

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN 1



BÁO CÁO BÀI TẬP LẦN 2

Nội dung báo cáo: Chương 1-7 – Quyển 2 (Deep Learning)

Học phần:

Giảng viên hướng dẫn:

Họ và tên:

Mã sinh viên:

Lớp hành chính:

Thực tập cơ sở

PGS.TS Trần Đình Quế

Đặng Quốc Khánh

B22DCCN444

D22CQCN12-B

HÀ NỘI – THÁNG 03 NĂM 2025

CHƯƠNG 1: HỌC SÂU LÀ GÌ?	6
1.1 Trí tuệ nhân tạo, học máy và học sâu	6
1.1.1 Trí tuệ nhân tạo	6
1.1.2 Học máy	7
1.1.3 Học các quy tắc và biểu diễn từ dữ liệu	7
1.1.4 Khái niệm “sâu” trong “học sâu”	10
1.1.5 Hiểu cách hoạt động của học sâu qua hình minh họa	11
1.2 Lịch sử ngắn gọn của học máy	15
1.2.1 Mô hình xác suất	15
1.2.2 Mạng nơ-ron ban đầu	15
1.2.3 Phương pháp nhân	15
1.2.4 Cây quyết định (decision tree), rừng ngẫu nhiên (random forest) và máy tang cườ gradient	16
1.2.5 Quay lại với mạng nơ-ron	17
1.2.6 Điều làm học sâu trở nên khác biệt	18
1.3 Tại sao là học sâu, tại thời điểm này?	20
1.3.1 Phần cứng (Hardware)	21
1.3.2 Dữ liệu (Data)	22
1.3.3 Thuật toán (Algorithms)	22
1.3.4 Một vài yếu tố khác	23
CHƯƠNG 2: CÁC KHỐI XÂY DỰNG TOÁN HỌC CỦA MẠNG NƠ-RO	24
2.0 Tổng quan	24
2.1 Một chút về mạng nơ-ron	24
2.2 Cách biểu diễn dữ liệu trong mạng nơ-ron	28
2.2.1 Tensor bậc n	29
2.2.2 Các thuộc tính khóa của tensor	29
2.2.3 Tensor Slicing	30
2.2.4 Các lô dữ liệu	31
2.2.5 Một số ví dụ đời thực của tensor	31

2.2.6 Dữ liệu vector	31
2.2.7 Dữ liệu chuỗi thời gian hoặc dữ liệu tuần tự.....	31
2.2.8 Dữ liệu hình ảnh.....	32
2.2.9 Dữ liệu video.....	32
2.3 Các phép toán tensor	33
2.3.1 Các phép toán theo từng phần tử (Element-wise operations)	33
2.3.2 Broadcasting (Phép phát sóng).....	34
2.3.3 Tích tensor (Tensor Product)	36
2.3.4 Phép reshape (định hình lại).....	36
2.3.5 Giải thích hình học của các phép toán tensor	37
2.3.6 Giải thích hình học của học sâu.....	39
2.4 Động cơ của mạng nơ ron: tối ưu hóa dựa trên gradient	40
2.4.1 Sơ qua về đạo hàm.....	41
2.4.2 Đạo hàm của 1 phép toán tensor : gradient	42
2.4.3 Hạ gradient ngẫu nhiên.....	43
2.4.4 Xâu chuỗi các đạo hàm: Thuật toán Backpropagation (thuật toán lan truyền ngược).....	45
CHƯƠNG 3: GIỚI THIỆU VỀ KERAS VÀ TENSORFLOW.....	48
3.1 Giới thiệu về Tensorflow và Keras.....	48
3.1.1 Tensorflow	48
3.1.2 Keras	48
3.1.3 Thiết lập môi trường học sâu.....	48
3.2 Bắt đầu với Tensorflow	49
3.2.1 Khởi tạo tensor và biến	49
3.2.2 Các phép toán tensor	50
3.2.3 Gradient Tape API	51
3.2.4 Ví dụ về phân loại tuyến tính.....	52
3.3 Các API của Keras.....	55
3.3.1 Tầng (Layers): Những khối xây dựng cơ bản của học sâu.....	55
3.3.2 Một số loại tầng.....	57
3.3.3 Một số loại hàm activate	58

3.3.4 Chuyển từ các tầng (Layer) sang mô hình	58
3.3.5 Một số bộ tối ưu hóa	58
3.3.6 Một số hàm mất mát.....	58
3.3.7 Một số chỉ số đo khi huấn luyện mô hình.....	58
3.3.8 Cách chọn hàm mất mát	59
3.3.9 Phương thức fit().....	59
3.3.10 Sử dụng mô hình sau huấn luyện.....	60
CHƯƠNG 4: BẮT ĐẦU VỚI MẠNG NƠON: PHÂN LOẠI VÀ HỒI QUY	61
4.1 Phân loại nhị phân thông qua ví dụ về phân loại đánh giá phim.....	61
4.1.2 Xử lý bài toán với code và giải thích code	62
4.2 Phân loại tin tức: Một ví dụ về phân loại đa lớp	65
4.2.1 Bộ dữ liệu tin tức từ Reuters	65
4.2.2 Học cách làm qua code và giải thích.....	66
4.3 Dự đoán giá nhà: Một ví dụ về hồi quy	68
4.3.1 Bộ dữ liệu giá nhà Boston	68
4.3.2 Cách thực hiện thông qua code và giải thích	69
CHƯƠNG 5: NHỮNG NGUYÊN TẮC CƠ BẢN CỦA HỌC MÁY	72
5.1 Mục tiêu của học máy.....	72
5.1.1 Underfitting và Overfitting.....	72
5.1.2 Bản chất của khái quát hóa trong học sâu	73
5.2 Đánh giá mô hình học máy	76
5.2.1 Tập huấn luyện, tập kiểm định và tập kiểm tra	76
5.2.2 Một số cách phân tập kiểm định	77
5.2.3 Đường cơ sở dựa trên ý thức thông thường.....	77
5.2.4 Những điều cần lưu ý khi đánh giá mô hình.....	78
5.3 Tối ưu hóa độ khớp fit.....	79
5.3.1 Điều chỉnh các tham số quan trọng của gradient descent	80
5.3.2 Tận dụng các kiến trúc tiên nghiệm	81
5.3.3 Tăng dung lượng mô hình (Increasing Model Capacity)	82
5.4 Cải thiện khả năng khái quát hóa	82

5.4.1 Quản lý dữ liệu.....	82
5.4.2 Kỹ thuật đặc trưng	83
5.4.3 Sử dụng dừng sớm	85
5.4.4 Điều chuẩn mô hình.....	85
CHƯƠNG 6: QUY TRÌNH CHUNG CỦA HỌC MÁY	89
6.1 Xác định bài toán và thu thập dữ liệu	89
6.1.1 Xác định bài toán	89
6.1.2 Thu thập dữ liệu.....	89
6.1.3 Hiệu dữ liệu	90
6.1.4 Chọn thước đo khả năng thành công.....	90
6.2 Phát triển mô hình	91
6.2.1 Chuẩn bị dữ liệu.....	91
6.2.2 Chọn phương pháp đánh giá	92
6.2.3 Vượt qua baseline	92
6.2.4 Tăng quy mô: Tạo mô hình quá khớp	93
6.2.5 Điều chỉnh và tối ưu.....	93
6.3 Triển khai mô hình	93
6.3.1 Giải thích và đặt kỳ vọng	93
6.3.2 Triển khai mô hình suy luận.....	94
6.3.3 Giám sát mô hình	94
6.3.4 Duy trì mô hình.....	94

Ghi chú: Phần được in đậm là những ý quan trọng

CHƯƠNG 1: HỌC SÂU LÀ GÌ?

1.1 Trí tuệ nhân tạo, học máy và học sâu

1.1.1 Trí tuệ nhân tạo

Trí tuệ nhân tạo ra đời vào những năm 1950, khi một nhóm nhỏ các nhà tiên phong từ lĩnh vực khoa học máy tính non trẻ bắt đầu tự hỏi liệu máy tính có thể “suy nghĩ” hay không? – một câu hỏi mà chúng ta vẫn đang khám phá cho đến ngày nay.

Trí tuệ nhân tạo cuối cùng đã trở thành một lĩnh vực nghiên cứu vào năm 1956, khi John McCarthy đã tổ chức một hội thảo và đặt ra vấn đề như sau:

“Sẽ có một nỗ lực để tìm ra cách làm cho máy móc sử dụng ngôn ngữ, hình thành các khái niệm trừu tượng, giải quyết các loại vấn đề hiện chỉ dành cho con người, và tự cải thiện chính mình.”

Cuộc hội thảo đã thu hút nhiều người sau này trở thành những nhà tiên phong trong lĩnh vực này, và khởi đầu một cuộc cách mạng trí tuệ vẫn đang tiếp diễn cho đến ngày nay.

Đơn giản hơn, AI có thể được mô tả là **việc tự động hóa các nhiệm vụ trí tuệ thường do con người thực hiện**. Do đó, AI là một lĩnh vực tổng quát bao gồm học máy và học sâu, nhưng cũng bao gồm nhiều cách tiếp cận khác không liên quan đến bất kỳ hình thức học nào. Cho đến những năm 1980, hầu hết các sách giáo khoa về AI không đề cập đến khái niệm “học”! Ví dụ, các chương trình cờ vua ban đầu chỉ bao gồm các quy tắc được lập trình viên lập trình bằng tay và không được xem là học máy. Thực tế, trong một thời gian khá dài, hầu hết các chuyên gia tin rằng trí tuệ nhân tạo có thể đạt được bằng cách để các lập trình viên tự tay xây dựng một bộ quy tắc rõ ràng và đủ lớn để thao tác và bộ quy tắc này được lưu trữ trong các cơ sở dữ liệu rõ ràng. Cách tiếp cận này được gọi là AI biểu tượng (symbolic AI).

Mặc dù AI biểu tượng tỏ ra phù hợp để giải quyết các vấn đề logic được xác định rõ ràng như chơi cờ vua, nhưng nó lại trở nên bất khả thi khi phải tìm ra các quy tắc để giải quyết các vấn đề phức tạp, mơ hồ hơn như phân loại hình ảnh, nhận diện giọng nói hoặc dịch ngôn ngữ tự nhiên. Một cách tiếp cận mới đã xuất hiện để thay thế AI biểu tượng: học máy.

Nói tóm lại, trí tuệ nhân tạo là một lĩnh vực nhằm tạo ra các hệ thống có khả năng mô phỏng trí thông minh của con người. Trí tuệ nhân tạo do con người tạo ra và nó có thể học hỏi, suy luận và đưa ra quyết định trực tiếp thay cho con người.

1.1.2 Học máy

Ban đầu, để máy móc thực hiện một công việc hữu ích, người ta viết ra từng bước hướng dẫn cho một cỗ máy gọi là “cỗ máy phân tích” thực hiện. Họ cho rằng: *“Cỗ máy phân tích không hề có tham vọng khởi tạo bất cứ điều gì. Nó chỉ có thể làm những gì chúng ta biết cách ra lệnh cho nó thực hiện... Vai trò của nó là hỗ trợ chúng ta tận dụng những gì chúng ta đã quen thuộc.”* Điều này có nghĩa là: cách thông thường để khiến máy tính thực hiện công việc hữu ích là yêu cầu lập trình viên viết ra các quy tắc, tạo thành một chương trình máy tính, sau đó máy tính tuân theo nhằm biến dữ liệu đầu vào thành câu trả lời phù hợp

Sau này, có ý kiến cho rằng: “Máy tính về nguyên tắc có thể được chế tạo để mô phỏng mọi khía cạnh của trí thông minh con người.” Đây chính là ý kiến định hình nên AI sau này. Học máy thực hiện như sau: máy tính quan sát dữ liệu đầu vào và các câu trả lời tương ứng, rồi tự tìm ra các quy tắc cần thiết. Một hệ thống học máy sẽ được huấn luyện thay vì được lập trình rõ ràng. Nó được cung cấp nhiều ví dụ liên quan đến một nhiệm vụ, và nó tìm ra cấu trúc thống kê trong các ví dụ này, cuối cùng cho phép hệ thống đưa ra các quy tắc để tự động hóa nhiệm vụ. Ví dụ như việc dựa vào tập dữ liệu thu được nhờ chẩn đoán các bệnh nhân từng đến khám tiểu đường, hệ thống sẽ thu thập dữ liệu, phân tích mối tương quan giữa các chỉ số và kết quả, đưa ra quy tắc để đưa ra dự đoán cụ thể cho bệnh nhân mới xem họ có bị tiểu đường hay không dựa trên các chỉ số đã có.

1.1.3 Học các quy tắc và biểu diễn từ dữ liệu

Để định nghĩa học sâu và hiểu sự khác biệt giữa học sâu với các phương pháp học máy khác, trước tiên chúng ta cần nắm rõ học máy làm gì. Học máy là việc khám phá các quy tắc để thực hiện một nhiệm vụ xử lý dữ liệu, dựa trên các ví dụ và kết quả. Vậy để thực hiện học máy, chúng ta cần ba yếu tố:

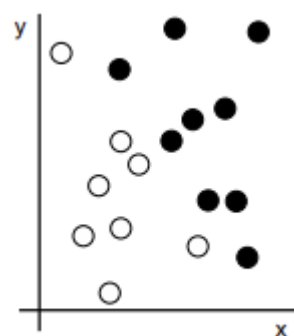
- + **Điểm dữ liệu đầu vào:** Ví dụ, nếu nhiệm vụ là nhận diện giọng nói, các điểm dữ liệu này có thể là các tệp âm thanh. Nếu nhiệm vụ là gắn nhãn ảnh, chúng có thể là các bức ảnh.

- + **Ví dụ về đầu ra mong đợi**—Trong nhiệm vụ nhận diện giọng nói, đó có thể là bản ghi chép được tạo ra từ các tệp âm thanh. Trong nhiệm vụ ảnh, đầu ra mong đợi có thể là các nhãn như “chó”, “mèo”, v.v.
- + **Cách đo lường hiệu quả của thuật toán**—Điều này cần thiết để xác định khoảng cách giữa đầu ra hiện tại của thuật toán và đầu ra mong đợi. Số đo này được phản hồi lại để điều chỉnh cách hoạt động của thuật toán. **Bước điều chỉnh này chính là cái mà chúng ta gọi là học.**

Một mô hình học máy biến đổi dữ liệu đầu vào thành các đầu ra có nghĩa, thông qua quá trình “học” từ việc tiếp xúc với các ví dụ đã biết về đầu vào và đầu ra. Do đó, **vấn đề cốt lõi trong học máy và học sâu là biến đổi dữ liệu một cách có ý nghĩa**: nói cách khác, học các biểu diễn hữu ích của dữ liệu đầu vào—những biểu diễn đưa ta gần hơn đến đầu ra mong đợi.

Vậy biểu diễn là gì? Về cơ bản, đó là một cách khác để nhìn nhận dữ liệu – để biểu diễn hoặc mã hóa dữ liệu. Chẳng hạn, một bức ảnh màu có thể được mã hóa ở định dạng RGB (đỏ-xanh-lam) hoặc HSV (sắc độ-độ bão hòa-giá trị): đây là hai biểu diễn khác nhau của cùng một dữ liệu. Một số nhiệm vụ khó với biểu diễn này có thể trở nên dễ dàng với biểu diễn khác. Ví dụ, nhiệm vụ “chọn tất cả các pixel màu đỏ trong ảnh” đơn giản hơn ở định dạng RGB, trong khi “giảm độ bão hòa của ảnh” lại đơn giản hơn ở định dạng HSV. Các mô hình học máy đều xoay quanh việc tìm kiếm các biểu diễn phù hợp cho dữ liệu đầu vào hay còn gọi là những biến đổi của dữ liệu giúp giải quyết nhiệm vụ dễ dàng hơn.

Hãy làm rõ điều này bằng một ví dụ cụ thể. Giả sử ta có một trục x , một trục y và một số điểm được biểu diễn bằng tọa độ (x, y) trong hệ thống này, như trong hình 1.3. Ta có một vài điểm trắng và một vài điểm đen. Giả định rằng ta muốn phát triển một thuật toán nhận tọa độ (x, y) của một điểm và dự đoán điểm đó có khả năng là đen hay trắng. Trong trường hợp này:

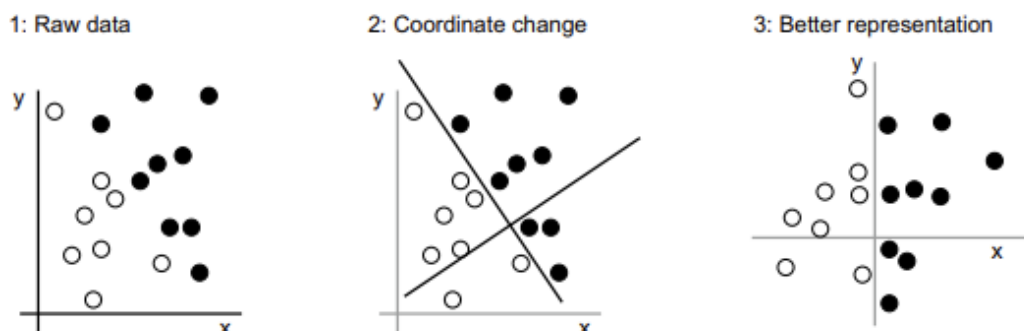


Hình 1.3

- + Đầu vào là tọa độ của các điểm.
- + Đầu ra mong đợi là màu sắc của các điểm.

- + Cách đo lường hiệu quả có thể là tỷ lệ phần trăm các điểm được phân loại đúng.

Điều ta cần là một biểu diễn mới của dữ liệu, giúp tách biệt rõ ràng các điểm trắng và điểm đen. Một biến đổi ta có thể dùng, trong số nhiều khả năng khác, là thay đổi tọa độ, như minh họa trong hình 1.4.



Hình 1.4

Trong hệ tọa độ mới này, tọa độ của các điểm có thể được xem là một biểu diễn mới của dữ liệu. Với biểu diễn này, vấn đề phân loại trắng/đen có thể được diễn đạt bằng một quy tắc đơn giản: “Điểm đen là những điểm có $x > 0$ ” hoặc “Điểm trắng là những điểm có $x < 0$ ”. Biểu diễn mới này đã giúp giải quyết gọn gàng vấn đề phân loại.

Cách giải quyết trên ổn với một vấn đề cực kỳ đơn giản, nhưng với nhiệm vụ như phân loại các chữ số viết tay thì nó còn khả thi hay không? Liệu bạn có thể biến đổi ảnh rõ ràng và thực thi bằng máy tính để làm nổi bật sự khác biệt giữa số 6 và số 8, giữa số 1 và số 7 trên mọi kiểu chữ viết tay không?

Điều này khả thi ở một mức độ nào đó. Các quy tắc dựa trên biểu diễn của chữ số như “số vòng kín” hoặc biểu đồ pixel ngang và dọc có thể làm tốt việc phân biệt các chữ số viết tay. Nhưng việc tìm ra các biểu diễn như vậy bằng tay là rất khó để thực hiện cũng như bảo trì. Mỗi khi gặp một ví dụ viết tay mới không nằm trong những quy tắc cũ, bạn sẽ phải thêm các biến đổi dữ liệu mới và các quy tắc mới, đồng thời cân nhắc sự tương tác của chúng với mọi quy tắc trước đó.

Nếu quá trình này quá khó khăn, liệu ta có thể tự động hóa nó không? Điều gì sẽ xảy ra nếu ta thử tìm kiếm một cách có hệ thống các tập hợp biểu diễn tự động tạo ra từ dữ liệu và các quy tắc dựa trên chúng, rồi xác định cái nào tốt bằng cách dùng phản hồi từ phần trăm chữ số được phân loại đúng trong một tập dữ liệu? Khi đó, ta đang làm học máy. **Học, trong bối cảnh học máy, mô tả một**

quá trình tìm kiếm tự động các biến đổi dữ liệu tạo ra biểu diễn hữu ích của dữ liệu nào đó, được dẫn dắt bởi một tín hiệu phản hồi – những biểu diễn phù hợp với các quy tắc đơn giản hơn để giải quyết nhiệm vụ.

Các biến đổi này có thể là thay đổi tọa độ (như trong ví dụ phân loại tọa độ 2D), hoặc lấy biểu đồ pixel và đếm vòng (như trong ví dụ phân loại chữ số), nhưng cũng có thể là phép chiếu tuyến tính, dịch chuyển, các phép toán phi tuyến (như “chọn tất cả các điểm có $x > 0$ ”),... Các thuật toán học máy thường không sáng tạo trong việc tìm ra các biến đổi này; chúng chỉ tìm kiếm trong một tập hợp các phép toán được xác định trước, gọi là *không gian giả thuyết*. Chẳng hạn, không gian của tất cả các thay đổi tọa độ có thể có sẽ là không gian giả thuyết trong ví dụ phân loại tọa độ 2D.

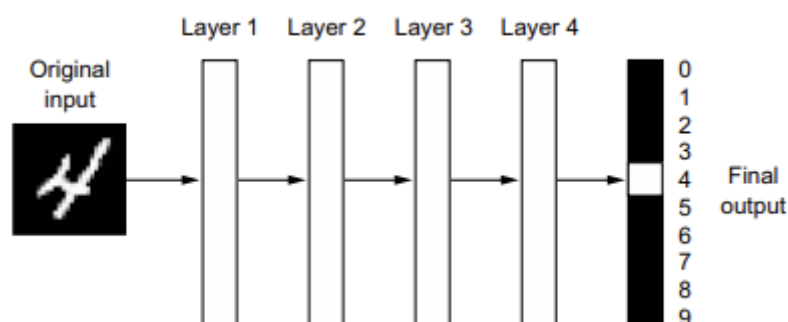
Vậy, học máy, nói một cách ngắn gọn, là: tìm kiếm các biểu diễn và quy tắc hữu ích từ dữ liệu đầu vào, trong một không gian khả năng được xác định trước, sử dụng hướng dẫn từ một tín hiệu phản hồi. Ý tưởng đơn giản này cho phép giải quyết một loạt nhiệm vụ trí tuệ đa dạng đáng kinh ngạc, từ nhận diện giọng nói đến lái xe tự động.

1.1.4 Khái niệm “sâu” trong “học sâu”

Học sâu là một nhánh cụ thể của học máy: **một cách tiếp cận mới để học các biểu diễn từ dữ liệu, nhấn mạnh vào việc học các tầng biểu diễn liên tiếp ngày càng có ý nghĩa hơn.** Chữ “sâu” trong “học sâu” không phải ám chỉ đến bất kỳ sự hiểu biết sâu sắc nào mà phương pháp này đạt được; thay vào đó, **nó đại diện cho ý tưởng về các tầng biểu diễn liên tiếp. Số lượng tầng đóng góp vào một mô hình dữ liệu được gọi là độ sâu của mô hình.** Nó còn có thể được gọi là học biểu diễn theo tầng hoặc học biểu diễn phân cấp. Học sâu hiện đại thường liên quan đến hàng chục hoặc thậm chí hàng trăm tầng biểu diễn liên tiếp, và tất cả chúng đều được học một cách tự động thông qua việc tiếp xúc với dữ liệu huấn luyện. Trong khi đó, các phương pháp học máy khác thường tập trung vào việc học chỉ một hoặc hai tầng biểu diễn của dữ liệu, do đó, chúng đôi khi được gọi là học nông.

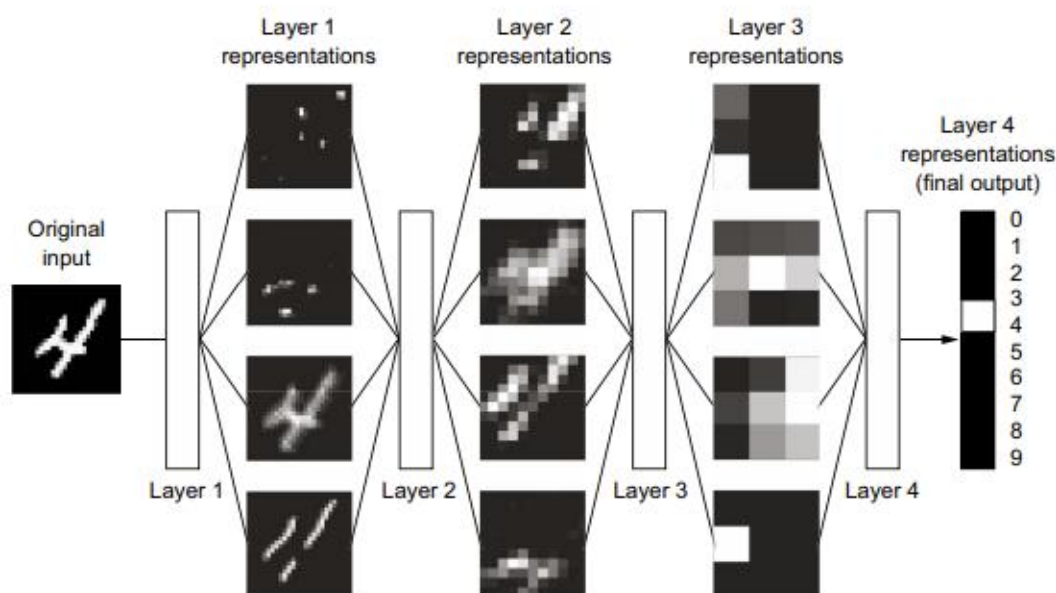
Trong học sâu, các biểu diễn theo tầng này được học thông qua các mô hình gọi là mạng nơ-ron, được cấu trúc theo các tầng thực sự chồng lên nhau. Đối với mục đích của chúng ta, học sâu là một thư viện toán học có sẵn để học các biểu diễn từ dữ liệu.

Các biểu diễn được học bởi một thuật toán học sâu trông như thế nào? Hãy xem xét cách một mạng có nhiều tầng sâu (xem hình 1.5) biến đổi một hình ảnh của một chữ số để nhận ra đó là chữ số nào.



Hình 1.5

Trong hình 1.6, mạng biến đổi hình ảnh chữ số thành các biểu diễn ngày càng khác biệt so với hình ảnh gốc và ngày càng cung cấp thông tin về kết quả cuối cùng. **Mô phỏng như một quá trình chưng cất thông tin nhiều giai đoạn, trong đó thông tin đi qua các bộ lọc liên tiếp và trở nên ngày càng tinh khiết hơn (nghĩa là hữu ích đối với một số nhiệm vụ).**



Hình 1.6

1.1.5 Hiểu cách hoạt động của học sâu qua hình minh họa

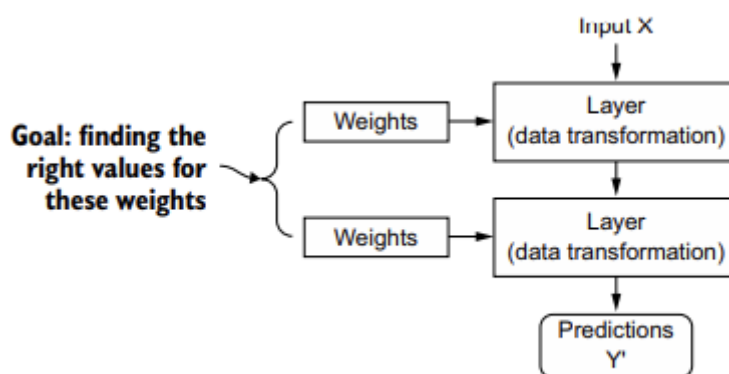
Tại thời điểm này, bạn đã biết rằng học máy là việc ánh xạ các đầu vào (như hình ảnh) sang các mục tiêu (như nhãn “mèo”), được thực hiện bằng cách quan

sát nhiều ví dụ về đầu vào và mục tiêu. Bạn cũng biết rằng mạng nơ-ron sâu thực hiện việc ánh xạ từ đầu vào sang mục tiêu này thông qua một chuỗi các phép biến đổi dữ liệu đơn giản (các tầng), và những phép biến đổi dữ liệu này được học từ việc tiếp xúc với dữ liệu huấn luyện. Bây giờ, hãy xem xét cách quá trình học này diễn ra một cách cụ thể.

