
Đắm mình vào Học Sâu

Release 0.7.0

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Mar 08, 2020

Contents

1 | Giới thiệu từ nhóm dịch

1.1 Mục tiêu của dự án

Trong những năm gần đây, học sâu là một trong các lĩnh vực được quan tâm nhiều nhất trong các trường đại học cũng như các công ty công nghệ. Ngày càng nhiều các diễn đàn liên quan đến học máy và học sâu với lượng thành viên và chủ đề trao đổi ngày một tăng. Một trong các diễn đàn tiếng Việt nổi bật nhất là [Forum Machine Learning cơ bản¹](https://www.facebook.com/groups/machinelearningcoban/) và [Diễn đàn Machine Learning cơ bản²](https://forum.machinelearningcoban.com/) với hơn 35 ngàn thành viên và hàng chục chủ đề mới mỗi ngày.

Qua các diễn đàn đó, chúng tôi nhận ra rằng nhu cầu tìm hiểu lĩnh vực này ngày một tăng trong khi lượng tài liệu tiếng Việt còn rất hạn chế. Đặc biệt, các tài liệu tiếng Việt còn chưaхват quan trọng cách dịch, khiến độc giả bối rối trước quá nhiều thông tin nhưng lại quá ít thông tin đầy đủ. Việc này thúc đẩy chúng tôi tìm và dịch những cuốn sách được quan tâm nhiều về lĩnh vực này.

Nhóm dịch đã bước đầu thành công khi dịch cuốn [Machine Learning Yearning³](#) của tác giả Andrew Ng. Cuốn sách này đề cập đến các vấn đề cần lưu ý khi xây dựng các hệ thống học máy, trong đó đề cập đến nhiều kiến thức thực tế khi thực hiện dự án. Tuy nhiên, cuốn sách này phần nào hướng tới những người đã có những kinh nghiệm nhất định đã đang tham gia các dự án học máy. Chúng tôi vẫn khao khát được mang một tài liệu đầy đủ hơn với đủ kiến thức toán nền tảng, cách triển khai các công thức toán bằng mã nguồn, cùng với cách triển khai một hệ thống thực tế trên một nền tảng học sâu được nhiều người sử dụng. Và quan trọng hơn, các kiến thức này phải cập nhật các xu hướng học máy mới nhất.

Sau nhiều ngày tìm kiếm các cuốn sách về học máy/học sâu được các trường đại học lớn trên thế giới sử dụng trong quá trình giảng dạy, chúng tôi quyết định dịch cuốn [Dive into Deep Learning⁴](#) của nhóm tác giả từ công ty Amazon. Cuốn này hội tụ đủ các yếu tố: có giải thích toán dễ hiểu, có code đi kèm cho những bạn muốn thực hành ngay khi học xong lý thuyết, cập nhật đầy đủ những khía cạnh của học sâu, và quan trọng nhất là không đòi hỏi bản quyền để dịch. Chúng tôi đã liên hệ với nhóm tác giả và họ rất vui mừng khi cuốn sách sắp được phổ biến rộng rãi hơn nữa.

Hiện cuốn sách vẫn đang được thực hiện và sắp ra mắt phiên bản 0.7.0. Nhóm tác giả có lời khuyên chúng tôi có thể dịch bản 0.7.0 này ở branch [numpy2⁵](#) và có thể cập nhật khi cuốn sách được xuất bản. Chúng tôi cũng chọn bản này vì nó sử dụng thư viện chính là numpy (tích hợp trong MXNet), một thư viện xử lý mảng nhiều chiều phổ biến mà theo chúng tôi, người làm về học máy, học sâu và khoa học dữ liệu cần biết.

