin lâu hơn. Cách này giúp phân tán tín hiệu đầu vào ra nhiều "kênh thời gian" khác nhau.
- Cập nhật rời rạc theo chu kỳ (Clockwork RNN): Một cách khác là chia các đơn vị ẩn thành các nhóm, mỗi nhóm chỉ được phép cập nhật trạng thái sau một số bước thời gian nhất định (theo đồng hồ nội bộ). Ví dụ, nhóm 1 cập nhật mỗi bước, nhóm 2 cập nhật sau mỗi 2 bước, nhóm 3 sau mỗi 4 bước, v.v. Ý tưởng này được El Hihi và Bengio (1996) đề xuất và phát triển thành mô hình Clockwork RNN (Koutník et al., 2014). Cách này đã cho thấy hiệu quả trên nhiều bài toán thực tế.

Những phương pháp trên giúp mạng hồi tiếp mô hình hóa hiệu quả các phụ thuộc đa thang thời gian, làm nền tảng cho nhiều kiến trúc hiện đại như LSTM, GRU và các biến thể sâu hơn sau này.

10.9.4 LSTM: Long Short-Term Memory

Mạng LSTM (Long Short-Term Memory), được giới thiệu bởi Hochreiter và Schmidhuber (1997), là một kiến trúc RNN với các đơn vị đặc biệt được thiết kế để lưu trữ thông tin

qua thời gian dài, đồng thời tránh được vấn đề gradient tiêu biến hoặc bùng nổ.

Trung tâm của một LSTM là một đơn vị lưu trữ trạng thái (gọi là cell state hay bộ nhớ) $c(t)$, được điều chỉnh bởi ba cổng:

- Cổng quên (*forget gate*) $f(t)$ – quyết định phần nào của trạng thái cũ $c(t-1)$ nên được giữ lại.
- Cổng vào (*input gate*) $i(t)$ – xác định thông tin mới nào từ đầu vào $x(t)$ sẽ được thêm vào trạng thái.
- Cổng đầu ra (*output gate*) $o(t)$ – xác định phần nào của trạng thái hiện tại $c(t)$ sẽ được đưa ra làm đầu ra $h(t)$.

Dưới đây là các phương trình lan truyền trong một khối LSTM điển hình:

$$f(t) = \sigma\left(W_f x(t) + U_f h(t-1) + b_f\right) \tag{10.34}$$

$$i(t) = \sigma\left(W_i x(t) + U_i h(t-1) + b_i\right) \tag{10.35}$$

$$\tilde{c}(t) = \tanh\left(W_c x(t) + U_c h(t-1) + b_c\right) \tag{10.36}$$

$$c(t) = f(t) \odot c(t-1) + i(t) \odot \tilde{c}(t) \tag{10.37}$$

$$o(t) = \sigma\left(W_o x(t) + U_o h(t-1) + b_o\right) \tag{10.38}$$

$$h(t) = o(t) \odot \tanh(c(t)) \tag{10.39}$$

Trong đó:

- σ là hàm sigmoid, cho đầu ra trong khoảng $(0, 1)$.
- $\tanh$ là hàm hyperbolic tangent.
- $\odot$ là phép nhân từng phần tử (element-wise product).

Như ta thấy, trạng thái $c(t)$ được cập nhật tuyến tính theo $c(t-1)$, điều này giúp giảm tình trạng gradient tiêu biến qua thời gian. Các cổng sigmoid đóng vai trò như công tắc điều chỉnh — cho phép mạng học cách điều phối thông tin nào nên giữ, quên hoặc cập nhật.

Ưu điểm của LSTM:

- Có khả năng ghi nhớ các thông tin quan trọng trong chuỗi rất lâu.
- Có thể học được các phụ thuộc dài hạn mà không cần cấu trúc đặc biệt như skip connections.
- Tự động quyết định khi nào nên quên hoặc nhớ — giảm thiểu sự phụ thuộc vào kỹ thuật thủ công.

Nhờ vào thiết kế này, LSTM đã trở thành nền tảng cho nhiều ứng dụng hiện đại trong

xử lý ngôn ngữ tự nhiên, dịch máy, nhận dạng tiếng nói, và nhiều bài toán chuỗi khác.

10.9.5 GRU: Gated Recurrent Unit

Một biến thể đơn giản hơn của LSTM được gọi là GRU (Gated Recurrent Unit), được đề xuất bởi Cho et al. (2014). GRU hợp nhất cổng quên và cổng vào thành một cổng duy nhất gọi là cổng cập nhật (*update gate*), đồng thời sử dụng một cổng đặt lại (*reset gate*) để điều khiển việc kết hợp thông tin mới.

Các phương trình của GRU như sau:

$$z(t) = \sigma\left(W_z x(t) + U_z h(t-1) + b_z\right) \tag{10.40}$$

$$r(t) = \sigma\left(W_r x(t) + U_r h(t-1) + b_r\right) \tag{10.41}$$

$$\tilde{h}(t) = \tanh\left(W_h x(t) + U_h\left(r(t) \odot h(t-1)\right) + b_h\right) \tag{10.42}$$

$$h(t) = (1 - z(t)) \odot h(t-1) + z(t) \odot \tilde{h}(t) \tag{10.43}$$

GRU có số lượng tham số ít hơn LSTM, do đó thường huấn luyện nhanh hơn, và trên một số bài toán có thể cho kết quả tương đương hoặc thậm chí tốt hơn.

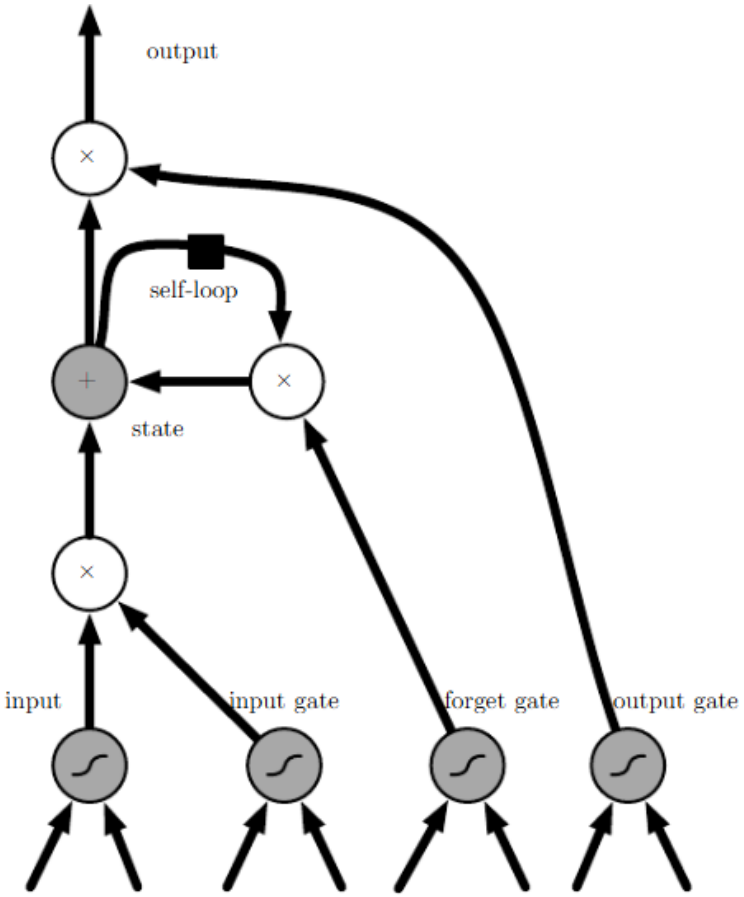
Tóm tắt:

- LSTM và GRU đều giúp giải quyết vấn đề gradient tiêu biến và hỗ trợ học các phụ thuộc dài hạn.
- LSTM có kiến trúc phức tạp hơn với ba cổng và một trạng thái ẩn phụ ($c(t)$).
- GRU đơn giản hơn, ít tham số hơn, và đôi khi học nhanh hơn.

Các mô hình RNN có cổng như LSTM và GRU hiện là lựa chọn mặc định trong nhiều ứng dụng học chuỗi và đã tạo nền tảng cho các mô hình sâu hơn và hiện đại hơn như attention, transformers,...

10.9.6 Bộ nhớ ngắn-dài hạn (LSTM)

Điểm sáng tạo quan trọng nhất của mô hình LSTM là việc thiết kế một *vòng lặp tự thân (self-loop)* trong mỗi đơn vị, cho phép tín hiệu đạo hàm truyền đi qua nhiều bước thời gian mà không bị mất mát đáng kể. Đây là phát minh nổi bật của Hochreiter và Schmidhuber (1997).



Hình 10.16: Sơ đồ khối của "tế bào" LSTM. Các tế bào này thay thế các đơn vị ẩn trong mạng hồi tiếp truyền thống. Một đầu vào sẽ được xử lý qua một neuron như bình thường. Nếu cổng vào sigmoid cho phép, giá trị này sẽ được cộng dồn vào trạng thái nội bộ. Trạng thái này có một vòng lặp tự thân tuyến tính, với trọng số được điều chỉnh bởi cổng quên. Đầu ra của tế bào sẽ được kiểm soát bởi cổng đầu ra. Các cổng điều khiển đều dùng hàm sigmoid, trong khi đơn vị đầu vào có thể sử dụng các hàm phi tuyến khác. Trạng thái nội bộ cũng có thể đóng vai trò là đầu vào cho các cổng.

Gers et al. (2000) sau đó đã đề xuất một cải tiến quan trọng: thay vì để trọng số vòng lặp là cố định, chúng ta có thể điều chỉnh nó dựa trên đầu vào bằng cách sử dụng một cổng mới – gọi là cổng quên (forget gate). Cơ chế này cho phép điều chỉnh linh hoạt khoảng thời gian thông tin được lưu trữ trong trạng thái nội bộ, dù các tham số của mạng vẫn cố định.

Một số ứng dụng thành công của LSTM: LSTM đã được áp dụng thành công trong nhiều bài toán khác nhau:

- Nhận dạng chữ viết tay tự do (Graves et al., 2009)
- Nhận dạng giọng nói (Graves et al., 2013)
- Sinh văn bản viết tay (Graves, 2013)
- Dịch máy (Sutskever et al., 2014)
- Sinh mô tả ảnh (Xu et al., 2015)

- Phân tích cú pháp câu (Vinyals et al., 2014a)

Cấu trúc của một LSTM Cell: Khác với RNN truyền thống chỉ có một hàm phi tuyến đơn giản, mỗi đơn vị LSTM bao gồm:

- Trạng thái nội bộ s_t với vòng lặp tự thân.
- Ba cổng để kiểm soát luồng thông tin:
  - Cổng quên (forget gate) f_t
  - Cổng vào (input gate) g_t
  - Cổng ra (output gate) q_t

Dưới đây là các công thức lan truyền tiến trong LSTM:

- Cổng quên quyết định phần nào của trạng thái cũ cần được loại bỏ:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

- Cập nhật trạng thái nội bộ:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \cdot \tanh\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

- Cổng vào điều khiển lượng thông tin mới được thêm vào trạng thái:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

- Cổng ra điều chỉnh phần nào của trạng thái sẽ trở thành đầu ra:

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

- Đầu ra của cell:

$$h_i^{(t)} = \tanh(s_i^{(t)}) \cdot q_i^{(t)}$$

Giải thích ký hiệu:

- $x^{(t)}$: đầu vào tại thời điểm t
- $h^{(t)}$: đầu ra ẩn tại thời điểm t
- $s^{(t)}$: trạng thái nội bộ tại thời điểm t
- U: trọng số từ đầu vào

- W: trọng số từ trạng thái trước
- b: độ lệch (bias)
- σ: hàm sigmoid
- $\tanh$: hàm tanh

Mở rộng thêm: Một số phiên bản LSTM mở rộng còn đưa cả $s^{(t)}$ vào làm đầu vào cho các cổng, giúp kiểm soát thông tin tốt hơn, nhưng điều này sẽ cần thêm ba bộ trọng số riêng biệt.

LSTM đã chứng minh hiệu quả trong việc học các mối quan hệ dài hạn, vượt trội hơn hẳn so với RNN đơn giản – kể cả trong các tác vụ nhân tạo lẫn ứng dụng thực tế có cấu trúc chuỗi phức tạp.

10.9.7 Các biến thể khác của mạng hồi tiếp có cổng (Other Gated RNNs)

Câu hỏi đặt ra: Liệu mọi thành phần trong kiến trúc LSTM đều cần thiết? Có thể rút gọn mô hình để đơn giản hơn mà vẫn hiệu quả không?

GRU (Gated Recurrent Unit) – được đề xuất bởi Cho et al. (2014b) – là một ví dụ tiêu biểu cho ý tưởng rút gọn LSTM. GRU đơn giản hơn nhưng vẫn giữ khả năng kiểm soát thông tin qua thời gian. Thay vì ba cổng như LSTM, GRU chỉ sử dụng hai cổng:

- Cổng cập nhật (update gate) u: quyết định nên giữ lại bao nhiêu trạng thái cũ.
- Cổng đặt lại (reset gate) r: điều chỉnh thông tin nào từ trạng thái cũ sẽ dùng để tính trạng thái mới.

Công thức cập nhật trạng thái ẩn tại thời điểm t như sau:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right) \quad (10.9)$$

Trong đó:

- $x^{(t)}$: đầu vào tại thời điểm t
- $h^{(t-1)}$: trạng thái ẩn tại thời điểm $t-1$
- u: cổng cập nhật
- r: cổng đặt lại

Các công thức cụ thể cho hai cổng này:

$$u_i^{(t)} = \sigma\left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)}\right) \tag{10.10}$$

$$r_i^{(t)} = \sigma\left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)}\right) \tag{10.11}$$

- Cổng cập nhật u giúp quyết định giữ lại hay thay thế trạng thái cũ. Khi u_i gần 1, mạng sẽ ưu tiên giữ $h_i^{(t-1)}$; khi gần 0, nó sẽ cập nhật hoàn toàn.
- Cổng đặt lại r cho phép mạng "tạm quên" trạng thái cũ khi tạo trạng thái mới – điều này tạo ra khả năng học phi tuyến mạnh mẽ.

Một số biến thể khác cũng được nghiên cứu:

- Dùng chung cổng forget/reset cho nhiều đơn vị ẩn
- Kết hợp cổng toàn cục (áp dụng cho toàn lớp) và cổng cục bộ (áp dụng từng đơn vị)

Tuy nhiên, theo các nghiên cứu thực nghiệm của Greff et al. (2015) và Jozefowicz et al. (2015), không có biến thể nào vượt trội hẳn so với LSTM hoặc GRU.

Greff et al. (2015) chỉ ra rằng cổng quên là yếu tố cốt lõi nhất trong kiến trúc LSTM. Ngoài ra, Jozefowicz et al. (2015) cũng phát hiện việc thêm một độ lệch bằng 1 vào cổng quên giúp mô hình học tốt hơn – mẹo này vốn được đề xuất bởi Gers et al. (2000).

10.9.8 Tối ưu hóa cho các mối quan hệ dài hạn (Optimization for Long-Term Dependencies)

Như đã trình bày ở các mục 8.2.5 và 10.7, các vấn đề như gradient biến mất và gradient bùng nổ thường xuyên xảy ra khi huấn luyện mạng hồi tiếp (RNN) qua nhiều bước thời gian. Khi làm việc với các chuỗi dài, gradient có thể trở nên cực kỳ nhỏ (gần như bằng 0) hoặc rất lớn, khiến cho việc huấn luyện trở nên khó khăn và dễ bị lỗi.

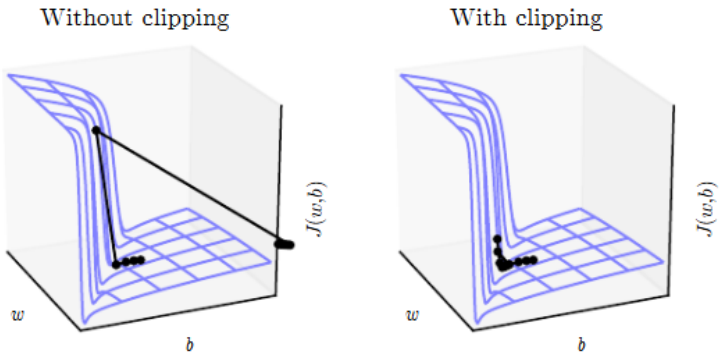
Martens và Sutskever (2011) đã đưa ra một ý tưởng thú vị: nếu gradient bậc nhất biến mất thì gradient bậc hai (đạo hàm cấp hai) cũng có thể biến mất theo. Các phương pháp tối ưu bậc hai, về mặt khái niệm, giống như việc *chia gradient bậc nhất cho gradient bậc hai* – hoặc trong không gian chiều cao, có thể hiểu là nhân gradient với nghịch đảo của ma trận Hessian.

Trong trường hợp cả hai loại đạo hàm này giảm cùng tốc độ, tỷ lệ giữa chúng có thể vẫn giữ ổn định, từ đó giúp cải thiện hiệu quả của quá trình tối ưu hóa. Tuy vậy, các phương pháp bậc hai có một số hạn chế: đòi hỏi chi phí tính toán cao, cần batch size lớn, và dễ bị mắc kẹt tại các *điểm yên ngựa (saddle points)* – nơi mà gradient bằng 0 nhưng không phải là cực tiểu hoặc cực đại.

Dù vậy, Martens và Sutskever (2011) vẫn đạt được kết quả khá hứa hẹn khi áp dụng các kỹ thuật tối ưu bậc hai. Sau này, Sutskever et al. (2013) cho thấy rằng các phương pháp đơn giản hơn – như sử dụng động lượng Nesterov kết hợp với khởi tạo cẩn thận – cũng có thể đạt hiệu quả tương đương. Độc giả quan tâm có thể xem thêm trong luận án

của Sutskever (2012) để hiểu rõ hơn.

Cuối cùng, cả hai phương pháp này đều bị thay thế bởi cách tiếp cận đơn giản hơn: sử dụng thuật toán SGD (Stochastic Gradient Descent) cho LSTM, thậm chí không cần dùng đến động lượng. Điều này phản ánh một xu hướng chung trong học máy: thay vì cố gắng phát triển thuật toán tối ưu phức tạp, chúng ta thường đạt kết quả tốt hơn bằng cách xây dựng mô hình dễ huấn luyện hơn.



Hình 10.17: Gradient clipping giúp gradient descent hoạt động ổn định hơn khi gặp các vùng dốc đứng trong không gian hàm mất mát. Những "vách đá" dốc đứng này xuất hiện trong mạng hồi tiếp khi mạng hành xử gần như tuyến tính – trọng số bị nhân lặp lại qua mỗi bước thời gian dẫn đến độ dốc tăng theo cấp số mũ. (Trái) Khi không dùng gradient clipping, thuật toán có thể rơi vào đáy hố, rồi bị gradient lớn đẩy xa khỏi vùng cần tìm. (Phải) Khi dùng gradient clipping, thuật toán phản ứng "êm ái" hơn – bước nhảy được giới hạn, giúp duy trì quá trình tối ưu trong vùng dốc hợp lý gần nghiệm. Hình được điều chỉnh từ Pascanu et al. (2013) với sự cho phép.

10.9.9 Cắt gradient

Như đã đề cập ở mục 8.2.4, khi làm việc với các hàm phi tuyến mạnh — chẳng hạn như những hàm do mạng hồi tiếp tạo ra qua nhiều bước thời gian — đạo hàm có thể trở nên rất lớn hoặc rất nhỏ. Điều này được minh họa rõ trong Hình 8.3 và Hình 10.17, nơi ta thấy rằng hàm mất mát theo tham số có thể có "cảnh quan" rất đặc biệt: các vùng rộng, bằng phẳng xen kẽ với những khu vực nhỏ nơi hàm thay đổi rất nhanh, tạo thành các "vách đá".

Khi gradient trở nên quá lớn, một bước cập nhật theo phương pháp gradient descent có thể đẩy tham số đi quá xa, rơi vào vùng mà giá trị hàm mất mát lại cao hơn, khiến công sức tối ưu trước đó bị "đạp đổ". Gradient chỉ thể hiện hướng dốc nhất trong một vùng nhỏ gần tham số hiện tại, nhưng nếu bước nhảy quá lớn, ta sẽ bước vào một vùng khác có độ cong cao và phản tác dụng. Do đó, ta thường dùng một tốc độ học nhỏ và ổn định để đảm bảo các bước nhảy hợp lý. Tuy nhiên, một bước phù hợp trong vùng gần như tuyến tính lại có thể không phù hợp khi ta bước vào vùng cong hơn — gây ra vấn đề nghiêm trọng.

Một giải pháp đơn giản nhưng hiệu quả là kỹ thuật cắt gradient. Ý tưởng này đã được áp dụng rộng rãi từ trước (Mikolov, 2012; Pascanu et al., 2013) với nhiều cách tiếp cận khác nhau. Một phương pháp là cắt từng phần tử của gradient trong từng phần tử của mini-batch (Mikolov, 2012), ngay trước khi cập nhật tham số. Cách khác là cắt toàn bộ chuẩn của vector gradient $||g||$ nếu nó vượt quá một ngưỡng định trước v (Pascanu et al.,

2013):

$$\text{if } ||g|| > v \tag{10.12}$$

$$g \leftarrow \frac{g}{||g||} v \tag{10.13}$$

Ở đây, v là ngưỡng giới hạn, và g là vector gradient sẽ được dùng để cập nhật các tham số. Phương pháp này áp dụng cùng một hệ số tỉ lệ để chuẩn hóa toàn bộ gradient của các tham số (bao gồm cả trọng số và bias), nhờ đó vẫn đảm bảo hướng di chuyển cùng chiều với gradient. Các nghiên cứu thực nghiệm cho thấy cả hai cách đều hoạt động hiệu quả tương đương.

Mặc dù hướng cập nhật vẫn giống gradient ban đầu, nhưng độ lớn của bước cập nhật bị giới hạn. Nhờ đó, nếu xảy ra hiện tượng gradient bùng nổ, bước nhảy sẽ không vượt khỏi giới hạn và không làm gián đoạn quá trình huấn luyện. Thực tế, nếu gradient quá lớn (thậm chí vô hạn hay NaN), ta có thể thực hiện một bước nhảy ngẫu nhiên có độ lớn bằng v, và điều này thường giúp thoát ra khỏi trạng thái bất ổn về mặt số học.

Cắt chuẩn gradient theo từng mini-batch không làm thay đổi hướng gradient trong từng mini-batch, nhưng nếu sau đó ta lấy trung bình các gradient đã bị cắt từ nhiều mini-batch thì kết quả sẽ khác với việc cắt chuẩn gradient sau khi cộng toàn bộ. Cụ thể, các ví dụ có gradient lớn hoặc nằm trong cùng một mini-batch với các ví dụ đó sẽ có ảnh hưởng nhỏ hơn đến hướng trung bình. Điều này trái ngược với phương pháp SGD thông thường, nơi hướng gradient là trung bình không thiên lệch của các gradient từ các ví dụ huấn luyện. Ngược lại, cắt chuẩn gradient làm xuất hiện một sai lệch heuristic — không đúng về mặt lý thuyết nhưng lại hữu ích về mặt thực nghiệm.

Đối với kỹ thuật cắt theo từng phần tử, hướng cập nhật không hẳn là hướng gradient đúng (hoặc của toàn mini-batch), nhưng nó vẫn đảm bảo là một hướng giúp giảm hàm mất mát. Một cách tiếp cận khác, được đề xuất bởi Graves (2013), là cắt gradient ngay trong quá trình lan truyền ngược tại các đơn vị ẩn. Tuy nhiên, hiện vẫn chưa có nghiên cứu so sánh hệ thống giữa các cách cắt khác nhau, nên ta có thể giả định rằng tất cả những phương pháp này đều mang lại hiệu quả tương tự.

10.9.10 Điều chỉnh để khuyến khích luồng thông tin (Regularizing to Encourage Information Flow)

Cắt gradient là một giải pháp hiệu quả cho vấn đề gradient bùng nổ, nhưng lại không giúp giải quyết được vấn đề gradient biến mất. Để khắc phục hiện tượng này và giúp mô hình học được các phụ thuộc dài hạn tốt hơn, một chiến lược đã được đề xuất là tạo ra những "con đường"trong đồ thị tính toán — dọc theo các con đường đó, tích của gradient trên các cung gần bằng 1.

Một cách để đạt được điều này là sử dụng kiến trúc LSTM, với cơ chế vòng lặp tự thân

và các cổng điều khiển như đã trình bày trong mục 10.10. Tuy nhiên, cũng có một hướng tiếp cận khác: điều chỉnh hoặc ràng buộc các tham số sao cho mô hình khuyến khích luồng thông tin đi qua mạng. Cụ thể, mục tiêu là đảm bảo gradient $\nabla h^{(t)} L$ — gradient của hàm mất mát đối với đầu ra ẩn tại thời điểm t — vẫn giữ được độ lớn khi lan truyền ngược qua các bước thời gian, ngay cả khi hàm mất mát chỉ được tính ở thời điểm cuối chuỗi.