1) Trọng số

Mạng nơ-ron sâu là cấu trúc gồm nhiều tầng (layers), mỗi tầng thực hiện một biến đổi dữ liệu. Ví dụ, trong phân loại chữ số, tầng đầu có thể phát hiện cạnh, tầng sau nhận diện hình dạng, và tầng sâu hơn xác định chữ số cụ thể. **Các biến đổi này được định nghĩa bởi trọng số (weights), là tập hợp các số lưu trữ thông tin về cách tầng xử lý dữ liệu đầu vào.**

Quá trình học trong mạng nơ-ron sâu là tìm tập giá trị trọng số cho tất cả các tầng, sao cho mạng ánh xạ đúng đầu vào ví dụ sang mục tiêu liên quan.

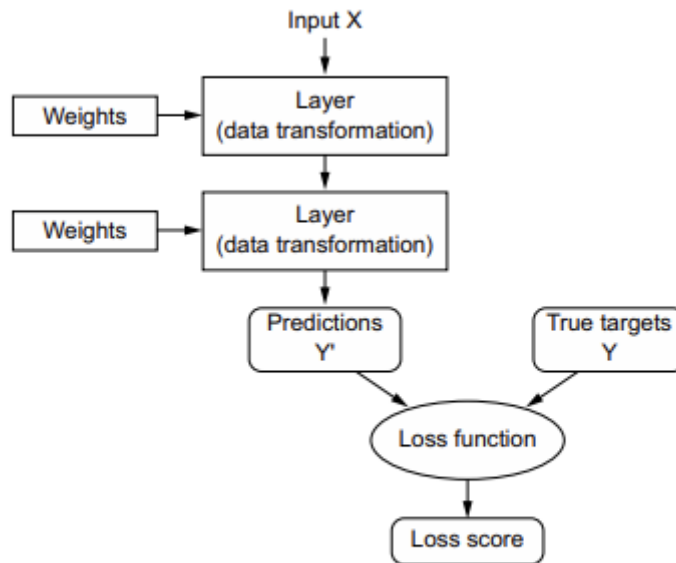


Mô tả hình ảnh: đầu vào X, qua tầng 1 có trọng số W1, sau đó qua tầng 2 có trọng số W2 để ra kết quả Y'

Một mạng nơ-ron sâu có thể chứa hàng chục triệu tham số, làm việc tìm giá trị đúng cho tất cả chúng trở nên phức tạp. Điều này đặc biệt khó vì thay đổi giá trị một trọng số sẽ ảnh hưởng đến hành vi của tất cả các trọng số khác, tạo ra sự phụ thuộc phức tạp.

b) Hàm mất mát

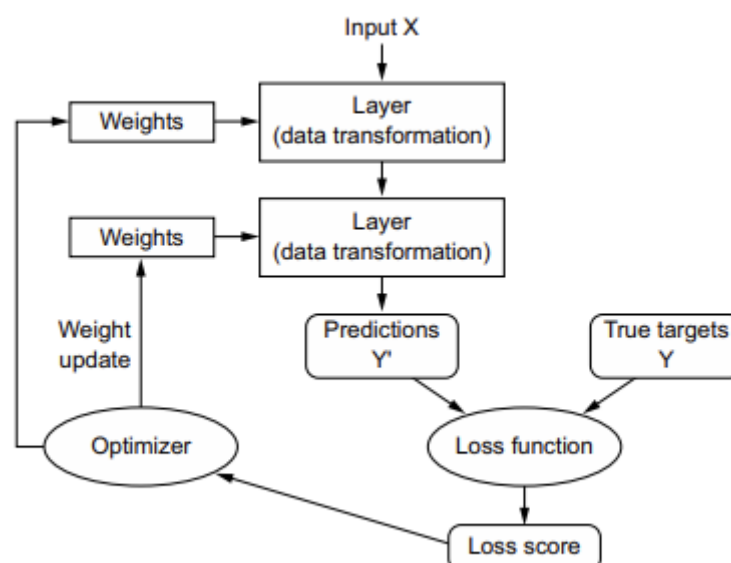
Để kiểm soát một thứ gì đó, trước tiên bạn cần phải quan sát được nó. Để kiểm soát đầu ra của một mạng nơ-ron, bạn cần phải đo lường được đầu ra này cách xa bao nhiêu so với kỳ vọng của bạn. Đây là công việc của **hàm mất mát** của mạng, đôi khi còn được gọi là hàm mục tiêu hoặc hàm chi phí. **Hàm mất mát nhận dự đoán của mạng và mục tiêu đúng, sau đó tính toán điểm số khoảng cách, phản ánh mức độ tốt của mạng trên ví dụ cụ thể.**



Mô tả hình ảnh: sau khi có dự đoán Y' ở trên, lấy kết quả chính xác Y để cùng với Y' đưa vào hàm mất mát, tính ra điểm mất mát.

c) Điều chỉnh trọng số nhờ thuật toán Backpropagation

Thủ thuật cơ bản trong học sâu là **dùng điểm số mất mát làm tín hiệu phản hồi để điều chỉnh trọng số, theo hướng giảm điểm số mất mát**. Việc này do bộ tối ưu hóa thực hiện, sử dụng **thuật toán backpropagation**, là thuật toán trung tâm trong học sâu. Backpropagation điều chỉnh trọng số từng chút một, dựa trên đạo hàm của hàm mất mát, để cải thiện dự đoán.



Mô tả hình ảnh: sau khi tính được điểm mất mát, đưa đến bộ tối ưu hóa để cập nhật lại trọng số theo hướng làm giảm điểm mất mát.

d) Vòng lặp huấn luyện và kết quả

Ban đầu, các trọng số của mạng được gán giá trị ngẫu nhiên, nên mạng chỉ thực hiện một loạt các phép biến đổi ngẫu nhiên, do đó điểm số mất mát tương ứng rất cao. Nhưng **với mỗi ví dụ được xử lý, các trọng số được điều chỉnh một chút theo hướng đúng**, và điểm số mất mát giảm xuống. Đây là **vòng lặp huấn luyện**. Thông thường, **vòng lặp này lặp lại hàng chục lần trên hàng nghìn ví dụ, cuối cùng tìm ra giá trị trọng số giảm thiểu hàm mất mát. Kết quả là mạng có mất mát tối thiểu, với đầu ra gần nhất có thể với mục tiêu, trở thành mạng đã huấn luyện.**

1.1.6 Thành tựu của học sâu

Học sâu đã mang lại những kết quả đáng kinh ngạc trong các nhiệm vụ tri giác và thậm chí cả các nhiệm vụ xử lý ngôn ngữ tự nhiên—những vấn đề liên quan đến các kỹ năng dường như tự nhiên và trực giác đối với con người nhưng từ lâu đã là thách thức lớn đối với máy móc.

Cụ thể, học sâu đã mang lại những bước đột phá sau đây, tất cả đều trong những lĩnh vực vốn khó khăn của học máy:

- + Phân loại hình ảnh gần mức con người
- + Chuyển đổi giọng nói gần mức con người
- + Chuyển đổi chữ viết tay gần mức con người
- + Cải thiện đáng kể dịch máy
- + Cải thiện đáng kể chuyển đổi văn bản thành giọng nói
- + Các trợ lý số như Google Assistant và Amazon Alexa
- + Lái xe tự động gần mức con người
- + Cải thiện nhắm mục tiêu quảng cáo, như được sử dụng bởi Google, Baidu, hoặc Bing
- + Cải thiện kết quả tìm kiếm trên web
- + Khả năng trả lời các câu hỏi bằng ngôn ngữ tự nhiên
- + Chơi cờ vây vượt trội hơn con người

Ngoài ra, học sâu còn giúp con người phát hiện và phân loại bệnh cây trồng trên đồng ruộng chỉ bằng một chiếc điện thoại thông minh đơn giản, hỗ trợ các bác sĩ ung bướu hoặc bác sĩ X-quang trong việc diễn giải dữ liệu hình ảnh y tế, dự đoán các thảm họa tự nhiên như lũ lụt, bão tố, hoặc thậm chí động đất,...

1.2 Lịch sử ngắn gọn của học máy

Có thể nói rằng phần lớn các thuật toán học máy được sử dụng trong ngành công nghiệp ngày nay không phải là học sâu. **Học sâu không phải lúc nào cũng là công cụ phù hợp, đặc biệt khi dữ liệu hạn chế hoặc vấn đề không phức tạp.** Hiểu lịch sử học máy trước học sâu giúp đặt học sâu vào bối cảnh rộng hơn và biết khi nào nên dùng các phương pháp khác.

1.2.1 Mô hình xác suất

Mô hình xác suất áp dụng nguyên lý thống kê vào phân tích dữ liệu, là một trong những hình thức học máy sớm nhất và vẫn được dùng đến nay.

- + **Naive Bayes:** Là bộ phân loại dựa trên định lý Bayes, giả định các đặc trưng trong dữ liệu đầu vào độc lập với nhau (giả định "naive" hoặc đơn giản). Phương pháp dựa trên định lý Bayes từ thế kỷ 18. Hiện nay, nó vẫn được dùng trong phân loại văn bản, lọc spam, và hệ thống đề xuất, nhờ tính đơn giản và khả năng mở rộng.
- + **Hồi quy logistic (logreg):** Thường được coi là "Hello World" của học máy hiện đại, là thuật toán phân loại, không phải hồi quy. Nó dự đoán xác suất kết quả thuộc một lớp nào đó, thường kết quả sẽ là 0, 1. Nó vẫn hữu ích nhờ tính đơn giản, linh hoạt, thường là lựa chọn đầu tiên của nhà khoa học dữ liệu để thử nghiệm ban đầu trên tập dữ liệu, đặc biệt trong y học, tài chính, và phân tích xã hội, nhờ khả năng giải thích rõ ràng.

1.2.2 Mạng nơ-ron ban đầu

Mạng nơ-ron sớm là bước đệm cho học sâu hiện đại, với ý tưởng ban đầu lấy cảm hứng từ cách não bộ hoạt động. Tuy nhiên, việc phát triển và ứng dụng thực tế gặp nhiều thách thức, đặc biệt là trong huấn luyện mạng lớn. Điểm đột phá xuất hiện khi thuật toán Backpropagation được tái khám phá, cho phép huấn luyện mạng nhiều tầng bằng tối ưu hóa gradient-descent.

Ứng dụng thực tế đầu tiên thành công của mạng nơ-ron đến từ Bell Labs vào năm 1989, khi Yann LeCun kết hợp các ý tưởng trước đó về mạng nơ-ron tích chập và backpropagation, áp dụng chúng vào vấn đề phân loại chữ số viết tay. Mạng kết quả, được gọi là LeNet, đã được sử dụng để tự động đọc mã ZIP trên phong bì thư.

1.2.3 Phương pháp nhân

Phương pháp nhân là một nhóm các thuật toán phân loại, trong đó nổi tiếng nhất là **Máy Vector Hỗ Trợ (SVM)**. SVM là một thuật toán phân loại

hoạt động bằng cách tìm “**ranh giới quyết định**” để phân tách hai lớp. SVM tiến hành tìm các ranh giới này qua hai bước:

- + Ánh xạ dữ liệu vào một không gian có chiều cao hơn, nơi ranh giới quyết định là một siêu phẳng (nếu dữ liệu hai chiều, siêu phẳng là một đường thẳng).
- + Tính toán siêu phẳng phân tách bằng cách tối đa hóa khoảng cách (lề) giữa siêu phẳng và các điểm dữ liệu gần nhất từ mỗi lớp, giúp ranh giới tổng quát hóa tốt cho dữ liệu mới ngoài tập huấn luyện.

Tuy nhiên, trong thực tế, SVM thường không thể tính toán được do chi phí tính toán quá lớn. Đó là lúc mà **thủ thuật hạt nhân** xuất hiện. Ý chính là: để tìm các siêu phẳng quyết định tốt trong không gian biểu diễn mới, bạn không cần phải tính toán rõ ràng tọa độ của các điểm trong không gian mới; bạn chỉ cần tính toán **khoảng cách giữa các cặp điểm** trong không gian đó, điều này có thể được thực hiện một cách hiệu quả bằng cách sử dụng **hàm hạt nhân**. **Hàm hạt nhân là một phép toán có thể tính toán được, ánh xạ bất kỳ hai điểm nào trong không gian ban đầu thành khoảng cách giữa các điểm đó trong không gian biểu diễn đích, hoàn toàn bỏ qua việc tính toán rõ ràng biểu diễn mới**. Các hàm hạt nhân thường được **thiết kế thủ công** thay vì học từ dữ liệu - trong trường hợp của SVM, chỉ có siêu phẳng phân tách là được học.

Vào thời điểm phát triển, SVM thể hiện hiệu suất tiên tiến cho các bài toán phân loại đơn giản, được hỗ trợ bởi lý thuyết thống kê sâu rộng. Điều này khiến SVM dễ phân tích toán học và dễ giải thích, dẫn đến sự phổ biến trong lĩnh vực học máy trong thời gian dài. Tuy nhiên, SVM gặp khó khăn khi mở rộng cho tập dữ liệu lớn, như được thảo luận trên do chi phí tính toán cao và yêu cầu giải quyết bài toán tối ưu hóa bậc hai.

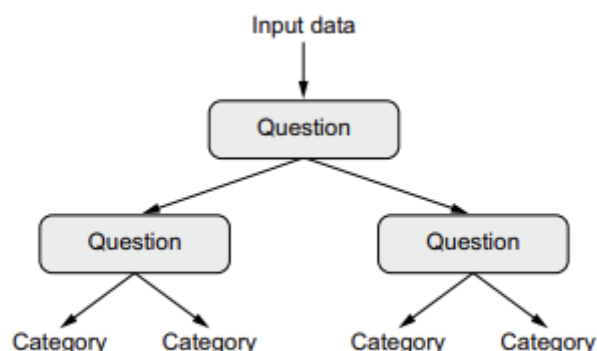
Đặc biệt, SVM không hiệu quả với các vấn đề nhận thức như phân loại hình ảnh. Vì là phương pháp nông, SVM đòi hỏi kỹ thuật đặc trưng thủ công (feature engineering), nghĩa là phải trích xuất thủ công các biểu diễn hữu ích từ dữ liệu thô, như biểu đồ pixel cho chữ số viết tay. Điều này khó khăn và không linh hoạt.

1.2.4 Cây quyết định (decision tree), rừng ngẫu nhiên (random forest) và máy tang cưỡi gradient

a) Decision tree

Decision tree là một thuật toán học có giám sát dùng cho phân loại và hồi quy. **Chúng mô hình hóa quyết định dưới dạng cấu trúc cây, nơi mỗi nút nội bộ đại diện cho một bài kiểm tra trên đặc trưng, các nhánh đại diện cho kết quả của bài kiểm tra, và các nút lá đại diện cho quyết định hoặc dự đoán**

cuối cùng. Ví dụ, một cây quyết định có thể hỏi: "Thu nhập có lớn hơn 50.000 USD không?" để quyết định phê duyệt vay. Sự đơn giản và dễ hiểu, như được ghi nhận trên khiến chúng trở thành nền tảng trong học máy.



b) Random forest

Random forest là một phương pháp tổng hợp kết hợp nhiều cây quyết định để nâng cao độ chính xác dự đoán. Mỗi cây được huấn luyện trên một tập con ngẫu nhiên của dữ liệu, và dự đoán cuối cùng được thực hiện bằng cách tổng hợp kết quả đầu ra của chúng. Rừng ngẫu nhiên có thể áp dụng cho nhiều loại vấn đề khác nhau—bạn có thể nói rằng chúng hầu như luôn là thuật toán tốt thứ hai cho bất kỳ nhiệm vụ học máy nông nào.

c) Gradient boosting machines

Gradient boosting machines xây dựng trên cây quyết định bằng cách lần lượt thêm các cây mới để sửa lỗi của các cây trước. Quá trình này, gọi là tăng cường gradient, tối ưu hóa hàm mất mát bằng cách tiếp cận giống như gradient descent. Các biến thể như XGBoost và LightGBM, đặc biệt, đã cho thấy hiệu suất vượt trội.

Nó có thể là một trong những thuật toán tốt nhất, nếu không muốn nói là tốt nhất, để xử lý **dữ liệu không nhận thức** ngày nay.

1.2.5 Quay lại với mạng nơ-ron

ConvNets là loại mạng nơ-ron chuyên biệt để xử lý dữ liệu dạng lưới, như hình ảnh hoặc chuỗi thời gian. Chúng sử dụng các lớp tích chập (convolutional layers) để áp dụng bộ lọc lên dữ liệu đầu vào, học các mẫu cục bộ như cạnh hoặc góc trong hình ảnh. Sau đó, các lớp gộp (pooling layers), như max pooling, giảm chiều dữ liệu để giảm tính toán và tránh quá khớp. Cuối cùng, các lớp kết nối đầy đủ (fully connected layers) đưa ra dự đoán cuối cùng, chẳng hạn như phân loại hình ảnh thành "mèo" hoặc "chó".

ConvNets học biểu diễn phân cấp, từ các đặc trưng cơ bản đến phức tạp, giống như cách mắt người nhận diện hình ảnh: bắt đầu từ các cạnh, sau đó là hình dạng, và cuối cùng là toàn bộ đối tượng. Chúng đã cách mạng hóa thị giác máy tính, đạt độ chính xác cao trong các nhiệm vụ như nhận diện khuôn mặt, phân loại hình ảnh, và phát hiện đối tượng. Ngoài ra, ConvNets cũng được áp dụng cho các dữ liệu khác, như xử lý ngôn ngữ tự nhiên (sử dụng các biến thể như TextCNN) hoặc phân tích chuỗi thời gian.

1.2.6 Điều làm học sâu trở nên khác biệt

Lý do chính khiến học sâu (deep learning) phát triển nhanh chóng là vì nó mang lại hiệu suất tốt hơn cho nhiều bài toán. Nhưng đó không phải là lý do duy nhất. **Học sâu cũng giúp việc giải quyết vấn đề trở nên dễ dàng hơn, bởi nó tự động hóa hoàn toàn bước quan trọng nhất trong quy trình học máy truyền thống: kỹ thuật đặc trưng (feature engineering).**

Các kỹ thuật học máy trước đây (học nông) chỉ bao gồm việc biến đổi dữ liệu đầu vào thành một hoặc hai không gian biểu diễn liên tiếp, thường thông qua các phép biến đổi đơn giản như SVM hoặc cây quyết định. Nhưng các biểu diễn cần thiết cho các bài toán phức tạp thường không thể đạt được bằng những kỹ thuật này. Do đó, con người phải nỗ lực rất nhiều để làm cho dữ liệu đầu vào dễ xử lý hơn. Họ phải tự tay thiết kế các tầng biểu diễn cho dữ liệu của mình. Quá trình này được gọi là kỹ thuật đặc trưng. **Ngược lại, học sâu tự động hóa hoàn toàn bước này: với học sâu, bạn học tất cả các đặc trưng trong một lần duy nhất thay vì phải tự thiết kế chúng.** Điều này đã đơn giản hóa đáng kể quy trình học máy, thường thay thế các pipeline nhiều giai đoạn phức tạp bằng một mô hình học sâu đơn giản, end-to-end.

Nếu vấn đề cốt lõi là có nhiều tầng biểu diễn liên tiếp, liệu các phương pháp học nông có thể được áp dụng lặp đi lặp lại để mô phỏng hiệu ứng của học sâu không? Thực tế, việc áp dụng liên tiếp các phương pháp học chỉ làm hiệu quả giảm nhanh, bởi vì tầng biểu diễn đầu tiên tối ưu trong một mô hình ba tầng không phải là tầng đầu tiên tối ưu trong một mô hình một hoặc hai tầng. Điều mang tính cách mạng của học sâu là nó cho phép một mô hình học tất cả các tầng biểu diễn cùng lúc, thay vì học từng tầng một cách tham lam (greedy). Với việc học đặc trưng chung, mỗi khi mô hình điều chỉnh một đặc trưng nội bộ, tất cả các đặc trưng khác phụ thuộc vào nó sẽ tự động thích nghi với thay đổi mà không cần sự can thiệp của con người. Mọi thứ được giám sát bởi một tín hiệu phản hồi duy nhất: mọi thay đổi trong mô hình đều phục vụ mục tiêu cuối cùng. Điều này mạnh mẽ hơn nhiều so với việc xếp chồng các mô hình học nông một cách tham lam, bởi vì nó cho phép học các biểu diễn phức tạp, trừu tượng bằng

cách chia nhỏ chúng thành một chuỗi dài các không gian trung gian (tầng); mỗi không gian chỉ là một phép biến đổi đơn giản từ không gian trước đó.

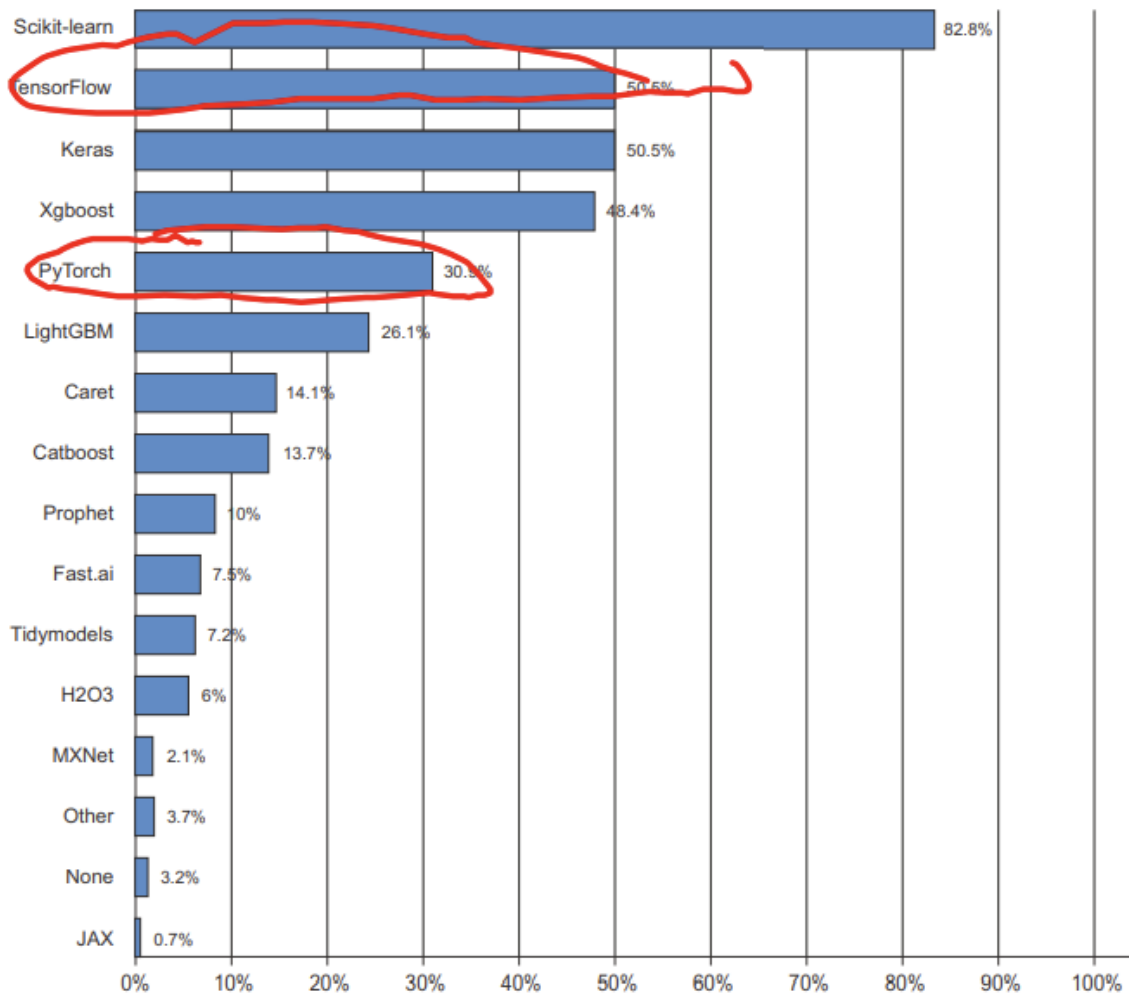
Đây là hai đặc điểm cốt lõi của cách học sâu học từ dữ liệu: cách thức từng bước, tầng-bổ-tầng mà các biểu diễn ngày càng phức tạp được phát triển, và việc các biểu diễn trung gian này được học cùng lúc, mỗi tầng được cập nhật để đáp ứng cả nhu cầu biểu diễn của tầng trên và tầng dưới. Cùng nhau, hai đặc tính này đã khiến học sâu thành công vượt trội hơn nhiều so với các phương pháp học máy trước đây.

1.2.7 Tổng quan về học máy hiện đại

Để hiểu về bối cảnh hiện tại của các thuật toán và công cụ học máy, ta xem xét các cuộc thi học máy trên Kaggle. Với môi trường cạnh tranh cao (một số cuộc thi có hàng ngàn người tham gia và giải thưởng lên đến hàng triệu đô-la) cùng với sự đa dạng của các bài toán học máy được đề cập, Kaggle cung cấp một cách thực tế để đánh giá những gì hiệu quả và những gì không. Vậy thuật toán nào thường xuyên giành chiến thắng trong các cuộc thi? Các đội hàng đầu sử dụng công cụ nào?

Vào đầu năm 2019, Kaggle đã thực hiện một khảo sát hỏi các đội đứng trong top 5 của bất kỳ cuộc thi nào kể từ năm 2017 về công cụ phần mềm chính mà họ đã sử dụng trong cuộc thi. Kết quả cho thấy các đội hàng đầu thường sử dụng hoặc phương pháp học sâu (deep learning, thường thông qua thư viện Keras) hoặc cây tăng cường gradient (gradient boosted trees, thường thông qua các thư viện LightGBM hoặc XGBoost).

Ngoài ra, Kaggle cũng tổ chức một khảo sát hàng năm với các chuyên gia học máy và khoa học dữ liệu trên toàn thế giới. Với hàng chục ngàn người trả lời, khảo sát này là một trong những nguồn đáng tin cậy nhất về tình trạng của ngành. Hình 1.13 cho thấy tỷ lệ sử dụng của các framework phần mềm học máy khác nhau.



Hình 1.3

Từ năm 2016 đến 2020, toàn bộ ngành học máy và khoa học dữ liệu đã bị chi phối bởi hai phương pháp: học sâu và cây tăng cường gradient. Cụ thể, cây tăng cường gradient được sử dụng cho các bài toán có dữ liệu cấu trúc, thông qua Scikit-learn, XGBoots hoặc LightGBM, trong khi học sâu được sử dụng cho các bài toán nhận thức (perceptual problems) như phân loại hình ảnh, sử dụng Keras, thường kết hợp với framework mẹ của nó là TensorFlow.

=> Điểm chung của các công cụ này là chúng đều là các thư viện Python: Python là ngôn ngữ được sử dụng rộng rãi nhất cho học máy và khoa học dữ liệu.

1.3 Tại sao là học sâu, tại thời điểm này?

Học sâu chỉ thực sự bùng nổ sau 2012, vậy tại sao lại như vậy, và điều gì đã xảy ra trong thời gian này? **Thực tế, có ba yếu tố kỹ thuật đang thúc đẩy sự tiến bộ trong học máy:**

- **Phần cứng (Hardware)**
- **Tập dữ liệu và chuẩn đánh giá (Datasets and benchmarks)**
- **Tiến bộ thuật toán (Algorithmic advances)**

Vậy tại sao lại cần ba yếu tố này?, Lý do vì lĩnh vực này được dẫn dắt bởi các kết quả thực nghiệm hơn là lý thuyết, các tiến bộ thuật toán chỉ có thể xảy ra khi có dữ liệu và phần cứng phù hợp để thử nghiệm các ý tưởng mới (hoặc mở rộng quy mô các ý tưởng cũ), chứ không đơn thuần là sử dụng bút và giấy để nghiên cứu như toán học hay vật lý. Nó là 1 ngành khoa học kỹ thuật.

1.3.1 Phần cứng (Hardware)

Từ năm 1990 đến 2010, các CPU thông thường đã nhanh hơn khoảng 5.000 lần. Kết quả là, ngày nay bạn có thể chạy các mô hình học sâu nhỏ trên laptop của mình, điều không thể thực hiện được cách đây 25 năm.