Để có thể thực hiện dự án dịch cuốn sách hơn 800 trang này, chúng tôi rất cần sự chung tay của cộng đồng. Mọi sự đóng góp đều đáng quý và sẽ được ghi nhận. Chúng tôi hy vọng cuốn sách sẽ được hoàn thành trong năm 2020. Và sau đó nó có thể trở thành giáo trình trong các trường đại học. Hy vọng một ngày chúng ta có thể nhìn thấy một trường của Việt Nam trong danh sách này:

¹ <https://www.facebook.com/groups/machinelearningcoban/>

² <https://forum.machinelearningcoban.com/>

³ https://github.com/aiivvn/Machine-Learning-Yearning-Vietnamese-Translation/blob/master/chapters/all_chapters.md

⁴ <https://www.d2l.ai/>

⁵ <https://github.com/d2l-ai/d2l-en/tree/numpy2>

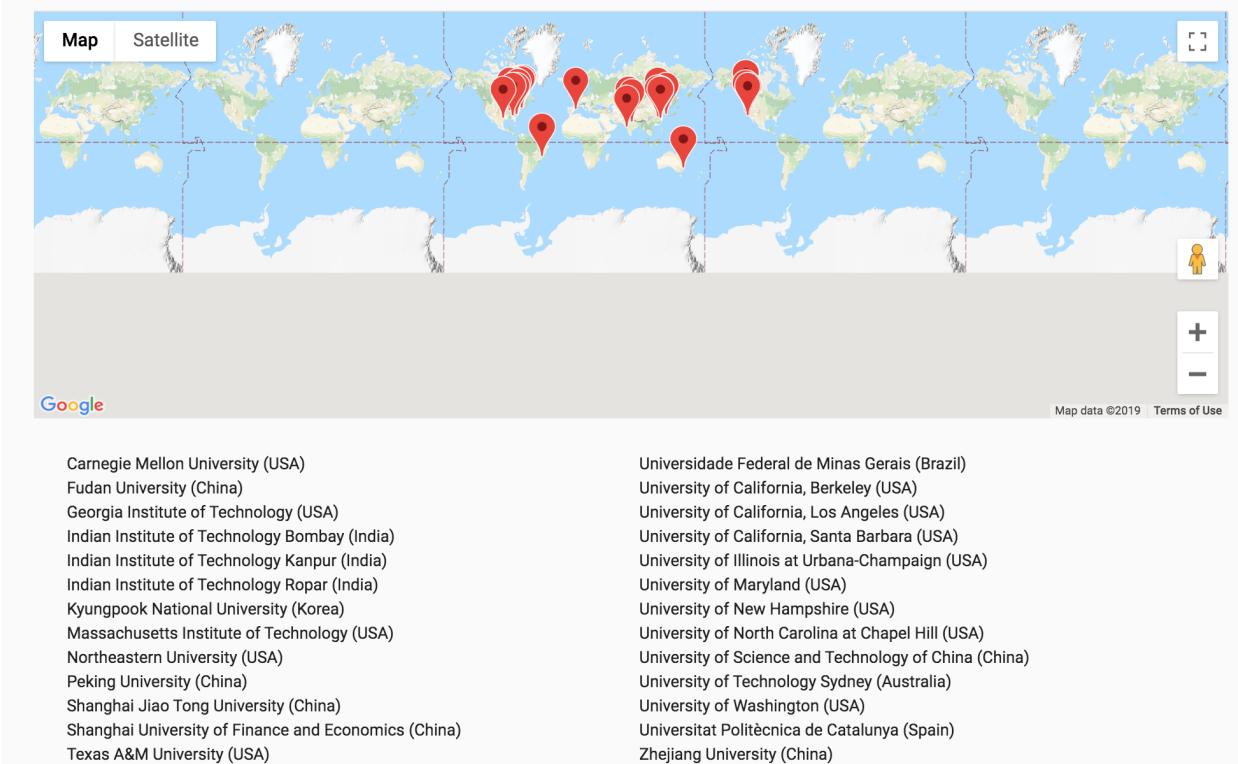


Fig. 1.1.1: Danh sách các trường đại học sử dụng cuốn sách này

1.1.1 Trình tự dịch

Đây cũng là các nội dung được đề cập trong cuốn sách:

- Preface
- Introduction
- Preliminaries
- Mathematics for Deep Learning
- Tools for Deep Learning
- Linear Neural Networks
- Multilayer Perceptrons
- Deep Learning Computation
- Convolutional Neural Networks
- Modern Convolutional Networks
- Recurrent Neural Networks
- Modern Recurrent Networks
- Attention Mechanisms
- Optimization Algorithms

- Computational Performance
- Computer Vision
- Natural Language Processing
- Recommender Systems
- Generative Adversarial Networks

1.1.2 Diễn đàn

Nội dung cuốn sách này rất phong phú và có nhiều bài tập ở cuối mỗi phần. Các bạn có thể tham gia thảo luận nội dung và bài tập của cuốn sách [tại đây](#)⁶.

1.2 Hướng dẫn đóng góp

Những việc bạn có thể làm để đóng góp vào dự án:

- Tham gia dịch thông qua các Pull Request
- Tham gia review các Pull Request
- Hỗ trợ kỹ thuật
- Sửa các lỗi chính tả, ngữ pháp, những điểm chưa nhát quán trong cách dịch
- Start GitHub repo của dự án
- Chia sẻ dự án tới nhiều người hơn

Bạn có thể tham gia thảo luận tại [Slack](#) của nhóm dịch⁷ hoặc đóng góp trực tiếp trên [GitHub repo](#)⁸.

Dưới đây là chi tiết về ba việc quan trọng nhất:

1.2.1 Dịch

Nếu bạn đã quen với GitHub, bạn có thể tham khảo cách [đóng góp](#) vào một dự án GitHub⁹. Cách này yêu cầu người đóng góp tạo một forked repo rồi tạo pull request từ forked repo đó. Sẽ có thể phức tạp với các bạn chưa quen với GitHub.

Ngoài ra, có một cách đơn giản hơn mà bạn có thể trực tiếp dịch trên trình duyệt mà không cần cài đặt Git hay fork repo này về GitHub của bạn, như trong [hướng dẫn tại đây](#)¹⁰

Tất nhiên bạn vẫn cần tạo một GitHub account để làm việc này.

⁶ <https://forum.machinelearningcoban.com/c/d2l>

⁷ https://docs.google.com/forms/d/e/1FAIpQLScYforPRBn0oDhqSV_zTpzkxCAF0F7Cke13QS2tqXrJ8LxisQ/viewform?usp=sf_link

⁸ <https://github.com/aivivn/d2l-vn>

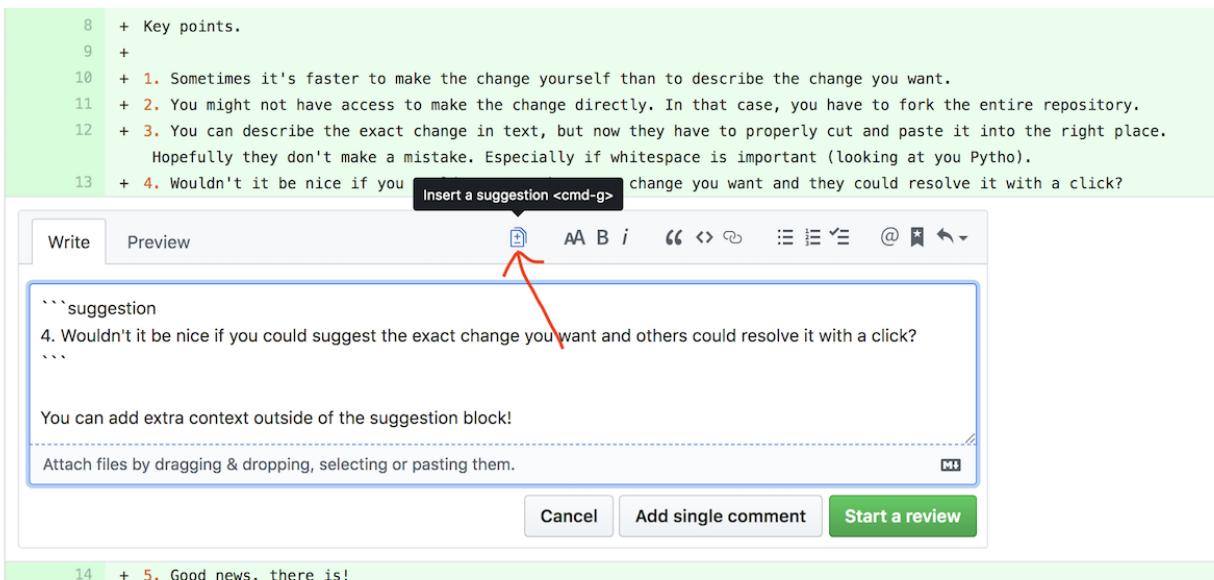
⁹ <https://codetot.net/contribute-github/>