Về mặt toán học, ta muốn điều kiện sau được đảm bảo:

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} \nabla h^{(t)} L \quad \text{duy trì độ lớn tương đương với} \quad \nabla h^{(t)} L. \tag{10.14}$$

Để hiện thực hóa mục tiêu đó, Pascanu et al. (2013) đã đề xuất một bộ điều chỉnh (regularizer) có dạng:

$$\Omega = \sum_t \left(\left\| \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \nabla h^{(t)} L \right\| - \|\nabla h^{(t)} L\| \right)^2. \tag{10.15}$$

Việc tính gradient của bộ điều chỉnh này có thể gây khó khăn, nhưng tác giả đề xuất một cách làm đơn giản: coi các vector $\nabla h^{(t)} L$ là hằng số. Do đó, trong quá trình lan truyền ngược, ta không cần tính đạo hàm qua các vector này. Dù đây là một phép xấp xỉ, nhưng nó cho thấy hiệu quả trong thực nghiệm.

Các thí nghiệm chỉ ra rằng khi kết hợp bộ điều chỉnh này với kỹ thuật cắt chuẩn gradient (đã giúp giải quyết vấn đề bùng nổ gradient), mô hình RNN có thể học được các phụ thuộc dài hơn một cách rõ rệt. Bởi vì bộ điều chỉnh này giữ mô hình RNN hoạt động gần ngưỡng bùng nổ gradient, nên việc cắt gradient trở nên cực kỳ quan trọng — nếu không, mô hình sẽ dễ bị phá vỡ bởi gradient quá lớn.

Tuy nhiên, cũng cần lưu ý rằng cách tiếp cận này không hiệu quả bằng LSTM trong các bài toán phức tạp với dữ liệu phong phú, ví dụ như trong mô hình ngôn ngữ. Trong những trường hợp như vậy, LSTM vẫn là lựa chọn đáng tin cậy hơn nhờ khả năng kiểm soát luồng thông tin tốt hơn.

10.10 Bộ nhớ rõ ràng (Explicit Memory)

Trong trí tuệ nhân tạo, hệ thống cần có khả năng lưu trữ kiến thức, và việc học tập chính là một cách để tiếp thu kiến thức đó. Đây cũng là động lực thúc đẩy sự phát triển của các kiến trúc mạng nơ-ron sâu với quy mô lớn. Tuy nhiên, không phải loại kiến thức nào cũng giống nhau. Có những kiến thức mang tính ngầm hiểu, khó diễn đạt thành lời — chẳng hạn như cách đi bộ, hoặc làm sao để phân biệt chó với mèo. Những kiến thức này thường được học một cách vô thức, và con người không cần phải lý giải rõ ràng để sử dụng chúng.

Ngược lại, còn có những kiến thức mang tính rõ ràng, có thể dễ dàng diễn đạt bằng ngôn ngữ — như các kiến thức phổ thông kiểu "mèo là một loài động vật", hoặc những thông tin cụ thể như "cuộc họp với nhóm bán hàng diễn ra lúc 3 giờ chiều ở phòng 141".

Mạng nơ-ron tỏ ra rất hiệu quả trong việc học các dạng kiến thức tiềm ẩn như trên, nhưng lại gặp nhiều khó khăn trong việc ghi nhớ các sự kiện hay sự thật cụ thể. Phương pháp gradient ngẫu nhiên (SGD) yêu cầu dữ liệu được lặp lại nhiều lần mới có thể "in sâu" vào các tham số của mạng. Ngay cả khi đã học xong, thông tin vẫn không được lưu trữ chính xác tuyệt đối.

Graves và các cộng sự (2014) cho rằng nguyên nhân chính là do mạng nơ-ron không có một hệ thống bộ nhớ làm việc giống như con người — tức là bộ nhớ có thể nhanh chóng lưu và xử lý các thông tin quan trọng để đạt được mục tiêu đang theo đuổi.

Việc bổ sung một thành phần bộ nhớ rõ ràng có thể giúp hệ thống AI không chỉ lưu và truy xuất thông tin cụ thể một cách nhanh chóng và có chủ đích, mà còn có khả năng lập luận tuần tự dựa trên những thông tin đó. Để làm được điều này, mạng nơ-ron cần phải xử lý thông tin theo nhiều bước liên tiếp, trong đó mỗi bước có thể thay đổi cách tiếp nhận dữ liệu đầu vào. Điều này rất cần thiết nếu ta muốn hệ thống có thể suy luận — thay vì chỉ phản xạ trực tiếp một cách "trực giác" như trước.

Một giải pháp cho vấn đề này là kiến trúc mạng bộ nhớ (memory networks) do Weston và cộng sự (2014) đề xuất. Các mạng này có một tập các ô bộ nhớ, có thể được truy cập thông qua một cơ chế địa chỉ. Tuy nhiên, phiên bản ban đầu của memory networks vẫn cần các tín hiệu giám sát chỉ dẫn cụ thể cách sử dụng bộ nhớ.

Để khắc phục hạn chế đó, Graves và cộng sự (2014) đã đề xuất kiến trúc Máy Turing nơ-ron (Neural Turing Machine – NTM), cho phép mô hình học cách đọc và ghi dữ liệu vào bộ nhớ một cách tự động — mà không cần giám sát rõ ràng về thao tác nào nên thực hiện. Cơ chế này sử dụng kỹ thuật chú ý mềm (soft attention mechanism), tương tự như phương pháp của Bahdanau et al. (2015) được trình bày trong mục 12.4.5.1.

Cơ chế địa chỉ mềm đã trở thành một chuẩn phổ biến trong các kiến trúc liên quan. Nó cho phép mô phỏng lại các thao tác truy cập bộ nhớ như một thuật toán, nhưng vẫn có thể tối ưu hóa bằng gradient. Mỗi ô bộ nhớ trong hệ thống này có thể xem như một dạng mở rộng của ô trạng thái trong LSTM hoặc GRU. Điểm khác biệt là: mạng nơ-ron sẽ sinh ra một trạng thái nội bộ, dùng để chọn ô bộ nhớ sẽ được đọc hoặc ghi — tương tự như việc CPU trong máy tính đọc hoặc ghi dữ liệu tại một địa chỉ cụ thể trong RAM.

Việc buộc mô hình phải sinh ra đúng địa chỉ ô nhớ (theo số nguyên) để truy cập là một bài toán tối ưu rất khó. Để tránh vấn đề này, NTM trong thực tế thường thực hiện đọc hoặc ghi từ nhiều ô bộ nhớ cùng lúc, chứ không chỉ một ô. Khi đọc, mô hình lấy giá trị trung bình có trọng số từ nhiều ô; khi ghi, mô hình cũng điều chỉnh nhiều ô với các mức độ khác nhau. Các trọng số này thường được tính sao cho tập trung vào một vài ô — ví dụ như sử dụng hàm softmax.

Nhờ việc các trọng số này vẫn có đạo hàm khác 0, nên ta có thể huấn luyện toàn bộ hệ thống bằng phương pháp gradient descent. Điều này giúp mạng học được các chiến lược truy xuất và ghi nhớ như một hệ thống bộ nhớ thực thụ, mà vẫn có thể tối ưu từ đầu đến cuối. Đạo hàm của các trọng số dùng để truy cập bộ nhớ cho ta biết rằng nên tăng hay

giảm mỗi trọng số. Tuy nhiên, trong thực tế, đạo hàm thường chỉ lớn đối với những địa chỉ bộ nhớ đang có trọng số cao — tức là những ô bộ nhớ mà mô hình đang tập trung đọc hoặc ghi vào. Các ô bộ nhớ này thường không chỉ chứa một giá trị đơn lẻ (scalar), mà được mở rộng thành các vector nhiều chiều.

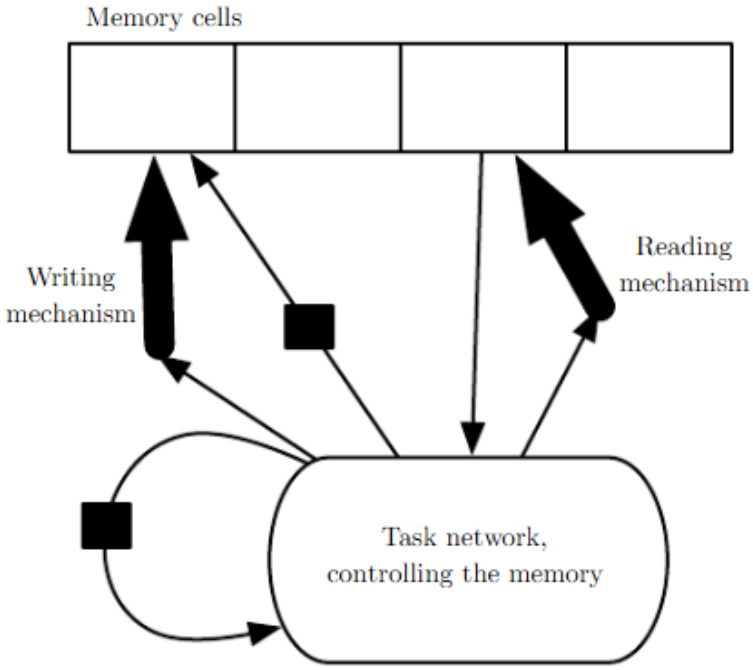
Có hai lý do chính để thiết kế ô bộ nhớ chứa vector thay vì chỉ một giá trị:

- Chi phí truy cập bộ nhớ: Khi truy cập bộ nhớ, chúng ta phải tính toán các hệ số địa chỉ cho nhiều ô, điều này có thể tốn kém về mặt tính toán. Nếu mỗi lần truy cập, ta thu được một vector thông tin đầy đủ thay vì một giá trị nhỏ, thì chi phí đó sẽ được sử dụng hiệu quả hơn.
- Địa chỉ hóa theo nội dung (content-based addressing): Khi ô bộ nhớ chứa vector, ta có thể tìm kiếm dựa trên nội dung bằng cách khớp một phần của vector. Cách này giống với khả năng của con người: chúng ta có thể nhớ một bài hát chỉ từ vài từ của đoạn điệp khúc. Ví dụ, mô hình có thể được yêu cầu “truy xuất bài hát có đoạn điệp khúc ’We all live in a yellow submarine.’”

Địa chỉ hóa theo nội dung trái ngược với địa chỉ hóa theo vị trí (location-based addressing), nơi ta chỉ truy xuất dữ liệu dựa trên chỉ mục ô bộ nhớ. Trong khi địa chỉ hóa theo vị trí đơn giản hơn, địa chỉ hóa theo nội dung linh hoạt và thông minh hơn — cho phép mô hình tìm kiếm dựa trên ý nghĩa.

Một lý do khác để sử dụng ô bộ nhớ có giá trị vector là khả năng giữ lại thông tin qua nhiều bước thời gian. Nếu nội dung bộ nhớ không bị ghi đè trong quá trình huấn luyện, nó có thể truyền thông tin hiệu quả về phía trước qua thời gian, và gradient cũng có thể truyền ngược trở lại qua thời gian mà không gặp vấn đề biến mất hoặc bùng nổ.

Cách tiếp cận bộ nhớ rõ ràng này được minh họa trong Hình 10.18, nơi một "mạng nơ-ron tác vụ"được kết nối với một bộ nhớ ngoài. Dù mạng nơ-ron tác vụ có thể là mạng lan truyền xuôi hoặc mạng hồi tiếp, nhưng hệ thống tổng thể vẫn là một mạng hồi tiếp. Mạng này học cách kiểm soát bộ nhớ — tức là quyết định đọc và ghi vào đâu trong bộ nhớ.



Hình 10.18: Sơ đồ của một mạng với bộ nhớ rõ ràng, thể hiện các thành phần chính của máy Turing nơ-ron. Trong hình, phần "mạng tác vụ" (mạng hồi tiếp ở phía dưới) xử lý thông tin và tương tác với "bộ nhớ" (các ô nhớ ở phía trên). Mạng học cách điều khiển việc đọc và ghi vào bộ nhớ, thể hiện bằng các mũi tên hướng đến các địa chỉ cụ thể.

Bộ nhớ rõ ràng giúp mô hình học được những tác vụ mà các RNN hay LSTM thông thường không xử lý hiệu quả. Một trong những lý do quan trọng là vì thông tin — và cả gradient — có thể truyền đi qua thời gian rất dài. Điều này giúp mô hình "ghi nhớ"và học từ những sự kiện cách nhau xa trong chuỗi dữ liệu.

Ngoài cách dùng trung bình có trọng số để truy cập bộ nhớ, một phương pháp khác là coi các hệ số địa chỉ như các xác suất, sau đó chỉ đọc ngẫu nhiên một ô duy nhất trong mỗi bước (Zaremba và Sutskever, 2015). Phương pháp này đòi hỏi phải dùng các kỹ thuật tối ưu hóa chuyên biệt để xử lý các quyết định rời rạc, sẽ được trình bày trong mục 20.9.1. Tuy nhiên, đến thời điểm hiện tại, huấn luyện các kiến trúc ngẫu nhiên như vậy vẫn khó khăn hơn nhiều so với các phương pháp xác định dùng quyết định mềm (soft decisions) cho phép lan truyền ngược (backpropagation).

Dù chọn phương pháp mềm hay ngẫu nhiên-cứng (stochastic and hard), cơ chế chọn địa chỉ bộ nhớ có hình thức tương tự như cơ chế chú ý (attention mechanism). Cơ chế chú ý lần đầu được phổ biến rộng rãi trong bài toán dịch máy (Bahdanau et al., 2015) và sẽ được thảo luận kỹ hơn ở mục 12.4.5.1. Thực tế, ý tưởng chú ý đã xuất hiện sớm hơn — trong bài toán tạo chữ viết tay (Graves, 2013), với cơ chế chú ý hạn chế chỉ tiến dần theo thời gian.

Trong các hệ thống dịch máy hay mạng bộ nhớ, chú ý cho phép tập trung vào bất kỳ vị trí nào trong bộ nhớ ở mỗi bước thời gian — không cần phải theo thứ tự tuần tự như trước.

Tổng kết: Mạng nơ-ron hồi tiếp (RNN) là một công cụ quan trọng để mở rộng học sâu

sang các loại dữ liệu dạng chuỗi. Qua các phần trên, chúng ta đã thấy cách kết hợp RNN với bộ nhớ rõ ràng để giúp mô hình học tốt hơn trong các bài toán cần ghi nhớ và suy luận lâu dài. Trong phần tiếp theo, chúng ta sẽ tìm hiểu cách chọn, sử dụng và áp dụng các công cụ này vào các ứng dụng thực tế.

CHƯƠNG 11. PHƯƠNG PHÁP LUẬN THỰC TIỄN

Để áp dụng thành công các kỹ thuật học sâu trong thực tế, người thực hành không chỉ cần hiểu biết về thuật toán và toán học, mà còn phải biết cách đưa ra các quyết định phù hợp khi thiết kế và huấn luyện hệ thống. Cụ thể, người làm học máy cần biết cách chọn thuật toán thích hợp với bài toán đang giải quyết, theo dõi quá trình huấn luyện, đánh giá hiệu quả dựa trên dữ liệu thu được, và điều chỉnh hệ thống theo cách hiệu quả nhất.

Trong thực tế, khi phát triển một hệ thống học máy, người ta thường xuyên đặt ra những câu hỏi như sau:

- Có cần phải thu thập thêm dữ liệu nữa không?
- Mô hình đang sử dụng liệu đã phù hợp hay quá phức tạp (overfitting) hoặc quá đơn giản (underfitting)?
- Có nên thêm hoặc bỏ đi các kỹ thuật regularization?
- Cần cải thiện thuật toán tối ưu hay không?
- Liệu có lỗi gì trong quá trình lập trình hay không?

Nếu ta cứ thử nghiệm ngẫu nhiên các câu hỏi này thì sẽ rất mất thời gian. Chính vì thế, cần có một phương pháp luận rõ ràng để định hướng tốt hơn.

Nhiều người nghĩ rằng một chuyên gia học máy giỏi là người phải biết rất nhiều thuật toán phức tạp và giỏi toán. Thực tế cho thấy, việc áp dụng đúng một thuật toán phổ biến nhưng đơn giản thường đem lại kết quả tốt hơn rất nhiều so với việc cố gắng dùng một kỹ thuật phức tạp mà không hiểu rõ.

Quá trình xây dựng hệ thống học máy trong thực tế thường gồm các bước sau:

1. Xác định mục tiêu rõ ràng: Cần chọn ra một chỉ số đánh giá hiệu quả (error metric) phù hợp với bài toán, ví dụ như: độ chính xác (Accuracy), F1-score, Precision, Recall, v.v.
2. Xây dựng hệ thống đầu-cuối: Nên cố gắng hoàn thiện một pipeline đơn giản nhưng có thể chạy xuyên suốt ngay từ ban đầu, sau đó mới dần cải thiện thêm.
3. Giám sát và chẩn đoán: Đặt ra các công cụ đo lường để phát hiện những điểm yếu (bottleneck) của hệ thống. Ví dụ kiểm tra xem vấn đề nằm ở overfitting, underfitting, do dữ liệu hay lỗi lập trình.
4. Lặp lại và điều chỉnh hệ thống: Dựa trên các phân tích thu được, thực hiện các thay đổi nhỏ như thu thập thêm dữ liệu, điều chỉnh các siêu tham số, hoặc thay đổi mô hình.

Một ví dụ thực tế nổi bật về việc áp dụng quy trình này là hệ thống nhận diện số địa chỉ từ hình ảnh Street View của Google (Goodfellow et al., 2014). Mục đích của dự án này là

bổ sung các địa chỉ nhà vào Google Maps:

- Xe Street View chụp ảnh các tòa nhà, đồng thời ghi lại tọa độ GPS của từng ảnh.
- Các ảnh được sử dụng để huấn luyện một mạng nơ-ron tích chập (CNN) nhằm nhận diện số nhà xuất hiện trong ảnh.
- Kết quả từ hệ thống được dùng để cập nhật thêm vị trí các địa chỉ mới vào bản đồ.

Việc phát triển thành công hệ thống này là một ví dụ tiêu biểu cho việc áp dụng hiệu quả phương pháp luận thực tế trong lĩnh vực học sâu.

11.1 Các chỉ số đánh giá hiệu năng (Performance Metrics)

Khi bắt đầu phát triển hệ thống học máy, điều quan trọng đầu tiên bạn cần làm là xác định rõ mục tiêu đánh giá—cụ thể là chọn ra một chỉ số lỗi (error metric) phù hợp. Việc này sẽ định hướng toàn bộ quá trình thiết kế mô hình về sau. Bạn cũng cần đặt ra một mức hiệu năng mong muốn để so sánh kết quả thực tế với mục tiêu đã đề ra.

Trong thực tế, không thể hy vọng một mô hình đạt được lỗi bằng 0 tuyệt đối. Mức lỗi thấp nhất có thể đạt được được gọi là Bayes error. Đây là mức lỗi lý tưởng trong trường hợp bạn có vô hạn dữ liệu huấn luyện và biết chính xác phân phối xác suất của dữ liệu. Tuy nhiên, vẫn luôn tồn tại một số rào cản khiến bạn không thể đạt đến Bayes error:

- Dữ liệu đầu vào có thể không chứa đầy đủ thông tin cần thiết để dự đoán kết quả chính xác.
- Hệ thống bạn đang xây dựng có thể mang tính ngẫu nhiên một cách tự nhiên (stochastic).
- Bạn bị hạn chế bởi lượng dữ liệu huấn luyện thực tế có được.

Có nhiều lý do khiến lượng dữ liệu thu thập được bị hạn chế, chẳng hạn chi phí cao, mất nhiều thời gian, hoặc rủi ro sức khỏe (như xét nghiệm y tế). Trong các ứng dụng thực tế, bạn cần cân nhắc kỹ giữa chi phí và lợi ích của việc thu thập thêm dữ liệu. Ngược lại, khi làm nghiên cứu trên các bộ dữ liệu tiêu chuẩn, bạn thường không thể tự ý thay đổi dữ liệu.

Vậy làm sao để biết mức hiệu năng mà bạn đặt ra có hợp lý không?

- Trong môi trường học thuật: tham khảo kết quả benchmark từ các nghiên cứu trước.
- Trong thực tế: xác định mức hiệu năng cần thiết dựa trên các tiêu chí an toàn, hiệu quả kinh tế, hoặc khả năng chấp nhận của người dùng.

Sau khi xác định mức hiệu năng mong muốn, bạn có thể dùng nó làm cơ sở để chọn kiến trúc mô hình, dữ liệu, và thuật toán tối ưu phù hợp nhất.

Không chỉ mức độ hiệu năng mong muốn mà loại chỉ số đánh giá cũng rất quan trọng. Trong học máy, chỉ số dùng để đánh giá hiệu năng (performance metric) thường khác với hàm mất mát (cost function) dùng khi huấn luyện mô hình.

Một chỉ số đánh giá đơn giản và phổ biến là độ chính xác (accuracy) hoặc tỉ lệ lỗi. Tuy

nhiên, trong nhiều ứng dụng, bạn cần những chỉ số cao cấp hơn.

Ví dụ, nếu xây dựng một hệ thống lọc thư rác (spam), bạn cần phân biệt rõ hai loại lỗi:

- False positive (FP): Email hợp lệ bị đánh dấu nhầm thành spam.
- False negative (FN): Email spam nhưng không bị phát hiện, lọt vào hộp thư chính.

Trong trường hợp này, lỗi FP nghiêm trọng hơn FN rất nhiều. Vì thế, thay vì chỉ tính tỉ lệ lỗi chung, bạn cần đo chi phí tổng thể với trọng số ưu tiên cho lỗi FP cao hơn FN.

Một ví dụ khác: giả sử bạn muốn phát hiện một căn bệnh rất hiếm gặp, tỷ lệ chỉ khoảng 1 trên 1 triệu. Một mô hình luôn dự đoán "không mắc bệnh"sẽ đạt độ chính xác tới 99,9999% nhưng lại hoàn toàn vô dụng. Đây là ví dụ cho thấy độ chính xác không phải luôn là chỉ số phản ánh đúng hiệu năng.

Trong những tình huống như vậy, người ta dùng hai chỉ số đánh giá khác:

- Precision (độ chính xác dự đoán): Tỉ lệ các dự đoán dương tính mà đúng thực sự.
- Recall (độ bao phủ): Tỉ lệ các trường hợp dương tính thật sự được phát hiện bởi mô hình.

Ví dụ cụ thể:

- Nếu mô hình luôn dự đoán "không bệnh"$\Rightarrow$ Precision = 100%, Recall = 0%.
- Nếu mô hình luôn dự đoán "có bệnh"$\Rightarrow$ Recall = 100%, Precision rất thấp.

Nếu mô hình đưa ra một xác suất dự đoán $\hat{y} = P(y = 1|x)$, bạn cần chọn một ngưỡng cụ thể để quyết định dương tính hay âm tính. Việc điều chỉnh ngưỡng sẽ dẫn đến sự đánh đổi giữa precision và recall.

Để đánh giá tổng hợp hai chỉ số này, ta dùng chỉ số F-score:

$$F = \frac{2pr}{p + r}$$

trong đó p là precision, r là recall.

Một cách đánh giá tổng quát hơn nữa là đo diện tích dưới đường cong Precision-Recall (PR-AUC).

Trong nhiều tình huống thực tế, hệ thống học máy có thể được thiết kế để từ chối đưa ra quyết định nếu nó không đủ tự tin. Cách làm này rất hữu ích trong những trường hợp quyết định sai có hậu quả nghiêm trọng và có sự hỗ trợ của con người.

Ví dụ rõ ràng là dự án trích xuất địa chỉ từ ảnh chụp Street View. Nhiệm vụ của hệ thống này là chuyển đổi số nhà trên ảnh thành địa chỉ trên bản đồ. Trong trường hợp này, thông tin sai có thể làm giảm chất lượng của bản đồ một cách đáng kể.

Vì vậy, một chiến lược an toàn hơn là chỉ cập nhật địa chỉ lên bản đồ khi hệ thống chắc chắn kết quả đúng. Khi hệ thống không đủ tự tin, ảnh sẽ được chuyển sang cho một người

xử lý thủ công.

Trong tình huống như vậy, chỉ số đánh giá hiệu năng phù hợp là coverage—tức tỉ lệ số mẫu mà hệ thống quyết định xử lý tự động:

- Coverage: *Tỉ lệ phần trăm các mẫu mà hệ thống đưa ra quyết định*.
- Có một sự đánh đổi rõ ràng giữa coverage và accuracy: bạn luôn có thể đạt được độ chính xác 100% bằng cách từ chối tất cả các mẫu, nhưng lúc này coverage = 0%.

Với bài toán Street View, mục tiêu đặt ra là đạt độ chính xác ngang bằng con người (khoảng 98%) và vẫn đảm bảo coverage ít nhất là 95%. Nghĩa là hệ thống phải xử lý tự động được đa số các trường hợp nhưng vẫn phải giữ độ chính xác cao.