Nhưng các mô hình học sâu điển hình dùng trong thị giác máy tính hoặc nhận diện giọng nói yêu cầu sức mạnh tính toán lớn hơn nhiều so với khả năng của laptop. Trong suốt những năm 2000, các công ty như NVIDIA và AMD đã đầu tư hàng tỷ đô-la để phát triển các chip nhanh, song song hóa cao (hay còn gọi là GPU) nhằm phục vụ đồ họa cho các trò chơi điện tử - những siêu máy tính giá rẻ, chuyên dụng được thiết kế để dựng các cảnh 3D phức tạp trên màn hình của bạn theo thời gian thực. Một số lượng nhỏ GPU bắt đầu thay thế các cụm CPU lớn trong các ứng dụng có khả năng song song hóa cao, bắt đầu từ mô phỏng vật lý.

Điều xảy ra là thị trường trò chơi điện tử đã tài trợ cho siêu máy tính cho thế hệ tiếp theo của các ứng dụng trí tuệ nhân tạo. Đôi khi, những điều lớn lao bắt đầu từ trò chơi. Ngày nay, NVIDIA Titan RTX, một GPU có giá 2.500 USD vào cuối năm 2019, có thể cung cấp hiệu suất tối đa 16 teraFLOPS ở độ chính xác đơn (16 nghìn tỷ phép toán float32 mỗi giây). Đó là sức mạnh tính toán gấp khoảng 500 lần so với siêu máy tính nhanh nhất thế giới năm 1990, Intel Touchstone Delta. Trên Titan RTX, chỉ mất vài giờ để huấn luyện một mô hình ImageNet tương tự như loại đã giành chiến thắng trong cuộc thi ILSVRC vào khoảng năm 2012 hoặc 2013. Trong khi đó, các công ty lớn huấn luyện mô hình học sâu trên cụm hàng trăm GPU.

Hơn nữa, ngành học sâu đã vượt qua GPU và đang đầu tư vào các chip chuyên dụng, hiệu quả hơn cho học sâu. Vào năm 2016, tại hội nghị I/O

hàng năm, Google đã công bố dự án **Tensor Processing Unit (TPU)**: một thiết kế chip mới được phát triển từ đầu để chạy mạng nơ-ron sâu nhanh hơn đáng kể và tiết kiệm năng lượng hơn nhiều so với GPU hàng đầu. Đến năm 2020, phiên bản thứ ba của thẻ TPU đạt hiệu suất 420 teraFLOPS. Đó là sức mạnh tính toán gấp 10.000 lần so với Intel Touchstone Delta năm 1990.

1.3.2 Dữ liệu (Data)

Trí tuệ nhân tạo (AI) đôi khi được ca ngợi là cuộc cách mạng công nghiệp mới. **Nếu học sâu là động cơ hơi nước của cuộc cách mạng này, thì dữ liệu là than đá**: nguyên liệu thô cung cấp năng lượng cho các máy thông minh của chúng ta. Về dữ liệu, ngoài sự tiến bộ vượt bậc trong phần cứng lưu trữ trong 20 năm qua (theo định luật Moore), **yếu tố thay đổi cuộc chơi là sự bùng nổ của internet, giúp việc thu thập và phân phối các tập dữ liệu lớn cho học máy trở nên khả thi**. Ngày nay, các công ty lớn làm việc với các tập dữ liệu hình ảnh, video và ngôn ngữ tự nhiên mà không thể thu thập được nếu không có internet. Các thẻ hình ảnh do người dùng tạo trên Flickr, ví dụ, đã là một kho báu dữ liệu cho thị giác máy tính. Video trên YouTube cũng vậy. Và Wikipedia là một tập dữ liệu quan trọng cho xử lý ngôn ngữ tự nhiên.

Nếu có một tập dữ liệu đã làm chất xúc tác cho sự phát triển của học sâu, thì đó là tập dữ liệu ImageNet, bao gồm 1,4 triệu hình ảnh được gắn nhãn thủ công với 1.000 danh mục hình ảnh (mỗi hình ảnh một danh mục). Nhưng điều làm ImageNet đặc biệt không chỉ là kích thước lớn, mà còn là cuộc thi hàng năm liên quan đến nó. Việc có các chuẩn đánh giá chung mà các nhà nghiên cứu cạnh tranh để vượt qua đã giúp rất nhiều cho sự phát triển của học sâu, bằng cách làm nổi bật thành công của nó so với các phương pháp học máy cổ điển.

1.3.3 Thuật toán (Algorithms)

Ngoài phần cứng và dữ liệu, cho đến cuối những năm 2000, chúng ta vẫn thiếu một cách đáng tin cậy để huấn luyện các mạng nơ-ron rất sâu. Kết quả là, các mạng nơ-ron vẫn còn khá nông, chỉ sử dụng một hoặc hai tầng biểu diễn; do đó, chúng không thể vượt trội hơn các phương pháp học nông như SVM và random forests. Vấn đề cốt lõi là lan truyền gradient qua các tầng sâu. Tín hiệu phản hồi dùng để huấn luyện mạng nơ-ron sẽ mờ dần khi số lượng tầng tăng lên.

Điều này đã thay đổi vào khoảng năm 2009–2010 với sự xuất hiện của một số cải tiến thuật toán đơn giản nhưng quan trọng, cho phép lan truyền gradient tốt hơn:

- Hàm kích hoạt tốt hơn cho các tầng nơ-ron.

- Phương pháp khởi tạo trọng số tốt hơn, bắt đầu với việc huấn luyện trước từng tầng (layer-wise pretraining), sau đó nhanh chóng bị bỏ qua.
- Phương pháp tối ưu hóa tốt hơn, như RMSProp và Adam.

Chỉ khi những cải tiến này bắt đầu cho phép huấn luyện các mô hình với 10 tầng trở lên, học sâu mới bắt đầu tỏa sáng.

Cuối cùng, vào các năm 2014, 2015 và 2016, các phương pháp tiên tiến hơn để cải thiện lan truyền gradient đã được phát hiện, như chuẩn hóa hàng loạt (batch normalization), kết nối dư (residual connections), và tích chập phân tách theo chiều sâu (depthwise separable convolutions).

Ngày nay, chúng ta có thể huấn luyện các mô hình sâu tùy ý từ đầu. Điều này đã mở ra việc sử dụng các mô hình cực lớn, có sức mạnh biểu diễn đáng kể—tức là mã hóa các không gian giả thuyết rất phong phú. Khả năng mở rộng cực lớn này là một trong những đặc điểm nổi bật của học sâu hiện đại. Các kiến trúc mô hình quy mô lớn, với hàng chục tầng và hàng chục triệu tham số, đã mang lại những tiến bộ quan trọng trong cả thị giác máy tính (ví dụ: các kiến trúc như ResNet, Inception, hoặc Xception) và xử lý ngôn ngữ tự nhiên (ví dụ: các kiến trúc dựa trên Transformer lớn như BERT, GPT-3, hoặc XLNet).

1.3.4 Một vài yếu tố khác

Học sâu bùng nổ sau 2012 nhờ làn sóng đầu tư lớn từ ngành công nghiệp (tăng từ dưới 1 tỷ USD năm 2011 lên 16 tỷ USD năm 2017), sự dân chủ hóa công cụ (như Keras, TensorFlow, Theano giúp dễ tiếp cận hơn), và các đặc tính vượt trội: tính đơn giản (loại bỏ feature engineering), khả năng mở rộng (song song hóa trên GPU/TPU), và tính linh hoạt (tái sử dụng mô hình). Học sâu là một cuộc cách mạng AI lâu dài. Đến năm 2021, học sâu đã qua giai đoạn bùng nổ ban đầu, nhưng vẫn còn tiềm năng lớn trong tương lai.

CHƯƠNG 2: CÁC KHỐI XÂY DỰNG TOÁN HỌC CỦA MẠNG NƠ-RO

2.0 Tổng quan

Chương này giới thiệu các khái niệm toán học cơ bản của học sâu (tensor, phép toán tensor, vi phân, gradient descent) một cách trực giác, tránh ký hiệu toán học phức tạp và sử dụng mã thực thi để giải thích. Bắt đầu với một ví dụ thực tế về mạng nơ-ron, chương sẽ giải thích từng khái niệm (tensor, lan truyền ngược, gradient descent), giúp bạn hiểu cách mạng nơ-ron học và chuẩn bị cho việc sử dụng Keras/TensorFlow ở chương sau.

2.1 Một chút về mạng nơ-ron

Hãy xem xét một ví dụ cụ thể về một mạng nơ-ron sử dụng thư viện Python **Keras** để học cách phân loại các chữ số viết tay.

Bài toán chúng ta đang cố gắng giải quyết ở đây là phân loại các hình ảnh của các chữ số viết tay thành 10 danh mục (từ 0 đến 9). Chúng ta sẽ sử dụng tập dữ liệu **MNIST**, một tập dữ liệu kinh điển trong cộng đồng học máy, đã tồn tại gần như từ khi lĩnh vực này ra đời và đã được nghiên cứu kỹ lưỡng. Đây là tập dữ liệu gồm 60.000 hình ảnh huấn luyện và 10.000 hình ảnh kiểm tra, được tập hợp bởi Viện Tiêu chuẩn và Công nghệ Quốc gia (National Institute of Standards and Technology - NIST, trong MNIST) vào những năm 1980.

=> Giải quyết MNIST được coi là bài toán “Hello World” của học sâu.

Bước 1: Tải bộ dữ liệu từ MNIST bằng Keras

```
1 #import tập dữ liệu mnist
2 from tensorflow.keras.datasets import mnist
3
4 # 2 set dữ liệu
5 (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

=> Chia ra làm 2 tập: tập huấn luyện và tập kiểm tra. Ở mỗi tập lại có phần hình ảnh và nhãn tương ứng với nó

a) Xem qua một chút về dữ liệu huấn luyện:

+ Lệnh 1: In ra shape của tập ảnh huấn luyện

```
1 print(train_images.shape)
✓ 0.0s Python
(60000, 28, 28)
```


Kết quả cho thấy: có 60000 ảnh, mỗi ảnh kích cỡ 28x28 pixel

+ Lệnh 2: in ra độ dài của mảng chứa giá trị nhãn huấn luyện

```
1 print(len(train_labels))
```

✓ 0.0s

60000

Kết quả: có 60000 nhãn, ứng với 60000 ảnh của train_images

+ Lệnh 3: in ra từng phần tử của mảng chứa các nhãn huấn luyện

```
1 print(train_labels)
```

✓ 0.0s

[5 0 4 ... 5 6 8]

Kết quả: in ra được danh sách các nhãn ứng với từng ảnh theo thứ tự

b) Xem tiếp dữ liệu kiểm tra với cùng 3 câu lệnh (giải thích y hệt)

```
1 print(test_images.shape)
```

✓ 0.0s Python

(10000, 28, 28)

```
1 print(len(test_labels))
```

✓ 0.0s Python

10000

```
1 print(test_labels)
```

✓ 0.0s Python

[7 2 1 ... 4 5 6]

Bước 2: Cung cấp dữ liệu huấn luyện cho mạng nơ ron. Mạng nơ ron sẽ học cách liên kết hình ảnh với các nhãn tương ứng, và cuối cùng, chúng ra sẽ yêu cầu mạng tạo ra các dự đoán cho dữ liệu kiểm tra, xem có khớp giữa test_images và test_label không

```
1 from tensorflow import keras
2 from tensorflow.keras import layers
3 model = keras.Sequential([
4     layers.Dense(512, activation="relu"),
5     layers.Dense(10, activation="softmax")
6 ])
4] ✓ 0.0s
```

Giải thích: đoạn code tạo ra một mô hình tuần tự (các tầng xếp chồng lên nhau một cách tuần tự), gồm 2 tầng (đoạn này chưa cần quá hiểu code), chỉ cần hiểu cấu trúc cốt lõi của mạng nơ ron là tầng (cho dữ liệu vào, và cho dữ liệu ra ở 1 dạng có ích hơn). **Phần lớn học sâu bao gồm việc xâu chuỗi các tầng đơn giản lại với nhau để tinh chế dữ liệu dần dần, nghĩa là sau mỗi tầng thì dữ liệu đầu ra ngày càng có ích và chính xác hơn.**

Bước 3: Chọn thêm một bộ tối ưu hóa, một hàm mất mát và cách theo dõi hiệu suất (công dụng như đã nói ở phần 1.1.5)

```
1 model.compile(optimizer="rmsprop",
2               loss="sparse_categorical_crossentropy",
3               metrics=["accuracy"])
✓ 0.1s
```

Giải thích: dùng hàm compile để cấu hình cho model, yêu cầu với 3 tham số chính là optimizer (bộ tối ưu hóa), loss (hàm mất mát) và metric (cách theo dõi hiệu suất)

Bước 4: Tiền xử lý dữ liệu: do mạng nơ ron yêu cầu đầu vào là vector 1 chiều, chứ không phải 2 chiều (28x28 pixel), do đó cần reshape và chuẩn hóa lại dạng

```
1 train_images = train_images.reshape((60000, 28 * 28))
2 train_images = train_images.astype("float32") / 255
3 test_images = test_images.reshape((10000, 28 * 28))
4 test_images = test_images.astype("float32") / 255
✓ 0.2s
```

Đến đây, dữ liệu đã sẵn sàng để huấn luyện và kiểm tra

Bước 5: Dùng hàm fit() để huấn luyện dữ liệu, truyền vào tập huấn luyện, epoch là số lần huấn luyện, batch_size=k là chia dữ liệu thành các lô mẫu huấn luyện (k ảnh/lô) để huấn luyện từng bước

```
1 model.fit(train_images, train_labels, epochs=5, batch_size=128)
✓ 9.9s

Epoch 1/5
469/469 ————— 3s 5ms/step - accuracy: 0.8704 - loss: 0.4427
Epoch 2/5
469/469 ————— 2s 4ms/step - accuracy: 0.9683 - loss: 0.1133
Epoch 3/5
469/469 ————— 2s 4ms/step - accuracy: 0.9785 - loss: 0.0732
Epoch 4/5
469/469 ————— 2s 4ms/step - accuracy: 0.9849 - loss: 0.0509
Epoch 5/5
469/469 ————— 2s 4ms/step - accuracy: 0.9885 - loss: 0.0385
```

Các thông số theo từng bước: as bms/step – accuracy: c – loss: d

Giải thích

- + a : thời gian thực hiện epoch
- + b: thời gian trung bình cho mỗi lô
- + c: độ chính xác sau mỗi epoch, ban đầu là 87.04%, dần dần tăng theo từng tầng đến 98.85%
- + d: giá trị hàm mất mát sau mỗi epoch, giá trị này sẽ giảm dần => mô hình ngày càng chính xác hơn

Kết luận: mô hình sau huấn luyện có độ chính xác lên đến 98.85%, kết quả rất tốt.

Bước 5: Dự đoán trên dữ liệu kiểm tra: model đã huấn luyện, giờ dùng hàm predict và truyền vào dữ liệu kiểm tra, giả sử là 10 giá trị đầu của mảng dữ liệu kiểm tra

Giải thích code:

- + hàm predict để dự đoán
- + truyền vào test_digits là 10 giá trị kiểm tra đầu vào đầu tiên của mảng test_images
- + in ra lần lượt:
- + prediction[i].argmax(): lấy ra và chuyển đổi kết quả dự đoán thành số 0->9 nhờ argmax() với giá trị của bộ dữ liệu thứ i (prediction[i])
- + test_labels[i]: nhãn của bộ dữ liệu thứ i
- + từ đây có thể nhìn và so sánh giữa dự đoán và kết quả

```
1 test_digits = test_images[0:10]
2 predictions = model.predict(test_digits)
3 for i in range(0,10):
4     print(predictions[i].argmax(), predictions[i][7])
5     print(test_labels[i])
6     print()
✓ 0.0s
```

1/1 ————— 0s 28ms/step
7 0.999869
7
2 1.4706832e-13
2
1 0.00010039983
1
0 2.8252448e-06
0
4 4.936688e-06
4
1 6.541762e-05
1
4 9.997266e-06
4
9 0.00012913147
9
...
9 0.00012452835
9

Bước 6: Kiểm tra độ chính xác

```
1 test_loss, test_acc = model.evaluate(test_images, test_labels)
2 print(f"Độ chính xác: {test_acc}")
✓ 0.5s
```

313/313 ————— 0s 1ms/step - accuracy: 0.9753 - loss: 0.0725
Độ chính xác: 0.979200005531311

313/313 vì mặc định batch_size là 32, 10000/32 là 313 lô, mỗi lô 312 ảnh

Độ chính xác thấp hơn (97.92%) so với accuracy ở trên (98.85%) là do overfitting – hiện tượng model học máy hoạt động kém hơn ở dữ liệu mới so với dữ liệu huấn luyện.

2.2 Cách biểu diễn dữ liệu trong mạng nơ ron

Trong ví dụ trước, chúng ta bắt đầu từ dữ liệu được lưu trữ trong các mảng NumPy đa chiều, còn được gọi là **tensor**. Nói chung, tất cả các hệ thống học máy hiện tại đều sử dụng tensor làm cấu trúc dữ liệu cơ bản của chúng. Tensor là nền tảng của lĩnh vực này—nền tảng đến mức TensorFlow được đặt tên theo chúng. Vậy tensor là gì?

Về cốt lõi, tensor là một **thùng chứa dữ liệu**—thường là dữ liệu số. Vì vậy, nó là một thùng chứa cho các con số. Bạn có thể đã quen thuộc với ma trận, là tensor bậc 2: tensor là một khái quát hóa của ma trận cho một số chiều bất kỳ (lưu ý rằng trong ngữ cảnh của tensor, một chiều thường được gọi là **trục** - axis).

2.2.1 Tensor bậc n

- + Tensor bậc 0: hay còn gọi là vô hướng, ví dụ như 1 số thực
- + Tensor bậc 1: là vector, biểu diễn trong NumPy ở dạng `np.array`, ví dụ: `x = np.array([1,2,3,4,5])`, là tensor bậc 1 (có thể kiểm tra bằng hàm `x.ndim`), và là vector 5 chiều (do `x` có 5 phần tử)
- + Tensor bậc 2: là ma trận
- + Tensor bậc 3 và cao hơn (n chiều): tương tự ma trận 3 và n chiều.

2.2.2 Các thuộc tính khóa của tensor

- + Số trục (axes), dùng thuộc tính `ndim`:

```
1 print(train_images.ndim)
✓ 0.0s Python
3
```

- + Hình dạng (shape), dùng thuộc tính `shape`

```
1 print(train_images.shape)
✓ 0.0s
(60000, 28, 28)
```

- + Kiểu dữ liệu (dtype), dùng thuộc tính `dtype`

```
1 print(train_images.dtype)
✓ 0.0s Python
float32
```

=> Chúng ta nhận thấy, tập ảnh huấn luyện là tensor bậc 3, là mảng có 60000 ảnh, mỗi ảnh được biểu diễn bởi ma trận 28x28 với các ô nhận giá trị là số nguyên từ 0 đến 255, đại diện cho độ xám của pixel đó

Bây giờ, biểu diễn ảnh bằng matplotlib, nhận thấy có 28x28 ô, mỗi ô nhỏ được tô màu đại tương ứng với độ xám trong ảnh gốc.



Giờ thì xem nhãn tương ứng:

```
1 print(train_labels[4])
```

✓ 0.0s

9

2.2.3 Tensor Slicing

Giống như trong mảng bình thường, tensor cũng có thể cắt thành các slice (lát cắt)

Lấy ví dụ với tập `train_images`:

- + `my_slice = train_images[10:100]` : lấy từ ảnh 10-99
- + do `train_images` là mảng, mỗi phần tử gồm 1 ảnh, và 2 mảng 28 pixel, nên khi cắt như trên, mỗi phần tử của slice sẽ giống như mỗi phần tử của tập `train_images`.
- + một số cách biểu diễn khác của cách trên: `train_images[10:100,:,:]` hoặc `[10:100,0:28,0:28]`
- + nếu chỉ lấy 14 pixel đầu của cả 2 mảng pixel: thay 2 cụm ở sau bằng `0:14,0:14` là được

2.2.4 Các lô dữ liệu

Mô hình học sâu không xử lý toàn bộ tập dữ liệu cùng một lúc; thay vào đó, chúng chia dữ liệu thành các **lô nhỏ**

Ví dụ: 1 lô lấy 128 ảnh, thì lô 1 là `train_images[0:128]`, lô 2 là `[128:256]`, lô thứ n là `[128*n : 128*(n+1)]`

Trục đầu tiên (trục 0, vì chỉ số bắt đầu từ 0) trong tất cả các tensor dữ liệu mà bạn sẽ gặp trong học sâu được gọi là trục mẫu

2.2.5 Một số ví dụ đời thực của tensor

- + Vector dữ liệu: tensor bậc 2, chứa các số mẫu và đặc trưng
- + Dữ liệu chuỗi thời gian, dữ liệu tuần tự: tensor bậc 3, gồm số mẫu, số bước nhảy thời gian và số đặc trưng.
- + Hình ảnh: tensor bậc 4: gồm số mẫu, chiều cao, chiều rộng, số kênh (số kênh: cách biểu diễn màu: ví dụ thang độ xám, giá trị từ 0-255 thì có 1 kênh, màu RGB có 3 kênh (giá trị red, giá trị green, giá trị blue), màu RGB thêm độ trong suốt thì có 4 kênh)
- + Video: tensor bậc 5, giống hình ảnh, nhưng có thêm giá trị số khung hình/ giây (FPS)

2.2.6 Dữ liệu vector

Đây là dạng dữ liệu phổ biến nhất. Trong một tập dữ liệu, mỗi điểm dữ liệu đơn lẻ có thể được mã hóa thành một vector, và do đó, một lô dữ liệu sẽ được mã hóa thành một tensor bậc 2 (tức là một mảng các vector), trong đó trục đầu tiên là trục mẫu (samples axis) và trục thứ hai là trục đặc trưng (features axis).

Ví dụ: Một tập dữ liệu bảo hiểm về con người, trong đó chúng ta xem xét tuổi, giới tính và thu nhập của mỗi người. Mỗi người có thể được biểu diễn dưới dạng một vector gồm 3 giá trị, và do đó, toàn bộ tập dữ liệu gồm 100.000 người có thể được lưu trữ trong một tensor bậc 2 có shape là (100000, 3).

2.2.7 Dữ liệu chuỗi thời gian hoặc dữ liệu tuần tự

Bất cứ khi nào thời gian (hoặc khái niệm thứ tự tuần tự) có vai trò quan trọng trong dữ liệu của bạn, việc lưu trữ dữ liệu đó trong một tensor bậc 3 với một trục thời gian rõ ràng là hợp lý. Mỗi mẫu (sample) có thể được mã hóa dưới dạng một tensor bậc 2, và do đó, một lô dữ liệu sẽ được mã hóa thành một tensor bậc 3.

Theo quy ước, trục thời gian luôn là trục thứ hai (trục có chỉ số 1). Ví dụ: Một tập dữ liệu về giá cổ phiếu: Cứ mỗi phút, chúng ta lưu trữ giá hiện tại của cổ phiếu, giá cao nhất trong phút vừa qua, và giá thấp nhất trong phút vừa qua.

Như vậy, mỗi phút được mã hóa thành một vector 3 chiều, một ngày giao

dịch đầy đủ được mã hóa thành một ma trận có hình dạng (390, 3) (có 390 phút trong một ngày giao dịch), và dữ liệu của 250 ngày có thể được lưu trữ trong một tensor bậc 3 có hình dạng (250, 390, 3). Ở đây, mỗi mẫu sẽ là dữ liệu của một ngày.

2.2.8 Dữ liệu hình ảnh

Hình ảnh thường có ba chiều: chiều cao, chiều rộng và độ sâu màu (color depth). Mặc dù các hình ảnh thang độ xám (như các chữ số trong MNIST) chỉ có một kênh màu và do đó có thể được lưu trữ trong tensor bậc 2, nhưng theo quy ước, tensor hình ảnh luôn là tensor bậc 3, với một kênh màu một chiều cho các hình ảnh thang độ xám. Một lô gồm 128 hình ảnh thang độ xám có kích thước 256×256 có thể được lưu trữ trong một tensor có hình dạng (128, 256, 256, 1), và một lô gồm 128 hình ảnh màu có thể được lưu trữ trong một tensor có hình dạng (128, 256, 256, 3).

Có hai quy ước về hình dạng của tensor hình ảnh: quy ước **channels-last** (được sử dụng chuẩn trong TensorFlow) và quy ước **channels-first** (ngày càng ít được ưa chuộng).

Quy ước channels-last đặt trục độ sâu màu ở cuối: (số mẫu, chiều cao, chiều rộng, độ sâu màu). Trong khi đó, quy ước channels-first đặt trục độ sâu màu ngay sau trục lô: (số mẫu, độ sâu màu, chiều cao, chiều rộng). Với quy ước channels-first, các ví dụ trước sẽ trở thành (128, 1, 256, 256) và (128, 3, 256, 256). API của Keras hỗ trợ cả hai định dạng này.

2.2.9 Dữ liệu video

Dữ liệu video là một trong số ít loại dữ liệu thực tế mà bạn sẽ cần sử dụng tensor bậc 5. Một video có thể được hiểu là một chuỗi các khung hình, mỗi khung hình là một hình ảnh màu. Vì mỗi khung hình có thể được lưu trữ trong một tensor bậc 3 (chiều cao, chiều rộng, độ sâu màu), một chuỗi các khung hình có thể được lưu trữ trong một tensor bậc 4 (số khung hình, chiều cao, chiều rộng, độ sâu màu), và do đó, một lô gồm nhiều video khác nhau có thể được lưu trữ trong một tensor bậc 5 có hình dạng (số mẫu, số khung hình, chiều cao, chiều rộng, độ sâu màu).

Ví dụ, một đoạn video YouTube dài 60 giây, kích thước 144×256 , được lấy mẫu với tốc độ 4 khung hình mỗi giây sẽ có 240 khung hình. Một lô gồm 4 đoạn video như vậy sẽ được lưu trữ trong một tensor có hình dạng (4, 240, 144, 256, 3). Đó là tổng cộng 106.168.320 giá trị! Nếu kiểu dữ liệu (dtype) của tensor là float32, mỗi giá trị sẽ được lưu trữ trong 32 bit, vì vậy tensor này sẽ chiếm 405 MB. Các video bạn gặp trong đời thực nhẹ hơn nhiều, vì chúng không được lưu trữ ở định dạng float32, và chúng thường được nén rất nhiều (chẳng hạn như ở định dạng MPEG).

2.3 Các phép toán tensor

Mọi biến đổi trong mạng nơ-ron sâu đều có thể được rút gọn thành các phép toán tensor (như cộng, nhân, tích vô hướng).

Ví dụ tầng Dense trong Keras (`keras.layers.Dense(512, activation = "relu")`) là một hàm:

- Nhận đầu vào là một ma trận (input).
- Thực hiện: $\text{output} = \text{relu}(\text{dot}(\text{input}, W) + b)$:
 - `dot(input, W)`: Tích vô hướng giữa đầu vào và ma trận trọng số W .
 - `+ b`: Cộng vector độ lệch b .
 - `relu()`: Hàm kích hoạt ReLU ($\max(x, 0)$: $x > 0$ trả về x , $x < 0$ trả về 0).

Các phép toán này được diễn đạt bằng code Python thay vì ký hiệu toán học để dễ hiểu.

2.3.1 Các phép toán theo từng phần tử (Element-wise operations)

Phép toán ReLU và phép cộng là các **phép toán theo từng phần tử**: các phép toán được áp dụng một cách độc lập cho từng phần tử trong các tensor được xem xét. Điều này có nghĩa là các phép toán này rất phù hợp để triển khai song song hóa ở quy mô lớn. Nếu bạn muốn viết một triển khai Python đơn giản cho một phép toán theo từng phần tử, bạn sẽ sử dụng vòng lặp `for`, như trong triển khai đơn giản sau của phép toán ReLU theo từng phần tử:

```
def naive_relu(x):  
    assert len(x.shape) == 2 # x là một tensor bậc 2 NumPy  
    x = x.copy() # Tránh ghi đè lên tensor đầu vào  
    for i in range(x.shape[0]):  
        for j in range(x.shape[1]):  
            x[i, j] = max(x[i, j], 0) #phép relu  
    return x
```

Python

Bạn cũng có thể làm tương tự cho phép cộng:

```
def naive_add(x, y):
    assert len(x.shape) == 2 # x và y là các tensor bậc 2 NumPy
    assert x.shape == y.shape
    x = x.copy() # Tránh ghi đè lên tensor đầu vào
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j] # Phép cộng
    return x
```

Theo cùng nguyên tắc, bạn có thể thực hiện phép nhân theo từng phần tử, phép trừ, v.v.

Trong thực tế, khi làm việc với các mảng NumPy, các phép toán này có sẵn dưới dạng các hàm NumPy được tối ưu hóa tốt, và chúng giao phó công việc nặng nhọc cho một triển khai **Basic Linear Algebra Subprograms (BLAS)**. BLAS là các routine thao tác tensor cấp thấp, song song hóa cao, hiệu quả, thường được triển khai bằng Fortran hoặc C.

Vì vậy, trong NumPy, bạn có thể thực hiện phép toán theo từng phần tử sau đây, và nó sẽ cực kỳ nhanh:

```
1 z = x + y # Phép cộng theo từng phần tử
2 z = np.maximum(z, 0.) # Phép ReLU theo từng phần tử
3
```

So sánh thời gian: Sử dụng NumPy cho kết quả trong 0.2s, trong khi hàm tự xây cho kết quả ở 2.45s

2.3.2 Broadcasting (Phép phát sóng)

Triển khai đơn giản trước đây của chúng ta cho naive_add chỉ hỗ trợ phép cộng giữa các tensor bậc 2 có hình dạng giống hệt nhau. Nhưng trong tầng Dense được giới thiệu trước đó, chúng ta đã cộng một tensor bậc 2 với một vector. Vậy phải làm gì để thực hiện được điều này?

Khi có thể và không có sự mơ hồ, tensor nhỏ hơn sẽ được **phát sóng (broadcast)** để khớp với hình dạng của tensor lớn hơn. Broadcasting bao gồm hai bước:

1. Các trục (còn gọi là trục phát sóng - broadcast axes) được thêm vào tensor nhỏ hơn để khớp với số trục của tensor lớn hơn.
2. Tensor nhỏ hơn được lặp lại dọc theo các trục mới này để khớp với toàn bộ hình dạng của tensor lớn hơn.

Hãy xem xét một ví dụ cụ thể. Xét X có hình dạng (32, 10) và Y có hình dạng (10,):

Đầu tiên, chúng ta thêm một trục đầu tiên rỗng vào y, khiến hình dạng của nó trở thành (1, 10):

Sau đó, chúng ta lặp lại y 32 lần dọc theo trục mới này, để cuối cùng ta có một tensor Y với hình dạng (32, 10)

```
1 import numpy as np
2 X = np.random.random((32, 10)) #random 1 ma trận có shape = 32,10
3 Y = np.random.random((10,)) # random vector có shape = 10
4 print("Shape của X và Y là:")
5 print(X.shape, Y.shape)
6 print()
7
8 # Thêm trục rỗng vào Y
9 Y = np.expand_dims(Y, axis=0)
10 print("Shape của Y sau khi thêm 1 trục rỗng")
11 print(Y.shape)
12 print("Thu được Y có shape là (1,10)")
13 print()
14
15 #Lặp lại 32 lần y dọc theo trục 0
16 Y = np.concatenate([Y] * 32, axis=0)
17 print("Shape của Y sau 32 lần lặp")
18 print(Y.shape)
```

✓ 0.0s Python

Shape của X và Y là:
(32, 10) (10,)

Shape của Y sau khi thêm 1 trục rỗng
(1, 10)
Thu được Y có shape là (1,10)

Shape của Y sau 32 lần lặp
(32, 10)

Tại thời điểm này, chúng ta có thể tiến hành cộng X và Y, vì chúng có cùng hình dạng.

Thực tế, việc lặp lại này chỉ diễn ra ở mức độ thuật toán chứ không diễn ra ở cấp độ bộ nhớ, nghĩa là chúng ta chỉ việc thực hiện luôn tất cả các phép toán mà không cần thực hiện việc chuẩn hóa vector Y như ở trên

Ví dụ sau áp dụng phép toán maximum (relu) theo từng phần tử cho hai tensor có hình dạng khác nhau thông qua broadcasting:

```
1 import numpy as np
2 x = np.random.random((64, 3, 32, 10))
3 y = np.random.random((32, 10))
4 z = np.maximum(x, y)
5 print(x.shape, y.shape, z.shape)
```

✓ 0.0s Python

(64, 3, 32, 10) (32, 10) (64, 3, 32, 10)

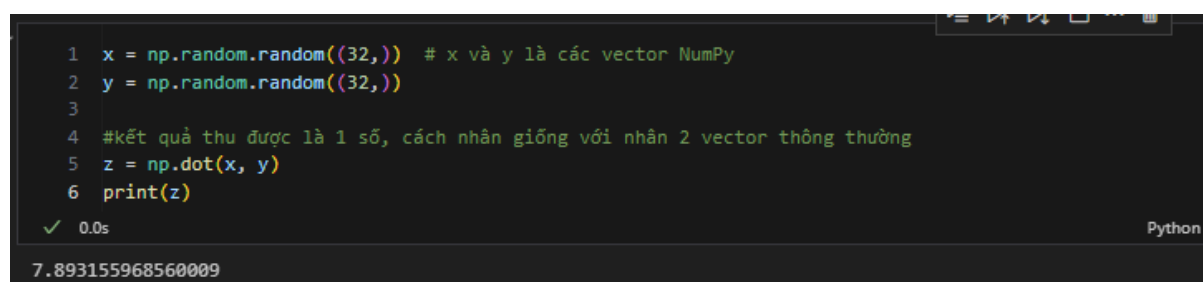
Dễ thấy shape của z sẽ giống shape của x, mặc dù shape y không thay đổi

2.3.3 Tích tensor (Tensor Product)

Tích tensor, hay còn gọi là tích vô hướng (dot product) (không nên nhầm lẫn với tích theo từng phần tử, toán tử $*$), là một trong những phép toán tensor phổ biến và hữu ích nhất.

Trong NumPy, tích tensor được thực hiện bằng hàm `np.dot` (vì ký hiệu toán học cho tích tensor thường là một dấu chấm), tuy nhiên shape của `x`, `y` phải tương thích (ví dụ giống như ma trận $n \times k$ thì phải nhân với ma trận dạng $k \times m$ chứ không thể nào là $j \times m$)

Ví dụ nhân 2 vector: (tương tự với nhân 2 ma trận,... giống như trong toán học)



```
1 x = np.random.random((32,)) # x và y là các vector NumPy
2 y = np.random.random((32,))
3
4 #kết quả thu được là 1 số, cách nhân giống với nhân 2 vector thông thường
5 z = np.dot(x, y)
6 print(z)
```

✓ 0.0s Python

7.893155968560009

(kết quả `z` có thể thay đổi vì `x`, `y` là random)

Lưu ý: nếu $\text{ndim} > 1$ thì `dot(x,y)` khác `dot(y,x)`

2.3.4 Phép reshape (định hình lại)

Định dạng lại một tensor có nghĩa là sắp xếp lại các hàng và cột của nó để khớp với một hình dạng mục tiêu. Tất nhiên, tensor sau khi định dạng lại sẽ có cùng tổng số hệ số như tensor ban đầu. Định dạng lại được hiểu rõ nhất thông qua các ví dụ đơn giản:

```
1 import numpy as np
2
3 # Tạo ma trận x ban đầu
4 x = np.array([[0., 1.],
5               [2., 3.],
6               [4., 5.]])
7 print("Hình dạng ban đầu của x:", x.shape) # (3, 2)
8
9 # Định dạng lại thành (6, 1)
10 x = x.reshape((6, 1))
11 print("x sau khi định dạng lại thành (6, 1):\n", x)
12
13 # Định dạng lại thành (2, 3)
14 x = x.reshape((2, 3))
15 print("x sau khi định dạng lại thành (2, 3):\n", x)
16
17 # Chuyển vị ma trận x
18 x = np.transpose(x)
19 print("Hình dạng của x sau khi chuyển vị:", x.shape) # (3, 2)
```

✓ 0.0s Python

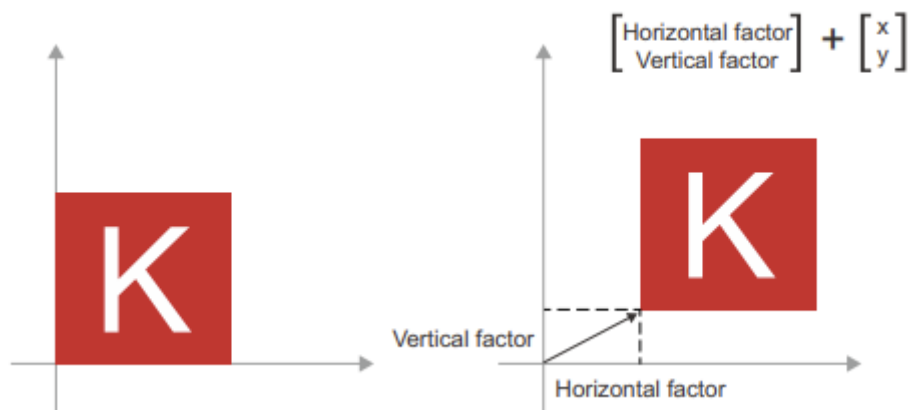
Hình dạng ban đầu của x: (3, 2)
x sau khi định dạng lại thành (6, 1):
[[0.]
[1.]
[2.]
[3.]
[4.]
[5.]]
x sau khi định dạng lại thành (2, 3):
[[0. 1. 2.]
[3. 4. 5.]]
Hình dạng của x sau khi chuyển vị: (3, 2)

2.3.5 Giải thích hình học của các phép toán tensor

Phần này không quá quan trọng, chỉ cần tóm gọn lại như sau:

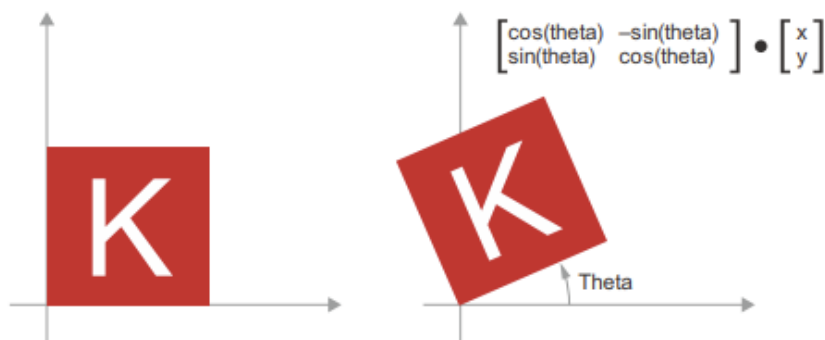
Các phép toán hình học cơ bản như dịch chuyển, xoay, co giãn, nghiêng, v.v. có thể được biểu diễn dưới dạng các phép toán tensor. Dưới đây là một vài ví dụ:

- + **Dịch chuyển (Translation)** Việc cộng một vector vào một điểm sẽ di chuyển điểm đó một khoảng cố định theo một hướng cố định. Khi áp dụng cho một tập hợp các điểm (như một đối tượng 2D), điều này được gọi là "dịch chuyển" (xem hình 2.9).



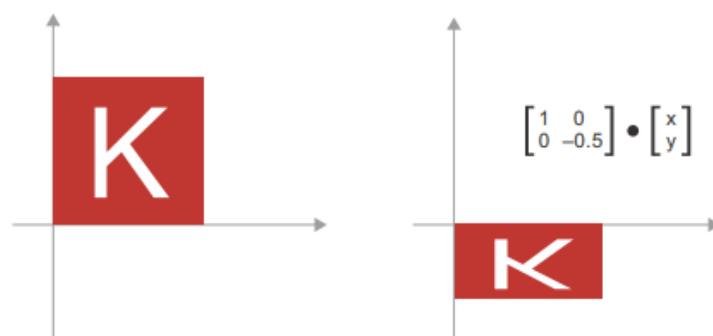
Hình 2.9

- + **Xoay (Rotation):** Một phép xoay ngược chiều kim đồng hồ của một vector 2D theo góc theta (xem hình 2.10) có thể được thực hiện thông qua một tích vô hướng với một ma trận 2×2 $R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$.



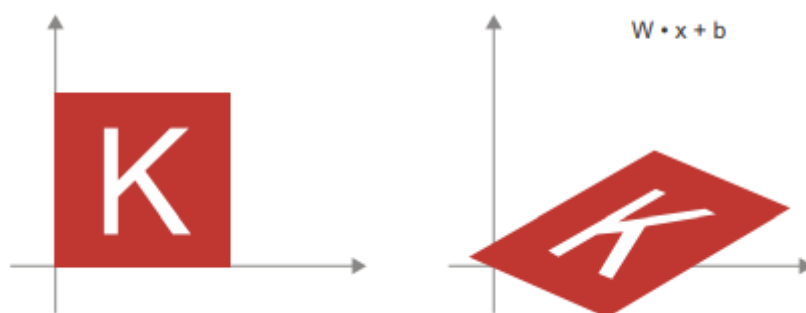
Hình 2.10

- + **Co giãn (Scaling):** Một phép co giãn theo chiều dọc và chiều ngang của hình ảnh (xem hình 2.11) có thể được thực hiện thông qua một tích vô hướng với một ma trận 2×2 $S = \begin{bmatrix} \text{độ_co_dãn_theo_trục_x} & 0 \\ 0 & \text{độ_co_dãn_theo_trục_y} \end{bmatrix}$



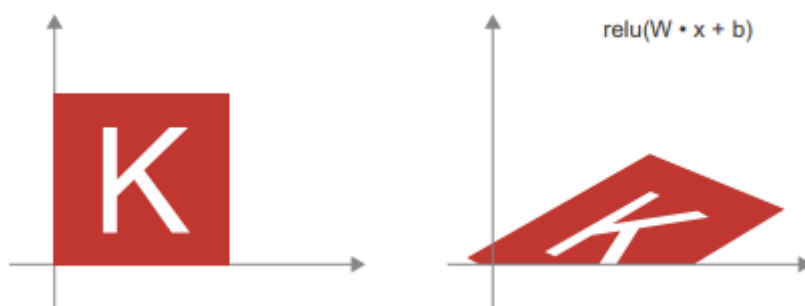
Giải thích: theo trục $x = 1 \Rightarrow$ không co dãn, theo trục $y = -0.5 \Rightarrow$ co 1 nửa và hướng theo đối xứng với vị trí hiện tại)

- + **Biến đổi tuyến tính (Linear transform)**: Một tích vô hướng với một ma trận bất kỳ sẽ thực hiện một biến đổi tuyến tính. Lưu ý rằng co giãn và xoay, như đã liệt kê trước đó, theo định nghĩa là các biến đổi tuyến tính.
- + **Biến đổi affine (Affine transform)**: Một biến đổi affine (xem hình 2.12) là sự kết hợp của một biến đổi tuyến tính (được thực hiện thông qua một tích vô hướng với một ma trận nào đó) và một phép dịch chuyển (được thực hiện thông qua một phép cộng vector). Như bạn có thể đã nhận ra, đó chính là phép tính $y = W \cdot x + b$ được triển khai bởi tầng Dense. Một tầng Dense không có hàm kích hoạt là một tầng affine.



Hình 2.12

- + **Tầng Dense với hàm kích hoạt ReLU**: Giống với affine, nhưng relu sẽ loại đi phần giá trị âm



2.3.6 Giải thích hình học của học sâu

Học sâu có thể minh họa thông qua ví dụ như sau: có 2 tờ giấy màu xanh và đỏ, chúng bị vò vào với nhau thành một quả bóng. Quả bóng giấy này chính là dữ liệu đầu vào, và mỗi tờ giấy đại diện cho một lớp dữ liệu trong bài toán phân loại. Điều mà một mạng nơ-ron cần làm là tìm ra một phép biến đổi của quả bóng giấy này để làm phẳng nó ra, sao cho hai lớp dữ liệu có thể được tách biệt rõ ràng một lần nữa. Việc làm phẳng các quả bóng giấy chính là mục tiêu

của học máy: tìm ra các biểu diễn gọn gàng cho các đa tạp dữ liệu phức tạp, bị gấp nếp nhiều trong không gian đa chiều. Học sâu vượt trội trong việc này vì nó áp dụng cách tiếp cận phân tách một biến đổi hình học phức tạp thành một chuỗi dài các biến đổi cơ bản, điều này khá giống với chiến lược mà con người sẽ dùng để làm phẳng một quả bóng giấy. Mỗi tầng trong một mạng sâu áp dụng một phép biến đổi để làm rõ dữ liệu một chút, và một chuỗi các tầng sâu sẽ làm cho quá trình làm rõ phức tạp này trở nên khả thi.

2.4 Động cơ của mạng nơ-ron: tối ưu hóa dựa trên gradient

Ở phần trước, mỗi tầng nơ-ron trong ví dụ mô hình đầu tiên của chúng ta biến đổi dữ liệu đầu vào nhờ hàm $\text{relu}(\text{dot}(\text{input}, W) + b)$, với: W và b là các tham số có thể huấn luyện của tầng. Các trọng số này chứa thông tin mà mô hình học được từ việc tiếp xúc với dữ liệu huấn luyện.

Ban đầu: W và b là ngẫu nhiên, khi đó, các biểu diễn đầu ra không thể hữu ích được. Điều tiếp theo là dần dần điều chỉnh các trọng số này, dựa trên một tín hiệu phản hồi. Sự điều chỉnh dần dần này, còn được gọi là **huấn luyện**, chính là quá trình học mà học máy hướng tới.

Quá trình này diễn ra trong một **vòng lặp huấn luyện**, hoạt động như sau. Lặp lại các bước này trong một vòng lặp cho đến khi giá trị mất mát (loss) đủ thấp:

1. Lấy một lô dữ liệu huấn luyện x và các mục tiêu tương ứng y_{true} .
2. Chạy mô hình trên x (một bước gọi là **lan truyền xuôi - forward pass**) để thu được dự đoán y_{pred} .
3. Tính giá trị mất mát (loss) của mô hình trên lô dữ liệu để tính toán sự khác nhau giữa y_{pred} và y_{true} .
4. Cập nhật tất cả trọng số của mô hình theo cách làm giảm giá trị mất mát dần dần.

Cuối cùng, bạn sẽ có một mô hình có giá trị mất mát rất thấp trên dữ liệu huấn luyện: sự khác biệt giữa dự đoán y_{pred} và mục tiêu mong đợi y_{true} sẽ rất thấp. Mô hình đã "học" được cách ánh xạ đầu vào của nó với các mục tiêu chính xác.

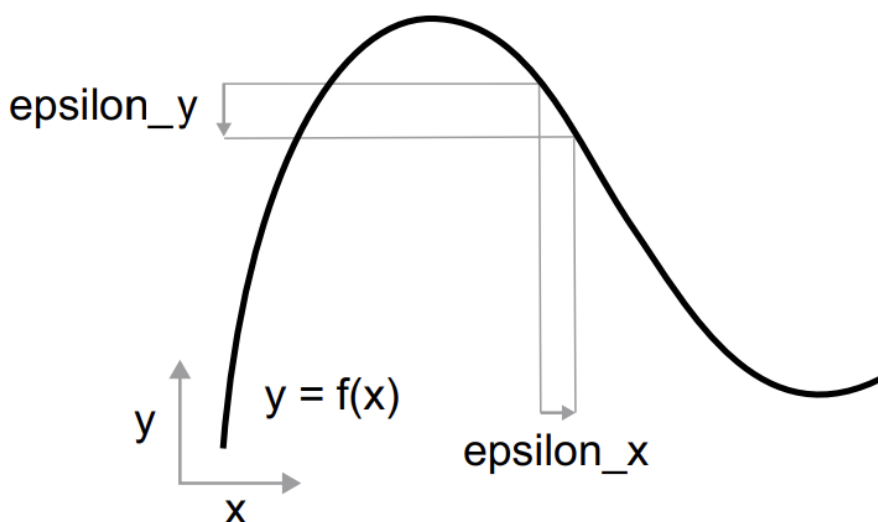
Bước 1 chỉ đơn giản là nhập/ xuất dữ liệu, bước 2,3 thì là một số phép tính toán tensor. Bước khó nhất ở đây là bước 4: cập nhật trọng số của mô hình. Với một hệ số trọng số riêng lẻ trong mô hình, làm thế nào bạn có thể tính toán liệu hệ số đó nên được tăng hay giảm, và tăng/giảm bao nhiêu?

Có một cách tiếp cận rất tốt để giải quyết vấn đề này: hạ gradient (gradient descent) Tổng quát về hạ gradient như sau:

Tất cả các hàm được sử dụng trong mô hình của chúng ta (như dot hoặc +) biến đổi đầu vào của chúng một cách mượt mà và liên tục: ví dụ, nếu bạn xét $z = x + y$, một thay đổi nhỏ ở y chỉ dẫn đến một thay đổi nhỏ ở z , và nếu bạn biết hướng của thay đổi ở y , bạn có thể suy ra hướng của thay đổi ở z . Về mặt toán học, bạn sẽ nói rằng các hàm này là **khả vi (differentiable)**. Nếu bạn kết hợp các hàm như vậy lại với nhau, hàm lớn hơn mà bạn thu được vẫn là khả vi. Đặc biệt, điều này áp dụng cho hàm ánh xạ các hệ số của mô hình đến giá trị mất mát của mô hình trên một lô dữ liệu: một thay đổi nhỏ trong các hệ số của mô hình dẫn đến một thay đổi nhỏ, có thể dự đoán được trong giá trị mất mát. Điều này cho phép bạn sử dụng một toán tử toán học gọi là **gradient** để mô tả cách giá trị mất mát thay đổi khi bạn di chuyển các hệ số của mô hình theo các hướng khác nhau. Tính toán được gradient cho biết hướng thay đổi trọng số để giảm mất mát, cho phép cập nhật tất cả trọng số cùng lúc.

2.4.1 Sơ qua về đạo hàm

Hãy xem xét một hàm liên tục, mượt mà $f(x) = y$, ánh xạ một số x thành một số mới y . Vì hàm này là liên tục, một thay đổi nhỏ ở x chỉ có thể dẫn đến một thay đổi nhỏ ở y nếu bạn tăng x lên một lượng nhỏ ϵ_x : điều này dẫn đến một thay đổi nhỏ ϵ_y ở y .



Mô tả: x tăng 1 đoạn nhỏ ϵ_x thì y giảm 1 đoạn nhỏ ϵ_y

Vì ϵ_x rất nhỏ \Rightarrow đoạn đồ thị từ $f(x)$ đến $f(x + \epsilon_x)$ cũng rất nhỏ \Rightarrow có thể coi là đoạn thẳng có độ dốc a . Đơn giản hơn, có thể thấy rõ ràng được sự thay đổi của y khi x thay đổi thông qua đạo hàm và độ dốc của nó. Xác

định được chiều thay đổi của x có thể dễ dàng xác định được chiều thay đổi của y (đạo hàm $f'(x)$ dương thì y tăng khi x tăng và $f'(x)$ âm thì y giảm khi x tăng). Độ dốc của đạo hàm cho biết được độ thay đổi nhanh chậm của y khi x thay đổi (càng dốc thì y thay đổi càng nhanh khi x thay đổi, còn càng thoải thì y thay đổi càng chậm).

Vậy khái niệm khả vi ở phần trên là gì? Khả vi là có thể lấy đạo hàm, nghĩa là tồn tại đạo hàm $f'(x)$ của $f(x)$ tại điểm x đó.

=> Sử dụng đạo hàm là một phương pháp mạnh mẽ để tối ưu hóa, bởi vì khi thay đổi x và biết đạo hàm của $f(x)$ thì nếu $f(x)$ thay đổi quá nhanh, chỉ cần di chuyển x ngược lại 1 chút là được.

2.4.2 Đạo hàm của 1 phép toán tensor : gradient

Khái niệm đạo hàm có thể được áp dụng cho bất kỳ hàm nào miễn là các bề mặt mà chúng mô tả là liên tục và mượt mà. Đạo hàm của một phép toán tensor (hoặc hàm tensor) được gọi là gradient. **Gradient là đạo hàm dành cho các hàm nhận tensor làm đầu vào.** Gradient của một hàm tensor biểu diễn độ cong (nhiều hay ít) của bề mặt đa chiều được mô tả bởi hàm đó => giống với độ dốc ở phần trên. Nó mô tả cách đầu ra của hàm thay đổi khi các tham số đầu vào của nó thay đổi.

Hãy xem xét một ví dụ cụ thể trong học máy. Xét:

- + Một vector đầu vào x (một mẫu trong tập dữ liệu).
- + Một ma trận W (trọng số của mô hình).
- + Một mục tiêu y_true (điều mà mô hình cần học để liên kết với x).
- + Một hàm mất mát loss (dùng để đo khoảng cách giữa dự đoán hiện tại của mô hình và y_true).

Bạn có thể sử dụng W để tính một dự đoán y_pred , sau đó tính giá trị mất mát giữa dự đoán y_pred và mục tiêu y_true (pred: prediction)