¹⁰ <https://github.com/aivivn/d2l-vn/blob/master/CONTRIBUTING.md>

1.2.2 Review

Chọn một Pull Request trong [danh sách này¹¹](#) và bắt đầu review.

Khi Review, bạn có thể đề xuất thay đổi cách dịch mỗi dòng trực tiếp như trong hình dưới đây:



Nếu bạn có những phản hồi hữu ích, tên của bạn sẽ được tác giả chính của Pull Request đó điền vào cuối file mục “Những người thực hiện”.

1.2.3 Hỗ trợ kỹ thuật

Để phục vụ cho việc dịch trên quy mô lớn, nhóm dịch cần một số bạn hỗ trợ kỹ thuật cho một số việc dưới đây:

- Lấy các bản gốc từ [bản tiếng Anh¹²](#). Vì bản này tác giả vẫn cập nhật nên dịch đến đâu chúng ta sẽ cập nhật đến đó.
- Tự động thêm comment vào các bản gốc (<!-- và -->) để các phần này không hiển thị trên [trang web chính¹³](#). Phần thêm này có thể được thực hiện tự động bằng cách chạy:

```
python3 utils --convert <path_to_file>.md
```

và tạo ra file <path_to_file>_vn.md.

- Chia các file lớn thành các mục nhỏ như trong [ví dụ này¹⁴](#). Phần này cần thực hiện bằng tay. Mỗi phần trong file nên bao gồm những mục cụ thể, không bắt đầu và kết thúc giữa chừng của một mục.
- Dịch các chữ trong hình vẽ theo Bảng thuật ngữ. Sẵn sàng đổi các bản dịch này nếu Bảng thuật ngữ thay đổi
- Hỗ trợ quản lý project trên github, slack và diễn đàn.

¹¹ <https://github.com/aivivn/d2l-vn/pulls>

¹² <https://github.com/d2l-ai/d2l-en/tree/numpy2>

¹³ <https://d2l.aiivivn.com/>

¹⁴ https://github.com/aivivn/d2l-vn/blame/master/chapter_preface/index_vn.md

Lưu ý: Chỉ bắt đầu thực hiện công việc nếu đã có một issue tương ứng được tạo. Nếu bạn thấy một việc cần thiết, hãy tạo issue và thảo luận trước khi thực hiện. Tránh việc dẫm lên chân nhau.