Ngoài coverage, tùy từng ứng dụng sẽ có thêm các chỉ số hiệu năng chuyên biệt khác. Một số ví dụ phổ biến gồm:

- Tỷ lệ nhấp chuột (CTR): dùng trong quảng cáo online.
- Mức độ hài lòng của người dùng: thường đo bằng khảo sát.
- Các tiêu chí đặc thù khác: phụ thuộc vào từng lĩnh vực cụ thể như y tế, tài chính, công nghiệp,...

Điều quan trọng nhất là, trước khi bắt đầu xây dựng hoặc cải tiến một hệ thống học máy, bạn phải xác định rõ đâu là chỉ số hiệu năng chính. Tất cả thiết kế, lựa chọn mô hình và đánh giá hiệu quả sau đó đều xoay quanh chỉ số này. Nếu không có một mục tiêu rõ ràng, rất khó để biết những thay đổi bạn làm đang cải thiện hay làm tệ đi hệ thống của mình.

11.2 Mô hình cơ sở mặc định (Default Baseline Models)

Sau khi đã xác định rõ các chỉ số hiệu suất và mục tiêu cụ thể, bước tiếp theo cần làm trong bất kỳ ứng dụng học máy thực tế nào là nhanh chóng xây dựng một hệ thống đầu-cuối (end-to-end) đơn giản. Việc này nhằm tạo ra một mô hình cơ sở ban đầu (baseline) để làm nền tảng cho các bước cải tiến sau này.

Tuỳ thuộc vào độ phức tạp của bài toán, đôi khi bạn thậm chí không cần phải dùng đến các kỹ thuật học sâu ngay từ đầu. Nếu bài toán đơn giản, có thể giải quyết chỉ bằng việc điều chỉnh vài trọng số tuyến tính thì tốt nhất bạn nên thử nghiệm trước với các mô hình thống kê đơn giản, chẳng hạn như hồi quy logistic (logistic regression).

Ngược lại, nếu bạn biết chắc chắn bài toán của mình thuộc loại rất khó—còn gọi là các bài toán AI-complete (ví dụ như nhận diện đối tượng, nhận dạng giọng nói hay dịch máy), thì tốt nhất nên bắt đầu luôn với một mô hình học sâu phù hợp.

Việc lựa chọn mô hình cơ sở chủ yếu phụ thuộc vào cấu trúc của dữ liệu đầu vào:

- Nếu dữ liệu đầu vào của bạn là các vector có kích thước cố định, hãy sử dụng mạng truyền thẳng (feedforward network) với các lớp kết nối đầy đủ (fully-connected layers).

- Nếu dữ liệu có cấu trúc không gian rõ ràng, ví dụ hình ảnh, bạn nên chọn mạng nơ-ron tích chập (CNN).
- Nếu dữ liệu đầu vào hoặc đầu ra là dạng chuỗi (sequence), thì hãy dùng mạng hồi tiếp có cổng (LSTM hoặc GRU).

Trong các mô hình này, thông thường bạn sẽ sử dụng các hàm kích hoạt dạng phi tuyến từng phần (piecewise linear), chẳng hạn như ReLU hoặc các phiên bản cải tiến của nó (Leaky ReLU, PReLU, Maxout).

Đối với thuật toán tối ưu hóa, có hai lựa chọn thường gặp và hiệu quả:

- SGD với momentum, thường đi kèm với việc giảm tốc độ học dần dần (learning rate decay).
  - Các phương pháp giảm tốc độ học gồm: giảm tuyến tính tới một giá trị nhỏ nhất, giảm theo hàm mũ, hoặc giảm mỗi khi hiệu năng trên tập kiểm định (validation) không còn cải thiện nữa (reduce LR on plateau).
- Adam là lựa chọn phổ biến khác, rất dễ dùng và hiệu quả trong nhiều trường hợp.

Batch normalization cũng thường rất hữu ích, đặc biệt khi bạn dùng CNN hoặc các hàm kích hoạt kiểu sigmoid. Bạn có thể chưa cần đưa batch normalization ngay vào mô hình ban đầu, nhưng nếu quá trình huấn luyện gặp khó khăn, hãy bổ sung nó sớm.

Trừ khi bạn có một lượng dữ liệu cực kỳ lớn (hàng chục triệu mẫu), bạn nên đưa vào các kỹ thuật regularization nhẹ ngay từ đầu để tránh overfitting:

- Early stopping (dừng sớm): gần như luôn luôn hữu ích.
- Dropout: rất dễ triển khai, hiệu quả cao, tương thích với hầu hết các loại mạng.
- Batch normalization đôi khi cũng đóng vai trò giảm overfitting, thậm chí thay thế dropout trong một số trường hợp.

Nếu bài toán bạn làm tương tự với một vấn đề đã từng được nghiên cứu kỹ trước đó, hãy bắt đầu bằng cách sao chép lại các mô hình và thuật toán đã chứng minh hiệu quả trong các nghiên cứu trước. Thậm chí, bạn có thể sử dụng trực tiếp các mô hình đã được huấn luyện sẵn để làm baseline.

Ví dụ điển hình là việc tái sử dụng các đặc trưng từ một mạng CNN đã được huấn luyện trên bộ dữ liệu ImageNet cho các bài toán thị giác máy tính khác.

Một câu hỏi thường gặp là liệu có nên bắt đầu bằng các kỹ thuật học không giám sát (unsupervised learning) hay không. Điều này thực ra phụ thuộc nhiều vào lĩnh vực ứng dụng:

- Với lĩnh vực xử lý ngôn ngữ tự nhiên (NLP), các kỹ thuật học không giám sát (như embedding từ ngữ) đem lại lợi ích rất rõ ràng.
- Trong thị giác máy tính, các phương pháp học không giám sát hiện tại thường chỉ

hữu ích trong các thiết lập bán giám sát (semi-supervised learning), khi lượng nhãn rất ít.

Nếu ứng dụng của bạn thuộc vào lĩnh vực mà học không giám sát đã chứng minh hiệu quả, hãy thêm nó vào mô hình baseline ngay từ đầu. Ngược lại, bạn chỉ nên dùng học không giám sát nếu bài toán của bạn vốn đã là bài toán không giám sát—hoặc bổ sung vào sau nếu mô hình của bạn đang bị overfit rõ rệt.

11.3 Xác định có nên thu thập thêm dữ liệu hay không

Sau khi đã xây dựng xong một hệ thống học sâu đầu tiên có thể chạy trọn vẹn từ đầu đến cuối, bước tiếp theo là đánh giá hiệu suất mô hình và tìm cách cải thiện nó. Một sai lầm phổ biến, đặc biệt với những người mới học, là thử thay đổi nhiều thuật toán khác nhau để cải thiện kết quả. Tuy nhiên, trong nhiều trường hợp, cách cải thiện hiệu quả hơn là thu thập thêm dữ liệu huấn luyện thay vì thay đổi thuật toán.

Trước tiên, cần đánh giá hiệu suất mô hình trên tập huấn luyện:

Nếu mô hình cho kết quả kém ngay trên tập huấn luyện, nghĩa là nó chưa khai thác được hết tiềm năng từ dữ liệu hiện có. Trong tình huống này, không nên thu thập thêm dữ liệu, mà nên tập trung vào các hướng sau:

- Tăng độ phức tạp của mô hình—ví dụ: thêm nhiều tầng hơn hoặc tăng số lượng nơ-ron.
- Tinh chỉnh các siêu tham số, đặc biệt là tốc độ học (learning rate).
- Kiểm tra lại chất lượng dữ liệu—dữ liệu có thể nhiễu hoặc thiếu các đặc trưng quan trọng.

Ngược lại, nếu mô hình hoạt động tốt trên tập huấn luyện nhưng lại kém trên tập kiểm tra, thì đây là dấu hiệu của hiện tượng overfitting. Trong trường hợp này, việc thu thập thêm dữ liệu huấn luyện có thể giúp cải thiện đáng kể khả năng tổng quát hóa của mô hình. Một số yếu tố cần cân nhắc khi quyết định có nên thu thập thêm dữ liệu hay không:

- Chi phí và tính khả thi của việc thu thập thêm dữ liệu trong ngữ cảnh thực tế.
- So sánh chi phí này với chi phí và hiệu quả của các phương pháp cải tiến khác, chẳng hạn như regularization.
- Ước lượng số lượng dữ liệu cần thêm để có cải thiện rõ rệt về hiệu suất.

Ví dụ, các công ty công nghệ lớn như Google hay Facebook thường có khả năng thu thập dữ liệu từ hàng triệu người dùng, nên họ có thể dễ dàng mở rộng tập dữ liệu để cải thiện mô hình. Tuy nhiên, trong các lĩnh vực như y học, việc thu thập dữ liệu mới lại rất tốn kém, khó khăn, hoặc tiềm ẩn rủi ro cho bệnh nhân.

Trong một số trường hợp, thay vì thu thập thêm dữ liệu, bạn có thể thử cải thiện khả năng tổng quát hóa của mô hình bằng cách áp dụng các kỹ thuật regularization:

- Điều chỉnh hệ số phạt (weight decay).

- Áp dụng dropout.
- Giảm độ phức tạp của mô hình.

Nếu bạn đã thử các biện pháp regularization mà mô hình vẫn bị overfit, thì đó là lúc nên nghiêm túc xem xét việc thu thập thêm dữ liệu.

Để định lượng xem cần thêm bao nhiêu dữ liệu, một phương pháp hữu ích là vẽ biểu đồ thể hiện mối quan hệ giữa kích thước tập huấn luyện và lỗi kiểm tra (generalization error). Từ đường cong đó, bạn có thể ngoại suy (extrapolate) để ước lượng lượng dữ liệu cần thiết để đạt được mức lỗi mong muốn.

Thông thường, việc tăng thêm một lượng nhỏ dữ liệu sẽ không tạo ra sự cải thiện đáng kể. Do đó, bạn nên thử các mức tăng theo cấp số nhân—ví dụ: nhân đôi kích thước tập huấn luyện mỗi lần thử nghiệm.

Tóm lại, việc quyết định có nên thu thập thêm dữ liệu hay không là một phần rất quan trọng trong quy trình phát triển mô hình học máy. Để đưa ra quyết định đúng đắn, bạn cần đánh giá kỹ lưỡng hiệu suất hiện tại của mô hình, chi phí thu thập dữ liệu, cũng như các lựa chọn thay thế khả thi khác.

11.4 Chọn hyperparameter

Hầu hết các thuật toán học sâu đều đi kèm với nhiều hyperparameter — tức là các tham số mà mô hình không tự học được từ dữ liệu, mà bạn phải chọn trước khi bắt đầu huấn luyện. Những hyperparameter này đóng vai trò then chốt, vì chúng ảnh hưởng đến cả quá trình huấn luyện lẫn hiệu suất của mô hình khi triển khai trên dữ liệu mới.

Một số hyperparameter chủ yếu quyết định đến thời gian huấn luyện và mức sử dụng tài nguyên, ví dụ như kích thước batch hay số tầng trong mạng. Một số khác thì ảnh hưởng trực tiếp đến chất lượng đầu ra của mô hình, như khả năng mô hình khái quát hóa tốt đến các dữ liệu chưa từng thấy.

Có hai phương pháp chính để lựa chọn các hyperparameter:

- Chọn thủ công: Người dùng tự điều chỉnh các giá trị hyperparameter dựa trên kinh nghiệm và kiến thức về mô hình. Phương pháp này thường được sử dụng khi bạn hiểu rõ các siêu tham số ảnh hưởng đến mô hình ra sao, và có thể thu hẹp không gian tìm kiếm dựa trên hiểu biết chuyên ngành.
- Chọn tự động: Trong phương pháp này, bạn sử dụng các thuật toán để tự động tìm ra tổ hợp hyperparameter tối ưu nhất. Ưu điểm là không cần hiểu sâu về từng siêu tham số, nhưng nhược điểm là rất tốn chi phí tính toán, vì phải huấn luyện mô hình nhiều lần với các cấu hình khác nhau. Thường thì không gian các hyperparameter là rất lớn, vì vậy người ta thường áp dụng các chiến lược như:
  - Tìm kiếm ngẫu nhiên (random search),
  - Tìm kiếm theo lưới (grid search),

– hoặc các kỹ thuật nâng cao hơn như tối ưu hóa Bayesian (Bayesian optimization).

11.4.1 Điều chỉnh siêu tham số thủ công

Khi tinh chỉnh các siêu tham số (hyperparameter) bằng tay, người dùng cần hiểu rõ mối quan hệ giữa các tham số này với lỗi huấn luyện, lỗi tổng quát và tài nguyên tính toán như thời gian chạy và bộ nhớ. Mục tiêu là tìm được tập hợp giá trị siêu tham số phù hợp để giảm lỗi tổng quát mà vẫn nằm trong giới hạn tài nguyên cho phép.

Hiệu suất của một mô hình học sâu phụ thuộc chủ yếu vào ba yếu tố:

- Khả năng biểu diễn (representational capacity): Mô hình càng sâu (nhiều tầng) và rộng (nhiều nơ-ron) thì càng có thể học được các hàm phức tạp.
- Khả năng tối ưu hóa (optimization ability): Thuật toán huấn luyện phải đủ mạnh để tìm được trọng số tốt giúp giảm hàm mất mát.
- Sự điều chuẩn (regularization): Các kỹ thuật như weight decay giúp ngăn mô hình học quá mức vào dữ liệu huấn luyện, từ đó giữ được khả năng khái quát hóa.

Trong nhiều trường hợp, lỗi tổng quát của mô hình thay đổi theo hình dạng đường cong chữ U khi ta thay đổi một siêu tham số:

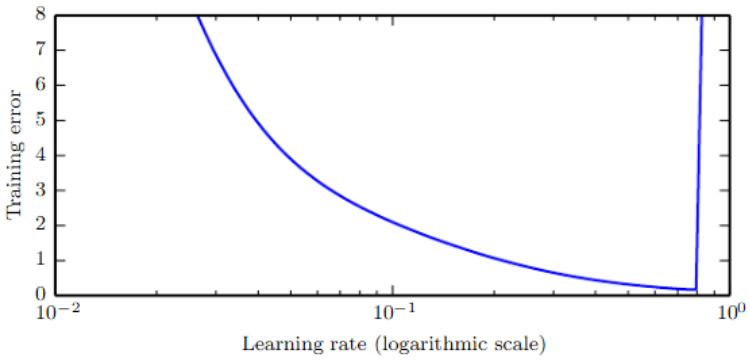
- Nếu giá trị siêu tham số quá thấp $\Rightarrow$ mô hình quá đơn giản $\Rightarrow$ underfitting.
- Nếu giá trị quá cao $\Rightarrow$ mô hình quá phức tạp $\Rightarrow$ overfitting.
- Có một khoảng giá trị vừa phải giúp tối ưu hóa lỗi tổng quát.

Một số siêu tham số như số tầng ẩn hay số lượng nơ-ron là các giá trị rời rạc nên khó xác định chính xác điểm tối ưu. Một số khác, chẳng hạn như việc bật hoặc tắt một kỹ thuật nào đó, chỉ có hai lựa chọn nên càng khó đánh giá theo đường cong U.

Trong tất cả các siêu tham số, tốc độ học (learning rate) thường là quan trọng nhất. Nếu chỉ có thể tinh chỉnh một tham số, hãy bắt đầu từ đây:

- Tốc độ học quá lớn $\Rightarrow$ mô hình dễ bị mất ổn định, lỗi huấn luyện có thể tăng.
- Tốc độ học quá nhỏ $\Rightarrow$ mô hình học rất chậm, dễ bị mắc kẹt ở điểm không tối ưu.

Thông thường, lỗi huấn luyện theo tốc độ học cũng có dạng đường cong chữ U, được minh họa trong Hình 11.1:



Hình 11.1: Mối quan hệ điển hình giữa tốc độ học và lỗi huấn luyện. Khi tốc độ học vượt quá mức tối ưu, lỗi huấn luyện tăng vọt. Nếu tốc độ học quá nhỏ, mô hình có thể không học đủ trong thời gian cho phép. Tốc độ học cũng ảnh hưởng đến khả năng regularization của mô hình.

Khi đánh giá mô hình, nếu lỗi trên tập kiểm tra (test error) cao, bạn cần xem xét hai thành phần:

- Lỗi huấn luyện (training error): mô hình học chưa tốt trên dữ liệu huấn luyện.
- Khoảng cách tổng quát hóa (generalization gap): sự chênh lệch giữa lỗi huấn luyện và lỗi kiểm tra.

Mục tiêu là giảm cả hai loại lỗi này. Trong thực tế, các mô hình học sâu thường cho kết quả tốt nhất khi lỗi huấn luyện gần như bằng 0, và phần lớn lỗi còn lại đến từ khoảng cách khái quát hóa. Chiến lược then chốt là: giảm khoảng cách khái quát hóa mà không làm tăng lỗi huấn luyện. Một cách hiệu quả là điều chỉnh các siêu tham số liên quan đến regularization như:

- Thêm dropout.
- Tăng weight decay.

Hầu hết các siêu tham số đều có thể xem như yếu tố làm tăng hoặc giảm độ phức tạp của mô hình. Ví dụ: tăng số lớp hoặc giảm regularization sẽ làm mô hình mạnh hơn nhưng cũng dễ bị overfit hơn. Bảng sau tóm tắt ảnh hưởng của một số siêu tham số phổ biến đến độ phức tạp của mô hình:

Một chiến lược đơn giản mà thực tế là tăng dần độ phức tạp mô hình và kích thước tập huấn luyện cho đến khi mô hình giải quyết được bài toán. Chiến lược này đòi hỏi tài nguyên tính toán lớn nên chỉ phù hợp nếu bạn có đủ khả năng. Dù trên lý thuyết, tăng độ phức tạp có thể làm quá trình tối ưu khó hơn, nhưng thực tế cho thấy, nếu bạn thiết kế mạng đúng cách, điều này hiếm khi là trở ngại lớn.

11.4.2 Thuật toán tối ưu siêu tham số tự động

Trong một thế giới lý tưởng, thuật toán học máy chỉ cần đầu vào là một tập dữ liệu và sẽ tự động trả về một hàm dự đoán tối ưu, mà không cần người dùng can thiệp vào bất kỳ siêu tham số (hyperparameter) nào. Sở dĩ các thuật toán như hồi quy logistic hay máy vector hỗ trợ (SVM) được sử dụng rộng rãi một phần là vì chúng hoạt động khá hiệu quả

Siêu tham số	Tăng độ phức tạp khi...	Lý do	Lưu ý
Số tầng ẩn	tăng	Mô hình sâu hơn có thể biểu diễn hàm phức tạp hơn	Tăng thời gian và bộ nhớ tính toán
Tốc độ học	tối ưu	Giá trị không phù hợp sẽ cản trở quá trình tối ưu	Nên là siêu tham số đầu tiên để tinh chỉnh
Độ rộng kernel tích chập	tăng	Thêm tham số, mô hình học được nhiều đặc trưng hơn	Tăng chi phí tính toán, giảm kích thước đặc trưng
Zero padding	tăng	Giữ nguyên kích thước qua nhiều lớp tích chập	Tăng tiêu tốn tài nguyên
Weight decay	giảm	Giới hạn ít hơn lên trọng số $\rightarrow$ mô hình linh hoạt hơn	Dễ overfit nếu giảm quá nhiều
Tỷ lệ dropout	giảm	Ít dropout hơn $\rightarrow$ mạng học mượt hơn nhưng dễ overfit	Quá ít thì không regularize tốt, quá nhiều thì khó hội tụ

Bảng 11.1: Tác động của các siêu tham số đến độ phức tạp mô hình

với chỉ một hoặc hai siêu tham số cần điều chỉnh.

Mạng nơ-ron đôi khi cũng cho kết quả tốt chỉ với vài siêu tham số cơ bản. Tuy nhiên, trong nhiều trường hợp thực tế, việc điều chỉnh đến hàng chục, thậm chí hơn 40 siêu tham số có thể giúp cải thiện hiệu suất đáng kể.

Tinh chỉnh thủ công có thể rất hiệu quả nếu bạn có điểm khởi đầu tốt—ví dụ: dựa vào kết quả từ các nghiên cứu trước đó trên bài toán tương tự, hoặc rút ra từ kinh nghiệm thực tiễn. Nhưng trong nhiều tình huống, đặc biệt là với các ứng dụng mới hoặc dữ liệu đặc thù, những điểm khởi đầu này lại không hề sẵn có. Khi đó, các thuật toán tự động tìm kiếm siêu tham số sẽ trở nên cực kỳ hữu ích.

Thực tế, quá trình con người tinh chỉnh siêu tham số cũng là một hình thức tối ưu hóa: tìm một tổ hợp giá trị sao cho một hàm mục tiêu—chẳng hạn như sai số trên tập validation—được giảm thiểu, có thể kèm thêm các ràng buộc như giới hạn thời gian huấn luyện, bộ nhớ sử dụng, hoặc tốc độ suy luận.

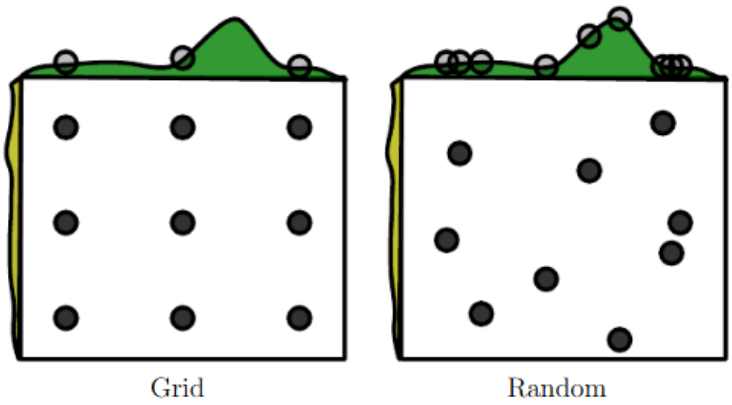
Từ nguyên lý này, người ta có thể xây dựng các *thuật toán tối ưu hoá siêu tham số tự động*, có thể xem như lớp bao bọc bên ngoài (wrapper) cho mô hình học máy chính. Các thuật toán này sẽ tự động lựa chọn các siêu tham số phù hợp mà không cần người dùng phải hiểu sâu về chi tiết kỹ thuật.

Tuy nhiên, một điểm nghịch lý là: các thuật toán tối ưu siêu tham số này lại thường có các *siêu tham số cấp hai* của chính chúng—ví dụ như khoảng giá trị cần khám phá cho từng siêu tham số của mô hình chính. Dù vậy, những siêu tham số cấp hai này thường

không quá khó để thiết lập, bởi vì có thể áp dụng các giá trị mặc định hoặc dải tìm kiếm phổ biến cho nhiều bài toán khác nhau mà không cần tinh chỉnh quá kỹ.

11.4.3 Tìm kiếm theo lưới (Grid Search)

Khi chỉ có một vài siêu tham số cần tinh chỉnh—thường là ba hoặc ít hơn—thì một trong những phương pháp đơn giản và phổ biến nhất là *tìm kiếm theo lưới* (grid search). Với phương pháp này, người dùng định nghĩa trước một tập hữu hạn các giá trị có thể thử cho mỗi siêu tham số. Sau đó, thuật toán sẽ huấn luyện mô hình với tất cả các tổ hợp giá trị có thể từ tích Descartes của các tập giá trị đã chọn.



Hình 11.2: So sánh giữa tìm kiếm theo lưới (grid search) và tìm kiếm ngẫu nhiên (random search). Ví dụ này minh họa với hai siêu tham số, nhưng trong thực tế có thể có nhiều hơn. (Trái) Với grid search, ta thử toàn bộ tổ hợp các giá trị đã định trước. (Phải) Với random search, ta định nghĩa một phân phối cho mỗi siêu tham số, rồi lấy mẫu ngẫu nhiên từ không gian tổ hợp. Khi chỉ một vài siêu tham số ảnh hưởng nhiều đến hiệu năng (trong ví dụ này là trục hoành), random search có xu hướng kiểm tra được nhiều giá trị khác nhau hơn của siêu tham số quan trọng, còn grid search lại tốn công vào những siêu tham số ít ảnh hưởng. Hình được trích từ Bergstra và Bengio (2012).

Sau khi đã huấn luyện xong tất cả mô hình, tổ hợp siêu tham số nào cho lỗi nhỏ nhất trên tập xác thực (validation set) sẽ được chọn là tốt nhất.

Chọn giá trị nào để đưa vào lưới? Với các siêu tham số có giá trị số học, người dùng thường chọn một tập giá trị rải đều trên thang *logarit* để tăng khả năng bao phủ vùng có giá trị tối ưu. Ví dụ:

- Tốc độ học (learning rate): {0.1, 0.01, 10^{-3}, 10^{-4}, 10^{-5}}
- Số lượng nơ-ron ẩn: {50, 100, 200, 500, 1000, 2000}

Tìm kiếm theo lưới thường hiệu quả hơn nếu được lặp lại theo từng vòng. Ví dụ, nếu bạn thử $\alpha \in \{-1, 0, 1\}$ và thấy 1 là tốt nhất, bạn có thể mở rộng phạm vi: thử tiếp với $\alpha \in \{1, 2, 3\}$. Nếu giá trị tối ưu là 0, bạn có thể thu hẹp lại: thử với $\{-0.1, 0, 0.1\}$.