```
y_pred = dot(W, x) # Chúng ta sử dụng trọng số mô hình W để dự đoán cho x
loss_value = loss(y_pred, y_true) # Chúng ta đo lường độ lệch của dự đoán
```

Python

Bây giờ chúng ta muốn sử dụng gradient để tìm cách cập nhật W sao cho làm giảm giá trị mất mát. Làm thế nào để thực hiện điều này?

Cũng giống như với một hàm $f(x)$, **bạn có thể giảm giá trị của $f(x)$ bằng cách di chuyển x một chút theo hướng ngược lại với đạo hàm**, với một hàm

$f(W)$ của một tensor, **bạn có thể giảm $\text{loss_value} = f(W)$ bằng cách di chuyển W theo hướng ngược lại với gradient.** Điều đó có nghĩa là đi ngược lại hướng tăng nhanh nhất của f , điều này trực giác sẽ đưa bạn đến vị trí thấp hơn trên đường cong.

Giải thích bằng công thức:

- + Giả sử hàm mất mát có dạng sau: loss_value (giá trị mất mát) $= f(X) \Rightarrow$ gọi gradient của hàm $f(W)$ tại W_0 là $\text{grad}(\text{loss_value}, W_0)$, mô tả độ dốc tăng giảm của hàm mất mát.
- + Vậy nếu khi gradient thay đổi quá nhanh \Rightarrow chỉ cần đi ngược lại là được, W_1 mới sẽ được điều chỉnh theo công thức
$$W_1 = W_0 - \text{step} * \text{grad}(f(W_0), W_0)$$
 với step rất nhỏ.

2.4.3 Hạ gradient ngẫu nhiên

Với hàm khả vi, có thể tìm được cực tiểu của nó nhờ phân tích. Điểm cực tiểu của 1 hàm là điểm ở đó đạo hàm bằng 0 \Rightarrow cần đi tìm tất cả các điểm có đạo hàm bằng 0 và kiểm tra xem điểm nào ở đó $f(x)$ có giá trị thấp nhất

Áp dụng cho mạng nơ ron: tìm ra cách phân tích, tìm trọng số sao cho hàm mất mát đạt giá trị bé nhất có thể. Để làm điều này, có thể sử dụng thuật toán bốn bước được nêu ở bên trên (xem ngay dưới phần 2.4) để nhằm điều chỉnh các tham số từng chút một dựa trên giá trị mất mát hiện tại cho một lô dữ liệu ngẫu nhiên. Vì bạn đang làm việc với một hàm khả vi, bạn có thể tính gradient của nó, điều này cung cấp cho bạn một cách hiệu quả để triển khai bước 4. Nếu bạn cập nhật trọng số theo hướng ngược lại với gradient, giá trị mất mát sẽ giảm đi một chút mỗi lần.

1. Lấy một lô dữ liệu huấn luyện x và các mục tiêu tương ứng y_{true} .
2. Chạy mô hình trên x để thu được dự đoán y_{pred} (bước này được gọi là lan truyền xuôi - forward pass).
3. Tính giá trị mất mát của mô hình trên lô dữ liệu, hay còn gọi là độ sai lệch giữa y_{pred} và y_{true} .
4. Tính gradient của giá trị mất mát đối với các tham số của mô hình (bước này được gọi là lan truyền ngược - backward pass).
5. Di chuyển các tham số một chút theo hướng ngược lại với gradient, ví dụ, $W = W - \text{learning_rate} * \text{gradient}$ để giảm giá trị mất mát trên lô dữ liệu một chút. Tỷ lệ học (learning rate) ở đây là một hệ số vô hướng điều chỉnh "tốc độ" của quá trình hạ gradient.

Những gì vừa được mô tả được gọi là **hạ gradient ngẫu nhiên theo lô nhỏ (mini-batch stochastic gradient descent - mini-batch SGD)**. Thuật ngữ "ngẫu nhiên" (stochastic) ám chỉ việc mỗi lô dữ liệu được lấy ngẫu nhiên

Việc chọn một giá trị hợp lý cho learning rate là rất quan trọng. Nếu nó quá nhỏ, việc hạ xuống đường cong sẽ cần nhiều lần lặp và có thể bị kẹt ở một cực tiểu cục bộ. Nếu learning_rate quá lớn, các cập nhật có thể nhảy đến những vị trí hoàn toàn ngẫu nhiên trên đường cong.

SGD thật sự là lấy một mẫu và mục tiêu duy nhất ngẫu nhiên tại mỗi lần lặp, thay vì lấy một lô dữ liệu, nó chính xác tuy nhiên tốn kém hơn. Có một cách cân bằng hơn là sử dụng những lô nhỏ với kích thước hợp lý

Trong thực tế, bạn sẽ sử dụng hạ gradient trong không gian đa chiều: mỗi hệ số trọng số trong mạng nơ-ron là một chiều tự do trong không gian, và có thể có hàng chục nghìn hoặc thậm chí hàng triệu chiều như vậy.

Ngoài ra, còn tồn tại nhiều biến thể của SGD khác nhau, chúng khác nhau ở chỗ lấy vào tài khoản các cập nhật trọng số trước đó khi tính toán cập nhật trọng số tiếp theo, thay vì chỉ nhìn vào giá trị gradient hiện tại. Ví dụ, có SGD với xung lượng (momentum), cũng như Adagrad, RMSprop, và một số biến thể khác. Các biến thể như vậy được gọi là **phương pháp tối ưu hóa** hoặc **bộ tối ưu hóa (optimizers)**. Đặc biệt, khái niệm xung lượng, được sử dụng trong nhiều biến thể này, đáng để bạn chú ý. Xung lượng giải quyết hai vấn đề của SGD: tốc độ hội tụ và các cực tiểu cục bộ. Hãy xem xét hình 2.20, minh họa đường cong của giá trị mất mát theo một tham số của mô hình.

Như bạn có thể thấy, quanh một giá trị tham số nhất định, có một **cực tiểu cục bộ**: quanh điểm đó, di chuyển sang trái sẽ làm tăng giá trị mất mát, nhưng di chuyển sang phải cũng vậy. Nếu tham số đang được tối ưu hóa bằng SGD với một tỷ lệ học nhỏ, quá trình tối ưu hóa có thể bị kẹt tại cực tiểu cục bộ thay vì tiến đến cực tiểu toàn cục.

Bạn có thể tránh những vấn đề như vậy bằng cách sử dụng xung lượng, lấy cảm hứng từ vật lý. Một hình ảnh tinh thần hữu ích ở đây là nghĩ về quá trình tối ưu hóa như một quả bóng nhỏ lăn xuống đường cong mất mát. Nếu nó có đủ xung lượng, quả bóng sẽ không bị kẹt trong một khe hẹp và sẽ đến được cực tiểu toàn cục. Xung lượng được triển khai bằng cách di chuyển quả bóng tại mỗi bước không chỉ dựa trên giá trị độ dốc hiện tại (gia tốc hiện tại) mà còn dựa trên vận tốc hiện tại (kết quả của các gia tốc trước đó). Trong thực tế, điều này có nghĩa là cập nhật tham số w không chỉ dựa trên giá trị gradient hiện tại mà còn dựa trên cập nhật tham số trước đó, như trong triển khai đơn giản sau:

2.4.4 Xâu chuỗi các đạo hàm: Thuật toán Backpropagation (thuật toán lan truyền ngược)

Trong thuật toán trước, chúng ta đã giả định một cách đơn giản rằng vì một hàm là khả vi, chúng ta có thể dễ dàng tính gradient của nó. Nhưng điều đó có đúng không? Làm thế nào để chúng ta tính gradient của các biểu thức phức tạp trong thực tế Đó là lúc thuật toán **Lan truyền ngược (Backpropagation)** xuất hiện.

* Quy tắc chuỗi

Ý tưởng cơ bản: Nếu bạn có một hàm phức tạp được tạo thành từ nhiều hàm đơn giản lồng nhau, bạn có thể tính đạo hàm của nó bằng cách nhân các đạo hàm riêng lẻ của từng hàm con. Đây là nền tảng của Backpropagation.

Ví dụ đơn giản: Giả sử bạn có hai hàm f và g , và hàm tổng hợp $fg(x) = f(g(x))$. Theo quy tắc chuỗi: đạo hàm của y theo x là: $\text{grad}(y, x) = \text{grad}(y, x1) * \text{grad}(x1, x)$; trong đó $x1 = g(x)$ và $y = f(x1)$.

Áp dụng vào mạng nơ-ron: Trong mạng nơ-ron, bạn có một chuỗi các phép toán tensor (như nhân ma trận dot, hàm kích hoạt relu, softmax, cộng +, v.v.) được lồng ghép với nhau. Ví dụ, hàm mất mát của một mô hình hai tầng có thể viết là:

$\text{loss_value} = \text{loss}(y_true, \text{softmax}(\text{dot}(\text{relu}(\text{dot}(\text{inputs}, W1) + b1), W2) + b2))$.

Mỗi phép toán trong đó đều có đạo hàm đơn giản và đã biết, nên ta có thể dùng quy tắc chuỗi để tính đạo hàm của loss_value theo các trọng số $W1, b1, W2, b2$.

* Đồ thị tính toán (Computation Graphs)

Đồ thị tính toán là gì?: Đây là một cấu trúc dữ liệu dạng đồ thị có hướng không vòng (directed acyclic graph - DAG), biểu diễn các phép toán tensor trong mạng nơ-ron. Mỗi nút (node) là một phép toán hoặc biến, và các cạnh (edge) thể hiện luồng dữ liệu giữa chúng. TensorFlow dùng đồ thị này để thực hiện tính toán.

Ví dụ: Với mô hình hai tầng, đồ thị tính toán có thể gồm các nút như $\text{dot}(\text{inputs}, W1)$, $+ b1$, relu , $\text{dot}(\cdot, W2)$, $+ b2$, softmax , và cuối cùng là loss . (Xem Hình 2.21 trong sách).

Tại sao hữu ích?: Đồ thị tính toán biến phép tính thành dữ liệu mà máy có thể xử lý. Nó cho phép:

- + Tự động hóa việc tính đạo hàm (automatic differentiation).

- + Phân phối tính toán trên nhiều máy mà không cần viết lại mã.

Ví dụ đơn giản hơn: Để dễ hiểu, sách đưa ra một đồ thị cơ bản với các biến vô hướng:

- + Đầu vào: x , trọng số w , bias b , mục tiêu y_true .
- + Phép toán: $x1 = x * w$, $x2 = x1 + b$, $loss_val = abs(y_true - x2)$.
- + Mục tiêu: Tính $grad(loss_val, w)$ và $grad(loss_val, b)$.

* Lan truyền xuôi (Forward Pass)

Cách hoạt động: Gán giá trị cụ thể cho các biến đầu vào ($x = 2$, $w = 3$, $b = 1$, $y_true = 4$), rồi tính lần lượt qua từng nút trong đồ thị:

- + $x1 = x * w = 2 * 3 = 6$.
- + $x2 = x1 + b = 6 + 1 = 7$.
- + $loss_val = abs(y_true - x2) = abs(4 - 7) = 3$.

Kết quả: Bạn có giá trị của $loss_val$ sau khi lan truyền xuôi.

* Lan truyền ngược (Backward Pass) - Backpropagation

Ý tưởng: Đi ngược từ $loss_val$ về các biến đầu vào để tính đạo hàm từng bước:

- + Tạo một đồ thị ngược, hỏi: “Khi A thay đổi, B thay đổi bao nhiêu?” (tức là $grad(B, A)$).
- + Dùng quy tắc chuỗi để nhân các đạo hàm dọc theo đường đi từ $loss_val$ đến biến cần tính (như w hoặc b).

Tính toán cụ thể:

- + $grad(loss_val, x2)$: Nếu $x2$ thay đổi một lượng nhỏ ϵ , thì $loss_val = abs(4 - x2)$ cũng thay đổi cùng lượng đó $\rightarrow grad(loss_val, x2) = 1$.
- + $grad(x2, x1)$: $x2 = x1 + b$, nên khi $x1$ thay đổi ϵ , $x2$ cũng thay đổi $\epsilon \rightarrow grad(x2, x1) = 1$.
- + $grad(x2, b)$: Tương tự, khi b thay đổi ϵ , $x2$ thay đổi $\epsilon \rightarrow grad(x2, b) = 1$.
- + $grad(x1, w)$: $x1 = x * w = 2 * w$, nên khi w thay đổi ϵ , $x1$ thay đổi $2 * \epsilon \rightarrow grad(x1, w) = 2$.

Kết hợp bằng quy tắc chuỗi:

$$+ \text{grad}(\text{loss_val}, w) = \text{grad}(\text{loss_val}, x2) * \text{grad}(x2, x1) * \text{grad}(x1, w) = 1 * 1 * 2 = 2.$$

$$+ \text{grad}(\text{loss_val}, b) = \text{grad}(\text{loss_val}, x2) * \text{grad}(x2, b) = 1 * 1 = 1.$$

Nó bắt đầu từ `loss_val`, đi ngược về các tham số, tính xem mỗi tham số đóng góp bao nhiêu vào mất mát.

* Tự động hóa với TensorFlow

Ngày xưa: Trước đây, người ta phải tự tay viết công thức đạo hàm

Bây giờ: Các framework như TensorFlow tự động tính đạo hàm (automatic differentiation) bằng đồ thị tính toán. Bạn chỉ cần viết lan truyền xuôi, phần còn lại để framework lo.

GradientTape trong TensorFlow: Đây là công cụ chính để tận dụng tính năng này:

- + Nó ghi lại mọi phép toán tensor trong một “băng” (tape).
- + Sau đó, bạn có thể yêu cầu tính đạo hàm của bất kỳ đầu ra nào theo bất kỳ biến nào (`tf.Variable`).

```

import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)

```

Instantiate a scalar Variable with an initial value of 0.

Open a GradientTape scope.

Inside the scope, apply some tensor operations to our variable.

Use the tape to retrieve the gradient of the output y with respect to our variable x.

The GradientTape works with tensor operations:

```

x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)

```

Instantiate a Variable with shape (2, 2) and an initial value of all zeros.

`grad_of_y_wrt_x` is a tensor of shape (2, 2) (like `x`) describing the curvature of $y = 2 * x + 3$ around $x = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$.

Với tensor phức tạp hơn: GradientTape cũng hoạt động với ma trận hoặc danh sách biến:

```

W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])

```

`matmul` is how you say “dot product” in TensorFlow.

`grad_of_y_wrt_W_and_b` is a list of two tensors with the same shapes as `W` and `b`, respectively.

CHƯƠNG 3: GIỚI THIỆU VỀ KERAS VÀ TENSORFLOW

3.1 Giới thiệu về Tensorflow và Keras

3.1.1 Tensorflow

Tensorflow là một nền tảng học máy mã nguồn mở, miễn phí, dựa trên Python, được phát triển chủ yếu bởi Google, cho phép các kỹ sư và nhà nghiên cứu thao tác với các biểu thức toán học trên các tensor số.

Tensorflow có một số ưu điểm vượt xa phạm vi của NumPy như sau:

- + Nó có thể tự động tính gradient của bất kỳ biểu thức khả vi nào (như bạn đã thấy ở chương 2), khiến nó rất phù hợp cho học máy.
- + Nó có thể chạy không chỉ trên CPU, mà còn trên GPU và TPU, các bộ tăng tốc phần cứng song song cao.
- + Các phép tính được định nghĩa trong TensorFlow có thể dễ dàng được phân tán trên nhiều máy.
- + Các chương trình TensorFlow có thể được xuất sang các môi trường chạy khác, như C++, JavaScript (cho các ứng dụng dựa trên trình duyệt), hoặc TensorFlow Lite (cho các ứng dụng chạy trên thiết bị di động hoặc thiết bị nhúng), v.v. Điều này khiến các ứng dụng TensorFlow dễ dàng triển khai trong các tình huống thực tế.

3.1.2 Keras

Keras là một API học sâu cho Python, được xây dựng trên TensorFlow, cung cấp một cách thuận tiện để định nghĩa và huấn luyện bất kỳ loại mô hình học sâu nào. Keras được phát triển cho nghiên cứu, với mục tiêu cho phép thử nghiệm học sâu nhanh chóng.

=> Có thể nói: Keras như Python của deep learning

3.1.3 Thiết lập môi trường học sâu

Để thực hiện học sâu trên GPU, có 3 lựa chọn:

- + Mua và cài đặt GPU NVIDIA vật lý trên máy tính
- + Sử dụng GPU trên đám mây của những công ty như Google (Google Cloud) hay Amazon (AWS EC2)
- + Sử dụng GPU miễn phí từ Colaboratory của Google (chạy Jupyter Notebook – ipynb)

=> Với nội dung trong sách: dùng Colab là đủ, nhưng với công việc nặng sau này, nên dùng cách 1 hoặc 2.

3.2 Bắt đầu với Tensorflow

3.2.1 Khởi tạo tensor và biến

Để sử dụng bất cứ điều gì, trong tensorflow, cần khởi tạo tensor, có thể là ngẫu nhiên, toàn số 0 hoặc toàn số 1, xem ví dụ code ở đây

```
import tensorflow as tf
x = tf.ones(shape=(2, 1)) # Tương đương với np.ones(shape=(2, 1))
print("Tensor toàn 1")
print(x)
print()

x = tf.zeros(shape=(2, 1)) # Tương đương với np.zeros(shape=(2, 1))
print("Tensor toàn 0")
print(x)
print()

x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
print("Tensor ngẫu nhiên với trung bình = 0 và độ lệch chuẩn 1")
print(x)
print()

x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
print("Tensor ngẫu nhiên với giá trị từ 0 đến 1")
print(x)
print()
```

```
Tensor toàn 1
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)

Tensor toàn 0
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)

Tensor ngẫu nhiên với trung bình = 0 và độ lệch chuẩn 1
tf.Tensor(
[[1.5279003]
 [1.1996255]
 [0.5056595]], shape=(3, 1), dtype=float32)

Tensor ngẫu nhiên với giá trị từ 0 đến 1
tf.Tensor(
[[0.28542173]
 [0.53997195]
 [0.289209  ]], shape=(3, 1), dtype=float32)
```

Tensor không thể gán giá trị mới, mà nó là 1 hằng số. Do đó, muốn thay đổi giá trị, cần dùng đến lớp Variable (biến), và dùng phương thức assign để thay đổi giá trị

Có thể dùng phương thức như assign với 1 phần tử trong tensor, cả 1 tensor, hoặc dùng assign_add, assign_sub để cộng trừ

```

v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
print("In ra tensor v")
print(v)
print()

#thay đổi v thành tensor toàn 1
v.assign(tf.ones((3, 1)))
print("tensor v sau khi thay bằng 1 tensor toàn 1")
print(v)
print()

#Thay đổi một phần tử duy nhất của tensor v (giả sử phần tử [0,0])
v[0,0].assign(3.)
print("tensor v sau khi thay đổi phần tử [0,0] thành 3")
print(v)
print()

# assign add (cộng), sub(trừ)
v.assign_add(tf.ones((3,1)))
print('tensor v sau khi cộng với tensor toàn 1')
print(v)
print()

v.assign_sub(tf.ones((3,1)))
print('tensor v sau khi trừ với tensor toàn 1')
print(v)
print()

```

```

In ra tensor v
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[ 1.574962 ],
       [-0.637406 ],
       [ 0.90942764]], dtype=float32)>

tensor v sau khi thay bằng 1 tensor toàn 1
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>

tensor v sau khi thay đổi phần tử [0,0] thành 3
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[3.],
       [1.],
       [1.]], dtype=float32)>

tensor v sau khi cộng với tensor toàn 1
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[4.],
       [2.],
       [2.]], dtype=float32)>

tensor v sau khi trừ với tensor toàn 1
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[3.],
       [1.],
       [1.]], dtype=float32)>

```

3.2.2 Các phép toán tensor

Có thể thực hiện các phép toán với tensor như cộng (+), bình phương (square), căn bậc 2 (sqrt), matmul(nhân vô hướng), nhân theo phần tử (*=)

```

a = tf.ones((2, 2))
print(a)
print()
a*=2
print(a)
print()
b = tf.square(a)
print(b)
print()
c = tf.sqrt(a)
print(c)
print()
d = b + c
print(d)
print()
e = tf.matmul(a, b)
print(e)
print()

```

```

tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)

tf.Tensor(
[[2. 2.]
 [2. 2.]], shape=(2, 2), dtype=float32)

tf.Tensor(
[[4. 4.]
 [4. 4.]], shape=(2, 2), dtype=float32)

tf.Tensor(
[[1.4142135 1.4142135]
 [1.4142135 1.4142135]], shape=(2, 2), dtype=float32)

tf.Tensor(
[[5.4142137 5.4142137]
 [5.4142137 5.4142137]], shape=(2, 2), dtype=float32)

tf.Tensor(
[[16. 16.]
 [16. 16.]], shape=(2, 2), dtype=float32)

```

3.2.3 Gradient Tape API

Ở phần trước, đã biết được tính gradient thông qua hàm `grad(loss, W0)` – tính đạo hàm của tensor, vậy thực hiện trong code như nào?

=> Sử dụng `gradient = tape.gradient(loss, weight)`

Với biến có thể huấn luyện, nó mặc định được theo dõi, nhưng với biến là tensor hằng số => cần đánh dấu bằng `tape.watch(tensor_name)` để tránh lãng phí tài nguyên

Ngoài ra có thể tính được đạo hàm cấp 2 bằng cách sử dụng gradient lồng vào nhau, ví dụ tính vận tốc v và gia tốc a theo vị trí s theo thời gian t :

```
v = inner_tape.gradient(s,t)
```

```
a = outer_tape.gradient(v,t)
```

3.2.4 Ví dụ về phân loại tuyến tính

Vậy chúng ta đã tính được gradient, là đủ để xây dựng một mô hình học máy dựa trên hạ gradient

Ví dụ: triển khai một bộ phân loại tuyến tính từ đầu trong TensorFlow:

Bước 1: tạo dữ liệu là 2 đám mây chứa các điểm để phân loại nhãn 0-1



Bước 2: Tạo một bộ phân loại tuyến tính có thể học cách tách biệt hai đám điểm này. Bộ phân loại tuyến tính là một phép biến đổi affine (prediction = $W \cdot \text{input} + b$) được huấn luyện để giảm thiểu bình phương của sự chênh lệch giữa dự đoán và mục tiêu.

```

# đầu vào là 2 chiều (x,y), đầu ra là 1 chiều (giá trị 1 hoặc 0)
input_dim = 2
output_dim = 1

# Khởi tạo W ngẫu nhiên, shape = (2,1) vì nhân với mỗi 1 điểm là (1,2)
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
# Khởi tạo b = 0
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))

# Hàm tạo model, thực ra là biến đổi tensor W*inp + b
def model(inputs):
    return tf.matmul(inputs, W) + b

# hàm mất mát (bình phương)
def square_loss(targets, predictions):
    # bình phương độ lệch
    per_sample_losses = tf.square(targets - predictions)
    # trả về giá trị trung bình
    return tf.reduce_mean(per_sample_losses)

# khởi tạo learning rate để điều khiển tốc độ thay đổi của hàm mất mát (độ dốc của hàm loss)
# learning rate càng lớn => dễ bị nhảy ngẫu nhiên, learning rate quá nhỏ => lâu, tốn thời gian
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        # dự đoán = cách biến đổi theo W*inp + b (hàm model)
        predictions = model(inputs)
        # tính giá trị mất mát
        loss = square_loss(predictions, targets)
        # tính gradient của W và b dựa trên giá trị mất mát
        grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
        # tính W và b mới bằng cách giảm theo hướng ngược lại của đạo hàm (dùng hàm trừ)
        # dùng learning_rate để quyết định tốc độ mất mát
        W.assign_sub(grad_loss_wrt_W * learning_rate)
        b.assign_sub(grad_loss_wrt_b * learning_rate)
        # trả về loss để xúy in ra mất mát từng bước để xem sự thay đổi
        return loss

# in ra giá trị mất mát sau từng bước
for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")

```

```

Loss at step 0: 5.8876
Loss at step 1: 0.8345
Loss at step 2: 0.2433
Loss at step 3: 0.1552
Loss at step 4: 0.1349
Loss at step 5: 0.1244
Loss at step 6: 0.1158
Loss at step 7: 0.1081
Loss at step 8: 0.1011
Loss at step 9: 0.0947
Loss at step 10: 0.0888
Loss at step 11: 0.0834
Loss at step 12: 0.0785
Loss at step 13: 0.0739
Loss at step 14: 0.0698
Loss at step 15: 0.0660
Loss at step 16: 0.0626
Loss at step 17: 0.0594
Loss at step 18: 0.0565
Loss at step 19: 0.0539
Loss at step 20: 0.0514

```

```

Loss at step 21: 0.0492
Loss at step 22: 0.0472
Loss at step 23: 0.0453
Loss at step 24: 0.0436
Loss at step 25: 0.0420
Loss at step 26: 0.0406
Loss at step 27: 0.0393
Loss at step 28: 0.0381
Loss at step 29: 0.0370
Loss at step 30: 0.0360
Loss at step 31: 0.0351
Loss at step 32: 0.0342
Loss at step 33: 0.0335
Loss at step 34: 0.0328
Loss at step 35: 0.0321
Loss at step 36: 0.0315
Loss at step 37: 0.0310
Loss at step 38: 0.0305
Loss at step 39: 0.0300

```

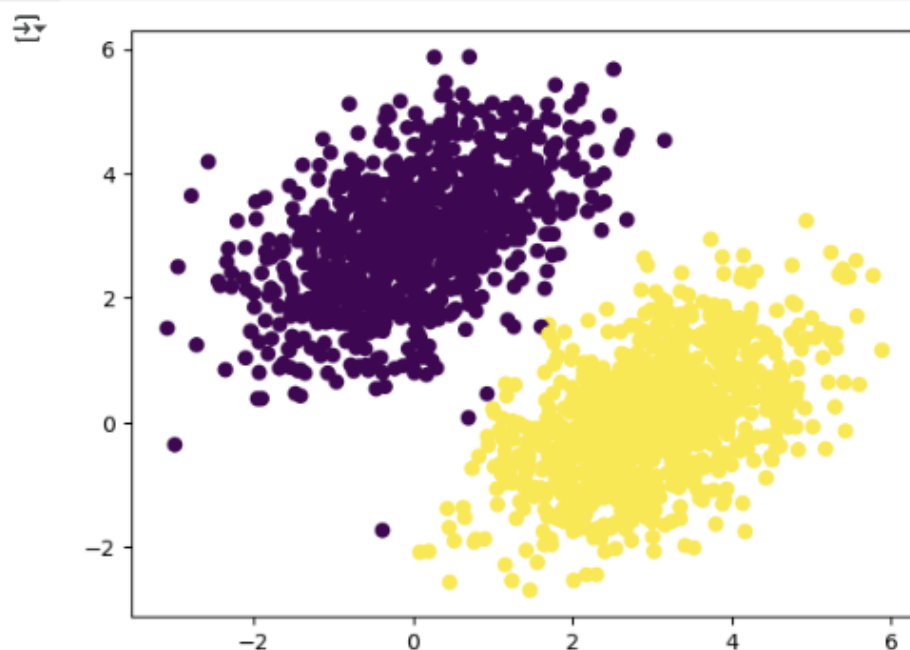
Sau 40 bước: giá trị hàm mất mát còn rất nhỏ (0.03)

Bước 3: Lúc này, đã dùng matplotlib để vẽ các điểm theo targets(kết quả), giờ vẽ theo prediction để so sánh

```

▶ predictions = model(inputs)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
plt.show()

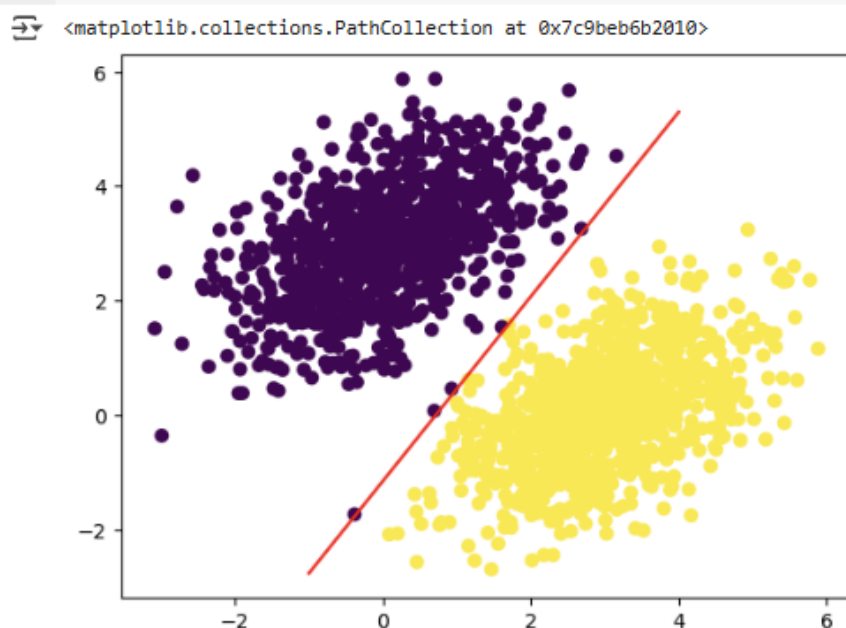
```



Phép tính $W \cdot \text{inp}$ thực ra là ma trận $[[W1], [W2]]$ 2×1 nhân vô hướng với 1 điểm $[x, y]$ là ma trận 1×2 cho ra kết quả $W1 \cdot x + W2 \cdot y$, sau đó cộng độ lệch $b \Rightarrow$ kết quả là số thực. Khi này, nhãn 0 được dán cho kết quả < 0.5 và nhãn 1 dán cho kết quả > 0.5

Bước 3: Vẽ đường thẳng phân biệt: đường thẳng phân biệt sẽ là đường thỏa mãn $W1*x + W2*y + b = 0.5 \Rightarrow y = -(W1/W2)*x + (0.5-b)/W2$

```
# tạo 100 số từ -1 đến 4 lưu vào mảng NumPy tên x
x = np.linspace(-1, 4, 100)
# tính mảng y từ x và các trọng số
y = - W[0] / W[1] * x + (0.5 - b) / W[1]
# Vẽ đường thẳng
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```



3.3 Các API của Keras

Đến thời điểm này, bạn đã nắm được những điều cơ bản về TensorFlow và có thể dùng nó để xây dựng một mô hình đơn giản từ đầu, như bộ phân loại tuyến tính theo lô ở phần trước, hay mạng nơ-ron đơn giản ở cuối chương 2. Đó là nền tảng vững chắc để bạn tiếp tục phát triển. Bây giờ, đã đến lúc chuyển sang một cách làm hiệu quả và mạnh mẽ hơn trong học sâu: **API Keras**.