2 | Lời nói đầu

Chỉ một vài năm trước, không có nhiều nhà khoa học học sâu (*deep learning*) phát triển các sản phẩm và dịch vụ thông minh tại các công ty lớn cũng như các công ty khởi nghiệp. Khi người trẻ nhất trong nhóm tác giả chúng tôi tiến vào lĩnh vực này, học máy (*machine learning*) còn chưa xuất hiện thường xuyên trên truyền thông. Cha mẹ chúng tôi còn không có ý niệm gì về học máy chứ chưa nói đến việc hiểu tại sao chúng tôi theo đuổi lĩnh vực này thay vì y khoa hay luật khoa. Học máy từng là một lĩnh vực nghiên cứu tiên phong với chỉ một số lượng nhỏ các ứng dụng thực tế. Những ứng dụng như nhận dạng giọng nói (*speech recognition*) hay thị giác máy tính (*computer vision*), đòi hỏi quá nhiều kiến thức chuyên biệt khiến chúng thường được phân thành các lĩnh vực hoàn toàn riêng mà trong đó học máy chỉ là một thành phần nhỏ. Các mạng nơ-ron (*neural network*), tiền đề của các mô hình học sâu mà chúng ta tập trung vào trong cuốn sách này, đã từng bị coi là các công cụ lỗi thời.

Chỉ trong khoảng năm năm gần đây, học sâu đã mang đến nhiều bất ngờ trên quy mô toàn cầu và dẫn đường cho những tiến triển nhanh chóng trong nhiều lĩnh vực khác nhau như thị giác máy tính, xử lý ngôn ngữ tự nhiên (*natural language processing*), nhận dạng giọng nói tự động (*automatic speech recognition*), học tăng cường (*reinforcement learning*), và mô hình hóa thống kê (*statistical modeling*). Với những tiến bộ này, chúng ta bây giờ có thể xây dựng xe tự lái với mức độ tự động ngày càng cao (nhưng chưa nhiều tới mức như vài công ty đang tuyên bố), xây dựng các hệ thống giúp trả lời thư tự động khi con người ngập trong núi email, hay lập trình phần mềm chơi cờ vây có thể thắng cả nhà vô địch thế giới, một kỷ tích từng được xem là không thể đạt được trong nhiều thập kỷ tới. Những công cụ này đã và đang gây ảnh hưởng rộng rãi tới các ngành công nghiệp và đời sống xã hội, thay đổi cách tạo ra các bộ phim, cách chẩn đoán bệnh và đóng một vài trò ngày càng tăng trong các ngành khoa học cơ bản – từ vật lý thiên văn tới sinh học.

2.1 Về cuốn sách này

Cuốn sách này được viết với mong muốn làm cho học sâu dễ tiếp cận hơn. Nó sẽ dạy bạn từ *khái niệm, bối cảnh*, cho tới cách *lập trình*.

2.1.1 Một phương tiện truyền tải kết hợp Mã nguồn, Toán, và HTML

Để một công nghệ điện toán đạt được tầm ảnh hưởng sâu rộng, nó phải dễ hiểu, có tài liệu đầy đủ, và được hỗ trợ bởi những công cụ cấp tiến được “bảo trì” thường xuyên. Các ý tưởng chính cần được chắt lọc rõ ràng, tối thiểu thời gian chuẩn bị cần thiết cho người mới bắt đầu để họ có thể trang bị các kiến thức đương thời. Các thư viện cấp tiến nên tự động hóa các tác vụ đơn giản, và các đoạn mã nguồn được lấy làm ví dụ cần phải đơn giản với những người mới bắt đầu sao cho họ có thể dễ dàng chỉnh sửa, áp dụng, và mở rộng những ứng dụng thông thường thành các ứng dụng họ cần. Lấy ứng dụng các trang web động làm ví dụ. Mặc dù các công ty công nghệ lớn như Amazon phát triển thành công các ứng dụng web định hướng bởi cơ sở dữ liệu từ những năm 1990, tiềm năng của công nghệ này để hỗ trợ các doanh nghiệp sáng tạo chỉ được nhân rộng lên ở một tầm cao mới từ khoảng mươi năm nay, nhờ vào sự phát triển của các nền tảng mạnh mẽ và với tài liệu đầy đủ.