Nhược điểm chính của Grid Search là chi phí tính toán tăng theo cấp số mũ khi số lượng siêu tham số tăng lên. Nếu có m siêu tham số và mỗi tham số thử n giá trị, số mô hình cần huấn luyện là $O(n^m)$.

Mặc dù các thử nghiệm trong grid search có thể thực hiện song song (vì không phụ thuộc lẫn nhau), nhưng việc chi phí tăng quá nhanh theo số siêu tham số khiến phương pháp này không phù hợp với các không gian tìm kiếm lớn. Khi số siêu tham số vượt quá một ngưỡng nhất định, grid search trở nên không hiệu quả và cần thay thế bằng các chiến lược khác như random search hoặc Bayesian optimization.

11.4.4 Tìm kiếm ngẫu nhiên (Random Search)

May mắn là có một phương pháp thay thế cho grid search vừa đơn giản trong lập trình, dễ triển khai, lại thường tìm ra các giá trị siêu tham số tốt nhanh hơn — đó là random search (tìm kiếm ngẫu nhiên), được đề xuất bởi Bergstra và Bengio (2012).

Nguyên lý hoạt động: Trước tiên, ta xác định một phân phối biên (marginal distribution) cho từng siêu tham số. Ví dụ:

- Với các siêu tham số rời rạc hoặc nhị phân, có thể dùng phân phối Bernoulli hoặc multinoulli.
- Với các siêu tham số số thực dương, nên dùng phân phối đều theo thang logarit. Ví dụ:

$$\text{log\_learning\_rate} \sim \mathcal{U}(-5, -1) \tag{11.1}$$

$$\text{learning\_rate} = 10^{\text{log\_learning\_rate}} \tag{11.2}$$

Tương tự, số lượng nơ-ron ẩn có thể chọn từ:

$$\text{log\_units} \sim \mathcal{U}(\log(50), \log(2000))$$

Ưu điểm chính:

- Không cần rời rạc hóa các giá trị như grid search, do đó có thể khám phá được không gian siêu tham số rộng hơn mà vẫn tiết kiệm chi phí tính toán.
- Khi số lượng siêu tham số lớn nhưng chỉ một số ít trong đó ảnh hưởng nhiều đến hiệu suất mô hình, random search thường vượt trội hơn grid search. Nghiên cứu của Bergstra và Bengio (2012) cho thấy random search giúp giảm lỗi nhanh hơn nếu xét trên cùng số lần thử.

So sánh với Grid Search: Grid search dễ bị lãng phí tài nguyên khi thử nhiều giá trị gần nhau của các siêu tham số ít quan trọng, còn random search thường tạo ra các tổ hợp đa dạng hơn — từ đó cung cấp thêm thông tin giá trị trong mỗi lần thử.

Lưu ý: Tương tự như grid search, random search cũng có thể được lặp lại nhiều lần để tinh

chỉnh lại không gian tìm kiếm, dựa trên các kết quả thu được từ các lần thử ban đầu.

11.4.5 Tối ưu siêu tham số dựa trên mô hình

Quá trình tìm kiếm siêu tham số tối ưu thực chất là một bài toán tối ưu hóa, trong đó:

- Biến đầu vào (decision variables) là các siêu tham số.
- Hàm mục tiêu là lỗi trên tập xác thực sau khi mô hình được huấn luyện với tập siêu tham số đó.

Trong những trường hợp đơn giản, nếu ta có thể tính đạo hàm của lỗi theo siêu tham số, thì có thể áp dụng các phương pháp tối ưu gradient-based **Bengio1999**, **Bengio2000**, **Maclaurin2015**. Tuy nhiên, trong thực tế, điều này hiếm khi khả thi do:

- Chi phí tính toán hoặc bộ nhớ quá lớn.
- Nhiều siêu tham số là rời rạc, làm cho hàm lỗi không khả vi.

11.4.6 Tối ưu hóa dựa trên mô hình (Model-based Optimization)

Thay vì cần gradient thực, ta có thể xây dựng một mô hình dự đoán lỗi trên tập xác thực dựa trên các cấu hình siêu tham số đã thử. Mô hình này sau đó được dùng để đề xuất các tổ hợp siêu tham số mới, bằng cách tối ưu hóa bên trong không gian dự đoán.

Phần lớn các phương pháp thuộc hướng tiếp cận này sử dụng một mô hình hồi quy Bayes để ước lượng:

- Kỳ vọng lỗi xác thực với mỗi tổ hợp siêu tham số.
- Mức độ không chắc chắn (uncertainty) của các ước lượng này.

Quá trình tìm kiếm khi đó trở thành một bài toán cân bằng giữa hai yếu tố:

- Exploration (thăm dò): thử những vùng mà mô hình còn chưa chắc chắn — có thể chứa siêu tham số rất tốt chưa từng biết.
- Exploitation (khai thác): ưu tiên thử những vùng mà mô hình tin rằng sẽ cho kết quả tốt — thường dựa trên dữ liệu lịch sử.

11.4.7 Một số phương pháp hiện đại:

- Spearmint **Snoek2012**
- TPE (Tree-structured Parzen Estimator) **Bergstra2011**
- SMAC (Sequential Model-based Algorithm Configuration) **Hutter2011**

11.4.8 Hạn chế hiện tại:

Dù các phương pháp tối ưu siêu tham số dựa trên mô hình — đặc biệt là tối ưu hóa Bayes (Bayesian Optimization) — đang ngày càng phổ biến và thu hút nhiều nghiên cứu, chúng vẫn còn nhiều hạn chế khiến việc áp dụng trong thực tế chưa hoàn toàn đáng tin cậy.

Hiện tại, chúng tôi chưa thể khẳng định rằng các phương pháp này là công cụ ổn định

và hiệu quả cao cho việc tối ưu siêu tham số trong deep learning. Trên một số bài toán cụ thể, các thuật toán như TPE hay SMAC có thể tìm ra các cấu hình siêu tham số tốt hơn hoặc ngang bằng với các chuyên gia con người nhiều kinh nghiệm. Tuy nhiên, cũng có không ít trường hợp chúng hoàn toàn thất bại — dẫn đến việc chọn ra những cấu hình không hiệu quả, thậm chí làm giảm mạnh hiệu năng mô hình.

Lý do là vì những thuật toán này hoạt động dựa trên mô hình hồi quy được huấn luyện từ kết quả các thử nghiệm trước. Nếu dữ liệu huấn luyện ban đầu không đại diện hoặc có sai lệch, mô hình này sẽ dẫn đến những đề xuất sai lệch. Ngoài ra, các yếu tố như đặc tính không ổn định của quá trình huấn luyện deep learning (chẳng hạn do ngẫu nhiên trong khởi tạo hoặc batch), hay các siêu tham số rời rạc không liên tục, cũng khiến mô hình dự đoán kém chính xác.

Vì vậy, cho đến hiện tại, tối ưu hóa Bayes và các phương pháp tương tự nên được xem như công cụ hỗ trợ cần được thử nghiệm thêm, chứ chưa phải là giải pháp tối ưu đảm bảo kết quả tốt trong mọi tình huống.

Tuy nhiên, lĩnh vực tối ưu siêu tham số tự động nói chung — đặc biệt là các kỹ thuật dựa trên mô hình — vẫn là một hướng nghiên cứu đầy tiềm năng. Không chỉ dành riêng cho deep learning, những phương pháp này có thể mang lại ảnh hưởng lớn đến toàn bộ ngành học máy (machine learning), và rộng hơn là tất cả các lĩnh vực trong khoa học kỹ thuật nơi các mô hình phức tạp cần điều chỉnh thông số để đạt hiệu suất tối ưu.

11.4.9 Một điểm yếu chung:

Một hạn chế lớn của phần lớn các phương pháp tối ưu siêu tham số hiện nay — đặc biệt là các kỹ thuật phức tạp hơn random search — là: chúng yêu cầu mỗi thử nghiệm phải được huấn luyện đến hết vòng đời để có thể đánh giá và rút ra thông tin phục vụ cho bước tiếp theo.

Cách tiếp cận này dẫn đến việc tiêu tốn rất nhiều tài nguyên. Trong khi đó, con người khi tinh chỉnh thủ công thường có thể nhanh chóng phát hiện ra nếu một tổ hợp siêu tham số là "thảm họa", ví dụ như khi mô hình không hội tụ, lỗi tăng vọt, hoặc quá trình huấn luyện quá chậm — và do đó có thể dừng sớm để tiết kiệm thời gian và tài nguyên.

Nhằm giải quyết hạn chế này, Swersky et al. (2014) đã đề xuất một phương pháp ban đầu cho phép duy trì nhiều thí nghiệm song song với khả năng quản lý linh hoạt hơn. Ý tưởng chính là:

- Tại mỗi thời điểm, hệ thống có thể khởi động thêm một thử nghiệm mới với một cấu hình siêu tham số mới.
- Nếu thấy một thử nghiệm đang chạy có biểu hiện không khả quan (dựa trên các chỉ số trung gian như lỗi hoặc tốc độ hội tụ), ta có thể "đóng băng" nó — tạm dừng và giải phóng tài nguyên.
- Nếu về sau, thử nghiệm đó được đánh giá là có tiềm năng hơn so với các thử nghiệm

khác, ta có thể "rã đông" nó — tiếp tục huấn luyện từ nơi đã dừng lại.

Cách làm này giúp cải thiện hiệu suất sử dụng tài nguyên, tránh việc huấn luyện lãng phí các mô hình yếu ngay từ đầu. Nó cũng đưa mô hình tối ưu siêu tham số tiến gần hơn đến cách mà con người ra quyết định — linh hoạt, phản xạ nhanh và biết loại bỏ sớm những lựa chọn tồi.

Trong tương lai, việc kết hợp các kỹ thuật như này với các thuật toán tối ưu hóa mạnh mẽ hơn có thể là chìa khóa để tối ưu siêu tham số hiệu quả, đặc biệt trong các môi trường có giới hạn về tài nguyên tính toán.

11.5 Chiến lược gỡ lỗi trong học sâu

Khi một hệ thống học máy hoạt động không tốt, việc xác định nguyên nhân không phải lúc nào cũng dễ dàng. Ta thường không rõ liệu vấn đề nằm ở chính thuật toán không phù hợp, dữ liệu chưa tốt, hay là do lỗi lập trình (bug) trong quá trình triển khai. Điều này khiến việc gỡ lỗi trong học sâu trở nên khó khăn hơn nhiều so với các hệ thống lập trình truyền thống.

Có một số nguyên nhân chính khiến gỡ lỗi trong học máy nói chung và học sâu nói riêng đặc biệt thách thức:

- Không có hành vi kỳ vọng rõ ràng: Trong các bài toán học máy, chúng ta hiếm khi biết chính xác mô hình "nên" hoạt động ra sao. Nếu đã có quy tắc cụ thể, có thể ta đã giải quyết bài toán bằng phương pháp truyền thống thay vì học máy. Do đó, việc xác định xem đầu ra của mô hình có "đúng" hay không là điều không rõ ràng, trừ khi hiệu suất cực kỳ tệ.
- Các thành phần có khả năng tự thích nghi: Mô hình học sâu thường bao gồm nhiều phần có khả năng điều chỉnh lẫn nhau. Nếu một phần bị lỗi, các phần còn lại có thể tự "bù đắp" để duy trì hiệu suất ở mức chấp nhận được. Điều này làm cho lỗi bị che khuất, khiến ta khó phát hiện sai sót.

Ví dụ minh họa: Giả sử bạn xây dựng một mạng nơ-ron nhiều lớp và tự cài đặt quá trình cập nhật gradient. Trong một lần vô tình, bạn viết dòng cập nhật độ lệch (bias) như sau:

$$b \leftarrow b - \alpha$$

trong khi lẽ ra phải là:

$$b \leftarrow b - \alpha \cdot \nabla_b J$$

Tức là bạn quên nhân với gradient $\nabla_b J$. Điều này tưởng như sẽ gây lỗi nghiêm trọng, nhưng trên thực tế mô hình vẫn có thể học được đến một mức độ nào đó. Lý do là vì các trọng số W có thể tự điều chỉnh để "bù trừ" cho phần sai lệch từ b, giúp mô hình duy trì

được kết quả đầu ra không quá tệ. Tuy nhiên, điều đó cũng đồng nghĩa với việc lỗi của bạn rất khó bị phát hiện thông qua kiểm tra hiệu suất tổng thể.

11.5.1 Chiến lược gỡ lỗi phổ biến

Để gỡ lỗi hiệu quả trong các mô hình học sâu, ta nên áp dụng các chiến lược có hệ thống, thay vì chỉ quan sát đầu ra và đoán mò. Dưới đây là hai chiến lược cốt lõi:

- Đơn giản hóa bài toán (Simplify the task): Cố gắng tạo ra một phiên bản đơn giản hóa của bài toán mà bạn có thể đoán trước được kết quả. Ví dụ: dùng một tập dữ liệu nhỏ (ví dụ vài mẫu) mà bạn biết rõ nhãn đúng, hoặc gán nhãn trùng khớp hoàn toàn với đầu vào. Nếu mô hình không thể khớp được dữ liệu đơn giản này, gần như chắc chắn có lỗi trong quá trình huấn luyện hoặc mô hình.
- Kiểm thử từng phần riêng lẻ (Unit testing for components): Hãy kiểm tra từng phần của pipeline một cách độc lập. Ví dụ:
  - Kiểm tra từng hàm tính toán — đảm bảo rằng đầu ra có giá trị mong đợi với đầu vào cụ thể.
  - Kiểm tra giá trị gradient — so sánh gradient tính theo lý thuyết với gradient tính xấp xỉ bằng vi phân hữu hạn.
  - Kiểm tra tính chất cơ bản — như kiểm tra xem loss có giảm qua từng epoch không, trọng số có thay đổi không, đầu ra của mô hình có thay đổi không khi cập nhật tham số.

Những chiến lược này giúp bạn xác định được lỗi nằm ở đâu: trong thiết kế mô hình, cài đặt huấn luyện, xử lý dữ liệu, hay thuật toán tối ưu. Gỡ lỗi hiệu quả là kỹ năng quan trọng không kém gì việc thiết kế mô hình tốt — bởi vì một mô hình đúng về lý thuyết nhưng có lỗi nhỏ trong code vẫn sẽ cho kết quả rất tệ.

Dưới đây là một số chiến lược quan trọng:

11.5.1.1 1. Trực quan hóa mô hình khi hoạt động

Khi huấn luyện một hệ thống học sâu, đặc biệt là các mô hình xử lý dữ liệu có thể cảm nhận được bằng thị giác hoặc thính giác, việc trực quan hóa đầu ra là một bước rất quan trọng nhưng thường bị bỏ qua. Ví dụ:

- Với bài toán nhận diện đối tượng trong ảnh, hãy trực quan hóa ảnh đầu vào với các khung phát hiện (bounding boxes) chồng lên.
- Với bài toán sinh giọng nói, hãy nghe các đoạn âm thanh mà mô hình tạo ra.

Việc này giúp phát hiện những bất thường hoặc lỗi rõ ràng mà các chỉ số như độ chính xác, log-likelihood không thể hiện ra. Nhiều khi, mô hình có thể đạt độ chính xác cao nhưng vẫn đưa ra những dự đoán vô nghĩa mà chỉ có con người mới nhận ra.

Lưu ý đặc biệt: *Lỗi trong khâu đánh giá (evaluation bugs)* là một trong những lỗi nguy hiểm nhất — vì nó khiến bạn tưởng rằng mô hình hoạt động tốt, trong khi thực tế thì

không. Việc trực quan hóa kết quả thực tế giúp kiểm chứng lại những gì bạn đo được bằng các chỉ số định lượng.

11.5.1.2 Trực quan hóa các lỗi tệ nhất

Hầu hết các mô hình phân loại đều cung cấp một số dạng "độ tin cậy"(confidence) cho mỗi dự đoán, chẳng hạn như xác suất từ lớp softmax. Dù các xác suất này có thể bị lệch — thường là quá tự tin — nhưng vẫn có thể tận dụng để tìm ra các lỗi đáng chú ý.

Hãy sắp xếp các mẫu mà mô hình dự đoán sai, nhưng lại đưa ra xác suất rất cao cho lựa chọn sai đó. Những lỗi này thường chỉ ra các vấn đề nghiêm trọng trong dữ liệu hoặc mô hình, ví dụ:

- Hình ảnh bị crop quá chặt, dẫn đến mất thông tin quan trọng.
- Nhiễu ảnh hoặc điều kiện ánh sáng không phù hợp.
- Gán nhãn sai trong tập huấn luyện.

Ví dụ thực tế: Trong hệ thống nhận diện số nhà từ ảnh Street View, mô hình ban đầu thường cắt ảnh quá sát, dẫn đến mất số. Mạng nơ-ron sau đó gán xác suất rất thấp cho nhãn đúng — và khi quan sát các ảnh bị dự đoán sai với độ tin cậy cao, lỗi crop này được phát hiện. Giải pháp là mở rộng khung ảnh, từ đó hiệu suất tăng rõ rệt.

11.5.1.3 Phân tích lỗi qua sai số huấn luyện và kiểm thử

Một cách tiếp cận đơn giản nhưng hiệu quả để chẩn đoán mô hình là so sánh sai số huấn luyện và kiểm thử:

- Huấn luyện tốt nhưng kiểm thử kém: Mô hình có thể đang overfit — tức là học quá sát vào dữ liệu huấn luyện. Cũng có thể do lỗi kỹ thuật như:
  - Dữ liệu kiểm thử được tiền xử lý khác với dữ liệu huấn luyện.
  - Mô hình không được lưu và nạp đúng cách.
- Cả huấn luyện và kiểm thử đều kém: Có thể do mô hình quá đơn giản (underfitting) hoặc có lỗi nghiêm trọng trong phần mềm, chẳng hạn như lỗi trong tính toán forward hoặc backward. Khi gặp tình huống này, cần thực hiện các kiểm thử sâu hơn để xác định nguyên nhân cụ thể.

11.5.2 Kiểm tra bằng tập dữ liệu nhỏ

Một trong những kiểm thử cơ bản và hiệu quả nhất là huấn luyện mô hình trên một tập dữ liệu rất nhỏ — ví dụ chỉ gồm một vài mẫu. Mục tiêu là kiểm tra xem mô hình có thể "ghi nhớ"dữ liệu này không:

- Nếu chỉ có một mẫu, mô hình phân loại hoàn toàn có thể đạt kết quả hoàn hảo — ví dụ bằng cách đặt bias đủ lớn ở lớp cuối.
- Nếu mô hình không học được đúng một mẫu duy nhất, rất có khả năng đang có lỗi phần mềm trong quá trình huấn luyện.

- Với autoencoder, đầu ra phải gần như trùng khớp với đầu vào nếu chỉ huấn luyện trên một ảnh.

Kiểm tra này rất hữu ích để phân biệt giữa lỗi do underfitting thực sự và lỗi do cài đặt sai.

11.5.3 So sánh đạo hàm lan truyền ngược với đạo hàm số

Nếu bạn tự xây dựng một framework học máy hoặc định nghĩa một hàm mất mát hoặc tầng mới, một lỗi rất phổ biến là sai trong việc tính đạo hàm. Để kiểm tra, bạn có thể so sánh đạo hàm từ backpropagation với đạo hàm xấp xỉ bằng sai phân hữu hạn (finite difference):

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

hoặc chính xác hơn, sử dụng sai phân trung tâm:

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}$$

Chọn ϵ cần đủ lớn để tránh sai số do tròn số trong tính toán số học (finite precision), nhưng đủ nhỏ để vẫn còn gần đúng đạo hàm thật.

11.5.4 Kiểm tra Gradient hoặc Ma trận Jacobian của một Hàm Vector

Khi làm việc với các mô hình có đầu ra là vector — ví dụ $g : \mathbb{R}^m \to \mathbb{R}^n$ — việc kiểm tra gradient đầy đủ bằng sai phân hữu hạn là rất tốn kém: cần thực hiện tổng cộng mn lần tính hàm.

Thay vào đó, ta có thể định nghĩa một hàm scalar mới:

$$f(x) = \mathbf{u}^T g(\mathbf{v}x)$$

với $\mathbf{u} \in \mathbb{R}^n$, $\mathbf{v} \in \mathbb{R}^m$ là các vector ngẫu nhiên. Hàm $f(x)$ lúc này có đầu vào và đầu ra là số thực, cho phép áp dụng các kiểm tra gradient đơn giản như trên. Phương pháp này rất hiệu quả và đủ mạnh để phát hiện lỗi trong quá trình lan truyền ngược qua $g(x)$.

Nên lặp lại quá trình này với nhiều lựa chọn khác nhau của $\mathbf{u}$ và $\mathbf{v}$ để đảm bảo không bỏ sót lỗi tiềm ẩn nào do lựa chọn chiếu ngẫu nhiên.

11.6 Phương pháp gradient số bằng số phức

Nếu môi trường lập trình hoặc thư viện học máy hỗ trợ tính toán với số phức, ta có thể tận dụng một kỹ thuật rất chính xác và ổn định để xấp xỉ đạo hàm — gọi là phương pháp gradient số bằng số phức.

Theo nghiên cứu của Squire và Trapp (1998), nếu đánh giá hàm f tại điểm $x + i\epsilon$, với $i = \sqrt{-1}$ và ϵ rất nhỏ, ta có thể khai triển như sau:

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + \mathcal{O}(\epsilon^2)$$

Từ đó, ta tách được:

$$\text{Re}(f(x + i\epsilon)) = f(x) + \mathcal{O}(\epsilon^2)$$
$$\frac{\text{Im}(f(x + i\epsilon))}{\epsilon} = f'(x) + \mathcal{O}(\epsilon^2)$$

Điểm mạnh nổi bật của phương pháp này là khả năng tránh hoàn toàn hiện tượng triệt tiêu số học — vốn là một vấn đề phổ biến trong phương pháp sai phân hữu hạn khi ϵ quá nhỏ. Nhờ đó, có thể lựa chọn giá trị ϵ cực kỳ nhỏ (ví dụ 10^{-150}) mà vẫn duy trì được độ chính xác cao, trong khi phương pháp sai phân thông thường sẽ gặp sai số nghiêm trọng.

Phương pháp này đặc biệt hữu ích để kiểm tra tính đúng đắn của việc lan truyền gradient (backpropagation) trong mô hình hoặc thư viện tự xây dựng. Tuy nhiên, nó chỉ áp dụng được cho các hàm nhận đầu vào là số thực và trả về giá trị thực — không mở rộng được trực tiếp cho các phép toán rời rạc hoặc xử lý số phức.

11.7 Giám sát thống kê của hoạt động và gradient

Một công cụ quan trọng trong huấn luyện mạng nơ-ron là việc theo dõi phân bố thống kê của các kích hoạt (activations) và gradient tại từng tầng qua nhiều vòng huấn luyện. Việc này thường được thực hiện dưới dạng biểu đồ histogram hoặc biểu đồ phân phối thời gian.

Cụ thể:

- Tiền kích hoạt (pre-activation): Với ReLU, cần kiểm tra xem bao nhiêu neuron luôn cho ra giá trị 0 — nếu nhiều quá, mạng có thể đang bị "tê liệt". Với tanh hoặc sigmoid, nên theo dõi xem đầu vào của chúng có bị đẩy quá xa khỏi vùng tuyến tính hay không (hiện tượng bão hòa).
- Gradient: Cần quan sát phân phối độ lớn của gradient qua các tầng. Nếu gradient gần như biến mất (vanishing gradient) hoặc tăng rất nhanh (exploding gradient), mô hình sẽ rất khó huấn luyện. Đây là một dấu hiệu quan trọng cần can thiệp kiến trúc hoặc kỹ thuật tối ưu hóa.
- Tỉ lệ cập nhật tham số: Theo Bottou (2015), một nguyên tắc hữu ích là độ lớn của mỗi cập nhật tham số (ví dụ trên mỗi mini-batch) nên vào khoảng 1% giá trị hiện tại của tham số đó. Nếu nhỏ hơn nhiều (ví dụ 0.001%) hoặc lớn hơn nhiều (ví dụ 50%), cần điều chỉnh tốc độ học hoặc regularization.

Một điểm cần chú ý là với dữ liệu thưa (sparse), như trong xử lý văn bản, có thể nhiều tham số chỉ được cập nhật hiếm hoi. Khi đó, bạn cần đặc biệt theo dõi liệu những tham số đó có đang học được gì hay không.

11.8 Kiểm tra tính đúng đắn của giải thuật tối ưu

Một số thuật toán tối ưu hoặc suy luận xấp xỉ trong học sâu đi kèm các đảm bảo lý thuyết. Khi bạn cài đặt các thuật toán này, việc kiểm tra các điều kiện đảm bảo là cách hiệu quả để phát hiện lỗi.

Ví dụ, ta có thể kiểm tra:

- Hàm mục tiêu có giảm sau mỗi bước cập nhật không?
- Gradient theo từng nhóm biến (ví dụ một tầng) có hội tụ về 0?
- Tổng độ lớn của gradient trên toàn bộ mô hình có giảm dần theo thời gian?