3.3.1 Tầng (Layers): Những khối xây dựng cơ bản của học sâu

Cấu trúc dữ liệu cơ bản nhất trong mạng nơ-ron chính là tầng (layer). Nói một cách dễ hiểu, tầng là một "bộ xử lý dữ liệu" nhận vào một hoặc nhiều tensor và cho ra một hoặc nhiều tensor khác. Một số tầng không có trạng thái (stateless), nhưng thường thì tầng sẽ có trạng thái: đó là trọng số (weights) của tầng, gồm một hoặc nhiều tensor được học qua quá trình hạ gradient ngẫu nhiên. Trọng số này chính là "kiến thức" mà mạng nơ-ron tích lũy được.

Mỗi loại tầng sẽ phù hợp với các định dạng tensor và kiểu xử lý dữ liệu khác nhau. Ví dụ:

- Dữ liệu vector đơn giản, được lưu trong tensor bậc 2 có dạng (số mẫu, số đặc trưng), thường được xử lý bởi các tầng kết nối đầy đủ (densely connected layers), hay còn gọi là tầng Dense trong Keras.
- Dữ liệu chuỗi (sequence data), được lưu trong tensor bậc 3 có dạng (số mẫu, số bước thời gian, số đặc trưng), thường được xử lý bởi các tầng hồi quy (recurrent layers) như tầng LSTM, hoặc tầng tích chập 1D (Conv1D).
- Dữ liệu hình ảnh, được lưu trong tensor bậc 4, thường được xử lý bởi các tầng tích chập 2D (Conv2D).

Việc xây dựng mô hình học sâu trong Keras chỉ đơn giản là ghép các tầng tương thích với nhau để tạo thành một dòng xử lý dữ liệu hữu ích.

* Layer cơ bản trong Keras

Một API đơn giản nên có một khái niệm cốt lõi mà mọi thứ xoay quanh nó. Trong Keras, đó chính là Layer. Mọi thứ trong Keras đều là một Layer hoặc một thứ gì đó tương tác chặt chẽ với Layer.

Một Layer là một đối tượng bao gồm:

- Trạng thái (state): Trọng số của tầng, thường là các tensor được học.
- Phép tính (computation): Quá trình xử lý dữ liệu, được định nghĩa trong tầng.

Trọng số thường được khởi tạo trong phương thức build() (dù cũng có thể được tạo trong hàm khởi tạo __init__()), còn phép tính được định nghĩa trong phương thức call().

Dưới đây là cách khởi tạo một tầng, kế thừa từ lớp Layer của Keras thông qua các hàm init, build và call


```

from tensorflow import keras

# Định nghĩa 1 loại tầng mới kế thừa lớp Layer của Keras
class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__init__()
        # số nơ ron trong tầng, hay số chiều của đầu ra
        self.units = units
        # hàm kích hoạt
        self.activation = activation

    def build(self, input_shape):
        # input_dim là số cuối cùng của input_shape (như lấy ảnh các số sẽ lấy giá trị 784=28*28 ở cuối)
        input_dim = input_shape[-1]
        # khởi tạo tensor trọng số ngẫu nhiên (tại sao shape như vậy đã giải thích ở các phần trên)
        self.W = self.add_weight(shape=(input_dim, self.units), initializer="random_normal")
        # khởi tạo tensor b = toàn giá trị 0
        self.b = self.add_weight(shape=(self.units,), initializer="zeros")

    def call(self, inputs):
        # biến đổi tensor
        y = tf.matmul(inputs, self.W) + self.b
        # nếu có hàm kích hoạt thì áp dụng nó với y (như relu chỉ lấy y dương)
        if self.activation is not None:
            y = self.activation(y)
        return y

```

Giờ xem đầu vào và đầu ra có dạng như nào:

```

my_dense = SimpleDense(units=32, activation=tf.nn.relu)
input_tensor = tf.ones(shape=(2, 784))
output_tensor = my_dense(input_tensor)
print(output_tensor.shape)

(2, 32)

```

Số chiều của đầu vào: 784, số unit/ số nơ ron của mạng: 32, số chiều của đầu ra 32 => số nơ ron của tầng là số chiều của đầu ra

Cách tạo 1 tầng:

```

layer1 = layers.tên_tầng (số_nơ_ron, activation = "hàm_kích_hoạt")

```

Cách tạo mô hình:

```

model = keras.Sequential([
    layer1 = layers.tên_tầng (số_nơ_ron, activation = "hàm_kích_hoạt")
    ...
    layern = layers.tên_tầng (số_nơ_ron, activation = "hàm_kích_hoạt")
])

```

3.3.2 Một số loại tầng

- + Tầng kết nối đầy đủ: Dense
- + Tầng tích chập: Conv1D, Conv2D, Conv3D

- + Tầng hồi quy: LSTM, GRU, SimpleRNN
- + Tầng làm phẳng: Flatten
- + Tầng bỏ qua: Dropout
- + Tầng chuẩn hóa: BatchNormalization, LayerNormalization
- + Tầng nhúng: Embedding
- + Tầng gộp: MaxPooling2D, AveragePooling2D, MaxPooling1D

3.3.3 Một số loại hàm activate


- + Linear/None: không có hàm kích hoạt
- + Sigmoid: dùng trong phân loại nhị phân
- + Softmax: chuyển đầu ra thành xác suất, tổng xác suất bằng 1
- + Tanh: dùng trong hồi quy, giá trị từ -1 đến 1
- + Relu: đã tìm hiểu ở trên
- + Softplus: là phiên bản khác của relu, luôn dương

3.3.4 Chuyển từ các tầng (Layer) sang mô hình

Sau khi tạo các tầng, cần phải chọn thêm một số thứ sau, như đã giải thích ở trước đây:

- + Hàm mất mát (loss)
- + Bộ tối ưu hóa (optimizer)
- + Chỉ số đo khi huấn luyện mô hình (metric)

Cách compile model:

```
 model.compile(optimizer='bộ tối ưu hóa', loss='hàm mất mát', metrics=mảng các metric)
```

3.3.5 Một số bộ tối ưu hóa

- + SGD (with or without momentum)
- + RMSprop
- + Adam
- + Adagrad

3.3.6 Một số hàm mất mát

- + CategoricalCrossentropy
- + SparseCategoricalCrossentropy
- + BinaryCrossentropy
- + MeanSquaredError
- + KLDivergence
- + CosineSimilarity

3.3.7 Một số chỉ số đo khi huấn luyện mô hình

- + CategoricalAccuracy

- + SparseCategoricalAccuracy
- + BinaryAccuracy
- + AUC
- + Precision
- + Recall

3.3.8 Cách chọn hàm mất mát

Việc chọn đúng hàm mất mát là cực kỳ quan trọng. Mạng nơ-ron sẽ tìm mọi cách để giảm giá trị mất mát, nên nếu hàm mất mát không thực sự phản ánh đúng mục tiêu thành công của bài toán, mạng có thể làm những điều mà bạn không mong muốn. Hãy tưởng tượng một trí tuệ nhân tạo (AI) siêu mạnh nhưng ngớ ngẩn, được huấn luyện bằng phương pháp hạ gradient ngẫu nhiên (SGD) với một hàm mất mát được chọn sai, ví dụ: "tối đa hóa mức độ hạnh phúc trung bình của tất cả con người trên Trái Đất". Để làm việc này dễ hơn, AI có thể chọn cách loại bỏ gần hết loài người, chỉ giữ lại một vài người, rồi tập trung tăng hạnh phúc cho số ít đó—vì mức hạnh phúc trung bình không phụ thuộc vào số lượng người còn lại. Chắc chắn đó không phải điều bạn muốn!

Vì mọi mạng nơ-ron bạn xây dựng sẽ "tàn nhẫn" trong việc giảm giá trị mất mát của nó - vì vậy, hãy chọn hàm mất mát một cách khôn ngoan, nếu không bạn sẽ phải đối mặt với những hậu quả không lường trước.

Với các bài toán phổ biến như phân loại, hồi quy hay dự đoán chuỗi, có những hướng dẫn đơn giản để bạn chọn hàm mất mát đúng. Chẳng hạn, bạn sẽ dùng hàm **Binary Crossentropy** cho bài toán phân loại nhị phân (hai lớp), **Categorical Crossentropy** cho bài toán phân loại đa lớp (nhiều lớp),...

3.3.9 Phương thức fit()

Công thức hàm fit()

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

với:


- + inputs là tập đầu vào
- + targets là tập nhãn gán cho input (mục tiêu)
- + epoch là số lần vòng lặp huấn luyện sẽ lặp lại trên toàn bộ dữ liệu được truyền vào.

- + `batch_size`: kích thước lô: mỗi lô sẽ chứa bao nhiêu mẫu huấn luyện (hay bộ dữ liệu)

Gọi hàm `fit()` sẽ trả về một đối tượng `History`. Đối tượng này chứa một trường `history`, là một từ điển (dict) ánh xạ các khóa như "loss" hoặc tên các chỉ số cụ thể (metrics) đến danh sách giá trị của chúng qua từng epoch

```
>>> history.history
{"binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],
 "loss": [0.6573270302042366,
          0.07434618508815766,
          0.07687718723714351,
          0.07412414988875389,
          0.07617757616937161]}
```

3.3.10 Sử dụng mô hình sau huấn luyện

```
 predictions = model.predict(new_inputs, batch_size=128) |
```

Cuối cùng, có thể in ra predictions để xem dự đoán mà mô hình cho ra

CHƯƠNG 4: BẮT ĐẦU VỚI MẠNG NƠ-ROK: PHÂN LOẠI VÀ HỒI QUY

Chương này được thiết kế để giúp bạn bắt đầu sử dụng mạng nơ-ron để giải quyết các bài toán thực tế. Bạn sẽ củng cố kiến thức đã học từ chương 2 và chương 3, đồng thời áp dụng những gì đã biết vào ba nhiệm vụ mới, bao gồm ba trường hợp sử dụng phổ biến nhất của mạng nơ-ron: **phân loại nhị phân**, **phân loại đa lớp**, và **hồi quy giá trị vô hướng**:

- **Phân loại đánh giá phim là tích cực hay tiêu cực** (phân loại nhị phân).
- **Phân loại tin tức theo chủ đề** (phân loại đa lớp).
- **Ước lượng giá nhà dựa trên dữ liệu bất động sản** (hồi quy giá trị vô hướng).

Những ví dụ này sẽ là lần đầu tiên bạn tiếp xúc với quy trình học máy từ đầu đến cuối: bạn sẽ được làm quen với **tiền xử lý dữ liệu**, các nguyên tắc cơ bản về kiến trúc mô hình, và cách đánh giá mô hình.

Chương này bao gồm

- Những ví dụ đầu tiên về quy trình học máy thực tế.
- Xử lý các bài toán phân loại trên dữ liệu dạng vector.
- Xử lý các bài toán hồi quy liên tục trên dữ liệu dạng vector.

Đến cuối chương này, bạn sẽ có khả năng sử dụng mạng nơ-ron để giải quyết các bài toán phân loại và hồi quy đơn giản trên dữ liệu dạng vector. Sau đó, bạn sẽ sẵn sàng để bắt đầu xây dựng một sự hiểu biết có hệ thống và lý thuyết hơn về học máy trong chương 5.

4.1 Phân loại nhị phân thông qua ví dụ về phân loại đánh giá phim

4.1.1 Bộ dữ liệu IMDB

Bạn sẽ làm việc với bộ dữ liệu **IMDB**, bao gồm 50,000 đánh giá phim có tính phân cực cao từ Internet Movie Database. Bộ dữ liệu được chia thành:

- 25,000 đánh giá để huấn luyện (training).
- 25,000 đánh giá để kiểm tra (testing).

Mỗi tập dữ liệu (huấn luyện và kiểm tra) đều có **50% đánh giá tiêu cực** và **50% đánh giá tích cực**.

Bộ dữ liệu được tích hợp sẵn trong Keras và được tiền xử lý, chúng ta chỉ cần đi huấn luyện mô hình.

4.1.2 Xử lý bài toán với code và giải thích code

Bước 1: Tải dữ liệu

```
# import imdb từ keras
from tensorflow.keras.datasets import imdb
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# dùng hàm load_data để tải các tập huấn luyện và tập kiểm tra
# num_words=10000: Giới hạn từ điển chỉ bao gồm 10,000 từ xuất hiện nhiều nhất trong dữ liệu huấn luyện
# nếu có từ xuất hiện ít lần => không lưu lại vì không hữu dụng
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

Bước 2: Tiền xử lý dữ liệu

```
#hàm chuyển đổi danh sách các chuỗi chỉ số từ (word indices) thành ma trận nhị phân.
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results

# sử dụng hàm ở trên
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)

# vector hóa nhãn
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

Bước 3: Xây mô hình

```
# Xây các tầng
model = keras.Sequential([
    # Dense là đủ để xử lý vector rồi, vì dữ liệu là chữ, string
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    # xây dựng tầng cuối với 1 nơ ron vì đầu ra là nhị phân, hàm sigmoid tính xác suất xem thuộc lớp nào
    layers.Dense(1, activation="sigmoid")
])

# chọn 3 thông số : bộ tối ưu, hàm mất mát và metric
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])

# chia tập huấn luyện thành tập kiểm định và tập huấn luyện còn lại
# như đã nói ở trên, mô hình có thể bị overfitting: độ chính xác cao trên tập huấn luyện nhưng thấp
# trên tập kiểm tra.
# dùng tập kiểm định từ tập huấn luyện để luôn luôn theo dõi giám sát để phát hiện overfitting
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

# phương thức fit
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Kết quả nhận được

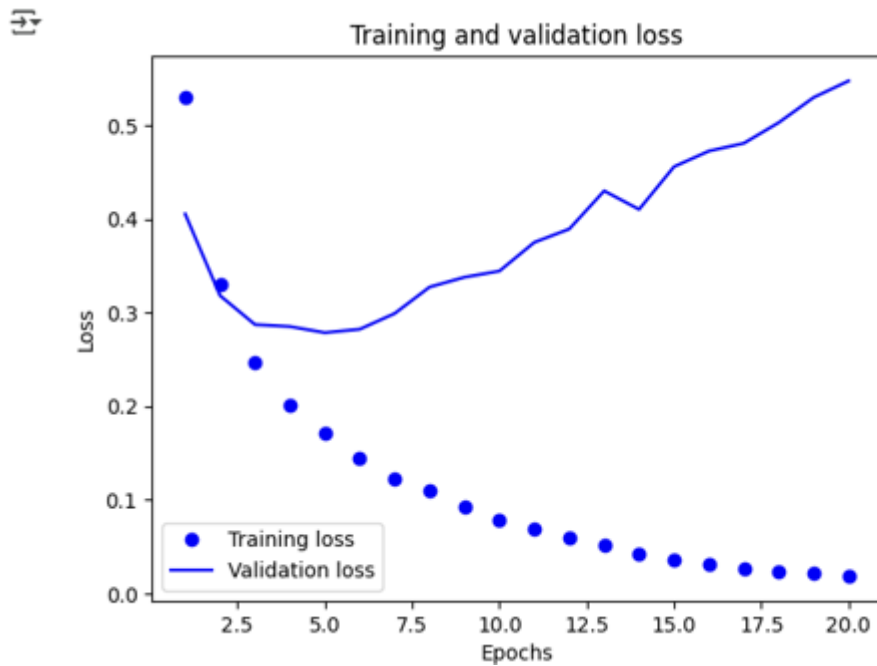
```
Epoch 1/20
30/30 ————— 4s 66ms/step - accuracy: 0.6956 - loss: 0.5982 - val_accuracy: 0.8618 - val_loss: 0.4053
Epoch 2/20
30/30 ————— 2s 49ms/step - accuracy: 0.8968 - loss: 0.3472 - val_accuracy: 0.8839 - val_loss: 0.3181
Epoch 3/20
30/30 ————— 1s 35ms/step - accuracy: 0.9224 - loss: 0.2513 - val_accuracy: 0.8884 - val_loss: 0.2872
Epoch 4/20
30/30 ————— 1s 36ms/step - accuracy: 0.9350 - loss: 0.2049 - val_accuracy: 0.8852 - val_loss: 0.2851
Epoch 5/20
30/30 ————— 1s 37ms/step - accuracy: 0.9448 - loss: 0.1733 - val_accuracy: 0.8886 - val_loss: 0.2785
Epoch 6/20
30/30 ————— 2s 47ms/step - accuracy: 0.9570 - loss: 0.1408 - val_accuracy: 0.8879 - val_loss: 0.2822
Epoch 7/20
30/30 ————— 1s 47ms/step - accuracy: 0.9624 - loss: 0.1261 - val_accuracy: 0.8835 - val_loss: 0.2990
Epoch 8/20
30/30 ————— 2s 61ms/step - accuracy: 0.9683 - loss: 0.1068 - val_accuracy: 0.8799 - val_loss: 0.3273
Epoch 9/20
30/30 ————— 1s 35ms/step - accuracy: 0.9771 - loss: 0.0895 - val_accuracy: 0.8806 - val_loss: 0.3378
Epoch 10/20
30/30 ————— 1s 35ms/step - accuracy: 0.9806 - loss: 0.0759 - val_accuracy: 0.8758 - val_loss: 0.3443
Epoch 11/20
30/30 ————— 1s 36ms/step - accuracy: 0.9829 - loss: 0.0680 - val_accuracy: 0.8713 - val_loss: 0.3750
Epoch 12/20
30/30 ————— 1s 35ms/step - accuracy: 0.9873 - loss: 0.0588 - val_accuracy: 0.8780 - val_loss: 0.3892
Epoch 13/20
30/30 ————— 1s 35ms/step - accuracy: 0.9909 - loss: 0.0494 - val_accuracy: 0.8747 - val_loss: 0.4299
Epoch 14/20
30/30 ————— 1s 38ms/step - accuracy: 0.9936 - loss: 0.0396 - val_accuracy: 0.8746 - val_loss: 0.4102
Epoch 15/20
30/30 ————— 1s 45ms/step - accuracy: 0.9952 - loss: 0.0331 - val_accuracy: 0.8654 - val_loss: 0.4558
Epoch 16/20
30/30 ————— 3s 46ms/step - accuracy: 0.9960 - loss: 0.0284 - val_accuracy: 0.8683 - val_loss: 0.4725
Epoch 17/20
30/30 ————— 2s 34ms/step - accuracy: 0.9975 - loss: 0.0252 - val_accuracy: 0.8738 - val_loss: 0.4808
Epoch 18/20
30/30 ————— 1s 34ms/step - accuracy: 0.9956 - loss: 0.0238 - val_accuracy: 0.8719 - val_loss: 0.5029
Epoch 19/20
30/30 ————— 1s 34ms/step - accuracy: 0.9967 - loss: 0.0192 - val_accuracy: 0.8726 - val_loss: 0.5297
Epoch 20/20
30/30 ————— 1s 34ms/step - accuracy: 0.9987 - loss: 0.0139 - val_accuracy: 0.8717 - val_loss: 0.5474
```

Có thể trực quan hóa bằng matplotlib

```

history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, "bo", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()

```



Đánh giá độ chính xác: 85,86%, khá cao

```

# đánh giá mô hình
results = model.evaluate(x_test, y_test)
print(results)

```

782/782 ————— 3s 4ms/step - accuracy: 0.8548 - loss: 0.5993
[0.5896691679954529, 0.858680009841919]

Bước 4: dự đoán

```

predictions = model.predict(x_test)
print(predictions)

```

782/782 ————— 1s 2ms/step
[[0.01617024]
 [1.]
 [0.9887774]
 ...
 [0.00612236]
 [0.0167008]
 [0.9542529]]

4.2 Phân loại tin tức: Một ví dụ về phân loại đa lớp

4.2.1 Bộ dữ liệu tin tức từ Reuters

Trong phần này, chúng ta sẽ xây dựng một mô hình để phân loại các tin tức từ Reuters thành **46 chủ đề** loại trừ lẫn nhau. Vì có nhiều lớp, bài toán này là một ví dụ về **phân loại đa lớp (multiclass classification)**. Hơn nữa, vì mỗi điểm dữ liệu chỉ được phân loại vào **một danh mục duy nhất**, bài toán này cụ thể hơn là một trường hợp của **phân loại đa lớp nhãn đơn (single-label multiclass classification)**.

4.2.2 Học cách làm qua code và giải thích

Các bước tương tự như ví dụ trước đó

```
# Import các thư viện cần thiết
from tensorflow.keras.datasets import reuters
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)

# - word_index: Từ điển ánh xạ từ (chuỗi) sang chỉ số (số nguyên)
# - reverse_word_index: Đảo ngược từ điển để ánh xạ từ chỉ số về từ
# - i - 3: Trừ 3 vì các chỉ số 0, 1, 2 dành cho ký tự đặc biệt (padding, start, unknown)
# - "?": Thay thế nếu không tìm thấy chỉ số
# - Kết quả: decoded_newswire là chuỗi văn bản tiếng Anh của tin tức đầu tiên
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_newswire = " ".join([reverse_word_index.get(i - 3, "?") for i in train_data[0]])

# Hàm vector hóa dữ liệu
# - sequences: Danh sách các tin tức (danh sách chỉ số từ)
# - dimension=10000: Kích thước vector đặc trưng (10,000 chiều)
# - Tạo ma trận nhị phân: 1 nếu từ có chỉ số j xuất hiện trong tin tức i, 0 nếu không
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results

# Vector hóa dữ liệu huấn luyện và kiểm tra
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)

# Hàm mã hóa one-hot cho nhãn
# - labels: Danh sách nhãn (chỉ số từ 0-45)
# - dimension=46: Số lớp (46 chủ đề)
# - Tạo ma trận one-hot: 1 tại vị trí tương ứng với nhãn, 0 ở các vị trí khác
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

# Mã hóa one-hot cho nhãn huấn luyện và kiểm tra
y_train = to_one_hot(train_labels)
y_test = to_one_hot(test_labels)
```

```

# Xây dựng mô hình
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    # 46 do có 46 nhân, softmax để tính xác suất cho lớp i => tổng xác suất của 46 lớp cho 1 mẫu là 1
    layers.Dense(46, activation="softmax")
])

# Thêm bộ tối ưu, hàm mất mát, theo dõi độ chính xác
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])

# tách để kiểm tra overfitting
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = y_train[:1000]
partial_y_train = y_train[1000:]

# Huấn luyện mô hình
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=512,
          validation_data=(x_val, y_val))

# Đánh giá mô hình trên tập kiểm tra
results = model.evaluate(x_test, y_test)
print(results)

# Dự đoán trên dữ liệu kiểm tra
# - predictions: Ma trận (2246, 46), chứa xác suất cho 46 lớp
predictions = model.predict(x_test)

# Lấy lớp có xác suất cao nhất cho mẫu đầu tiên
# - np.argmax(predictions[0]): Chỉ số lớp có xác suất cao nhất
print(np.argmax(predictions[0]))

```

Kết quả thu được:

```

Epoch 1/20
16/16 ————— 2s 73ms/step - accuracy: 0.3691 - loss: 3.2536 - val_accuracy: 0.6370 - val_loss: 1.8447
Epoch 2/20
16/16 ————— 1s 58ms/step - accuracy: 0.6766 - loss: 1.6335 - val_accuracy: 0.7010 - val_loss: 1.3769
Epoch 3/20
16/16 ————— 1s 63ms/step - accuracy: 0.7329 - loss: 1.2142 - val_accuracy: 0.7390 - val_loss: 1.1851
Epoch 4/20
16/16 ————— 1s 53ms/step - accuracy: 0.7870 - loss: 0.9555 - val_accuracy: 0.7510 - val_loss: 1.0884
Epoch 5/20
16/16 ————— 1s 50ms/step - accuracy: 0.8245 - loss: 0.7825 - val_accuracy: 0.7720 - val_loss: 1.0050
Epoch 6/20
16/16 ————— 2s 66ms/step - accuracy: 0.8501 - loss: 0.6870 - val_accuracy: 0.7880 - val_loss: 0.9491
Epoch 7/20
16/16 ————— 2s 92ms/step - accuracy: 0.8807 - loss: 0.5683 - val_accuracy: 0.8060 - val_loss: 0.9016
Epoch 8/20
16/16 ————— 1s 59ms/step - accuracy: 0.9002 - loss: 0.4773 - val_accuracy: 0.8130 - val_loss: 0.8812
Epoch 9/20
16/16 ————— 1s 50ms/step - accuracy: 0.9198 - loss: 0.3904 - val_accuracy: 0.8130 - val_loss: 0.8748
Epoch 10/20
16/16 ————— 1s 52ms/step - accuracy: 0.9300 - loss: 0.3387 - val_accuracy: 0.8140 - val_loss: 0.8684

```

```

Epoch 11/20
16/16 ----- 1s 52ms/step - accuracy: 0.9396 - loss: 0.2865 - val_accuracy: 0.8120 - val_loss: 0.8621
Epoch 12/20
16/16 ----- 1s 53ms/step - accuracy: 0.9443 - loss: 0.2482 - val_accuracy: 0.8150 - val_loss: 0.8644
Epoch 13/20
16/16 ----- 1s 50ms/step - accuracy: 0.9464 - loss: 0.2206 - val_accuracy: 0.8090 - val_loss: 0.9048
Epoch 14/20
16/16 ----- 1s 50ms/step - accuracy: 0.9519 - loss: 0.2014 - val_accuracy: 0.8200 - val_loss: 0.8722
Epoch 15/20
16/16 ----- 1s 52ms/step - accuracy: 0.9590 - loss: 0.1700 - val_accuracy: 0.8110 - val_loss: 0.9045
Epoch 16/20
16/16 ----- 1s 53ms/step - accuracy: 0.9561 - loss: 0.1614 - val_accuracy: 0.8130 - val_loss: 0.9473
Epoch 17/20
16/16 ----- 1s 52ms/step - accuracy: 0.9560 - loss: 0.1606 - val_accuracy: 0.8050 - val_loss: 0.9248
Epoch 18/20
16/16 ----- 2s 86ms/step - accuracy: 0.9575 - loss: 0.1480 - val_accuracy: 0.8130 - val_loss: 0.9455
Epoch 19/20
16/16 ----- 1s 85ms/step - accuracy: 0.9616 - loss: 0.1287 - val_accuracy: 0.8080 - val_loss: 0.9599
Epoch 20/20
16/16 ----- 2s 50ms/step - accuracy: 0.9598 - loss: 0.1258 - val_accuracy: 0.8080 - val_loss: 0.9592
71/71 ----- 0s 4ms/step - accuracy: 0.8027 - loss: 1.0142
[1.0477114915847778, 0.791629528411865]
71/71 ----- 0s 4ms/step
3

```

```
[15] print(test_labels[0])
```

↩ 3

- + Nhận xét:
độ chính xác: 79.16%, không quá cao
- + với test đầu tiên, kết quả thu được trùng khớp, đều là 3

4.3 Dự đoán giá nhà: Một ví dụ về hồi quy

Còn một loại bài toán học máy phổ biến khác là **hồi quy (regression)**, với mục tiêu là dự đoán một **giá trị liên tục (continuous value)** thay vì một nhãn rời rạc. Ví dụ, dự đoán nhiệt độ ngày mai dựa trên dữ liệu khí tượng, hoặc dự đoán thời gian hoàn thành một dự án phần mềm dựa trên thông số kỹ thuật của nó.

4.3.1 Bộ dữ liệu giá nhà Boston

Trong phần này, chúng ta sẽ cố gắng dự đoán **giá trị trung bình của nhà ở tại một khu vực ngoại ô Boston vào giữa những năm 1970**, dựa trên các điểm dữ liệu về khu vực đó tại thời điểm, chẳng hạn như tỷ lệ tội phạm, thuế suất tài sản địa phương, v.v. Bộ dữ liệu mà chúng ta sẽ sử dụng có một số điểm khác biệt thú vị so với hai ví dụ trước:

- **Số lượng dữ liệu ít:** Bộ dữ liệu chỉ có 506 mẫu, được chia thành 404 mẫu huấn luyện và 102 mẫu kiểm tra.
- **Đặc trưng có thang đo khác nhau:** Mỗi đặc trưng trong dữ liệu đầu vào (ví dụ: tỷ lệ tội phạm) có thang đo khác nhau. Một số giá trị là tỷ lệ (proportions), nằm trong khoảng từ 0 đến 1; một số giá trị nằm trong khoảng từ 1 đến 12; một số khác nằm trong khoảng từ 0 đến 100, v.v.

Lưu ý: vì số bộ dữ liệu ít, nên ở đây, dùng thêm kỹ thuật k-fold: chia dữ liệu ra làm k phần để thực hiện huấn luyện k lần, lần 1 lấy phần 1 để kiểm tra,

còn lại để huấn luyện, lần 2 lấy phần 2 kiểm tra, còn lại huấn luyện,... lần n lấy phần n kiểm tra, còn lại huấn luyện), thực hiện đủ k lần

=> Phù hợp với lượng dữ liệu ít vì bảo đảm tất cả dữ liệu đều tham gia vào huấn luyện và kiểm tra.

4.3.2 Cách thực hiện thông qua code và giải thích