Kiểm định tiềm năng của học sâu có những thách thức riêng biệt vì bất kỳ ứng dụng riêng lẻ nào cũng bao gồm nhiều lĩnh vực khác nhau. Ứng dụng học sâu đòi hỏi những hiểu biết đồng thời về (i) động lực để mô hình hoá một bài toán theo một hướng cụ thể; (ii) kiến thức toán học của một phương pháp mô hình hoá; (iii) những thuật toán tối ưu để khớp mô hình với dữ liệu; và (iv) phần kỹ thuật yêu cầu để huấn luyện mô hình một cách hiệu quả, xử lý những khó khăn trong tính toán và tận dụng thật tốt phần cứng hiện có. Việc đào tạo kỹ năng suy nghĩ thẩm đáo cần thiết để định hình bài toán, cung cấp kiến thức toán để giải chúng, và hướng dẫn cách dùng các công cụ phần mềm để triển khai những giải pháp đó, tất cả trong một nơi, hàm chứa nhiều thách thức lớn. Mục tiêu của chúng tôi trong cuốn sách này là trình bày một nguồn tài liệu tổng hợp giúp những học viên nhanh chóng bắt kịp.

Chúng tôi bắt đầu dự án sách này từ tháng 7/2017 khi cần trình bày giao diện MXNet Gluon (khi đó còn mới) tới người dùng. Tại thời điểm đó, không có một nguồn tài liệu nào vừa đồng thời (i) cập nhật; (ii) bao gồm đầy đủ các khía cạnh của học máy hiện đại với đầy đủ chiều sâu kỹ thuật; và (iii) xem kẽ các giải trình mà người ta mong đợi từ một cuốn sách giáo trình với mã nguồn có thể thực thi, điều thường được tìm thấy trong các bài hướng dẫn thực hành. Chúng tôi tìm thấy một lượng lớn các đoạn mã ví dụ về việc sử dụng một nền tảng học sâu (ví dụ làm thế nào để thực hiện các phép toán cơ bản với ma trận trên TensorFlow) hoặc để triển khai những kỹ thuật cụ thể (ví dụ các đoạn mã cho LeNet, AlexNet, ResNet,...) trong các bài blog hoặc là trên GitHub. Tuy nhiên, những ví dụ này thường tập trung vào khía cạnh *làm thế nào* để triển khai một hướng tiếp cận cho trước, mà bỏ qua việc thảo luận *tại sao* một thuật toán được tạo như thế. Nhiều chủ đề đã được đề cập đến trong các bài blog, ví dụ như trang [Distill¹⁵](#) hoặc các trang cá nhân, chúng thường chỉ đề cập đến một vài chủ đề được chọn về học sâu và thường thiếu mã nguồn đi kèm. Một mặt khác, trong khi nhiều sách giáo trình đã ra đời, đáng chú ý nhất là (?) (cuốn này cung cấp một bản khảo sát xuất sắc về các khái niệm phía sau học sâu), những nguồn tài liệu này lại không đi kèm với việc diễn giải dưới dạng mã nguồn để làm rõ hơn các khái niệm. Điều này khiến người đọc đôi khi mơ hồ về cách thực thi chúng. Bên cạnh đó, rất nhiều tài liệu lại được cung cấp dưới dạng các khóa học có phí.

Chúng tôi đặt mục tiêu tạo ra một tài liệu mà có thể (1) miễn phí cho mọi người; (2) cung cấp chiều sâu kỹ thuật đầy đủ, là điểm khởi đầu trên con đường trở thành một nhà khoa học học máy ứng dụng; (3) bao gồm mã nguồn thực thi được, trình bày cho người đọc *làm thế nào* giải quyết các bài toán trên thực tế; (4) tài liệu này có thể cập nhật một cách nhanh chóng bởi các tác giả cũng như cộng động ở quy mô lớn; và (5) được bổ sung bởi một [diễn đàn¹⁶](#) (và [diễn đàn tiếng Việt¹⁷](#) của nhóm dịch) để nhanh chóng thảo luận và hỏi đáp về các chi tiết kỹ thuật.