Do tính chất số học hữu hạn, không phải lúc nào các điều kiện trên cũng chính xác tuyệt đối. Vì vậy, cần thiết lập ngưỡng dung sai (tolerance) hợp lý để phân biệt giữa nhiễu số học và lỗi thật.

11.9 Ví dụ: nhận dạng số đa chữ số

Để minh họa cách áp dụng toàn bộ phương pháp luận đã trình bày, chúng tôi giới thiệu hệ thống nhận dạng số địa chỉ từ ảnh Street View — một dự án thành công nổi bật của Google.

Hệ thống bắt đầu từ giai đoạn thu thập dữ liệu: các xe Street View ghi lại ảnh các tòa nhà, sau đó nhân viên gán nhãn chính xác số địa chỉ. Bài toán chính là nhận dạng chuỗi số từ ảnh, với độ chính xác tương đương người — tức là khoảng 98%.

11.9.1 Mục tiêu ban đầu và chỉ số hiệu suất

Dự án chọn độ chính xác 98% làm mục tiêu chính, vì đây là mức hiệu năng con người có thể đạt được. Tuy nhiên, để đạt độ chính xác cao, hệ thống phải hy sinh độ bao phủ (coverage): chỉ trả về kết quả khi chắc chắn.

Do đó, chỉ số hiệu suất chính được tối ưu là coverage, với độ chính xác cố định ở mức 98%. Khi các kỹ thuật như CNN cải thiện, hệ thống dần có thể giảm ngưỡng tự tin mà vẫn duy trì độ chính xác cao, nhờ đó tăng coverage lên vượt 95%.

11.10 Xây dựng hệ thống cơ bản

Bước đầu tiên là xây dựng hệ thống đơn giản nhất có thể — một CNN dùng ReLU. Vì chưa có kỹ thuật dự đoán chuỗi tốt vào thời điểm đó, mô hình ban đầu sử dụng nhiều lớp softmax độc lập để dự đoán từng chữ số trong chuỗi.

Các lớp softmax được huấn luyện riêng biệt, như các bài toán phân loại độc lập. Hệ thống này đã đủ để chạy được đầu-cuối, tạo nền tảng cho các cải tiến tiếp theo.

Một trong những cải tiến đầu tiên là thay đổi cách tính $p(y|x)$ (xác suất của chuỗi ký tự) sao cho phản ánh tốt hơn khả năng mô hình tin tưởng vào dự đoán của nó. Thay vì lấy tích đơn giản các softmax, nhóm phát triển đã chuyển sang hàm log-likelihood chính quy, giúp việc từ chối đầu vào thiếu chắc chắn trở nên chính xác hơn.

11.11 Phát hiện vấn đề và cải thiện hệ thống

Dù mô hình ban đầu có hiệu quả nhất định, coverage vẫn dưới 90%. Phân tích lỗi cho thấy mô hình không overfit (lỗi huấn luyện và kiểm thử tương đương), mà đơn giản là chưa đủ khả năng biểu diễn — tức là underfitting.

Đặc biệt, khi sắp xếp các ví dụ mà mô hình sai nhưng lại rất tự tin, nhóm phát hiện lỗi crop ảnh: nhiều ảnh bị cắt quá chặt, dẫn đến mất một phần số. Ví dụ "1849"có thể bị cắt thành "849".

Thay vì cải tiến module phát hiện, nhóm quyết định mở rộng vùng crop ảnh thủ công. Điều chỉnh này đã tăng coverage lên hơn 10%.

11.12 Cải tiến cuối cùng và kết quả

Các cải tiến sau đó chủ yếu xoay quanh tăng kích thước mô hình và tinh chỉnh siêu tham số — trong giới hạn tài nguyên có thể chấp nhận được. Vì lỗi huấn luyện và kiểm thử vẫn gần như bằng nhau, các cải tiến này giúp giảm underfitting.

Kết quả cuối cùng là một hệ thống phiên âm số có hiệu suất cao, vượt ngưỡng 95% coverage với độ chính xác tương đương con người. Dự án thành công lớn, giúp tự động hóa quá trình phiên âm hàng trăm triệu địa chỉ, tiết kiệm rất nhiều chi phí so với làm thủ công.

Những nguyên lý thiết kế thực tế trong chương này đã đóng vai trò quan trọng trong thành công đó, và có thể áp dụng cho nhiều hệ thống học sâu tương tự.

CHƯƠNG 12. ỨNG DỤNG CỦA DEEP LEARNING

Trong chương này, chúng ta sẽ tìm hiểu cách áp dụng deep learning để giải quyết các bài toán trong thực tế — đặc biệt là trong những lĩnh vực như thị giác máy tính, nhận dạng giọng nói, xử lý ngôn ngữ tự nhiên và các ứng dụng khác mang lại giá trị thương mại.

Mở đầu, chúng ta sẽ thảo luận về việc triển khai các mạng nơ-ron sâu ở quy mô lớn, điều gần như là bắt buộc khi xây dựng các hệ thống AI thực tế.

Tiếp theo, chương trình bày một số lĩnh vực cụ thể nơi deep learning đã được ứng dụng hiệu quả. Dù một trong những mục tiêu dài hạn của deep learning là phát triển các thuật toán có thể xử lý được nhiều loại nhiệm vụ khác nhau, trên thực tế, hiện vẫn cần có sự điều chỉnh chuyên biệt cho từng bài toán.

Chẳng hạn, các bài toán thị giác máy tính thường phải xử lý khối lượng lớn đặc trưng đầu vào — ví dụ như hàng triệu điểm ảnh trong mỗi hình ảnh. Trong khi đó, các bài toán về ngôn ngữ lại đòi hỏi khả năng mô hình hóa một không gian rất lớn các giá trị đầu ra tiềm năng, chẳng hạn như toàn bộ từ vựng, cho mỗi đơn vị đầu vào.

Những đặc thù như vậy đặt ra các yêu cầu riêng đối với mô hình và kỹ thuật huấn luyện, mà chúng ta sẽ lần lượt khám phá trong các phần tiếp theo của chương.

12.1 Deep learning quy mô lớn

Deep learning được xây dựng dựa trên triết lý của chủ nghĩa kết nối (connectionism): trong khi một nơ-ron sinh học riêng lẻ hay một đặc trưng đơn lẻ trong mô hình máy học không biểu hiện trí thông minh, thì một tập hợp lớn các nơ-ron hoặc đặc trưng, khi phối hợp hoạt động với nhau, lại có thể tạo ra hành vi thông minh. Điểm mấu chốt ở đây là **số lượng nơ-ron phải đủ lớn**.

Một yếu tố quan trọng giúp cải thiện độ chính xác của mạng nơ-ron và khả năng giải các bài toán phức tạp kể từ thập niên 1980 đến nay chính là việc mở rộng quy mô mạng. Như đã đề cập ở mục 1.2.3, kích thước mạng nơ-ron đã tăng theo cấp số nhân trong suốt ba thập kỷ qua — dù vậy, các mạng hiện đại vẫn còn nhỏ hơn nhiều so với hệ thần kinh của một số loài côn trùng.

Chính vì vậy, deep learning yêu cầu phần cứng mạnh mẽ và hạ tầng phần mềm hiệu quả để có thể huấn luyện và chạy được các mạng lớn.

12.1.1 Các triển khai CPU nhanh

Trước đây, mạng nơ-ron thường được huấn luyện trên CPU của một máy tính đơn. Nhưng ngày nay, phương pháp này thường không đủ hiệu quả. Thay vào đó, chúng ta thường sử dụng GPU hoặc các hệ thống phân tán nhiều CPU kết nối với nhau. Trước khi các hệ thống đắt tiền này trở nên phổ biến, một số nhà nghiên cứu đã chứng minh rằng CPU không còn phù hợp để xử lý khối lượng tính toán lớn của mạng nơ-ron.

Việc tối ưu hoá tính toán trên CPU cụ thể vượt quá phạm vi của cuốn sách này, nhưng

có một nguyên lý quan trọng: nếu ta triển khai số học và thao tác bộ nhớ một cách chuyên biệt cho từng dòng CPU, hiệu suất có thể tăng đáng kể.

Ví dụ, Vanhoucke et al. (2011) cho thấy rằng trên các CPU hiện đại thời điểm đó, việc dùng số học kiểu số nguyên cố định (fixed-point) thay vì dấu chấm động (floating-point) có thể tăng tốc độ huấn luyện mạng nơ-ron gấp ba lần. Tuy nhiên, tùy thuộc vào loại CPU, số học dấu chấm động cũng có thể hoạt động tốt hơn.

Ngoài lựa chọn loại số học, các kỹ thuật khác để tăng tốc bao gồm: tối ưu hoá cấu trúc dữ liệu để tránh cache miss, và sử dụng các chỉ thị xử lý song song theo vectơ (vectorized instructions). Mặc dù những chi tiết này thường bị bỏ qua trong nghiên cứu học máy, nhưng chúng có thể ảnh hưởng lớn đến khả năng mở rộng mô hình — và theo đó là độ chính xác.

12.1.2 Các triển khai GPU

Phần lớn các mạng nơ-ron hiện đại được huấn luyện trên GPU — các phần cứng ban đầu được thiết kế cho đồ họa máy tính. Thị trường trò chơi điện tử là động lực thúc đẩy sự phát triển của GPU, và thật tình cờ, các yêu cầu hiệu năng trong trò chơi cũng rất phù hợp với deep learning.

Để kết xuất đồ họa, GPU cần thực hiện nhiều phép toán ma trận và pixel đơn giản trên các phần tử độc lập như đỉnh và điểm ảnh, với tốc độ rất cao. Các phép toán này đơn giản, ít nhánh điều kiện, có thể song song hóa dễ dàng, và đòi hỏi băng thông bộ nhớ rất lớn — tất cả đều trùng khớp với đặc điểm của mạng nơ-ron.

Trong mạng nơ-ron, ta cũng cần xử lý lượng lớn bộ nhớ chứa tham số, kích hoạt, và gradient. Các thao tác này diễn ra liên tục trong suốt quá trình huấn luyện. Do bộ nhớ lớn, chúng thường vượt quá dung lượng cache, khiến băng thông bộ nhớ trở thành yếu tố quyết định hiệu năng. GPU có lợi thế rõ rệt so với CPU nhờ băng thông bộ nhớ cao và khả năng song song hóa mạnh.

Ban đầu, GPU chỉ phục vụ cho đồ họa. Nhưng theo thời gian, chúng đã trở nên linh hoạt hơn, cho phép người dùng chạy mã tùy chỉnh. Ngôn ngữ CUDA của NVIDIA là ví dụ điển hình, cung cấp một nền tảng gần giống C để lập trình GPU. Kết hợp khả năng song song, băng thông cao và ngôn ngữ tiện lợi, GPU đã trở thành nền tảng lý tưởng cho deep learning (Raina et al., 2009; Ciresan et al., 2010).

Tuy nhiên, viết mã hiệu quả cho GPU là một công việc phức tạp. Ví dụ, thay vì tận dụng cache như trên CPU, GPU đôi khi cần tính lại một giá trị nhiều lần hơn là đọc lại từ bộ nhớ. Các thao tác bộ nhớ cũng cần được "hợp nhất" giữa các luồng để tăng hiệu năng, và mỗi nhóm luồng (warp) phải thực thi cùng một lệnh tại một thời điểm, nên việc rẽ nhánh phải được kiểm soát kỹ lưỡng.

Vì vậy, các nhà nghiên cứu nên xây dựng quy trình thử nghiệm của mình sao cho có thể tái sử dụng các thư viện tính toán hiệu năng cao có sẵn (như phép tích chập hoặc nhân ma trận). Ví dụ, thư viện Pylearn2 dựa vào Theano và cuda-convnet để chạy các phép toán

chính. Tương tự, TensorFlow và Torch cũng cung cấp các giao diện trừu tượng giúp mô hình hóa mà không cần viết mã GPU thủ công.

12.1.3 Triển khai phân tán quy mô lớn

Trong nhiều trường hợp, một máy tính đơn là không đủ để huấn luyện hoặc chạy mô hình. Do đó, ta cần đến các chiến lược tính toán phân tán.

Phân tán suy luận: Đây là phần dễ hơn — mỗi mẫu dữ liệu đầu vào có thể được xử lý độc lập trên các máy khác nhau, được gọi là *song song dữ liệu (data parallelism)*.

Phân tán mô hình: Trong một số trường hợp, một mô hình quá lớn nên không thể chứa vừa trên một máy. Khi đó, ta chia mô hình thành nhiều phần, mỗi phần xử lý trên một máy khác nhau. Cách này gọi là *song song mô hình (model parallelism)* và có thể áp dụng cho cả huấn luyện lẫn suy luận.

Phân tán dữ liệu trong huấn luyện: Việc huấn luyện phân tán phức tạp hơn. Tăng kích thước minibatch là cách đơn giản nhất, nhưng hiệu quả tối ưu hóa thường không tăng tương ứng. Cách tốt hơn là cho nhiều máy tính tính toán các bước SGD độc lập. Tuy nhiên, SGD chuẩn là thuật toán tuần tự — gradient tại bước t phụ thuộc vào tham số từ bước $t - 1$.

Gradient descent ngẫu nhiên bất đồng bộ: Cách khắc phục là dùng SGD *bất đồng bộ* (asynchronous), trong đó các lõi xử lý chia sẻ tham số trong bộ nhớ, và mỗi lõi có thể đọc/ghi mà không cần khóa (Bengio et al., 2001; Recht et al., 2011). Dù điều này có thể làm nhiễu lẫn nhau một chút, nhưng tổng thể tốc độ huấn luyện tăng lên rõ rệt.

Máy chủ tham số: Dean et al. (2012) mở rộng ý tưởng này bằng cách dùng một hoặc nhiều *máy chủ tham số* để lưu trữ tham số và cho phép các máy khác gửi cập nhật gradient. Đây hiện vẫn là chiến lược phổ biến nhất để huấn luyện các mạng deep learning lớn trong công nghiệp (Chilimbi et al., 2014; Wu et al., 2015).

Dù các nhóm nghiên cứu trong học thuật ít có điều kiện để xây dựng hệ thống lớn như vậy, một số công trình đã thử triển khai học phân tán trên phần cứng giá rẻ hơn, khả thi trong môi trường đại học (Coates et al., 2013).

12.1.4 Nén mô hình

Trong nhiều ứng dụng thực tế, chi phí thời gian và bộ nhớ để chạy suy luận trên một mô hình học máy thường quan trọng hơn nhiều so với chi phí huấn luyện. Khi không cần cá nhân hóa theo người dùng, ta chỉ cần huấn luyện mô hình một lần rồi triển khai cho hàng tỷ người dùng. Trong tình huống này, thiết bị của người dùng — chẳng hạn như điện thoại di động — thường hạn chế hơn rất nhiều so với cụm máy tính dùng để huấn luyện.

Nén mô hình là một chiến lược phổ biến giúp giảm chi phí suy luận (Buciluǎ et al., 2006). Ý tưởng chính là thay thế một mô hình lớn, tốn bộ nhớ và thời gian xử lý, bằng một mô hình nhỏ hơn nhưng vẫn cho kết quả tương đương.

Khi nào nên nén mô hình? Việc nén đặc biệt phù hợp khi mô hình gốc được thiết kế

lớn để tránh hiện tượng quá khớp (overfitting). Trong nhiều trường hợp, mô hình tốt nhất là tập hợp của nhiều mô hình nhỏ — nhưng đánh giá toàn bộ tổ hợp này là rất tốn kém. Ngay cả khi chỉ dùng một mô hình duy nhất, việc tăng kích thước cũng giúp tổng quát hóa tốt hơn (ví dụ như khi dùng dropout), nhưng lại sử dụng nhiều tham số hơn mức cần thiết.

Khi đã có mô hình lớn học được một hàm $f(x)$ mong muốn, ta có thể sinh ra vô số dữ liệu huấn luyện bằng cách lấy ngẫu nhiên các điểm x rồi áp dụng $f(x)$ để tạo ra nhãn. Sau đó, một mô hình nhỏ hơn sẽ được huấn luyện để bắt chước lại đầu ra của $f(x)$. Để kết quả tốt, nên lấy mẫu x từ phân phối tương tự như dữ liệu thực tế, bằng cách làm nhiễu dữ liệu huấn luyện hoặc dùng mô hình sinh.

Một phương pháp khác là huấn luyện mô hình nhỏ trên chính tập huấn luyện gốc, nhưng thay vì bắt chước nhãn thật, mô hình học cách mô phỏng lại đầu ra phân phối của mô hình lớn, bao gồm cả xác suất của các lớp sai (Hinton et al., 2014, 2015).

12.1.5 Cấu trúc động

Một cách để tăng tốc hệ thống xử lý là sử dụng **cấu trúc động**, tức là mô hình chỉ thực hiện một phần tính toán cần thiết tùy thuộc vào đầu vào. Điều này có thể xảy ra ở mức hệ thống — chọn mạng nào cần chạy, hoặc bên trong mạng — chọn phần nào của mạng cần được kích hoạt.

Kiến trúc động còn được gọi là **tính toán có điều kiện** (conditional computation) (Bengio, 2013; Bengio et al., 2013b), và thường mang lại hiệu quả cao khi mỗi đặc trưng chỉ liên quan tới một số ít đầu vào.

Chiến lược cascade: Một cách đơn giản để thực hiện điều này là **xâu chuỗi các bộ phân loại (cascade)**. Các bộ phân loại đơn giản được dùng trước để loại bỏ nhanh những đầu vào không chứa đối tượng. Nếu đầu vào vượt qua các bước này, nó sẽ được xử lý bởi một bộ phân loại chính xác hơn ở cuối chuỗi. Phương pháp này đặc biệt hiệu quả khi đối tượng cần phát hiện là hiếm gặp.

Viola và Jones (2001) đã áp dụng ý tưởng này để phát hiện khuôn mặt bằng cách quét hình ảnh với cửa sổ trượt và loại bỏ sớm các vùng không tiềm năng.

Mạng lựa chọn chuyên gia: Một biến thể khác là **mixture of experts**, trong đó một mạng gater quyết định chuyên gia nào được sử dụng (Nowlan, 1990; Jacobs et al., 1991). Nếu chỉ một chuyên gia được chọn tại mỗi bước, ta có **hard mixture of experts** — giúp tiết kiệm chi phí tính toán (Collobert et al., 2001, 2002).

Một số nghiên cứu đã cố gắng huấn luyện gater rời rạc bằng kỹ thuật như policy gradient (Bengio et al., 2013b; Bacon et al., 2015) để học dạng dropout có điều kiện.

Khó khăn trong triển khai: Tuy nhiên, cấu trúc động gây ra thách thức với phần cứng. Ví dụ, khi các đầu vào khác nhau kích hoạt nhánh khác nhau, việc song song hóa trên GPU sẽ kém hiệu quả. Có thể nhóm các đầu vào giống nhau lại để xử lý chung, nhưng cách này gây mất cân bằng tải trong môi trường thời gian thực.

12.1.6 Các triển khai phần cứng chuyên biệt của mạng sâu

Từ những năm đầu của nghiên cứu mạng nơ-ron, các nhà nghiên cứu phần cứng đã phát triển các hệ thống **phần cứng chuyên biệt** nhằm tăng tốc huấn luyện và suy luận. Các thiết kế bao gồm:

- **ASIC**: mạch tích hợp chuyên dụng, có thể là kỹ thuật số, analog hoặc kết hợp.
- **FPGA**: mảng cổng lập trình trường, có thể tái cấu hình linh hoạt.

Mặc dù CPU/GPU hiện nay dùng số thực 32 hoặc 64 bit, nhưng nhiều nghiên cứu đã chỉ ra rằng suy luận chỉ cần 8–16 bit (Holt và Baker, 1991; Simard và Gra

12.2 Thị giác máy tính (Computer Vision)

Thị giác máy tính từ lâu đã là một trong những lĩnh vực ứng dụng nổi bật và sôi động nhất của học sâu. Lý do là vì — mặc dù nhìn là một nhiệm vụ tưởng chừng đơn giản với con người và nhiều loài động vật — nhưng lại cực kỳ khó để máy tính thực hiện một cách chính xác **Ballard1983**.

Nhiều bộ dữ liệu đánh giá nổi tiếng nhất cho các thuật toán học sâu đều xoay quanh các bài toán nhận diện đối tượng hoặc nhận dạng ký tự quang học (OCR), phản ánh tầm quan trọng trung tâm của thị giác máy tính trong AI.

Thị giác máy tính là một lĩnh vực rộng, bao gồm nhiều phương pháp xử lý ảnh và vô số ứng dụng thực tế. Các ứng dụng có thể trải dài từ những mục tiêu mô phỏng thị giác con người, như nhận diện khuôn mặt, đến những tác vụ mà con người không thể làm được — ví dụ như trích xuất âm thanh từ các rung động nhỏ trên vật thể trong video **Davis2014**.

Tuy nhiên, phần lớn các nghiên cứu deep learning trong thị giác máy tính vẫn tập trung vào việc mô phỏng năng lực thị giác của con người. Các nhiệm vụ phổ biến bao gồm:

- Nhận diện đối tượng trong ảnh (object recognition).
- Phát hiện đối tượng và xác định hộp chứa (bounding box).
- Phiên âm chuỗi ký tự từ hình ảnh (ví dụ như OCR).
- Phân đoạn ngữ nghĩa (semantic segmentation) — gán nhãn từng điểm ảnh theo lớp đối tượng.

Vì mô hình sinh (generative models) là một hướng nghiên cứu quan trọng trong học sâu, nhiều nghiên cứu cũng tập trung vào việc **sinh ảnh** (image synthesis). Mặc dù sinh ảnh không hoàn toàn thuộc lĩnh vực thị giác máy tính, nhưng nó rất hữu ích cho các tác vụ như:

- Phục hồi ảnh hỏng (image inpainting).
- Xóa các đối tượng không mong muốn khỏi ảnh.

12.2.1 Tiền xử lý (Preprocessing)

Trong nhiều bài toán học sâu khác, dữ liệu đầu vào thường có cấu trúc phức tạp, cần qua nhiều bước xử lý trước khi đưa vào mạng. Ngược lại, trong thị giác máy tính, dữ liệu đầu vào là ảnh — thường có định dạng thống nhất và dễ xử lý hơn.

Một bước tiền xử lý quan trọng là **chuẩn hóa ảnh** (image normalization), ví dụ: chuyển giá trị pixel từ khoảng $[0, 255]$ về $[0, 1]$ hoặc $[-1, 1]$. Trộn lẫn dữ liệu chuẩn hóa và dữ liệu chưa chuẩn hóa có thể làm giảm hiệu quả học hoặc gây lỗi cho mô hình.

Phần lớn các mạng nơ-ron trong thị giác yêu cầu ảnh đầu vào có kích thước cố định, nên ảnh thường phải được *resize* hoặc *crop* trước khi đưa vào mạng. Tuy nhiên, một số kiến trúc hiện đại có thể xử lý ảnh có kích thước thay đổi. Ví dụ:

- Một số mạng tích chập điều chỉnh kích thước vùng pooling để đầu ra giữ nguyên kích thước **Waibel1989**.
- Các mạng khác sinh đầu ra có kích thước tỷ lệ với ảnh gốc — thường dùng trong tác vụ như phân đoạn từng điểm ảnh hoặc làm sạch ảnh **Hadsell2007**.

Một kỹ thuật quan trọng khác là **tăng cường dữ liệu (data augmentation)**, được áp dụng ở giai đoạn huấn luyện để giảm lỗi khái quát hoá. Ví dụ, ta có thể xoay, lật, dịch ảnh, hoặc thay đổi độ sáng.

Tại thời điểm kiểm thử, một kỹ thuật tương tự có thể được dùng: cung cấp nhiều biến thể của cùng một ảnh đầu vào (như nhiều bản crop khác nhau), và dùng trung bình hoặc bỏ phiếu từ nhiều đầu ra của mô hình để đưa ra kết quả cuối cùng. Đây là một hình thức **kết hợp mô hình (ensemble)** giúp giảm sai số dự đoán.

Một số kỹ thuật tiền xử lý khác có thể áp dụng cả trong huấn luyện và kiểm thử, nhằm chuẩn hóa dữ liệu và giảm độ biến thiên không cần thiết, giúp:

- Giảm lỗi khái quát hoá.
- Giảm kích thước mô hình cần thiết.
- Giúp mô hình đơn giản hơn nhưng vẫn hiệu quả.

Những yếu tố bị loại bỏ trong bước tiền xử lý thường là các biến thiên mà con người biết là không liên quan tới bài toán. Tuy nhiên, khi có đủ dữ liệu và mô hình đủ lớn, ta có thể bỏ qua bước tiền xử lý và để mạng tự học cách bỏ qua các yếu tố không cần thiết.

Ví dụ, trong mạng AlexNet dùng cho phân loại ảnh ImageNet, bước tiền xử lý duy nhất là: *trừ trung bình giá trị của từng pixel tính trên toàn bộ tập huấn luyện* **Krizhevsky2012**.