```
from tensorflow.keras.datasets import boston_housing
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers

(train_data, train_targets), (test_data, test_targets) = (boston_housing.load_data())

#tính TB và độ lệch chuẩn để chuẩn hóa dữ liệu (do đề có ghi các thang dữ liệu k đồng nhất)
mean = train_data.mean(axis=0)
std = train_data.std(axis=0)
# chuẩn hóa dữ liệu
train_data -= mean
train_data /= std
test_data -= mean
test_data /= std

# cấu hình mô hình
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        # Hồi quy vô hướng, không cần hàm kích hoạt
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

```
# số fold
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
```

```

for i in range(k):
    # tạo tập kiểm định
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    # phần còn lại để train
    partial_train_data = np.concatenate([train_data[:i * num_val_samples],
                                          train_data[(i + 1) * num_val_samples:]],
                                         axis=0)
    partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
                                           train_targets[(i + 1) * num_val_samples:]],
                                           axis=0)

    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=16, verbose=0)
    # Đánh giá trên tập kiểm định
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    # Lưu MAE của fold này
    all_scores.append(val_mae)

```

```

# huấn luyện 500 lần
num_epochs = 500
all_mae_histories = []
for i in range(k):
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate([train_data[:i * num_val_samples],
                                          train_data[(i + 1) * num_val_samples:]],
                                         axis=0)
    partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
                                           train_targets[(i + 1) * num_val_samples:]],
                                           axis=0)

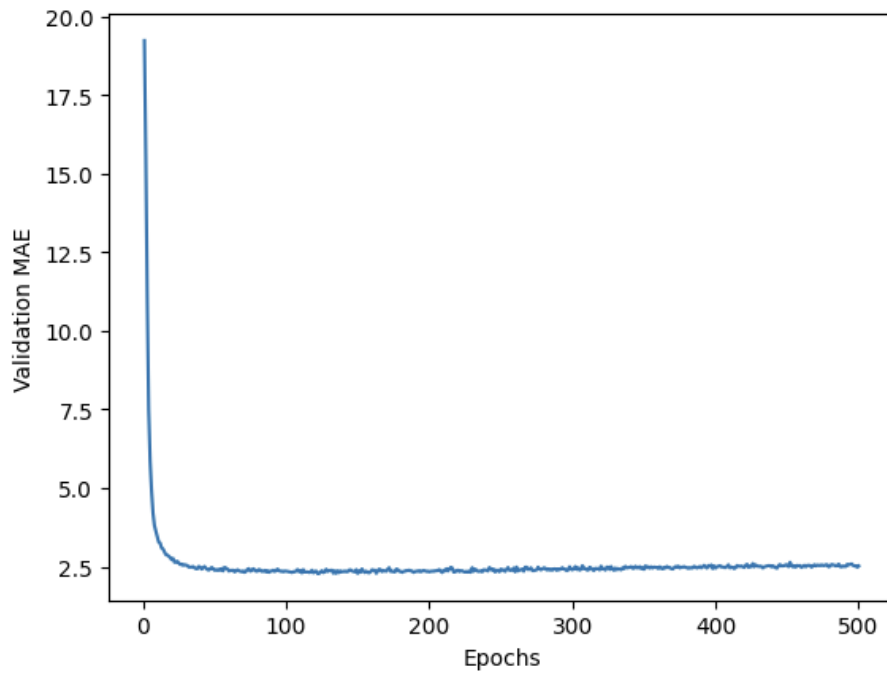
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                       validation_data=(val_data, val_targets),
                       epochs=num_epochs, batch_size=16, verbose=0)
    # lưu lịch sử MAE trên tập kiểm định
    mae_history = history.history["val_mae"]
    all_mae_histories.append(mae_history)

# Tính trung bình MAE qua từng epoch cho tất cả fold
average_mae_history = [np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()

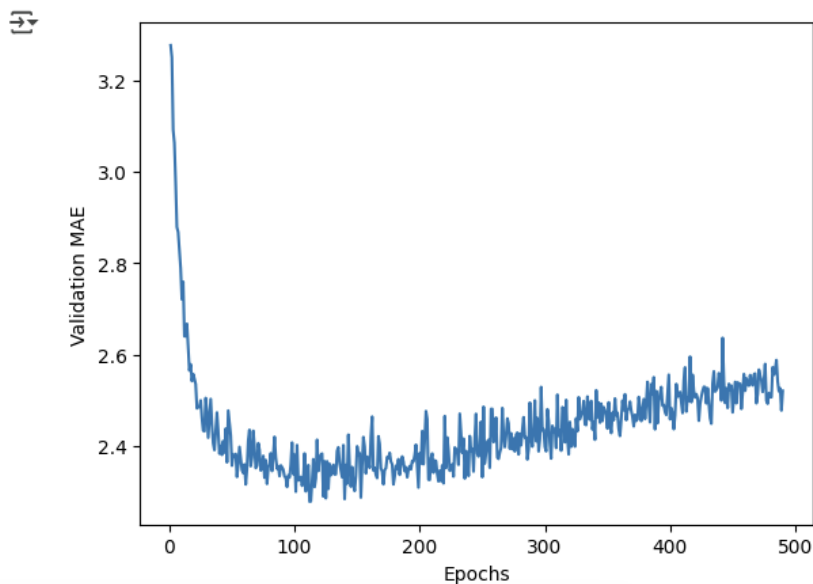
```

Kết quả:



Do 10 điểm đầu không quan sát tốt được => bỏ đi và vẽ lại (do 10 điểm đó thang đo quá cao làm tỉ lệ sọc đồ thay đổi => khó quan sát chi tiết)

```
# 10 điểm đầu không thực => bỏ đoạn đó đi
truncated_mae_history = average_mae_history[10:]
plt.plot(range(1, len(truncated_mae_history) + 1), truncated_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```



Nhận xét: epoch 120-140 ổn định, còn về sau bị overfitting => lấy 130 epoch để huấn luyện

```
model = build_model()
model.fit(train_data, train_targets,
epochs=130, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)