Các mục tiêu này thường không tương thích với nhau. Các công thức, định lý, và các trích dẫn được quản lý tốt nhất trên LaTex. Mã được giải thích tốt nhất bằng Python. Và trang web phù hợp với HTML và JavaScript. Hơn nữa, chúng tôi muốn nội dung của nó vừa có thể được truy cập dưới dạng mã nguồn có thể thực thi, vừa có thể tải về như một cuốn sách dưới định dạng PDF, và lại ở trên internet như một trang web. Hiện tại không có một công cụ nào là hoàn hảo cho những nhu cầu này, bởi vậy chúng tôi phải tự tạo công cụ cho riêng mình. Chúng tôi mô tả hướng tiếp cận một cách chi tiết trong `chapter_contribute`. Chúng tôi tổ chức dự án trên GitHub để chia sẻ mã nguồn và cho phép sửa đổi, Jupyter notebook để kết hợp đoạn mã, phương trình toán và nội dung chữ, sử dụng Sphinx như một bộ máy tạo nhiều tập tin đầu ra, và Discourse để tạo diễn đàn. Trong khi hệ thống này còn chưa hoàn hảo, những lựa chọn này cung cấp một giải pháp chấp nhận được trong số các giải pháp tương tự. Chúng tôi tin rằng đây có thể là cuốn sách đầu tiên được xuất bản dưới dạng kết hợp này.

¹⁵ <http://distill.pub>

¹⁶ <http://discuss.mxnet.io>

¹⁷ <https://forum.machinelearningcoban.com/c/d21>

2.1.2 Học thông qua thực hành

Có nhiều cuốn sách dạy rất chi tiết về một chuỗi các chủ đề khác nhau. Ví dụ như trong cuốn sách tuyệt vời (?) này của Bishop, mỗi chủ đề được dạy rất kỹ lưỡng tới nỗi để đến được chương hồi quy tuyến tính cũng đòi hỏi không ít công sức phải bỏ ra. Các chuyên gia yêu thích quyển sách này chính vì sự kỹ lưỡng mà nó mang lại, nhưng với những người mới bắt đầu thì đây là điểm hạn chế việc sử dụng cuốn sách này như một tài liệu nhập môn.

Trong cuốn sách này, chúng tôi sẽ dạy hầu hết các khái niệm *ở mức vừa đủ*. Hay nói cách khác, bạn sẽ chỉ học và hiểu các khái niệm cần thiết đủ để bạn hoàn tất phần thực hành. Trong khi chúng tôi sẽ dành một chút thời gian để dạy kiến thức căn bản sơ bộ như đại số tuyến tính và xác suất, chúng tôi muốn các bạn được tận hưởng cảm giác mãn nguyện của việc huấn luyện được mô hình đầu tiên trước khi bạn tâm tới các lý thuyết phân phôi xác suất.

Bên cạnh một vài notebook cơ bản cung cấp một khoá học cấp tốc về nền tảng toán học, mỗi chương tiếp theo sẽ giới thiệu một lượng hợp lý các khái niệm mới và đồng thời cung cấp các ví dụ đơn hoàn chỉnh—sử dụng các tập dữ liệu thực tế. Và đây là cả thách thức về cách tổ chức nội dung. Một vài mô hình có thể được nhóm lại một cách có logic trong một notebook riêng lẻ. Và một vài ý tưởng có thể được dạy tốt nhất bằng cách thực thi một số mô hình kế tiếp nhau. Mặt khác, có một lợi thế lớn về việc tuân thủ theo chính sách *mỗi notebook là một ví dụ hoàn chỉnh*: Điều này giúp bạn bắt đầu các dự án nghiên cứu của mình một cách dễ dàng nhất có thể bằng cách tận dụng mã nguồn của chúng tôi. Bạn chỉ cần sao chép một notebook và bắt đầu sửa đổi ở trên đó.