12.2.1.1 Chuẩn hóa độ tương phản (Contrast Normalization)

Một trong những nguồn biến thiên dễ xử lý nhất trong ảnh là **độ tương phản** — mức độ chênh lệch giữa vùng sáng và vùng tối. Trong học sâu, độ tương phản thường được đo bằng *độ lệch chuẩn* của các giá trị pixel trong toàn ảnh hoặc trong từng vùng nhỏ.

Giả sử ảnh được biểu diễn bởi tensor $X \in \mathbb{R}^{r \times c \times 3}$, với:

- $X_{i,j,1}$: giá trị kênh đỏ tại vị trí dòng i, cột j,
- $X_{i,j,2}$: kênh xanh lá,
- $X_{i,j,3}$: kênh xanh dương.

Ta có thể tính **độ tương phản toàn cục (global contrast)** của ảnh bằng:

$$\text{Contrast}(X) = \sqrt{\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k} - \bar{X})^2} \tag{12.1}$$

trong đó:

$$\bar{X} = \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}X_{i,j,k} \tag{12.2}$$

là giá trị trung bình của toàn ảnh.

Phương pháp **GCN (Global Contrast Normalization)** gồm hai bước chính:

1. Trừ đi trung bình $\bar{X}$ khỏi mỗi pixel.
2. Chia cho độ lệch chuẩn (có điều chỉnh) để đưa ảnh về cùng thang đo.

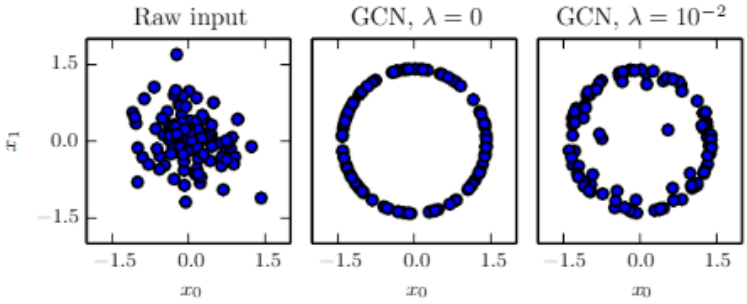
Với ảnh có độ tương phản quá thấp, chia cho độ lệch chuẩn có thể gây khuếch đại nhiễu. Để tránh điều này, ta thêm một hằng số nhỏ λ hoặc ngưỡng ϵ vào mẫu số.

Công thức GCN hoàn chỉnh:

$$X'_{i,j,k} = s \cdot \frac{X_{i,j,k} - \bar{X}}{\max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k} - \bar{X})^2}\right\}} \tag{12.3}$$

Một số thiết lập thường dùng:

- Với ảnh từ các tập như CIFAR-10: $\lambda = 0$, $\epsilon = 10^{-8}$
- Với các patch ảnh nhỏ: Coates et al. (2011) dùng $\lambda = 10$, $\epsilon = 0$
- Hệ số tỉ lệ s: thường đặt bằng 1, hoặc được chọn sao cho pixel có độ lệch chuẩn gần 1 trên toàn tập dữ liệu.



Hình 12.1: GCN ánh xạ dữ liệu đầu vào lên bề mặt cầu. Với $\lambda = 0$, dữ liệu gần như được đưa lên một hình cầu, tuy không hoàn toàn chính xác. Với $\lambda > 0$, dữ liệu được kéo về phía hình cầu nhưng vẫn giữ một phần biến thiên ban đầu.

Ta có thể hình dung GCN như một phép chiếu dữ liệu lên một lớp vỏ cầu trong không gian. Điều này có ích vì mạng nơ-ron thường nhạy hơn với hướng của vector đầu vào hơn là độ lớn của chúng. Việc chuẩn hóa giúp mạng học dễ hơn trong việc phát hiện các mẫu đặc trưng.

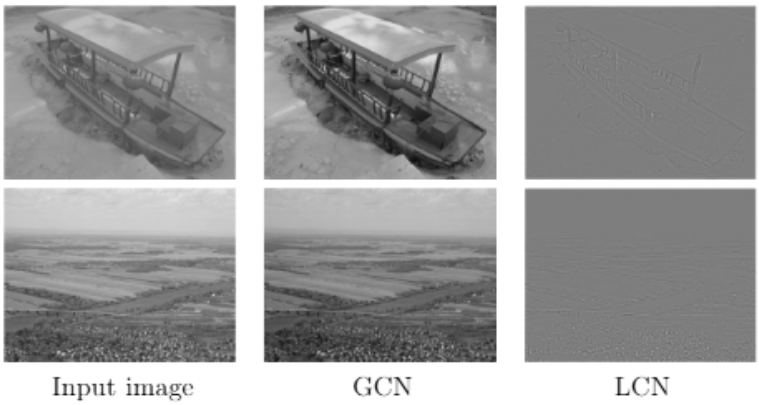
Lưu ý: **GCN không giống sphering**. Dù cả hai đều chuẩn hóa, sphering là biến đổi để làm cho các phương sai bằng nhau sau PCA, còn GCN chuẩn hóa trên toàn ảnh.

GCN không làm nổi bật các đặc trưng như cạnh hay góc — lý do khiến ta cần đến **LCN (Local Contrast Normalization)**.

LCN – Chuẩn hóa độ tương phản cục bộ:

Khác với GCN, LCN chuẩn hóa ảnh theo từng vùng nhỏ, giúp làm nổi bật các cạnh, góc — những đặc trưng quan trọng trong thị giác.

- Trừ đi trung bình cục bộ quanh mỗi điểm ảnh.
- Chia cho độ lệch chuẩn cục bộ (có thể áp dụng trọng số Gauss để ưu tiên điểm gần tâm).



Hình 12.2: So sánh hiệu ứng trực quan giữa GCN và LCN. GCN đưa ảnh về cùng thang đo tổng thể, trong khi LCN làm nổi bật cấu trúc cục bộ như đường viền và biên cạnh.

Với ảnh màu, LCN có thể được áp dụng riêng cho từng kênh hoặc kết hợp từ nhiều

kênh.

LCN là phép biến đổi khả vi — có thể dùng như một lớp trong mạng, hoặc là bước tiền xử lý.

Vì LCN hoạt động trên cửa sổ nhỏ, nên xác suất độ lệch chuẩn bằng 0 cao hơn. Khi đó, cần thêm ϵ vào mẫu số để tránh lỗi chia cho 0.

12.2.1.2 Tăng cường tập dữ liệu (Dataset Augmentation)

Như đã trình bày ở mục 7.4, tăng cường dữ liệu là một cách đơn giản nhưng hiệu quả để giúp mô hình tổng quát tốt hơn.

Trong bài toán **nhận dạng đối tượng**, ta có thể áp dụng nhiều phép biến đổi mà không làm thay đổi nhãn của ảnh. Ví dụ:

- Dịch chuyển ảnh (translation).
- Xoay ảnh (rotation).
- Lật ảnh (flip).
- Thay đổi độ sáng, tương phản.

Các kỹ thuật nâng cao hơn:

- **Nhiễu màu ngẫu nhiên (random color perturbation):** Thay đổi sắc độ, độ sáng hoặc tương phản một cách ngẫu nhiên mà không làm sai lệch nhãn **Krizhevsky2012**.
- **Biến dạng hình học phi tuyến (nonlinear distortions):** Cong, bóp méo ảnh theo cách khó dự đoán để tăng tính đa dạng, như trong **LeCun1998b**.

Tăng cường dữ liệu không chỉ giúp tăng kích thước tập huấn luyện mà còn làm mô hình **chống chịu tốt hơn với các biến đổi ngoài thực tế**, giúp cải thiện độ chính xác khi suy luận.

12.3 Nhận dạng giọng nói (Speech Recognition)

Nhận dạng giọng nói là bài toán chuyển đổi một tín hiệu âm thanh — trong đó chứa một câu nói bằng ngôn ngữ tự nhiên — thành một chuỗi từ đại diện đúng cho nội dung và ý định của người nói. Đây là một trong những ứng dụng thực tiễn và quan trọng nhất của học sâu trong xử lý tín hiệu.

Gọi $\boldsymbol{X} = (x^{(1)}, x^{(2)}, \ldots, x^{(T)})$ là chuỗi các vector đặc trưng đầu vào, được trích xuất từ tín hiệu âm thanh bằng cách chia nhỏ thành các khung (frame) khoảng 20 mili giây. Mỗi $x^{(t)}$ là một vector thể hiện đặc trưng âm thanh tại thời điểm t.

Trước đây, đặc trưng đầu vào thường là các đặc trưng thủ công như MFCC (Mel-frequency cepstral coefficients). Tuy nhiên, một số nghiên cứu học sâu gần đây (Jaitly và Hinton, 2011) đã cho thấy mạng nơ-ron có thể học đặc trưng trực tiếp từ tín hiệu âm thanh thô.

Gọi $\boldsymbol{y} = (y_1, y_2, \ldots, y_N)$ là chuỗi đầu ra mục tiêu — có thể là từ, ký tự hoặc âm vị

(phoneme) — đại diện cho nội dung lời nói.

Mục tiêu của hệ thống nhận dạng giọng nói tự động (ASR - Automatic Speech Recognition) là xây dựng một hàm f^*_{ASR} ánh xạ đầu vào $\boldsymbol{X}$ sang chuỗi ngôn ngữ $\boldsymbol{y}$ có xác suất cao nhất:

$$f^*_{\text{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\boldsymbol{y} \mid \boldsymbol{X} = \boldsymbol{X}) \tag{12.4}$$

Ở đây, P^* là phân phối xác suất thật sự mô tả mối quan hệ giữa tín hiệu âm thanh và nội dung phát ngôn.

Giai đoạn truyền thống: GMM-HMM

Từ những năm 1980 đến khoảng 2009–2012, các hệ thống ASR tiên tiến nhất dựa trên kết hợp giữa:

- **GMM** (Gaussian Mixture Models): mô hình hóa mối liên hệ giữa vector đặc trưng và các âm vị [21],
- **HMM** (Hidden Markov Models): mô hình hóa chuỗi các âm vị và trạng thái con của chúng (đầu, giữa, cuối).

Hệ thống hoạt động theo quy trình:

1. HMM sinh ra chuỗi âm vị và trạng thái rời rạc.
2. GMM ánh xạ các trạng thái này thành tín hiệu âm thanh tương ứng.

Dù GMM-HMM là chuẩn mực trong nhiều năm, mạng nơ-ron cũng đã được thử nghiệm từ rất sớm (cuối thập niên 1980), ví dụ trong các nghiên cứu của **robinson1991**, [17], [22], [23], [24].

Robinson và Fallside (1991) đạt tỷ lệ lỗi âm vị (phoneme error rate) 26% trên tập TIMIT — tương đương hoặc vượt hơn HMM thời điểm đó. Điều này khiến TIMIT trở thành một bộ dữ liệu chuẩn cho đánh giá hệ thống ASR.

Tuy nhiên, vì kiến trúc mạng nơ-ron khó tích hợp vào hệ thống GMM-HMM đã tồn tại và được đầu tư công phu, nên công nghiệp lúc đó không sẵn sàng chuyển đổi.

Bước ngoặt: học sâu với pretraining không giám sát

Khoảng năm 2009, một hướng mới nổi lên: dùng học sâu không giám sát để khởi tạo các tầng của mạng nơ-ron. Cụ thể:

- Dùng **RBM** (Restricted Boltzmann Machine) để học đặc trưng từ âm thanh.
- Dùng nhiều RBM chồng nhau để khởi tạo mạng nhiều tầng.

Mạng được huấn luyện với đầu vào là cửa sổ đặc trưng âm thanh quanh một khung trung tâm, và đầu ra là xác suất trạng thái HMM tương ứng. Kết quả:

- Lỗi âm vị trên TIMIT giảm từ 26% còn 20.7% **mohamed2009**, [25].

- Thêm đặc trưng theo người nói còn giúp giảm lỗi sâu hơn nữa **mohamed2011**.

Từ nhận dạng âm vị sang từ vựng lớn

Dahl et al. (2012) mở rộng từ nhận dạng âm vị sang nhận dạng từ có từ vựng lớn (large-vocabulary ASR), mở ra khả năng áp dụng vào các hệ thống thực tế.

Sau đó, cộng đồng dần chuyển từ:

- RBM và pretraining không giám sát,
- sang dùng ReLU và dropout để huấn luyện mạng sâu từ đầu [26], [27].

Hinton et al. (2012a) tổng hợp kết quả ban đầu của các nhóm nghiên cứu lớn, và nhiều hệ thống thực tế đã được triển khai — ví dụ, trên điện thoại di động.

Sự bùng nổ của học sâu trong ASR

Khi tập dữ liệu được gán nhãn lớn dần lên, cộng với các kỹ thuật hiện đại trong khởi tạo và huấn luyện mạng, các nghiên cứu cho thấy pretraining không còn cần thiết. Một bước tiến đột phá xuất hiện: **giảm WER (word error rate)** khoảng 30% — điều chưa từng đạt được trong gần một thập kỷ trước đó với GMM-HMM [28].

Trong chỉ 2 năm, phần lớn hệ thống ASR trong công nghiệp đã chuyển sang dùng mạng nơ-ron sâu (DNN). Điều này cũng dẫn đến một làn sóng đổi mới về thuật toán và kiến trúc trong cộng đồng ASR.

CNN và RNN cho ASR

- **CNN**: Sainath et al. (2013) dùng mạng tích chập để khai thác cấu trúc hai chiều của phổ âm thanh (thời gian × tần số). Khác với TDNN chỉ chia sẻ theo thời gian, CNN chia sẻ trọng số cả theo tần số.
- **RNN**: Graves et al. (2013) dùng RNN nhiều tầng (deep RNN với LSTM), huấn luyện đầu-cuối bằng CTC (Connectionist Temporal Classification). Kết quả: lỗi âm vị trên TIMIT giảm còn 17.7%.

CTC cho phép mô hình học cách căn chỉnh tự động giữa chuỗi đầu vào và đầu ra, không cần gắn nhãn từng khung. Hướng này tiếp tục được mở rộng với cơ chế **attention**, giúp mô hình học cách tập trung vào phần âm thanh liên quan (Chorowski et al., 2014; Lu et al., 2015).

Tổng kết

Nhờ học sâu, ASR đã có một bước nhảy vọt, không chỉ về độ chính xác mà còn về khả năng huấn luyện đầu-cuối, giảm phụ thuộc vào các mô hình thống kê truyền thống như HMM. Học sâu đã trở thành nền tảng chính cho nhiều hệ thống nhận dạng giọng nói hiện đại.

12.4 Xử lý ngôn ngữ tự nhiên (Natural Language Processing)

Xử lý ngôn ngữ tự nhiên (Natural Language Processing - NLP) là lĩnh vực nghiên cứu cách máy tính hiểu, phân tích và tạo ra ngôn ngữ của con người, như tiếng Anh hoặc tiếng Pháp. Khác với các ngôn ngữ lập trình được thiết kế để dễ phân tích cú pháp, ngôn ngữ tự nhiên thường rất mơ hồ, không rõ ràng, và khó mô tả chặt chẽ bằng các quy tắc hình thức.

NLP bao gồm nhiều ứng dụng thực tế, chẳng hạn như dịch máy, trong đó hệ thống cần đọc một câu ở ngôn ngữ nguồn và tạo ra câu tương đương ở ngôn ngữ đích. Phần lớn các hệ thống NLP hiện đại dựa trên **mô hình ngôn ngữ** — một mô hình xác suất trên các chuỗi từ, ký tự, hoặc byte trong ngôn ngữ tự nhiên.

Tương tự các lĩnh vực khác, mạng nơ-ron nói chung có thể được áp dụng hiệu quả vào NLP. Tuy nhiên, để đạt hiệu quả tối ưu và có khả năng mở rộng, ta cần một số chiến lược chuyên biệt dành riêng cho ngữ liệu tuần tự.

Đặc biệt, khi xây dựng mô hình ngôn ngữ, chúng ta thường coi văn bản là chuỗi các từ. Vì số lượng từ rất lớn nên không gian đầu vào có kích thước cao và rất thưa, đòi hỏi các kỹ thuật mô hình hóa hiệu quả cả về tính toán lẫn thống kê.

12.5 Mô hình N-gram

Mô hình ngôn ngữ là mô hình xác suất định nghĩa phân phối trên các chuỗi token trong ngôn ngữ tự nhiên. Một *token* có thể là từ, ký tự hoặc byte, và luôn được coi là một đơn vị rời rạc.

Một trong những mô hình ngôn ngữ sớm và đơn giản nhất là **mô hình n-gram**, trong đó chuỗi token được phân chia thành các cụm liên tiếp có độ dài n:

- $n = 1$: unigram (một từ),
- $n = 2$: bigram (hai từ),
- $n = 3$: trigram (ba từ), ...

Ý tưởng chính của n-gram là: xác suất của một từ trong câu phụ thuộc vào $n - 1$ từ đứng trước nó. Ta ước lượng xác suất có điều kiện:

$$P(x_t \mid x_{t-n+1}, \ldots, x_{t-1})$$

Sau đó, xác suất toàn chuỗi được tính bằng quy tắc chuỗi:

$$P(x_1, x_2, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \cdot \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \tag{12.5}$$

Ở đây:

- $P(x_1, \ldots, x_{n-1})$ là phân phối ban đầu cho $n - 1$ từ đầu tiên,

• từ $t = n$ trở đi, xác suất được tính theo ngữ cảnh $n - 1$ từ trước đó.

Việc huấn luyện n-gram khá đơn giản nhờ *ước lượng hợp lý cực đại (MLE)* — chỉ cần đếm tần suất n-gram trong tập huấn luyện.

Trong nhiều năm, các mô hình n-gram là nền tảng chính cho NLP thống kê, với các công trình như Jelinek và Mercer (1980), Katz (1987), và Chen & Goodman (1999).

Một phương pháp phổ biến là huấn luyện đồng thời cả mô hình n-gram và $(n-1)$-gram để tính xác suất có điều kiện:

$$P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \ldots, x_t)}{P_{n-1}(x_{t-n+1}, \ldots, x_{t-1})} \tag{12.6}$$

Ví dụ: Với mô hình trigram, xác suất câu "THE DOG RAN AWAY" được tính như sau:

$$P(\text{THE DOG RAN AWAY}) = \frac{P_3(\text{THE DOG RAN}) \cdot P_3(\text{DOG RAN AWAY})}{P_2(\text{DOG RAN})} \tag{12.7}$$

Hạn chế của MLE: Với MLE, bất kỳ n-gram nào không xuất hiện trong tập huấn luyện đều có xác suất bằng 0, gây ra hai vấn đề:

• Nếu $P_{n-1} = 0$: công thức (12.6) không xác định.

• Nếu $P_n = 0$: log-likelihood của câu sẽ là $-\infty$.

Giải pháp: Làm mượt (smoothing)

Để khắc phục, ta áp dụng các kỹ thuật làm mượt như:

• **Làm mượt cộng thêm (additive smoothing):** thêm lượng nhỏ vào mọi n-gram.

• **Mô hình hỗn hợp (mixture models):** kết hợp trigram, bigram, unigram...

• **Back-off:** nếu ngữ cảnh ít xuất hiện, quay lui về ngữ cảnh ngắn hơn.

Vấn đề: Lời nguyền chiều không gian

Với từ vựng kích thước $|V|$, số lượng n-gram có thể là $|V|^n$. Khi n lớn, số lượng này tăng rất nhanh, khiến hầu hết n-gram không xuất hiện trong tập huấn luyện — kể cả khi tập dữ liệu rất lớn.

Nhìn theo cách khác, n-gram giống như một hệ thống dự đoán dựa trên *tìm kiếm lân cận gần nhất* trong không gian one-hot — nơi mọi từ đều cách nhau bằng nhau. Điều này làm cho việc chia sẻ thông tin giữa các từ tương tự trở nên bất khả thi.

Giải pháp: Mô hình dựa trên lớp từ (class-based models)

Một hướng khắc phục là phân nhóm từ thành các *lớp* (cluster), rồi chia sẻ thống kê giữa các từ cùng lớp. Các mô hình n-gram sau đó sử dụng mã lớp thay cho mã từ cụ thể ở

phần điều kiện.

Đây là hướng tiếp cận của các công trình như Brown et al. (1992), Ney và Kneser (1993), Niesler et al. (1998).

Dù giúp mô hình tổng quát hóa tốt hơn, việc biểu diễn theo lớp cũng khiến ta mất đi một số thông tin chi tiết quan trọng.

12.5.1 Mô hình ngôn ngữ thần kinh (Neural Language Models)

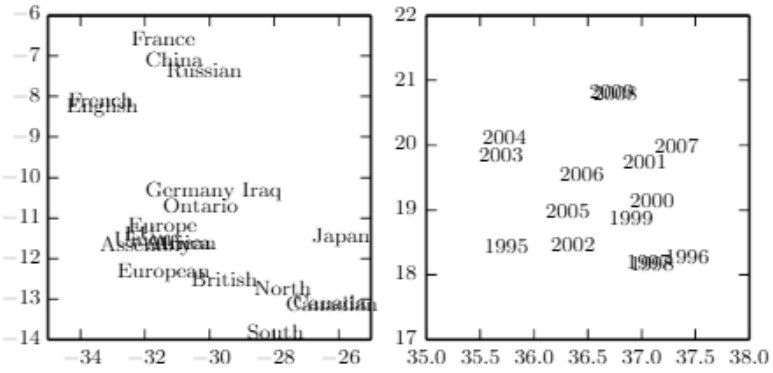
Mô hình ngôn ngữ thần kinh (Neural Language Models - NLMs) là một lớp mô hình ngôn ngữ được thiết kế để vượt qua hạn chế của *lời nguyền chiều không gian* — vấn đề xảy ra khi cố gắng mô hình hóa xác suất trên các chuỗi từ có độ dài thay đổi và không gian rời rạc kích thước rất lớn. Giải pháp chính của NLM là sử dụng **biểu diễn phân tán** (distributed representation) cho các từ (Bengio et al., 2001).

Khác với mô hình n-gram dựa trên phân lớp (class-based), vốn chỉ gom các từ vào nhóm và dùng đại diện nhóm để khái quát hóa, NLM có thể vừa nhận biết sự tương đồng giữa các từ, vừa giữ được bản sắc riêng biệt của từng từ. Cụ thể, các mô hình này chia sẻ thông tin thống kê giữa các từ có ngữ cảnh tương tự, thông qua việc học các biểu diễn véc-tơ trong không gian liên tục — nơi mà những từ tương đồng sẽ có véc-tơ gần nhau.

Ví dụ, nếu từ *"dog"* và *"cat"* có biểu diễn véc-tơ gần nhau, thì mô hình có thể sử dụng kinh nghiệm học được từ câu chứa *"cat"* để cải thiện dự đoán cho các câu chứa *"dog"*, và ngược lại. Nhờ biểu diễn bằng nhiều đặc trưng (features), mô hình có khả năng khái quát mạnh — thông tin từ mỗi câu huấn luyện có thể được truyền sang nhiều câu tương đồng về mặt ngữ nghĩa.

Một trong những nguyên nhân khiến mô hình truyền thống như n-gram kém hiệu quả là vì sử dụng biểu diễn one-hot cho từ: mỗi từ là một véc-tơ toàn số 0, chỉ có đúng một phần tử là 1 tại vị trí đại diện cho từ đó. Điều này khiến mọi cặp từ đều cách nhau bằng nhau trong không gian Euclid (khoảng cách $\sqrt{2}$), không phản ánh bất kỳ mối liên hệ ngữ nghĩa nào.

Ngược lại, trong các mô hình ngôn ngữ thần kinh, mỗi từ được ánh xạ sang một không gian nhúng (embedding space) có số chiều thấp hơn, nơi các từ cùng ngữ cảnh sẽ gần nhau hơn. Đây chính là các **word embeddings** — véc-tơ từ được học sao cho phản ánh các mối liên hệ ngữ nghĩa. Chẳng hạn, *"king"* và *"queen"*, hay *"run"* và *"walk"*, sẽ có véc-tơ gần nhau trong không gian embedding.



Hình 12.3: Trực quan hóa word embedding hai chiều được trích xuất từ mô hình dịch máy sử dụng mạng nơ-ron (Bahdanau et al., 2015). Các từ liên quan ngữ nghĩa có vector nhúng gần nhau. Cụ thể, các quốc gia nằm bên trái và các số nằm bên phải. Hình ảnh này sử dụng kỹ thuật giảm chiều để trực quan hóa; trong thực tế, vector embedding thường có hàng trăm chiều để biểu diễn nhiều kiểu quan hệ ngữ nghĩa khác nhau giữa các từ.

Thách thức chính của bài toán mô hình ngôn ngữ là phải học từ một lượng dữ liệu huấn luyện hữu hạn, nhưng có thể dự đoán tốt cho số lượng chuỗi đầu ra khổng lồ — tăng theo cấp số mũ theo độ dài chuỗi. Các mô hình ngôn ngữ thần kinh giúp khái quát tốt nhờ tận dụng sự tương đồng ngữ nghĩa và cấu trúc ngữ cảnh từ dữ liệu.