predictions = model.predict(test_data)
print(predictions[0])
print(test_targets[0])
```

4/4 ————— 0s 9ms/step - loss: 11.6946 - mae: 2.4074
4/4 ————— 0s 18ms/step
[9.391381]
7.2

CHƯƠNG 5: NHỮNG NGUYÊN TẮC CƠ BẢN CỦA HỌC MÁY

5.1 Mục tiêu của học máy

5.1.1 Underfitting và Overfitting

Quá trình huấn luyện mô hình trải qua:

- + Underfitting: khi mô hình chưa học đủ các mẫu quan trọng (như 10 điểm đầu ở phần trên của chương 4)
- + Robust fit: mô hình cân bằng, khái quát tốt
- + Overfitting: mô hình học phải những mẫu lạ, không khái quát, làm cho tỉ lệ chính xác trên tập kiểm tra giảm dù tỉ lệ chính xác trên tập huấn luyện cao

Nguyên nhân gây ra Overfitting

- + Đầu vào không hợp lệ: ví dụ đầu vào là ảnh từ 1-9 nhưng ảnh đưa vào không liên quan, kỳ lạ
- + Dữ liệu bị nhiễu
- + Một số đầu vào không khái quát (ví dụ như cách viết số của người đó đặc biệt hơn so với mọi người, hay như bạn gặp 2 con mèo cam không thân thiện thì bạn sẽ nghĩ mèo cam không thân thiện mặc dù màu lông chẳng liên quan) => mô hình cần học những xu hướng chung
- + Mẫu bị gán nhãn sai
- + Mối quan hệ giả: ví dụ, 1 từ ngữ mang 46% tích cực, 54% tiêu cực, tuy nhiên nó tùy trường hợp, nhưng mô hình lại dựa vào ưu thế 54% để phân loại => quá khớp (ví dụ như chảy nước mắt có thể do vui hoặc buồn)

5.1.2 Bản chất của khái quát hóa trong học sâu

Các mô hình học sâu có thể được huấn luyện để khớp với bất kỳ thứ gì, miễn là chúng có đủ khả năng biểu diễn (representational power). Tuy nhiên, nếu chỉ "ghi nhớ" dữ liệu huấn luyện, mô hình sẽ quá khớp và không khái quát hóa được trên dữ liệu mới.

* Giả thuyết đa tạp (Manifold Hypothesis)

- Dữ liệu tự nhiên (như hình ảnh MNIST) không phân bố ngẫu nhiên trong không gian đầu vào mà nằm trên một đa tạp (manifold) có chiều thấp, có cấu trúc cao.
 - Ví dụ: Các chữ số MNIST chỉ chiếm một phần nhỏ trong không gian 28×28 , và các mẫu hợp lệ có tính liên tục (có thể biến đổi mượt mà từ chữ số này sang chữ số khác).
- Đa tạp: Một không gian con có chiều thấp, cục bộ giống không gian Euclide (như đường cong 1D trong không gian 2D, mặt phẳng 2D trong không gian 3D).

* Khả năng khái quát hóa nhờ nội suy (Interpolation)

- Giả thuyết đa tạp cho rằng:
 - Mô hình học máy chỉ cần học các đa tạp tiềm tàng (latent manifolds) đơn giản, có chiều thấp trong không gian đầu vào.
 - Trong đa tạp, có thể nội suy (interpolate) giữa các điểm dữ liệu, tức là biến đổi mượt mà từ một điểm này sang điểm khác.
- Nội suy giúp mô hình hiểu các điểm dữ liệu mới bằng cách liên hệ chúng với các điểm đã thấy, từ đó đạt được khái quát hóa cục bộ (local generalization).
- Lưu ý: Nội suy trên đa tạp khác với nội suy tuyến tính trong không gian gốc (ví dụ: trung bình pixel giữa hai chữ số MNIST không tạo ra chữ số hợp lệ).

* Tại sao học sâu hoạt động tốt?

- Học sâu hoạt động tốt nhờ cấu trúc thông tin trong thế giới thực (theo giả thuyết đa tạp), không chỉ nhờ khả năng biểu diễn của mô hình.
- Mô hình học sâu:

- Là một đường cong đa chiều (manifold) có tính mượt mà và liên tục, được điều chỉnh dần qua gradient descent để khớp với dữ liệu huấn luyện.
- Trong quá trình huấn luyện, mô hình đi qua trạng thái khớp tốt (robust fit), nơi nó xấp xỉ đa tạp tiềm tàng của dữ liệu, trước khi quá khớp.
- Đặc điểm của học sâu:
 - Ánh xạ mượt mà và liên tục từ đầu vào đến đầu ra (do yêu cầu khả vi để dùng gradient descent).
 - Cấu trúc mô hình (như phân cấp và mô-đun) phản ánh cấu trúc của dữ liệu tự nhiên (ví dụ: mô hình xử lý ảnh, chuỗi).

* Hạn chế của nội suy và khái quát hóa

- Nội suy chỉ giúp khái quát hóa cục bộ (cho các điểm gần với dữ liệu đã thấy).
- Con người có khả năng khái quát hóa cực đoan (extreme generalization) nhờ các cơ chế nhận thức khác (trừu tượng, lý luận, logic, kiến thức chung), không chỉ dựa vào nội suy.
- Học sâu chủ yếu dựa vào nội suy, nhưng để đạt khái quát hóa tốt hơn, cần kết hợp các kỹ thuật khác

* Dữ liệu huấn luyện là yếu tố then chốt

1. Khả năng khái quát hóa phụ thuộc vào dữ liệu

- Mặc dù học sâu rất phù hợp để học các đa tạp (manifold learning), khả năng khái quát hóa (generalization) chủ yếu là kết quả của cấu trúc tự nhiên của dữ liệu, chứ không phải do đặc tính của mô hình.
- Mô hình chỉ có thể khái quát hóa nếu dữ liệu huấn luyện hình thành một đa tạp mà tại đó các điểm dữ liệu có thể được nội suy (interpolated).
- Dữ liệu càng thông tin và ít nhiễu thì khả năng khái quát hóa càng tốt, vì không gian đầu vào sẽ đơn giản và có cấu trúc rõ ràng hơn.
- Quản lý dữ liệu (data curation) và kỹ thuật đặc trưng (feature engineering) là yếu tố thiết yếu để đạt được khái quát hóa tốt.

2. Cần lấy mẫu dày đặc (Dense Sampling)

- Học sâu là quá trình khớp đường cong (curve fitting), nên để mô hình hoạt động tốt, nó cần được huấn luyện trên một lấy mẫu dày đặc (dense sampling) của không gian đầu vào.
- Lấy mẫu dày đặc nghĩa là dữ liệu huấn luyện phải bao phủ đầy đủ toàn bộ đa tạp dữ liệu đầu vào (input data manifold), đặc biệt là ở các vùng gần ranh giới quyết định (decision boundaries).
- Với lấy mẫu dày đặc, mô hình có thể hiểu các đầu vào mới bằng cách nội suy giữa các điểm huấn luyện mà không cần đến kiến thức chung (common sense), lý luận trừu tượng (abstract reasoning), hay kiến thức bên ngoài (external knowledge)—những thứ mà mô hình học máy không có.

3. Cách cải thiện khái quát hóa

- Cách tốt nhất: Huấn luyện mô hình trên nhiều dữ liệu hơn hoặc dữ liệu chất lượng cao hơn.
 - Dữ liệu nhiều hoặc không chính xác sẽ làm giảm khả năng khái quát hóa.
 - Lấy mẫu dày đặc hơn sẽ giúp mô hình khái quát hóa tốt hơn.
- Không nên kỳ vọng quá mức: Mô hình học sâu chỉ có thể thực hiện nội suy thô (crude interpolation) giữa các mẫu huấn luyện, không thể suy luận vượt ra ngoài dữ liệu đã thấy.
- Dữ liệu quyết định tất cả: Mô hình học sâu chỉ học được những gì bạn đưa vào—các kiến thức tiên nghiệm (priors) được mã hóa trong kiến trúc mô hình và dữ liệu huấn luyện.

4. Khi không thể có thêm dữ liệu

- Nếu không thể thu thập thêm dữ liệu, cách tiếp theo là:
 - Điều chỉnh lượng thông tin mô hình được phép lưu trữ: Giới hạn số lượng mẫu mà mô hình có thể ghi nhớ.
 - Thêm ràng buộc về độ mượt của đường cong mô hình: Đảm bảo mô hình không học các mẫu quá phức tạp.
- Khi mô hình chỉ có thể ghi nhớ một số lượng nhỏ mẫu hoặc các mẫu rất đều (regular patterns), quá trình tối ưu hóa sẽ buộc nó tập trung vào các mẫu nổi bật nhất, có khả năng khái quát hóa tốt hơn.

- Quá trình này được gọi là điều chuẩn (regularization), sẽ được thảo luận chi tiết ở phần 5.4.4.

5. Đánh giá mô hình trước khi cải thiện

- Trước khi điều chỉnh mô hình để cải thiện khái quát hóa, bạn cần một cách để đánh giá hiệu suất hiện tại của mô hình.
- Phần tiếp theo sẽ giới thiệu các phương pháp đánh giá mô hình (model evaluation) để theo dõi khả năng khái quát hóa trong quá trình phát triển mô hình.

5.2 Đánh giá mô hình học máy

5.2.1 Tập huấn luyện, tập kiểm định và tập kiểm tra

Việc đánh giá một mô hình luôn xoay quanh việc chia dữ liệu có sẵn thành ba tập: tập huấn luyện (training set), tập kiểm định (validation set), và tập kiểm tra (test set):

- Bạn huấn luyện mô hình trên tập huấn luyện và đánh giá mô hình trên tập kiểm định.
- Khi mô hình đã sẵn sàng để triển khai thực tế (prime time), bạn kiểm tra lần cuối trên tập kiểm tra, tập này được thiết kế để giống với dữ liệu thực tế (production data) nhất có thể

* Tại sao không chỉ dùng hai tập: huấn luyện và kiểm tra?

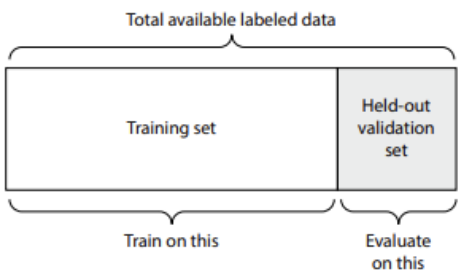
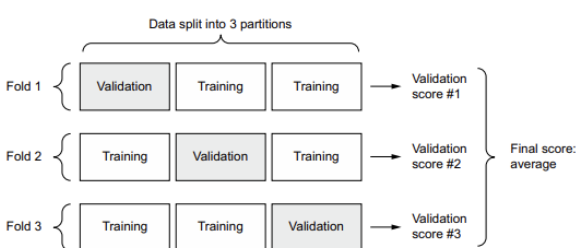
- + Khi phát triển mô hình, bạn luôn cần điều chỉnh cấu hình của nó
- + Việc điều chỉnh này được thực hiện bằng cách sử dụng hiệu suất của mô hình trên tập kiểm định như một tín hiệu phản hồi (feedback signal).
- + Về bản chất, việc điều chỉnh siêu tham số này là một dạng học: tìm kiếm một cấu hình tốt trong không gian siêu tham số.
- + Kết quả là, nếu bạn điều chỉnh cấu hình mô hình dựa trên hiệu suất trên tập kiểm định, bạn có thể nhanh chóng gây ra quá khớp với tập kiểm định, ngay cả khi mô hình không được huấn luyện trực tiếp trên tập này.

* Rò rỉ thông tin (Information Leaks)

- Mỗi lần bạn điều chỉnh một siêu tham số dựa trên hiệu suất của mô hình trên tập kiểm định, một phần thông tin về tập kiểm định sẽ rò rỉ vào mô hình.
- Nếu bạn chỉ làm điều này một lần, với một siêu tham số, lượng thông tin rò rỉ sẽ rất nhỏ, và tập kiểm định vẫn đáng tin cậy để đánh giá mô hình.

- Nhưng nếu bạn lặp lại quá trình này nhiều lần - chạy một thí nghiệm, đánh giá trên tập kiểm định, rồi điều chỉnh mô hình - bạn sẽ rò rỉ một lượng thông tin ngày càng lớn về tập kiểm định vào mô hình.

5.2.2 Một số cách phân tập kiểm định

Chia đơn giản	K-fold
Chia tập huấn luyện thành 2 phần: huấn luyện và kiểm định	Chia tập huấn luyện thành k phần như nhau
Mô hình sẽ dùng tập huấn luyện để huấn luyện và tập kiểm định để kiểm soát	Lặp lại K lần, mỗi lần sẽ dùng 1 phần để làm tập kiểm định để kiểm soát, phần còn lại là tập huấn luyện để huấn luyện
	

5.2.3 Đường cơ sở dựa trên ý thức thông thường

Ngoài các giao thức đánh giá khác nhau mà bạn có thể sử dụng, một điều cuối cùng bạn cần biết là việc sử dụng đường cơ sở dựa trên ý thức thông thường (common-sense baselines).

Tầm quan trọng của đường cơ sở

- Huấn luyện một mô hình học sâu giống như nhấn một nút phóng tên lửa trong một thế giới song song. Bạn không thể nghe hay thấy nó. Bạn không thể quan sát trực tiếp quá trình học đa tạp (manifold learning)—nó diễn ra trong không gian hàng nghìn chiều, và ngay cả khi bạn chiếu nó xuống không gian 3D, bạn cũng không thể diễn giải được.
- Phản hồi duy nhất bạn có là các chỉ số kiểm định (validation metrics)—giống như một đồng hồ đo độ cao trên tên lửa vô hình của bạn.
- Điều đặc biệt quan trọng là bạn cần biết liệu mô hình của mình có đang "cất cánh" hay không. Độ cao ban đầu của bạn là bao nhiêu? Mô hình của bạn đạt độ chính xác 15%—liệu đó có tốt không?

Đường cơ sở dựa trên ý thức thông thường

- Trước khi bắt đầu làm việc với một tập dữ liệu, bạn nên luôn chọn một đường cơ sở đơn giản để vượt qua. Nếu bạn vượt qua ngưỡng này, bạn sẽ biết mình đang đi đúng hướng: mô hình của bạn thực sự đang sử dụng thông tin trong dữ liệu đầu vào để đưa ra dự đoán có khả năng khái quát hóa, và bạn có thể tiếp tục.
- Đường cơ sở có thể là:
 - Hiệu suất của một bộ phân loại ngẫu nhiên (random classifier).
 - Hiệu suất của kỹ thuật không học máy đơn giản nhất mà bạn có thể nghĩ ra.

Ví dụ về đường cơ sở

- Phân loại chữ số MNIST:
 - Đường cơ sở: Độ chính xác kiểm định lớn hơn 0.1 (10%), vì một bộ phân loại ngẫu nhiên (random classifier) sẽ đạt độ chính xác 10% khi chọn ngẫu nhiên giữa 10 lớp (0-9).
- Phân loại đánh giá phim IMDB:
 - Đường cơ sở: Độ chính xác kiểm định lớn hơn 0.5 (50%), vì có hai lớp (tích cực/tiêu cực) và phân bố nhãn là cân bằng.
- Phân loại tin tức Reuters:
 - Đường cơ sở: Độ chính xác kiểm định khoảng 0.18-0.19, do có sự mất cân bằng lớp (class imbalance) trong 46 lớp.

5.2.4 Những điều cần lưu ý khi đánh giá mô hình

Khi chọn một giao thức đánh giá, bạn cần chú ý đến các yếu tố sau:

1. Tính đại diện của dữ liệu (Data Representativeness)

- Bạn muốn cả tập huấn luyện và tập kiểm tra đều đại diện cho dữ liệu thực tế.
- Ví dụ sai lầm:
 - Nếu bạn phân loại hình ảnh chữ số và dữ liệu ban đầu được sắp xếp theo lớp (ví dụ: các mẫu được sắp xếp từ lớp 0 đến lớp 9), việc lấy 80% đầu tiên làm tập huấn luyện và 20% còn lại làm tập kiểm tra sẽ dẫn đến:
 - Tập huấn luyện chỉ chứa các lớp 0-7.

- Tập kiểm tra chỉ chứa các lớp 8-9.
- Giải pháp:
 - Bạn nên xáo trộn ngẫu nhiên (randomly shuffle) dữ liệu trước khi chia thành tập huấn luyện và kiểm tra để đảm bảo tính đại diện.

2. Dòng thời gian

- Nếu bạn đang dự đoán tương lai dựa trên quá khứ (ví dụ: thời tiết ngày mai, biến động chứng khoán, v.v.), bạn không được xáo trộn ngẫu nhiên dữ liệu trước khi chia.
- Lý do:
 - Xáo trộn sẽ gây ra rò rỉ thời gian: mô hình sẽ được huấn luyện trên dữ liệu từ tương lai, điều này không thực tế.
- Giải pháp:
 - Đảm bảo tất cả dữ liệu trong tập kiểm tra đều sau dữ liệu trong tập huấn luyện, để mô phỏng đúng tình huống thực tế.

3. Sự trùng lặp trong dữ liệu

- Nếu một số điểm dữ liệu trong tập dữ liệu của bạn xuất hiện nhiều lần (khá phổ biến với dữ liệu thực tế), việc xáo trộn và chia dữ liệu thành tập huấn luyện và kiểm định có thể dẫn đến trùng lặp giữa hai tập.
- Hậu quả:
 - Bạn sẽ vô tình kiểm tra trên một phần dữ liệu huấn luyện, điều này là tối kỵ trong học máy.
- Giải pháp:
 - Đảm bảo tập huấn luyện và tập kiểm định là tách biệt (disjoint), không có mẫu nào xuất hiện ở cả hai tập.

5.3 Tối ưu hóa độ khớp fit

Ba vấn đề phổ biến khi huấn luyện mô hình

Ở giai đoạn này, bạn có thể gặp ba vấn đề phổ biến:

1. Huấn luyện không bắt đầu được: Giá trị mất mát huấn luyện (training loss) không giảm theo thời gian.

2. Huấn luyện bắt đầu tốt, nhưng mô hình không khái quát hóa một cách có ý nghĩa: Bạn không thể vượt qua đường cơ sở dựa trên ý thức thông thường (common-sense baseline) đã đặt ra.
3. Mất mát huấn luyện và kiểm định đều giảm theo thời gian, và bạn có thể vượt qua đường cơ sở, nhưng mô hình không thể quá khớp: Điều này cho thấy bạn vẫn đang thiếu khớp (underfitting).

5.3.1 Điều chỉnh các tham số quan trọng của gradient descent

Đôi khi huấn luyện không bắt đầu được, hoặc dừng lại quá sớm. Giá trị mất mát của bạn bị kẹt (stuck). Đây luôn là vấn đề có thể khắc phục: hãy nhớ rằng bạn có thể huấn luyện một mô hình để khớp với dữ liệu ngẫu nhiên. Ngay cả khi bài toán của bạn không có ý nghĩa, bạn vẫn có thể huấn luyện được một thứ gì đó - dù chỉ là ghi nhớ dữ liệu huấn luyện.

Nguyên nhân của vấn đề

- Khi huấn luyện không bắt đầu được, vấn đề luôn nằm ở cấu hình của quá trình gradient descent:
 - Lựa chọn bộ tối ưu hóa (optimizer).
 - Phân phối giá trị ban đầu của trọng số (weights) trong mô hình.
 - Tỷ lệ học (learning rate).
 - Kích thước lô (batch size).
- Tất cả các tham số này đều phụ thuộc lẫn nhau, nên thường chỉ cần điều chỉnh tỷ lệ học và kích thước lô trong khi giữ các tham số còn lại không đổi là đủ.

Ví dụ cụ thể: Huấn luyện mô hình MNIST với tỷ lệ học không phù hợp

- Tình huống:
 - Thông thường, tỷ lệ học được chọn nhỏ (ví dụ: 0.001 hoặc 0.01) để đảm bảo gradient descent cập nhật trọng số một cách mượt mà.
 - Nếu tỷ lệ học quá lớn (như 1), các bước cập nhật trọng số sẽ quá lớn, khiến mô hình "nhảy" qua các điểm tối ưu và không hội tụ.
- Kết quả:
 - Giá trị mất mát có thể dao động mạnh hoặc không giảm, vì mô hình không thể tìm được hướng đi đúng để tối ưu hóa.

- Trong trường hợp này, huấn luyện sẽ không bắt đầu được hoặc dừng lại quá sớm.

Cách khắc phục

- Giảm tỷ lệ học:
 - Thử giảm tỷ lệ học xuống một giá trị nhỏ hơn, ví dụ: 0.01 hoặc 0.001, và quan sát xem giá trị mất mát có bắt đầu giảm hay không.
- Điều chỉnh kích thước lô:
 - Kích thước lô lớn (như 512) có thể làm gradient ổn định hơn, nhưng nếu quá lớn, mô hình có thể bỏ qua các chi tiết nhỏ trong dữ liệu.
 - Thử giảm kích thước lô (ví dụ: xuống 32 hoặc 64) để gradient được cập nhật thường xuyên hơn.
- Thay đổi bộ tối ưu hóa:
 - Nếu vẫn không hiệu quả, thử một bộ tối ưu hóa khác (như Adam thay vì RMSprop), vì các bộ tối ưu hóa khác nhau có cách xử lý gradient khác nhau.
- Kiểm tra phân phối trọng số ban đầu:
 - Đảm bảo trọng số ban đầu được khởi tạo hợp lý (ví dụ: sử dụng khởi tạo Xavier/Glorot hoặc He initialization), để tránh các giá trị ban đầu quá lớn hoặc quá nhỏ.

5.3.2 Tận dụng các kiến trúc tiên nghiệm

- Tình huống: Mô hình huấn luyện được (giá trị mất mát giảm), nhưng không khái quát hóa tốt—chỉ số kiểm định không cải thiện, không vượt qua được đường cơ sở dựa trên ý thức thông thường.
- Nguyên nhân:
 - Dữ liệu không đủ thông tin: Đầu vào không chứa đủ thông tin để dự đoán đầu ra (như trong ví dụ MNIST với nhãn xáo trộn, không thể khái quát hóa).
 - Kiến trúc mô hình không phù hợp: Mô hình không được thiết kế để giải quyết bài toán cụ thể.
 - Ví dụ: Một mô hình kết nối đầy đủ (densely connected) không hiệu quả cho bài toán dự đoán chuỗi thời gian, trong khi mô hình hồi quy (recurrent) lại hoạt động tốt.

- Giải pháp:
 - Sử dụng kiến trúc phù hợp với bài toán, dựa trên các kiến thức tiên nghiệm (architecture priors).
 - Tìm hiểu các phương pháp tốt nhất (best practices) cho loại bài toán bạn đang giải quyết (hình ảnh, văn bản, chuỗi thời gian, v.v.).

5.3.3 Tăng dung lượng mô hình (Increasing Model Capacity)

- Tình huống: Mô hình huấn luyện được, chỉ số kiểm định cải thiện và vượt qua đường cơ sở, nhưng không thể quá khớp—giá trị mất mát kiểm định chững lại hoặc giảm rất chậm, cho thấy mô hình vẫn thiếu khớp (underfitting).
- Nguyên nhân:
 - Mô hình có dung lượng (capacity) không đủ để học các mẫu phức tạp trong dữ liệu.
 - Ví dụ: Một mô hình logistic regression đơn giản (chỉ có 1 tầng Dense) trên MNIST không thể học các mẫu phức tạp, dẫn đến thiếu khớp.
- Giải pháp:
 - Tăng dung lượng mô hình để có khả năng biểu diễn (representational power) lớn hơn:
 - Thêm nhiều tầng (layers).
 - Tăng số nơ-ron trong mỗi tầng (bigger layers).
 - Sử dụng các loại tầng phù hợp hơn với bài toán (better architecture priors).
- Kết quả:
 - Một mô hình lớn hơn (ví dụ: 2 tầng Dense với 96 nơ-ron mỗi tầng) có thể học nhanh hơn, đạt độ khớp tốt, và bắt đầu quá khớp sau một số epoch (như 8 epoch trong ví dụ MNIST).

5.4 Cải thiện khả năng khái quát hóa

Khi mô hình của bạn đã cho thấy khả năng khái quát hóa nhất định và có thể quá khớp, đã đến lúc tập trung vào việc tối đa hóa khả năng khái quát hóa.

5.4.1 Quản lý dữ liệu

- Khái quát hóa phụ thuộc vào dữ liệu:

- Khả năng khái quát hóa trong học sâu đến từ cấu trúc tiềm tàng (latent structure) của dữ liệu.
- Nếu dữ liệu cho phép nội suy mượt mà giữa các mẫu, bạn có thể huấn luyện một mô hình học sâu để khái quát hóa.
- Nếu bài toán quá nhiều hoặc mang tính rời rạc (như sắp xếp danh sách), học sâu sẽ không hiệu quả, vì học sâu là khớp đường cong (curve fitting), không phải phép màu.
- Tầm quan trọng của dữ liệu phù hợp:
 - Đảm bảo bạn làm việc với một tập dữ liệu phù hợp.
 - Đầu tư vào việc thu thập dữ liệu thường mang lại lợi tức cao hơn so với việc phát triển một mô hình tốt hơn.
- Các bước quản lý dữ liệu:
 - Đảm bảo đủ dữ liệu: Cần lấy mẫu dày đặc (dense sampling) trong không gian đầu vào-đầu ra. Dữ liệu càng nhiều, mô hình càng tốt. Một số bài toán tưởng chừng không thể giải quyết được sẽ trở nên khả thi với tập dữ liệu lớn hơn.
 - Giảm lỗi gán nhãn: Kiểm tra dữ liệu đầu vào để phát hiện bất thường, và rà soát nhãn.
 - Làm sạch dữ liệu: Xử lý giá trị thiếu (missing values) (sẽ được đề cập ở chương sau).
 - Lựa chọn đặc trưng (feature selection): Nếu có nhiều đặc trưng nhưng không chắc cái nào hữu ích, hãy chọn lọc đặc trưng.

5.4.2 Kỹ thuật đặc trưng

- Kỹ thuật đặc trưng là gì?:
 - Là quá trình sử dụng kiến thức của bạn về dữ liệu và thuật toán học máy (ở đây là mạng nơ-ron) để biến đổi dữ liệu đầu vào một cách thủ công (hardcoded, non-learned) trước khi đưa vào mô hình, giúp thuật toán hoạt động tốt hơn.
 - Dữ liệu thô thường không đủ để mô hình học trực tiếp; cần trình bày dữ liệu theo cách giúp mô hình dễ học hơn.
- Ví dụ minh họa: Đọc giờ trên đồng hồ:
 - Dữ liệu thô: Hình ảnh đồng hồ (lưới pixel).

- Nếu dùng pixel thô làm đầu vào, bài toán trở nên khó, cần một mạng nơ-ron tích chập (CNN) và nhiều tài nguyên tính toán.
- Đặc trưng tốt hơn: Tọa độ (x, y) của đầu kim đồng hồ.
 - Dùng một đoạn code ngắn để tìm tọa độ đầu kim, sau đó mô hình học máy đơn giản có thể học để liên kết tọa độ với thời gian.
- Đặc trưng tốt hơn nữa: Góc theta của kim đồng hồ (dùng tọa độ cực).
 - Biến đổi tọa độ (x, y) thành góc theta, bài toán trở nên rất dễ, không cần học máy—chỉ cần làm tròn và tra cứu là đủ.
- Bản chất của kỹ thuật đặc trưng:
 - Làm cho bài toán dễ hơn bằng cách biểu diễn dữ liệu theo cách đơn giản hơn.
 - Làm cho đa tạp tiềm tàng (latent manifold) mượt mà hơn, đơn giản hơn, có tổ chức tốt hơn.
 - Yêu cầu hiểu sâu về bài toán.
- Vai trò của kỹ thuật đặc trưng:
 - Trước thời học sâu, kỹ thuật đặc trưng là phần quan trọng nhất trong quy trình học máy, vì các thuật toán nông (shallow algorithms) không thể tự học các đặc trưng hữu ích.
 - Học sâu hiện đại giảm nhu cầu kỹ thuật đặc trưng, vì mạng nơ-ron có thể tự động trích xuất đặc trưng từ dữ liệu thô.
- Tuy nhiên, kỹ thuật đặc trưng vẫn quan trọng:
 - Giải quyết bài toán hiệu quả hơn: Đặc trưng tốt giúp giải bài toán với ít tài nguyên hơn (ví dụ: đọc giờ trên đồng hồ không cần CNN).
 - Giảm nhu cầu dữ liệu: Với ít mẫu dữ liệu, giá trị thông tin của đặc trưng trở nên quan trọng, vì học sâu cần nhiều dữ liệu để tự học đặc trưng.

5.4.3 Sử dụng dừng sớm

- Mô hình học sâu luôn quá tham số hóa:
 - Mô hình học sâu có nhiều tham số hơn mức cần thiết để khớp với đa tạp tiềm tàng của dữ liệu.
 - Quá tham số hóa không phải vấn đề, vì bạn không bao giờ khớp hoàn toàn mô hình—khớp hoàn toàn sẽ không khái quát hóa.
- Dừng sớm để tối ưu khái quát hóa:
 - Bạn cần dừng huấn luyện trước khi đạt giá trị mất mát huấn luyện nhỏ nhất, tại điểm mà mô hình đạt độ khớp khái quát hóa tốt nhất (ranh giới giữa thiếu khớp và quá khớp).
- Phương pháp dừng sớm:
 - Huấn luyện mô hình lâu hơn cần thiết để tìm số epoch tối ưu (dựa trên chỉ số kiểm định).
 - Huấn luyện lại mô hình mới với số epoch tối ưu.
 - Cách hiệu quả hơn: Sử dụng EarlyStopping callback trong Keras để tự động dừng huấn luyện khi chỉ số kiểm định ngừng cải thiện, đồng thời lưu lại trạng thái mô hình tốt nhất (sẽ được đề cập ở chương 7).
- Lợi ích:
 - Tránh huấn luyện dư thừa, tiết kiệm tài nguyên.
 - Đảm bảo mô hình dừng tại điểm khái quát hóa tốt nhất.

5.4.4 Điều chuẩn mô hình

- Điều chuẩn là gì?:
 - Là tập hợp các kỹ thuật ngăn cản mô hình khớp hoàn toàn với dữ liệu huấn luyện, nhằm cải thiện hiệu suất trên tập kiểm định.
 - Điều chuẩn làm cho mô hình đơn giản hơn, mượt mà hơn, ít đặc thù với tập huấn luyện, từ đó khái quát hóa tốt hơn.
- Điều chuẩn cần đánh giá chính xác:
 - Điều chuẩn chỉ hiệu quả nếu bạn có phương pháp đánh giá đáng tin cậy để đo lường khái quát hóa.

Giảm kích thước mạng (Reducing the Network's Size)

- Nguyên lý:

- Mô hình nhỏ hơn (ít tham số hơn) sẽ ít quá khớp hơn.
- Nếu mô hình có ít tài nguyên ghi nhớ, nó không thể ghi nhớ toàn bộ dữ liệu huấn luyện, buộc phải học các biểu diễn nén (compressed representations) có khả năng dự đoán tốt—loại biểu diễn mà chúng ta mong muốn.
- Cân bằng:
 - Mô hình không được quá nhỏ để tránh thiếu khớp (underfitting).
 - Cần tìm sự cân bằng giữa quá nhiều dung lượng và không đủ dung lượng.
- Thực nghiệm trên mô hình IMDB:
 - Mô hình gốc: 2 tầng Dense(16, activation="relu"), quá khớp sau 4 epoch.
 - Mô hình nhỏ hơn: 2 tầng Dense(4, activation="relu"), quá khớp muộn hơn (sau 6 epoch) và suy giảm chậm hơn.
 - Mô hình lớn hơn: 2 tầng Dense(512, activation="relu"), quá khớp ngay sau 1 epoch, mất mát kiểm định dao động mạnh (noisier).
- Cách thực hiện:
 - Bắt đầu với mô hình nhỏ, tăng dần số tầng hoặc số nơ-ron cho đến khi thấy lợi ích giảm dần (diminishing returns) trên mất mát kiểm định.

Thêm điều chuẩn trọng số (Adding Weight Regularization)

- Nguyên lý:
 - Dựa trên dao cạo Occam (Occam's razor): Giải thích đơn giản nhất thường là đúng nhất.
 - Trong học máy: Mô hình đơn giản (có phân phối tham số ít entropy hoặc ít tham số hơn) ít có khả năng quá khớp hơn mô hình phức tạp.
- Điều chuẩn trọng số:
 - Áp đặt ràng buộc lên độ phức tạp của mô hình bằng cách buộc trọng số chỉ nhận các giá trị nhỏ, làm cho phân phối trọng số đều hơn (regular).

- Thêm một chi phí (cost) vào hàm mất mát, liên quan đến việc có trọng số lớn.
- Hai loại điều chuẩn trọng số:
 - L1 regularization: Chi phí tỷ lệ với giá trị tuyệt đối của trọng số (L1 norm).
 - L2 regularization (weight decay): Chi phí tỷ lệ với bình phương giá trị trọng số (L2 norm).
- Thực nghiệm trên mô hình IMDB:
 - Thêm L2 regularization với hệ số 0.002 vào mô hình gốc.
 - Kết quả: Mô hình kháng quá khớp tốt hơn, mất mát kiểm định thấp hơn so với mô hình gốc, dù cả hai có cùng số tham số.
- Lưu ý:
 - Điều chuẩn trọng số thường được dùng cho mô hình nhỏ.
 - Với mô hình lớn (quá tham số hóa), điều chuẩn trọng số ít ảnh hưởng đến dung lượng và khái quát hóa.

Thêm Dropout

- Dropout là gì?:
 - Là một trong những kỹ thuật điều chuẩn hiệu quả và phổ biến nhất cho mạng nơ-ron, do Geoff Hinton và nhóm của ông tại Đại học Toronto phát triển.
 - Dropout ngẫu nhiên bỏ qua (drop out) (đặt về 0) một số đặc trưng đầu ra của tầng trong quá trình huấn luyện.
 - Ví dụ: Một tầng trả về vector $[0.2, 0.5, 1.3, 0.8, 1.1]$, sau khi áp dụng dropout (tỷ lệ 0.5), vector có thể là $[0, 0.5, 1.3, 0, 1.1]$.
- Tỷ lệ dropout:
 - Tỷ lệ đặc trưng bị bỏ qua, thường từ 0.2 đến 0.5.
- Tại thời điểm kiểm tra (test time):
 - Không bỏ qua đơn vị nào, nhưng đầu ra của tầng được giảm tỷ lệ (scaled down) theo tỷ lệ dropout (ví dụ: nhân với 0.5 nếu tỷ lệ dropout là 0.5), để cân bằng với việc có nhiều đơn vị hoạt động hơn so với lúc huấn luyện.

- Cơ chế hoạt động:
 - Trong huấn luyện: Ngẫu nhiên đặt một số giá trị về 0.
 - Trong kiểm tra: Giảm tỷ lệ đầu ra hoặc áp dụng cả hai thao tác trong huấn luyện (phổ biến hơn).
- Lý do hiệu quả:
 - Dropout phá vỡ các mẫu ngẫu nhiên không quan trọng mà mô hình có thể ghi nhớ nếu không có nhiễu.
 - Giới thiệu nhiễu vào đầu ra của tầng, ngăn overfitting
- Thực nghiệm trên mô hình IMDB:
 - Thêm 2 tầng Dropout(0.5) sau các tầng Dense.
 - Kết quả: Mô hình kháng quá khớp tốt hơn, đạt mất mát kiểm định thấp hơn so với mô hình gốc và cả mô hình dùng L2 regularization.
- Lưu ý:
 - Dropout là kỹ thuật điều chuẩn phổ biến cho mạng nơ-ron lớn, nơi điều chuẩn trọng số ít hiệu quả.

CHƯƠNG 6: QUY TRÌNH CHUNG CỦA HỌC MÁY

Quy trình chung của học máy trải qua 3 giai đoạn

- Định hình nhiệm vụ: hiểu mục đích, nhiệm vụ đang làm, hiểu công việc được giao, thu thập dữ liệu, hiểu nó và tìm phương pháp đo đặc độ thành công của nhiệm vụ
- Phát triển mô hình: Chuẩn bị dữ liệu, xác định phương thức và một lối đi, huấn luyện 1 mô hình và dần cải thiện nó
- Chạy mô hình: Trình bày với người làm chung, đưa mô hình lên một web/ app hoặc thiết bị, rồi dần dần thu thập thêm dữ liệu cho các mô hình tiếp theo.

6.1 Xác định bài toán và thu thập dữ liệu

6.1.1 Xác định bài toán

- Câu hỏi quan trọng:
 - Dữ liệu đầu vào và đầu ra là gì? Có đủ dữ liệu huấn luyện không?
 - Loại bài toán học máy: phân loại nhị phân, đa lớp, hồi quy, v.v.?
 - Giải pháp hiện tại là gì? Có ràng buộc nào (như độ trễ, bảo mật)?
- Ví dụ:
 - Tìm kiếm ảnh: Phân loại đa lớp, đa nhãn.
 - Phát hiện spam: Phân loại nhị phân.
 - Đề xuất nhạc: Không dùng học sâu, mà dùng phân tích ma trận.
- Giả thuyết:
 - Đầu vào có thể dự đoán đầu ra.
 - Dữ liệu đủ thông tin để học mối quan hệ đầu vào-đầu ra.

6.1.2 Thu thập dữ liệu

- Thu thập dữ liệu là bước tốn kém, mất thời gian nhất.
- Ví dụ:
 - Tìm kiếm ảnh: Gắn nhãn hàng trăm nghìn ảnh.
 - Phát hiện spam: Sử dụng dữ liệu bên ngoài, gắn nhãn thủ công.
 - Đề xuất nhạc: Dùng dữ liệu "likes" sẵn có.
- Tầm quan trọng của dữ liệu:

- Dữ liệu chất lượng cao (nhiều mẫu, nhãn đáng tin cậy, đặc trưng tốt) quyết định khả năng khái quát hóa.
 - Đầu tư vào dữ liệu thường hiệu quả hơn cải thiện mô hình.
- Gắn nhãn dữ liệu:
 - Tự gắn nhãn, dùng nền tảng đông người (Mechanical Turk), hoặc công ty chuyên nghiệp.
 - Cân nhắc: Cần chuyên gia không? Có thể đào tạo người gắn nhãn không? Phần mềm gắn nhãn nào hiệu quả?
- Dữ liệu không đại diện:
 - Dữ liệu huấn luyện phải đại diện cho dữ liệu thực tế, nếu không mô hình sẽ thất bại (ví dụ: ảnh món ăn từ mạng xã hội không đại diện cho ảnh chụp từ điện thoại).
 - Concept drift: Dữ liệu thay đổi theo thời gian (như sở thích âm nhạc, xu hướng spam), cần liên tục thu thập và huấn luyện lại.
- Sampling bias:
 - Lỗi lấy mẫu: Dữ liệu không đại diện do cách thu thập (ví dụ: khảo sát qua điện thoại năm 1948 thiên về người giàu, dẫn đến dự đoán sai kết quả bầu cử).

6.1.3 Hiểu dữ liệu

- Không nên coi dữ liệu như hộp đen; cần khám phá và trực quan hóa dữ liệu trước khi huấn luyện:
 - Xem mẫu dữ liệu (hình ảnh, văn bản) và nhãn.
 - Vẽ histogram giá trị đặc trưng, kiểm tra phân bố.
 - Vẽ dữ liệu vị trí trên bản đồ, tìm mẫu.
 - Kiểm tra giá trị thiếu, mất cân bằng lớp, rò rỉ mục tiêu (target leaking).
- Hiểu dữ liệu giúp phát triển đặc trưng và phát hiện vấn đề.

6.1.4 Chọn thước đo khả năng thành công

- Cần định nghĩa thước đo thành công (accuracy, precision, recall, ROC AUC, v.v.) để hướng dẫn các lựa chọn kỹ thuật.
- Thước đo phải phù hợp với mục tiêu cao hơn (như thành công kinh doanh):

- Phân loại cân bằng: Accuracy, ROC AUC.
- Phân loại mất cân bằng: Precision, recall, weighted accuracy.
- Có thể cần thước đo tùy chỉnh (xem các cuộc thi trên Kaggle để hiểu đa dạng thước đo).

6.2 Phát triển mô hình

Thực tế, trong học máy, phát triển mô hình không phải là bước khó nhất. Những thử thách lớn hơn nằm ở việc xác định đúng vấn đề cần giải quyết và quá trình thu thập, làm sạch, chú thích dữ liệu.

6.2.1 Chuẩn bị dữ liệu

Trước khi đưa dữ liệu thô vào mô hình học sâu, bạn cần xử lý nó để phù hợp với yêu cầu của mạng nơ-ron. Quá trình này gọi là tiền xử lý dữ liệu, bao gồm các bước như vector hóa, chuẩn hóa và xử lý giá trị thiếu. Tùy thuộc vào loại dữ liệu (văn bản, hình ảnh, âm thanh), sẽ có các kỹ thuật riêng biệt mà ta sẽ khám phá trong các chương sau. Dưới đây là những bước cơ bản áp dụng cho mọi lĩnh vực:

- **Vector hóa:** Mạng nơ-ron yêu cầu dữ liệu đầu vào và đầu ra phải ở dạng tensor số thực (hoặc đôi khi là số nguyên, chuỗi). Dù là âm thanh, hình ảnh hay văn bản, bạn cần chuyển đổi chúng thành tensor – quá trình này gọi là vector hóa. Ví dụ, trong phân loại văn bản, ta chuyển danh sách từ thành số nguyên rồi mã hóa one-hot để tạo tensor float32. Với bài toán phân loại chữ số MNIST hay dự đoán giá nhà, dữ liệu đã được vector hóa sẵn, nên ta không cần làm bước này.
- **Chuẩn hóa giá trị:** Dữ liệu đầu vào cần được chuẩn hóa để mạng học hiệu quả hơn. Chẳng hạn, trong bài MNIST, ảnh gốc có giá trị từ 0-255 (mức xám), ta chuyển thành float32 và chia cho 255 để đưa về khoảng 0-1. Trong dự đoán giá nhà, các đặc trưng có phạm vi khác nhau (số nhỏ, số lớn), nên ta chuẩn hóa từng đặc trưng sao cho có trung bình bằng 0 và độ lệch chuẩn bằng 1. Tại sao cần làm vậy? Nếu dữ liệu có giá trị lớn (như số nguyên nhiều chữ số) hoặc không đồng nhất (một đặc trưng từ 0-1, đặc trưng khác từ 100-200), gradient sẽ bị ảnh hưởng mạnh, khiến mạng khó hội tụ. Dữ liệu lý tưởng nên:
 - Có giá trị nhỏ, thường trong khoảng 0-1.
 - Đồng nhất, các đặc trưng có phạm vi tương đương nhau.
 - (Tùy chọn) Chuẩn hóa thêm để mỗi đặc trưng có trung bình 0 và độ lệch chuẩn 1, dùng công thức: $x \text{ -= } x.\text{mean}(\text{axis}=0)$; $x \text{ /= } x.\text{std}(\text{axis}=0)$ (với x là ma trận dữ liệu).

- **Xử lý giá trị thiếu:** Thỉnh thoảng dữ liệu của bạn sẽ có giá trị bị thiếu. Chẳng hạn, trong bài toán giá nhà, nếu cột “tỷ lệ tội phạm bình quân đầu người” không có dữ liệu cho một số mẫu, ta cần xử lý. Có vài cách:
 - Với đặc trưng phân loại: Tạo thêm một hạng mục mới như “giá trị bị thiếu” – mô hình sẽ tự học ý nghĩa của nó.
 - Với đặc trưng số: Không nên điền 0 tùy tiện vì có thể gây gián đoạn trong không gian tiềm ẩn, làm khó khái quát hóa. Thay vào đó, dùng trung bình hoặc trung vị của đặc trưng đó. Bạn cũng có thể huấn luyện một mô hình phụ để dự đoán giá trị thiếu dựa trên các đặc trưng khác.

6.2.2 Chọn phương pháp đánh giá

Mục tiêu của mô hình là khái quát hóa tốt, và mọi quyết định trong quá trình phát triển đều dựa trên các chỉ số đánh giá. Bạn cần chọn một phương pháp đánh giá để ước lượng chính xác chỉ số thành công (như độ chính xác) trên dữ liệu thực tế. Các lựa chọn phổ biến gồm:

- **Tách tập kiểm tra riêng:** Dùng khi bạn có nhiều dữ liệu, giữ một phần làm tập kiểm tra để đánh giá.
- **K-fold cross-validation:** Phù hợp khi dữ liệu ít, chia dữ liệu thành K phần, luân phiên dùng từng phần làm tập kiểm tra.
- **Lặp lại K-fold:** Dùng khi dữ liệu rất ít, lặp lại K-fold nhiều lần để đánh giá chính xác hơn.
Hãy đảm bảo tập kiểm tra đại diện tốt cho dữ liệu thực tế và không trùng lặp với tập huấn luyện.

6.2.3 Vượt qua baseline

Khi bắt đầu xây dựng mô hình, mục tiêu đầu tiên là đạt được sức mạnh thống kê – tức là tạo một mô hình nhỏ vượt qua một baseline đơn giản (ví dụ: dự đoán ngẫu nhiên). Ba yếu tố cần chú ý:

- **Kỹ thuật đặc trưng:** Lọc bỏ đặc trưng không hữu ích và tạo đặc trưng mới dựa trên hiểu biết về vấn đề.
- **Chọn kiến trúc:** Quyết định dùng mạng nào – mạng dày đặc, CNN, RNN hay Transformer? Học sâu có phải lựa chọn tốt nhất không?
- **Cấu hình huấn luyện:** Chọn hàm mất mát, kích thước batch, tốc độ học phù hợp. Ví dụ, với phân loại nhị phân, dùng sigmoid và `binary_crossentropy`; với phân loại đa nhãn, dùng softmax và `categorical_crossentropy`.

Nếu thử nhiều kiến trúc hợp lý mà vẫn không vượt qua baseline, có thể dữ liệu không chứa đủ thông tin để giải quyết vấn đề – lúc đó cần xem lại giả thuyết ban đầu.

6.2.4 Tăng quy mô: Tạo mô hình quá khớp

Sau khi có mô hình vượt baseline, hãy kiểm tra xem nó đủ mạnh chưa bằng cách làm nó quá khớp (overfit). Cách làm đơn giản:

- Thêm tầng, tăng số đơn vị trong mỗi tầng, huấn luyện thêm epoch.
- Theo dõi mất mát và chỉ số trên tập huấn luyện/kiểm tra. Khi hiệu suất trên tập kiểm tra giảm, bạn đã đạt quá khớp.
Mục đích là tìm ranh giới giữa thiếu khớp (undercapacity) và quá khớp (overcapacity) để biết mô hình cần bao lớn.

6.2.5 Điều chỉnh và tối ưu

Khi đã quá khớp, bạn chuyển sang tối ưu khái quát hóa – bước này tốn nhiều thời gian nhất. Hãy thử:

- Thay đổi kiến trúc: thêm/bớt tầng, dùng dropout, hoặc thêm L1/L2 regularization nếu mô hình nhỏ.
- Tinh chỉnh siêu tham số: số đơn vị, tốc độ học, v.v.
- Cải thiện dữ liệu: thu thập thêm, tạo đặc trưng tốt hơn, bỏ đặc trưng không hữu ích.
Lưu ý: Mỗi lần dùng phản hồi từ tập kiểm tra để chỉnh mô hình, bạn vô tình làm rò rỉ thông tin, dẫn đến quá khớp lên tập kiểm tra nếu lặp lại quá nhiều

6.3 Triển khai mô hình

Mô hình đã vượt qua đánh giá cuối trên tập test, sẵn sàng triển khai.

6.3.1 Giải thích và đặt kỳ vọng

- Giải thích rõ cho các bên liên quan về khả năng và hạn chế của mô hình, tránh kỳ vọng không thực tế (như hiểu biết giống người).
- Trình bày hiệu suất cụ thể (ví dụ: tỷ lệ lỗi âm tính/dương tính) thay vì số liệu chung chung (như 98% chính xác).
- Thảo luận các tham số quan trọng (ngưỡng xác suất, v.v.) dựa trên mục tiêu kinh doanh.

6.3.2 Triển khai mô hình suy luận

- **API REST:** Dùng TensorFlow Serving hoặc dịch vụ đám mây để triển khai mô hình qua API, phù hợp khi cần truy cập internet, không yêu cầu độ trễ thấp.
- **Trên thiết bị:** Dùng TensorFlow Lite cho điện thoại hoặc thiết bị nhúng khi cần độ trễ thấp, dữ liệu nhạy cảm, hoặc không có mạng.
- **Trên trình duyệt:** Dùng TensorFlow.js để chạy mô hình trên máy người dùng, giảm chi phí server, phù hợp ứng dụng JavaScript.
- **Tối ưu hóa:** Giảm kích thước mô hình bằng pruning (cắt tỉa trọng số) hoặc quantization (nén trọng số).

6.3.3 Giám sát mô hình

- Theo dõi hiệu suất thực tế, tác động kinh doanh (ví dụ: tỷ lệ nhấp quảng cáo).
- Dùng A/B testing để đánh giá ảnh hưởng mô hình.
- Kiểm tra thủ công định kỳ hoặc khảo sát người dùng nếu không thể kiểm tra tay.

6.3.4 Duy trì mô hình

- Theo dõi sự thay đổi của dữ liệu thực tế (concept drift).
- Thu thập, chú thích dữ liệu mới, cải thiện mô hình để thay thế phiên bản cũ khi cần.