Ý tưởng biểu diễn phân tán không chỉ có trong NLP. Trong các mạng tích chập (CNN), các lớp ẩn cũng học được biểu diễn embedding cho ảnh đầu vào. Tuy nhiên, với NLP, embedding đặc biệt quan trọng bởi vì ngôn ngữ tự nhiên không có biểu diễn liên tục sẵn có — các từ ban đầu là các token rời rạc.

Việc học biểu diễn phân tán không chỉ giới hạn trong mô hình thần kinh. Một số mô hình đồ thị (graphical models) cũng có thể học embedding thông qua các biến tiềm ẩn (latent variables), như trong nghiên cứu của Mnih và Hinton (2007).

12.6 Đầu ra có chiều cao (High-Dimensional Outputs)

Trong nhiều ứng dụng NLP, chúng ta thường mong muốn mô hình đầu ra là các từ, thay vì ký tự. Tuy nhiên, khi từ vựng có kích thước rất lớn (hàng trăm nghìn từ), việc mô hình hóa phân phối xác suất trên toàn bộ tập từ vựng trở nên cực kỳ tốn kém.

Giả sử tập từ vựng có kích thước $|V|$. Cách tiếp cận phổ biến là ánh xạ véc-tơ ẩn $h \in \mathbb{R}^{n_h}$ sang không gian từ vựng bằng một phép biến đổi affine, rồi áp dụng softmax để tạo phân phối xác suất. Cụ thể:

- Trọng số của phép biến đổi là ma trận $W \in \mathbb{R}^{|V| \times n_h}$.
- Phép nhân ma trận W với h cho ra véc-tơ logit $a \in \mathbb{R}^{|V|}$.
- Phân phối softmax được tính như sau:

$$a_i = b_i + \sum_j W_{ij} h_j, \quad forall i \in \{1, \ldots, |V|\} \tag{12.5}$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|V|} e^{a_{i'}}} \tag{12.6}$$

Khi n_h khoảng vài nghìn và $|V|$ lên tới hàng trăm nghìn, phép biến đổi này có độ phức tạp $O(|V| \cdot n_h)$, dẫn đến các vấn đề:

- **Tốn bộ nhớ**: do ma trận W quá lớn.
- **Tốn thời gian tính toán**, cả trong huấn luyện và suy luận:
  - Trong huấn luyện: phải tính toàn bộ phân phối để tối ưu cross-entropy.
  - Trong suy luận: phải xét toàn bộ từ vựng để chọn từ có xác suất cao nhất.
- Không thể tối ưu bằng cách chỉ tính toán cho từ đúng, vì softmax yêu cầu chuẩn hóa toàn bộ.

Một số hàm mất mát đặc biệt (ví dụ Vincent et al., 2015) có thể giúp giảm chi phí tính gradient. Tuy nhiên, khi dùng softmax và cross-entropy tiêu chuẩn, tầng đầu ra thường là phần tốn kém nhất trong mô hình ngôn ngữ thần kinh hiện đại.

12.6.0.1 Sử dụng danh sách ngắn (Use of a Short List)

Các mô hình ngôn ngữ thần kinh đầu tiên (Bengio et al., 2001, 2003) xử lý chi phí tính toán cao bằng cách chỉ xét một danh sách ngắn khoảng 10.000–20.000 từ phổ biến.

Ý tưởng này được mở rộng bởi Schwenk và Gauvain (2002), sau đó bởi Schwenk (2007), bằng cách chia tập từ vựng V thành hai phần:

- **Danh sách ngắn** L: gồm các từ phổ biến nhất, được xử lý bằng mạng nơ-ron.
- **Phần đuôi** $T = V \setminus L$: gồm các từ hiếm, được xử lý bởi mô hình n-gram truyền thống.

Để kết hợp đầu ra từ hai mô hình, mạng nơ-ron sẽ học thêm một xác suất:

$$P(i \in T \mid C)$$

tức là xác suất một từ thuộc phần đuôi T khi đã thấy ngữ cảnh C.

Khi đó, xác suất đầy đủ cho mỗi từ i được tính bằng:

$$P(y = i \mid C) = \begin{cases} P(y = i \mid C, i \in L) \cdot (1 - P(i \in T \mid C)) & \text{nếu } i \in L \\ P(y = i \mid C, i \in T) \cdot P(i \in T \mid C) & \text{nếu } i \in T \end{cases} \tag{12.7}$$

Trong đó:

- $P(y = i \mid C, i \in L)$ được sinh từ mô hình mạng nơ-ron.
- $P(y = i \mid C, i \in T)$ lấy từ mô hình n-gram.

Xác suất $P(i \in T \mid C)$ có thể được ước lượng bằng một nút sigmoid riêng, hoặc có thể biểu diễn như một giá trị đặc biệt trong tầng softmax.

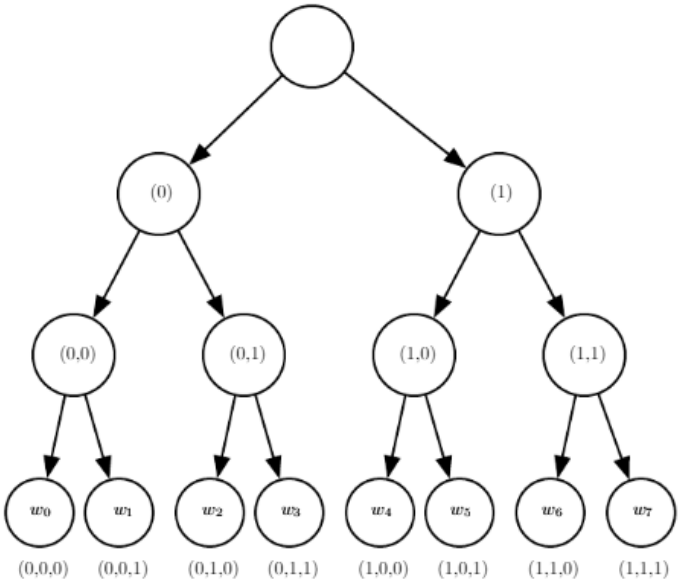
Hạn chế: Với phương pháp danh sách ngắn, khả năng khái quát hóa của mạng nơ-ron chỉ áp dụng cho các từ phổ biến — trong khi chính các từ hiếm mới cần được mô hình hóa hiệu quả hơn. Điều này đã dẫn đến sự ra đời của nhiều kỹ thuật mới để xử lý bài toán đầu ra có chiều cao, sẽ được trình bày trong các phần tiếp theo.

12.6.0.2 Softmax phân cấp (Hierarchical Softmax)

Một cách tiếp cận cổ điển để giảm chi phí tính toán trong tầng đầu ra có chiều cao là phân rã phân phối xác suất theo cấu trúc phân cấp (Goodman, 2001). Thay vì phải thực hiện số lượng phép tính tỷ lệ với $|V|$ (kích thước từ vựng), ta có thể giảm xuống chỉ còn $O(\log |V|)$ nếu tổ chức từ vựng thành một cây nhị phân.

Phương pháp này được Bengio (2002) và Morin và Bengio (2005) đưa vào áp dụng trong các mô hình ngôn ngữ thần kinh.

Ý tưởng chính là tổ chức các từ thành một cây, trong đó mỗi từ là một lá, và việc chọn một từ được thực hiện bằng cách đi qua các nút trên đường từ gốc đến lá đó. Mỗi nút tương ứng với một quyết định nhị phân: đi trái hay phải. Nếu cây cân bằng tốt, độ sâu sẽ vào khoảng $\log |V|$, từ đó giảm đáng kể chi phí tính toán.



Hình 12.4: Minh họa một hệ phân cấp đơn giản gồm 8 từ $w_0, \ldots, w_7$. Các từ được tổ chức thành cây ba cấp. Các lá là các từ, còn các nút trong là các nhóm từ. Mỗi từ có thể được xác định bởi một chuỗi các quyết định nhị phân (0 = trái, 1 = phải). Ví dụ, để đến được w_4, ta đi qua các nút với nhãn nhị phân $(1, 0, 0)$, tương đương với: $P(y = w_4) = P(b_0 = 1) \cdot P(b_1 = 0 \mid b_0 = 1) \cdot P(b_2 = 0 \mid b_0 = 1, b_1 = 0)$

Để dự đoán xác suất tại mỗi nút, ta thường dùng hồi quy logistic. Tất cả các nút đều nhận cùng một ngữ cảnh đầu vào C. Vì ta biết đường đi đúng đến từ mục tiêu, việc huấn luyện các mô hình logistic này có thể thực hiện độc lập bằng entropy chéo (cross-entropy loss), tương ứng với tối đa hóa log-xác suất của chuỗi quyết định đúng.

Lợi ích rõ ràng là log-likelihood và gradient của nó đều có thể tính nhanh với chi phí $O(\log |V|)$.

Mnih và Hinton (2009) đề xuất một biến thể mở rộng, trong đó một từ có thể có nhiều đường đi đến nó trong cây. Khi đó, xác suất của từ là tổng các xác suất trên tất cả các đường đi đó, giúp cải thiện khả năng mô hình hóa những từ đa nghĩa.

Về lý thuyết, người ta có thể tối ưu hóa cấu trúc cây để giảm số phép tính trung bình. Ví dụ, dựa trên lý thuyết thông tin, có thể xây dựng cây sao cho độ dài đường đi đến mỗi từ tỷ lệ nghịch với tần suất xuất hiện của từ đó. Điều này giúp giảm kỳ vọng chi phí tính toán trên toàn tập từ vựng.

Tuy nhiên, trong thực tế, chi phí tính toán ở tầng đầu ra thường không chiếm phần lớn thời gian tổng thể, đặc biệt khi so với chi phí ở các tầng ẩn. Giả sử mô hình có l tầng fully-connected, mỗi tầng có n_h đơn vị, và n_b là số bit trung bình cần thiết để mã hóa một từ, ta có:

$$\text{số phép tính tầng ẩn} = O(ln_h^2), \quad \text{số phép tính đầu ra} = O(n_h n_b)$$

Miễn là $n_b \leq \ln n_h$, thì giảm n_h vẫn là cách hiệu quả hơn để giảm tổng chi phí.

Trong thực tế, n_b khá nhỏ. Với từ vựng có khoảng 10^6 từ, thì $\log_2(10^6) \approx 20$, trong khi n_h thường khoảng 1000 hoặc hơn.

Một biến thể đơn giản là xây cây chỉ với độ sâu 2, mỗi nút có khoảng $\sqrt{|V|}$ nhánh, tức là chia từ vựng thành các lớp rời nhau (word classes). Dù đơn giản, cách này vẫn giúp giảm đáng kể chi phí tính toán và giữ lại phần lớn hiệu quả của softmax phân cấp.

Một thách thức lớn là làm sao để xây dựng cây phân cấp hoặc nhóm từ một cách tối ưu. Các nghiên cứu ban đầu sử dụng cấu trúc cây cố định (Morin và Bengio, 2005). Trong khi lý tưởng là học được cấu trúc cây cùng lúc với tham số mô hình, nhưng điều này rất khó thực hiện vì không thể tối ưu bằng gradient do lựa chọn cấu trúc là bài toán rời rạc.

Tuy nhiên, có thể áp dụng các kỹ thuật heuristic hoặc tối ưu hóa rời rạc để phân chia từ vựng thành các lớp.

Một ưu điểm lớn của phương pháp softmax phân cấp là: nó giúp tăng tốc cả trong huấn luyện và suy luận, đặc biệt khi chỉ cần tính xác suất của một từ cụ thể trong một ngữ cảnh.

Tuy nhiên, nếu ta muốn tính xác suất cho tất cả các từ trong từ vựng, thì chi phí vẫn rất cao — ngay cả với phân cấp.

Một hạn chế khác là việc tìm từ có xác suất cao nhất cũng không dễ với cây phân cấp, vì cây không được tổ chức theo giá trị xác suất.

Ngoài ra, trên thực nghiệm, các mô hình softmax phân cấp thường cho kết quả kém hơn so với các phương pháp dựa trên mẫu (sampling-based), sẽ được trình bày trong phần sau. Nguyên nhân có thể do cấu trúc lớp từ không tối ưu hoặc quá thô.

12.6.0.3 Lấy mẫu quan trọng (Importance Sampling)

Một cách để tăng tốc quá trình huấn luyện mô hình ngôn ngữ học sâu là tránh việc phải tính gradient cho toàn bộ từ vựng tại mỗi bước. Mỗi từ sai (không phải từ đúng tại vị trí cần dự đoán) cần có xác suất thấp, nhưng việc tính xác suất cho tất cả các từ là rất tốn kém.

Thay vào đó, ta có thể chọn ngẫu nhiên một tập con các từ sai để ước lượng phần gradient này. Sử dụng ký hiệu từ phương trình (12.8), ta có:

$$\frac{\partial \log P(y \mid C)}{\partial \theta} = \frac{\partial \log \text{softmax}_y(a)}{\partial \theta} \tag{12.13}$$

$$= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \tag{12.14}$$

$$= \frac{\partial}{\partial \theta} \left(a_y - \log \sum_i e^{a_i} \right) \tag{12.15}$$

$$= \frac{\partial a_y}{\partial \theta} - \sum_i P(i \mid C) \frac{\partial a_i}{\partial \theta} \tag{12.16}$$

Trong đó:

- Thành phần đầu tiên là **pha dương**, nhằm đẩy điểm số a_y của từ đúng lên cao hơn.
- Thành phần thứ hai là **pha âm**, kéo điểm số các từ sai xuống thấp, với trọng số là xác suất mô hình $P(i \mid C)$.

Do pha âm là một kỳ vọng, ta có thể xấp xỉ nó bằng phương pháp Monte Carlo. Tuy nhiên, nếu lấy mẫu từ chính mô hình $P(i \mid C)$ thì vẫn cần tính toán toàn bộ phân phối — điều mà ta đang muốn tránh.

Giải pháp là sử dụng một **phân phối lấy mẫu thay thế** $q(i)$ (gọi là *proposal distribution*), thường chọn là phân phối unigram hoặc bigram — dễ lấy mẫu và ước lượng.

Để hiệu chỉnh độ lệch do lấy mẫu từ q thay vì từ mô hình P, ta dùng *lấy mẫu quan trọng (importance sampling)*. Tuy nhiên, nếu dùng lấy mẫu quan trọng chính xác thì vẫn cần biết $P(i \mid C)$, nên không hiệu quả. Thay vào đó, ta dùng *phiên bản lệch (biased)*:

$$w_i = \frac{p_{n_i}/q_{n_i}}{\sum_{j=1}^{m} p_{n_j}/q_{n_j}} \tag{12.17}$$

Gradient của pha âm được xấp xỉ bằng:

$$\sum_{i=1}^{|V|} P(i \mid C) \frac{\partial a_i}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^{m} w_i \frac{\partial a_{n_i}}{\partial \theta} \tag{12.18}$$

Trong đó:

- m là số lượng từ âm được lấy mẫu.
- p_{n_i} là xác suất mô hình (không cần chuẩn hóa).
- q_{n_i} là xác suất từ n_i theo phân phối lấy mẫu.

Lưu ý: q nên phản ánh xác suất xuất hiện thực tế của các từ âm — ví dụ như phân phối unigram. Điều này giúp mô hình tập trung vào những từ âm dễ gây nhầm lẫn.

Lấy mẫu quan trọng không chỉ giúp giảm chi phí tính toán ở tầng softmax có từ vựng lớn, mà còn hữu ích với các mô hình có đầu ra thưa (sparse outputs), chẳng hạn như đầu ra dạng *bag of words*.

Dauphin et al. (2011) đề xuất:

- Chỉ tính gradient cho các từ mục tiêu (các từ "dương"),
- Lấy thêm một số từ "âm" bằng một chiến lược lấy mẫu,
- Dùng trọng số lấy mẫu quan trọng để hiệu chỉnh sai lệch.

Kết luận: Lấy mẫu quan trọng giúp giảm chi phí gradient ở lớp đầu ra từ $O(|V|)$ xuống $O(m)$ — với m là số từ âm được lấy mẫu, rất nhỏ so với $|V|$. Đây là một chiến lược đặc biệt hiệu quả cho các mô hình ngôn ngữ học sâu.

12.6.0.4 Ước lượng đối lập với nhiễu và hàm mất mát xếp hạng

Một hướng tiếp cận khác để tránh chi phí tính toán lớn của tầng softmax là sử dụng các hàm mất mát thay thế không cần tính toàn bộ phân phối xác suất.

Một ví dụ đầu tiên là **hàm mất mát xếp hạng (ranking loss)** của Collobert và Weston (2008a). Mỗi từ được gán một điểm số a_i và mục tiêu là đảm bảo từ đúng y có điểm cao hơn các từ sai i:

$$L = \sum_i \max(0, 1 - a_y + a_i) \tag{12.8}$$

Nếu a_y lớn hơn a_i ít nhất 1 đơn vị thì không bị phạt; nếu không, hàm mất mát sẽ đẩy a_y lên và a_i xuống.

Hạn chế: hàm xếp hạng không cho ra phân phối xác suất có điều kiện $P(w_t \mid w_{<t})$, nên không dùng được cho các ứng dụng cần xác suất — như sinh ngôn ngữ hay dịch máy.

Một giải pháp tốt hơn là **Ước lượng đối lập với nhiễu (Noise-Contrastive Estimation**

- NCE), sẽ được trình bày chi tiết ở Mục 18.6. NCE giúp huấn luyện mô hình sinh xác suất mà không cần tính chuẩn hóa toàn phần — vừa chính xác, vừa hiệu quả, và đã được áp dụng thành công cho các mô hình ngôn ngữ thần kinh (Mnih và Teh, 2012; Mnih và Kavukcuoglu, 2013).

12.7 Kết hợp mô hình ngôn ngữ neural với n-gram

Một ưu điểm lớn của mô hình ngôn ngữ truyền thống dựa trên n-gram so với mạng nơ-ron là khả năng lưu trữ rất nhiều tổ hợp từ (tuples), từ đó đạt được dung lượng mô hình cao. Hơn nữa, việc xử lý một ví dụ mới trong mô hình n-gram thường rất nhanh, vì chỉ cần tra cứu một số lượng nhỏ các tuple có ngữ cảnh phù hợp.

Nếu sử dụng các cấu trúc dữ liệu hiệu quả như bảng băm hoặc cây tìm kiếm, thì thời gian xử lý không phụ thuộc nhiều vào số lượng tuple đã lưu. Nói cách khác, có thể tăng dung lượng của mô hình n-gram lên rất lớn mà không làm tăng đáng kể thời gian xử lý.

Ngược lại, trong các mạng nơ-ron, khi tăng gấp đôi số lượng tham số thì chi phí tính toán thường cũng tăng tương ứng. Tuy nhiên, vẫn có một số ngoại lệ:

- **Lớp embedding:** Chỉ truy xuất một vector tương ứng với từ đang xét, nên có thể mở rộng từ vựng mà không làm chậm mỗi lần xử lý.
- **Một số mô hình như tiled CNN:** Có thể tăng số tham số bằng cách giảm chia sẻ trọng số mà không làm tăng đáng kể chi phí tính toán.

Tuy vậy, trong hầu hết các lớp mạng chuẩn dựa trên nhân ma trận, chi phí tính toán tỷ lệ thuận với số lượng tham số.

Một chiến lược đơn giản để vừa tăng dung lượng mô hình mà không tăng quá nhiều chi phí là kết hợp hai mô hình: một mạng nơ-ron và một mô hình n-gram, thông qua kỹ thuật tổ hợp mô hình (ensemble), như được đề xuất bởi Bengio et al. (2001, 2003).

Cũng giống như các tổ hợp mô hình khác, ý tưởng là giảm lỗi kiểm tra bằng cách kết hợp các mô hình mắc lỗi khác nhau. Có thể đơn giản cộng trung bình các phân phối đầu ra hoặc dùng trọng số tối ưu hóa dựa trên tập xác thực.

Mikolov et al. (2011a) đã mở rộng phương pháp này bằng cách kết hợp nhiều loại mô hình ngôn ngữ khác nhau, không chỉ dừng lại ở hai mô hình.

Một hướng tiếp cận khác là kết hợp mạng nơ-ron với mô hình entropy cực đại, bằng cách huấn luyện cả hai đồng thời (Mikolov et al., 2011b). Cách này tương đương với việc thêm một tập đặc trưng rời rạc trực tiếp vào đầu ra.

Các đặc trưng này là chỉ báo (indicator features) cho sự có mặt của một n-gram cụ thể trong ngữ cảnh đầu vào. Chúng có tính chất rất thưa (sparse) và chiều cao (high-dimensional), với số đặc trưng tiềm năng có thể lên tới $|\mathcal{V}|^n$ nếu từ vựng là $\mathcal{V}$. Tuy nhiên, vì mỗi ví dụ chỉ kích hoạt một số nhỏ đặc trưng, nên chi phí tính toán tăng thêm là rất nhỏ.

12.8 Dịch máy bằng mạng nơ-ron (Neural Machine Translation)

Dịch máy (machine translation) là bài toán chuyển một câu từ ngôn ngữ nguồn sang ngôn ngữ đích sao cho vẫn giữ nguyên ý nghĩa. Các hệ thống dịch máy hiện đại thường được chia thành nhiều thành phần.

Một thành phần đầu tiên đưa ra các bản dịch ứng viên. Tuy nhiên, do khác biệt cú pháp giữa các ngôn ngữ, nhiều bản dịch có thể không đúng ngữ pháp. Ví dụ, thay vì "red apple", hệ thống có thể đề xuất "apple red".

Để đánh giá các bản dịch này, một **mô hình ngôn ngữ** được dùng để chấm điểm và chọn bản dịch có khả năng đúng hơn. Ví dụ, mô hình sẽ xác định rằng "red apple" hợp ngữ pháp tiếng Anh hơn "apple red".

Ngay từ rất sớm, các nghiên cứu về mạng nơ-ron đã đưa ra ý tưởng encoder và decoder (Allen, 1987; Chrisman, 1991). Tuy nhiên, ứng dụng lớn đầu tiên của mạng nơ-ron trong dịch máy là thay thế mô hình n-gram bằng mạng MLP để mô hình hóa xác suất trong ngôn ngữ đích (Schwenk, 2010).

Thay vì chỉ mô hình hóa xác suất $P(t_1, \ldots, t_k)$ cho câu đích, mục tiêu được mở rộng thành mô hình hóa xác suất có điều kiện:

$$P(t_1, \ldots, t_k \mid s_1, \ldots, s_n)$$

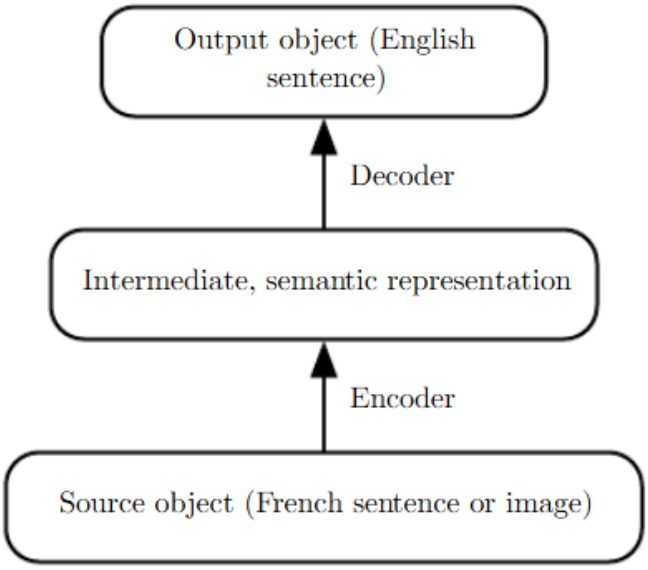
trong đó s_i là từ trong câu nguồn và t_j là từ trong câu đích. Điều này cho phép mô hình tạo ra bản dịch phù hợp hơn với ngữ cảnh của câu nguồn.

Tuy nhiên, mạng MLP yêu cầu độ dài chuỗi cố định, không phù hợp với thực tế. Vì thế, người ta chuyển sang dùng mạng hồi tiếp (RNN), vốn có thể xử lý chuỗi với độ dài linh hoạt.

Kiến trúc chính được sử dụng là **encoder-decoder**:

- **Encoder:** thường là RNN hoặc CNN, đọc toàn bộ câu nguồn và sinh ra một vector ngữ cảnh C.
- **Decoder:** một RNN khác, sử dụng C để sinh ra chuỗi đầu ra trong ngôn ngữ đích.

Sơ đồ tổng quát được trình bày trong Hình 12.5.



Hình 12.5: Kiến trúc *encoder-decoder*: encoder ánh xạ đầu vào (câu tiếng Pháp) sang một vector ngữ nghĩa, decoder chuyển vector đó thành đầu ra (câu tiếng Anh). Ý tưởng này cũng áp dụng cho các tác vụ chuyển đổi giữa các loại dữ liệu khác như ảnh $\rightarrow$ chú thích.

Từ góc độ học biểu diễn, mục tiêu là xây dựng một không gian đặc trưng nơi các câu mang cùng ý nghĩa (dù viết bằng các ngôn ngữ khác nhau) sẽ có vector gần nhau.

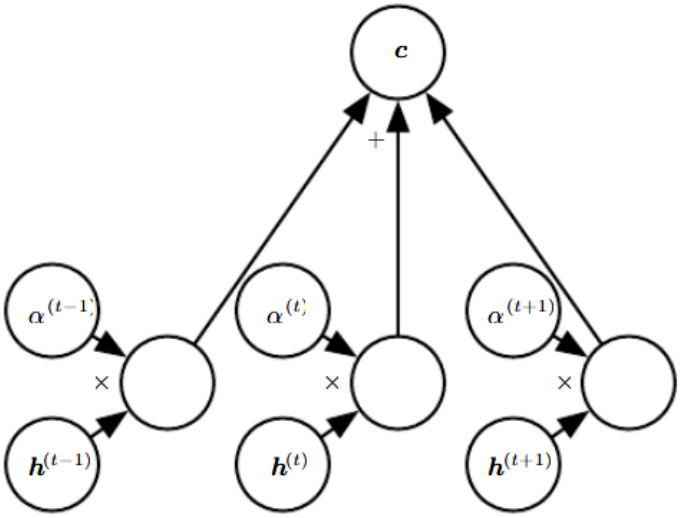
Kalchbrenner và Blunsom (2013) đề xuất mô hình kết hợp CNN và RNN cho dịch máy. Các công trình sau như Cho et al. (2014a) và Sutskever et al. (2014) đã phát triển mô hình encoder-decoder bằng RNN để không chỉ đánh giá mà còn trực tiếp sinh bản dịch.

Jean et al. (2014) mở rộng mô hình để áp dụng cho từ vựng rất lớn, sử dụng các kỹ thuật giảm chi phí tính toán tại tầng softmax.

12.8.0.1 Sử dụng cơ chế attention và căn chỉnh dữ liệu

Việc ép một câu dài (ví dụ 60 từ) vào một vector cố định để truyền tải toàn bộ ý nghĩa là một thách thức lớn. Dù về lý thuyết có thể làm được với các mạng RNN đủ lớn (Cho et al., 2014a; Sutskever et al., 2014), nhưng trong thực tế, cách này không hiệu quả.

Một giải pháp hiệu quả hơn được đề xuất bởi **Bahdanau et al. (2015)**. Ý tưởng là sau khi encoder đọc xong toàn bộ câu nguồn, decoder không sử dụng một vector ngữ cảnh cố định mà sẽ **chọn lọc thông tin từ các phần khác nhau của câu nguồn** mỗi khi sinh ra một từ mới trong câu đích. Cách tiếp cận này gọi là **cơ chế attention**.



Hình 12.6: Cơ chế *attention* được giới thiệu bởi Bahdanau et al. (2015), về bản chất là một phép trung bình có trọng số. Một vector ngữ cảnh $\mathbf{c}$ được tạo ra bằng cách lấy trung bình có trọng số của các vector đặc trưng $\mathbf{h}(t)$ với trọng số $\alpha(t)$. Các trọng số này được sinh ra bởi mô hình, thường bằng cách đưa các điểm liên quan qua hàm *softmax*. Phép trung bình có trọng số là một xấp xỉ trơn và khả vi của việc "truy xuất"thông tin cụ thể — giúp mô hình có thể huấn luyện hiệu quả bằng lan truyền ngược.

Quy trình hoạt động của mô hình attention gồm ba phần:

1. **Trình đọc (Encoder):** Ánh xạ mỗi từ trong câu nguồn thành một vector đặc trưng, giữ nguyên thứ tự xuất hiện.
2. **Bộ nhớ (Memory):** Danh sách các vector đặc trưng này tạo thành một bộ nhớ, giống như một chuỗi các sự kiện ngữ nghĩa.
3. **Trình sinh (Decoder):** Tại mỗi bước dịch, decoder có thể "chú ý"vào một vài vector đặc trưng, truy xuất thông tin từ bộ nhớ thông qua cơ chế attention để sinh ra từ tiếp theo.

Khi các từ trong câu nguồn và câu đích có liên hệ rõ ràng (được căn chỉnh), mô hình có thể học cách ánh xạ giữa các từ tương ứng, kể cả ở hai ngôn ngữ khác nhau. Ví dụ, Kočiský et al. (2014) chỉ ra rằng có thể học một *ma trận dịch chuyển tuyến tính* giữa các embedding từ của hai ngôn ngữ — và việc đó cho kết quả căn chỉnh chính xác hơn so với các phương pháp thống kê truyền thống.

Những nghiên cứu trước đó, như của Klementiev et al. (2012), cũng đã xây dựng các vector từ đa ngôn ngữ (cross-lingual embeddings), giúp ánh xạ từ ngôn ngữ này sang ngôn ngữ khác trong cùng một không gian véc-tơ. Gouws et al. (2014) sau đó còn mở rộng cách tiếp cận này để huấn luyện trên tập dữ liệu lớn hơn, hiệu quả hơn.

12.8.1 Góc nhìn lịch sử

Khái niệm **biểu diễn phân tán** (distributed representation) cho các ký hiệu được giới thiệu lần đầu bởi **Rumelhart et al. (1986)** trong một nghiên cứu về lan truyền ngược. Trong nghiên cứu này, các đối tượng như thành viên trong gia đình được ánh xạ sang vector và mạng học cách mô tả quan hệ như (Colin, Mother, Victoria). Lớp đầu tiên học cách biểu diễn Colin thông qua các đặc trưng như vị trí trong cây gia phả, nhánh, thế hệ,...

Sau đó, ý tưởng embedding được mở rộng cho từ vựng bởi **Deerwester et al. (1990)** thông qua kỹ thuật phân rã SVD. Sau này, việc học embedding từ được thực hiện trực tiếp bằng mạng nơ-ron thay vì dùng các phương pháp thống kê tuyến tính.

Lịch sử của NLP cũng chứng kiến sự chuyển đổi trong cách biểu diễn đầu vào cho mô hình:

- Ban đầu, đầu vào là chuỗi ký tự (Miikkulainen và Dyer, 1991; Schmidhuber và Heil, 1996).
- Sau đó, Bengio et al. (2001) đưa ra mô hình ngôn ngữ neural đầu tiên dựa trên từ vựng, học được embedding từ diễn giải được.

Ngày nay, các mô hình hiện đại có thể biểu diễn hàng triệu từ, bao gồm cả tên riêng, lỗi chính tả,... Những khả năng này mở ra nhiều hướng phát triển mới cho NLP. Các kỹ thuật nền tảng như giảm chi phí tại tầng softmax đã được trình bày trong mục 12.4.3.

Mô hình ngôn ngữ neural cũng được áp dụng rộng rãi trong các tác vụ khác của NLP:

- Phân tích cú pháp (parsing) (Henderson, 2003; 2004),
- Gán nhãn từ loại (POS tagging),
- Gán vai nghĩa (semantic role labeling),
- Tách cụm từ (chunking).

Collobert và Weston (2008) đề xuất kiến trúc học đa nhiệm (multitask learning), trong đó các tác vụ chia sẻ cùng một embedding từ.

Việc trực quan hóa embedding cũng đóng vai trò quan trọng trong việc phát triển và hiểu mô hình. Sau khi thuật toán t-SNE được đề xuất (van der Maaten và Hinton, 2008), việc trực quan hóa các embedding từ trở nên dễ dàng và phổ biến hơn. Joseph Turian là một trong những người đầu tiên sử dụng t-SNE để minh họa trực quan embedding từ vào năm 2009, từ đó thúc đẩy sự quan tâm lớn hơn đến việc phân tích embedding trong NLP.

12.9 Các ứng dụng khác

Trong phần này, chúng tôi giới thiệu một số ứng dụng khác của học sâu, không nằm trong các nhóm nhiệm vụ tiêu chuẩn như nhận dạng đối tượng, nhận dạng giọng nói hay xử lý ngôn ngữ tự nhiên đã trình bày ở các mục trước. Các ứng dụng này mở rộng thêm tầm ảnh hưởng của học sâu, và sẽ được đề cập sâu hơn trong Phần III của cuốn sách với các nhiệm vụ vẫn còn là chủ đề nghiên cứu.

12.9.1 Hệ thống gợi ý (Recommender systems)

Một trong những ứng dụng phổ biến nhất của học máy trong ngành công nghệ là khả năng đề xuất các mục tiêu như sản phẩm, bài viết hay quảng cáo đến người dùng. Hai dạng ứng dụng chính là:

- **Quảng cáo trực tuyến (Online Advertising)**
- **Gợi ý sản phẩm (Item Recommendation)** — ví dụ như phim, bài hát, hay bài viết.

Cả hai đều xoay quanh việc dự đoán mối liên hệ giữa người dùng và đối tượng: chẳng hạn xác suất nhấp vào quảng cáo, khả năng mua hàng, hoặc mức độ quan tâm.

Ban đầu, các hệ thống này chỉ dùng thông tin đơn giản như ID người dùng và ID sản phẩm. Một phương pháp hiệu quả là **lọc cộng tác (collaborative filtering)**, giả định rằng những người dùng có hành vi giống nhau sẽ có sở thích tương tự.

Một dạng mô hình gợi ý điển hình có dạng:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}$$

Trong đó:

- A: embedding biểu diễn người dùng,
- B: embedding biểu diễn sản phẩm,
- b_u: bias riêng của người dùng u,
- c_i: bias riêng của sản phẩm i.

Mục tiêu của bài toán là tối thiểu hóa sai số giữa giá trị dự đoán $\hat{R}_{u,i}$ và giá trị thực tế $R_{u,i}$. Việc học các embedding có thể thực hiện qua phân rã ma trận (SVD) hoặc tối ưu gradient.

Học sâu mở ra nhiều hướng phát triển mạnh mẽ trong hệ thống gợi ý, chẳng hạn:

- **Restricted Boltzmann Machines (RBMs):** từng được dùng trong hệ thống đạt giải Netflix Prize.
- **Mạng tích chập (CNN):** dùng để học embedding từ nội dung gốc như audio (van den Oord et al., 2013).
- **Mô hình deep multitask:** học biểu diễn chung cho người dùng và sản phẩm từ nhiều nguồn dữ liệu (Huang et al., 2013; Elkahky et al., 2015).

Một bài toán phổ biến trong hệ thống gợi ý là **cold-start** — khi gặp người dùng mới hoặc sản phẩm mới, chưa có dữ liệu lịch sử. Cách khắc phục bao gồm:

- Sử dụng thông tin hồ sơ người dùng (như độ tuổi, vị trí, sở thích),
- Dùng đặc trưng nội dung sản phẩm (ví dụ mô tả văn bản),

- Học biểu diễn từ những đặc trưng này bằng học sâu.

12.9.1.1 Khám phá và khai thác (Exploration vs. Exploitation)

Một vấn đề quan trọng trong hệ thống gợi ý vượt ra ngoài phạm vi học có giám sát truyền thống: đó là sự đánh đổi giữa **khám phá (exploration)** và **khai thác (exploitation)**. Vấn đề này thuộc về lĩnh vực **học tăng cường (reinforcement learning)**, đặc biệt là biến thể gọi là **contextual bandits**.

Thiên lệch dữ liệu: Giả sử hệ thống hiển thị sản phẩm A cho người dùng và nhận được phản hồi, nó không biết điều gì sẽ xảy ra nếu hiển thị sản phẩm B. Với các mục tiêu không được chọn (ví dụ không thắng trong phiên đấu giá quảng cáo), hệ thống không nhận được bất kỳ phản hồi nào. Điều này tạo ra sự thiên lệch trong dữ liệu quan sát được.

Contextual Bandit là gì? Đây là một mô hình đơn giản hóa của học tăng cường: tại mỗi thời điểm, mô hình chỉ chọn một hành động (ví dụ: đề xuất một sản phẩm) dựa trên ngữ cảnh (ví dụ: thông tin người dùng), và chỉ nhận được một phần thưởng cho hành động đó.

Khám phá vs. Khai thác:

- **Khai thác (Exploitation):** Chọn hành động mà mô hình tin là sẽ đem lại phần thưởng cao nhất dựa trên dữ liệu đã học.
- **Khám phá (Exploration):** Cố tình thử các hành động khác để thu thập thêm dữ liệu — có thể dẫn đến phần thưởng cao hơn trong tương lai.

Ví dụ: Nếu sản phẩm A đã từng cho phần thưởng = 1, và sản phẩm B thì chưa có dữ liệu, ta vẫn nên thử B để khám phá khả năng có thể đạt phần thưởng cao hơn.

Chiến lược khám phá:

- Chọn hành động ngẫu nhiên với xác suất nhỏ (*epsilon-greedy*),
- Ưu tiên các hành động có độ bất định cao trong ước lượng phần thưởng.

Ảnh hưởng của thời gian: Nếu hệ thống chỉ hoạt động trong thời gian ngắn, thì nên ưu tiên khai thác. Nếu hoạt động lâu dài, cần khám phá mạnh ở giai đoạn đầu để thu thập dữ liệu, sau đó khai thác kết quả học được.

Đánh giá mô hình khó khăn: Không giống như học có giám sát, trong học tăng cường, chính sách của mô hình ảnh hưởng đến dữ liệu nó thu thập — khiến việc đánh giá khách quan trở nên khó. Các kỹ thuật như của **Dudik et al. (2011)** được sử dụng để đánh giá hiệu suất chính sách trong bối cảnh contextual bandits.

12.9.2 Đại diện tri thức, suy luận và trả lời câu hỏi

Học sâu đã đạt được nhiều thành công ấn tượng trong các lĩnh vực như mô hình ngôn ngữ, dịch máy và xử lý ngôn ngữ tự nhiên. Một phần quan trọng tạo nên thành công này là việc sử dụng *embedding* — tức là biểu diễn các từ và ký hiệu dưới dạng vector trong không gian liên tục. Những nền tảng ban đầu được đặt ra bởi các công trình kinh điển của

Rumelhart et al. (1986a), Deerwester et al. (1990), và Bengio et al. (2001).

Embedding không chỉ giúp mô hình hiểu rõ ý nghĩa của từng từ mà còn cho phép mô hình học được mối liên hệ giữa các khái niệm trong ngôn ngữ. Tuy nhiên, một thách thức hiện tại là làm sao mở rộng khả năng biểu diễn này để áp dụng cho cụm từ, các mối quan hệ, và cả những tri thức cụ thể (facts) trong thế giới.

Một số công cụ tìm kiếm hiện đại đã bắt đầu sử dụng học máy để xây dựng những biểu diễn như vậy. Tuy nhiên, lĩnh vực này vẫn còn rất nhiều câu hỏi mở — đặc biệt trong việc cải thiện khả năng suy luận, hiểu ngữ nghĩa sâu và trả lời câu hỏi tự động.

12.9.2.1 Kiến thức, quan hệ và trả lời câu hỏi

Một hướng nghiên cứu quan trọng là xây dựng các biểu diễn phân tán (distributed representations) có thể học được **mối quan hệ** giữa hai thực thể. Các mối quan hệ này giúp mô hình nắm bắt tri thức về các đối tượng và cách chúng tương tác với nhau.

Về mặt toán học, quan hệ nhị phân có thể hiểu đơn giản là một tập các cặp (a, b). Nếu (a, b) nằm trong tập, thì có quan hệ giữa a và b. Ví dụ:

$$S = \{(1, 2), (1, 3), (2, 3)\}$$

Ở đây, ta có thể nói $1 < 2$ vì $(1, 2) \in S$, nhưng không thể nói $2 < 1$ vì $(2, 1) \notin S$.

Thay vì chỉ dùng số, ta có thể dùng các khái niệm ngữ nghĩa như:

$$(\texttt{dog}, \texttt{is\_a\_type\_of}, \texttt{mammal})$$

Các quan hệ này thường được biểu diễn dưới dạng bộ ba:

$$(\text{subject}, \text{relation}, \text{object}) \tag{12.9}$$

hay viết chung là:

$$(\text{entity}_i, \text{relation}_j, \text{entity}_k) \tag{12.10}$$

Ngoài ra, còn có **thuộc tính (attribute)**, là các thông tin đơn lẻ gắn với một thực thể:

$$(\text{entity}_i, \text{attribute}_j) \tag{12.11}$$

Ví dụ: `(dog, has_fur)`.

Tri thức này có thể đến từ văn bản chưa có cấu trúc hoặc từ các **cơ sở dữ liệu quan hệ (relational database)**. Khi dữ liệu được thiết kế để truyền đạt kiến thức cho hệ thống AI, ta gọi đó là **cơ sở tri thức (knowledge base)** — ví dụ như `Freebase`, `WordNet`, hoặc

`Wikidata`.

Biểu diễn cho thực thể và quan hệ có thể được học bằng cách xem từng bộ ba trong cơ sở tri thức như một mẫu huấn luyện. Một số nghiên cứu như **bordes2013translating** đề xuất tối ưu hàm mục tiêu biểu diễn phân phối chung giữa thực thể và quan hệ.

Một số hướng tiếp cận:

- **Linear relational embeddings paccanaro2000learning**: biểu diễn mỗi thực thể bằng một vector, mỗi quan hệ bằng một ma trận tuyến tính.
- **Joint embeddings bordes2012joint**: quan hệ cũng được biểu diễn bằng vector như thực thể, cho phép mô hình hóa quan hệ phức tạp và linh hoạt hơn.

Một ứng dụng tiêu biểu là **dự đoán liên kết (link prediction)** — tức là suy ra các mối quan hệ còn thiếu trong đồ thị tri thức. Đây là một dạng suy luận để bổ sung kiến thức chưa có trong cơ sở dữ liệu.

Vì dữ liệu huấn luyện thường chỉ chứa các quan hệ đúng (positive examples), một kỹ thuật phổ biến là tạo thêm các **quan hệ giả (negative samples)** bằng cách thay thế một thực thể ngẫu nhiên. Ví dụ:

$$(\texttt{Paris}, \texttt{isCapitalOf}, \texttt{France}) \rightarrow (\texttt{Paris}, \texttt{isCapitalOf}, \texttt{Japan})$$

Chỉ số đánh giá thường dùng là **precision@10%**, kiểm tra xem các quan hệ đúng có được xếp hạng cao hơn các quan hệ sai hay không.

Một ứng dụng khác là **phân biệt nghĩa của từ (word-sense disambiguation)**: mô hình phải xác định nghĩa phù hợp của từ tùy theo ngữ cảnh.

Trong tương lai, kết hợp tri thức với khả năng suy luận có thể giúp xây dựng hệ thống **trả lời câu hỏi tổng quát (general-purpose question answering)** — nơi mô hình không chỉ ghi nhớ dữ liệu mà còn phải suy luận để tìm ra câu trả lời.

Hiện tại, các mô hình như **Memory Networks weston2014memory** hoặc các phiên bản mở rộng sử dụng GRU như Kumar, Irsoy, Su, Bradbury, English, Pierce, Ondruska, Iyyer, Gulrajani **and** Socher [29] đang được dùng để thử nghiệm các hệ thống như vậy trong các môi trường đơn giản.

Tổng kết

Học sâu đang ngày càng được mở rộng sang nhiều lĩnh vực ứng dụng khác nhau. Những gì trình bày trong chương này chỉ là một lát cắt đại diện. Từ nhận dạng hình ảnh, giọng nói, đến xử lý ngôn ngữ, hệ thống gợi ý, và suy luận tri thức — học sâu đã chứng minh khả năng vượt trội khi có đủ dữ liệu huấn luyện và công cụ tính toán mạnh.

Phần II đến đây kết thúc. Phần này đã trình bày hầu hết các phương pháp thực tiễn hiện đại trong học sâu, với điểm chung là tối ưu các hàm mục tiêu bằng gradient descent để

học xấp xỉ một hàm mục tiêu nào đó. Cách tiếp cận này tỏ ra rất mạnh khi có đủ dữ liệu.

Tài liệu tham khảo

[1] B. E. Boser, I. M. Guyon **and** V. N. Vapnik, "A training algorithm for optimal margin classifiers," **in***COLT '92: Proceedings of the fifth annual workshop on Computational learning theory* ACM, 1992, **pages** 144–152.

[2] C. Cortes **and** V. Vapnik, "Support vector networks," *Machine Learning*, **jourvol** 20, **pages** 273–297, 1995.

[3] L. Breiman, J. H. Friedman, R. A. Olshen **and** C. J. Stone, *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group, 1984.

[4] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] T. Hastie, R. Tibshirani **and** J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference and Prediction* (Springer Series in Statistics). Springer Verlag, 2001.

[6] Y. Bengio **and** M. Monperrus, "Non-local manifold tangent learning," **in***Advances in Neural Information Processing Systems 17 (NIPS'04)* L. Saul, Y. Weiss **and** L. Bottou, **editors**, MIT Press, 2005, **pages** 129–136.

[7] Y. Bengio, H. Larochelle **and** P. Vincent, "Non-local manifold Parzen windows," **in***NIPS'2005* MIT Press, 2006.

[8] C. M. Bishop, "Regularization and complexity control in feed-forward networks," **in***Proceedings International Conference on Artificial Neural Networks ICANN'95* **volume** 1, 1995, **pages** 141–148.

[9] C. M. Bishop, "Training with noise is equivalent to Tikhonov regularization," *Neural Computation*, **jourvol** 7, **number** 1, **pages** 108–116, 1995.

[10] A. Graves, "Practical variational inference for neural networks," **in***NIPS'2011* 2011.

[11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens **and** Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *arXiv e-prints*, 2015.

[12] M. Belkin **and** P. Niyogi, "Laplacian eigenmaps and spectral techniques for embedding and clustering," **in***Advances in Neural Information Processing Systems 14 (NIPS'01)* T. Dietterich, S. Becker **and** Z. Ghahramani, **editors**, MIT Press, 2002.

[13] O. Chapelle, J. Weston **and** B. Schölkopf, "Cluster kernels for semi-supervised learning," **in***Advances in Neural Information Processing Systems 15 (NIPS'02)* MIT Press, 2003, **pages** 585–592.

[14] O. Chapelle, B. Schölkopf **and** e. Zien A., *Semi-Supervised Learning*. Cambridge, MA: MIT Press, 2006.

[15] D. Warde-Farley, I. J. Goodfellow, A. Courville **and** Y. Bengio, "An empirical analysis of dropout in piecewise linear networks," **in***ICLR'2014* 2014, **pages** 259, 263, 264.

[16] P. Smolensky, "Information processing in dynamical systems: Foundations of harmony theory," **in***Parallel Distributed Processing* D. E. Rumelhart **and** J. L. McClelland, **editors**, **volume** 1, MIT Press, 1986, **chapter** 6, **pages** 194–281.

[17] A. Waibel, T. Hanazawa, G. E. Hinton, K. Shikano **and** K. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **jourvol** 37, **pages** 328–339, 1989.

[18] K. J. Lang, A. H. Waibel **and** G. E. Hinton, "A Time-Delay Neural Network Architecture for Isolated Word Recognition," *Neural Networks*, **jourvol** 3, **number** 1, **pages** 23–43, 1990.

[19] K. J. Lang **and** G. E. Hinton, "The Development of the Time-Delay Neural Network Architecture for Speech Recognition," Carnegie-Mellon University, Technical Report CMU-CS-88-152, 1988.

[20] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks* (Studies in Computational Intelligence). Springer, 2012.

[21] L. R. Bahl, P. Brown, P. V. de Souza **and** R. L. Mercer, "Speech recognition with continuous-parameter hidden Markov models," *Computer, Speech and Language*, **jourvol** 2, **pages** 219–234, 1987.

[22] H. Bourlard **and** C. Wellekens, "Speech pattern discrimination and multi-layered perceptrons," *Computer Speech and Language*, **jourvol** 3, **pages** 1–19, 1989.

[23] Y. Bengio, "Artificial Neural Networks and their Application to Sequence Recognition," phdthesis, McGill University, Computer Science, Montreal, Canada, 1991.

[24] Y. Bengio, R. De Mori, G. Flammia **and** R. Kompe, "Neural network-Gaussian mixture hybrid for speech recognition or density estimation," **in***NIPS 4* Morgan Kaufmann, 1992, **pages** 175–182.

[25] A. Mohamed, G. Dahl **and** G. Hinton, "Acoustic modeling using deep belief networks," *IEEE Trans. on Audio, Speech and Language Processing*, **jourvol** 20, **number** 1, **pages** 14–22, 2012.

[26] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean **and** G. E. Hinton, "On rectified linear units for speech processing," **in***ICASSP 2013* 2013, **page** 454.

[27] G. E. Dahl, T. N. Sainath **and** G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," **in***ICASSP'2013* 2013.

[28] L. Deng **and** D. Yu, "Deep learning – methods and applications," *Foundations and Trends in Signal Processing*, 2014.

[29] A. Kumar, O. Irsoy, J. Su, J. Bradbury, R. English, B. Pierce, P. Ondruska, M. Iyyer, I. Gulrajani **and** R. Socher, *Ask Me Anything: Dynamic Memory Networks for Natural Language Processing*, 2015. arXiv: 1506.07285 [cs.CL].