Chúng tôi sẽ xen kẽ mã nguồn có thể thực thi với kiến thức nền tảng khi cần thiết. Thông thường, chúng tôi sẽ tập trung vào việc tạo ra những công cụ trước khi giải thích chúng đầy đủ (và chúng tôi sẽ theo sát bằng cách giải thích phần kiến thức nền tảng sau). Ví dụ, chúng tôi có thể sử dụng *hà gradient ngẫu nhiên* trước khi giải thích đầy đủ tại sao nó lại hữu ích hoặc tại sao nó lại hoạt động. Điều này giúp cung cấp cho người thực hành những phương tiện cần thiết để giải quyết vấn đề nhanh chóng và đòi hỏi người đọc phải tin tưởng vào một số quyết định triển khai của chúng tôi.

Xuyên suốt cuốn sách, chúng ta sẽ làm việc với thư viện MXNet; đây là một thư viện với một đặc tính hiếm có, đó là vừa đủ linh hoạt để nghiên cứu và đủ nhanh để tạo ra sản phẩm. Cuốn sách này sẽ dạy về khái niệm học sâu từ đầu. Thỉnh thoảng, chúng tôi sẽ muốn đào sâu hơn vào những chi tiết về mô hình mà thông thường sẽ được che giấu khỏi người dùng bởi những lớp trừu tượng bậc cao Gluon. Điều này đặc biệt hay xuất hiện trong các hướng dẫn cơ bản, nơi chúng tôi muốn bạn hiểu về tất cả mọi thứ đang diễn ra trong một tầng hoặc bộ tối ưu nào đó. Trong những trường hợp này, chúng tôi sẽ thường trình bày hai phiên bản của một ví dụ: một phiên bản trong đó chúng tôi hiện thực mọi thứ từ đầu, chỉ dựa vào giao diện Numpy và việc tính đạo hàm tự động; và một phiên bản khác thực tế hơn, khi chúng tôi viết mã ngắn gọn sử dụng Gluon. Một khi chúng tôi đã dạy bạn cách một số thành phần hoạt động cụ thể như thế nào, chúng tôi có thể chỉ sử dụng phiên bản Gluon trong những hướng dẫn tiếp theo.

2.1.3 Nội dung và Bố cục

Cuốn sách này có thể được chia thành ba phần, với các phần được thể hiện bởi những màu khác nhau trong `fig_book.org`:

Fig. 2.1.1: Bố cục cuốn sách

- Phần đầu cuốn sách trình bày các kiến thức cơ bản và những việc cần chuẩn bị sơ bộ. `chap_introduction` giới thiệu về học sâu. Sau đó, qua Section ??, chúng tôi nhanh chóng trang bị cho bạn những kiến thức nền cần thiết để thực hành học sâu như cách lưu trữ, thao tác dữ liệu và cách áp dụng những phép tính dựa trên những khái niệm cơ bản trong đại số tuyến tính, giải tích và xác suất. Section ?? và

chap_perceptrons giới thiệu những khái niệm và kỹ thuật cơ bản của học sâu, ví dụ như hồi quy tuyến tính, mạng perceptron đa lớp và điều chuẩn.

- Năm chương tiếp theo tập trung vào những kỹ thuật học sâu hiện đại. chap_computation miêu tả những thành phần thiết yếu của các phép tính trong học sâu và tạo nền tảng để chúng tôi triển khai những mô hình phức tạp hơn. Sau đó, chúng tôi sẽ giới thiệu mạng nơ-ron tích chập (Convolutional Neural Networks/CNNs), một công cụ mạnh mẽ đang là nền tảng của hầu hết các hệ thống thị giác máy tính hiện đại. Tiếp đến, trong chap_rnn và chap_modern_rnn, chúng tôi giới thiệu mạng nơ-ron truy hồi (Recurrent Neural Networks/RNNs), một loại mô hình khai thác cấu trúc tạm thời hoặc tuần tự trong dữ liệu và thường được sử dụng để xử lý ngôn ngữ tự nhiên và dự đoán chuỗi thời gian. Trong chap_attention, chúng tôi giới thiệu một lớp mô hình mới sử dụng kỹ thuật cơ chế chú ý (attention mechanisms), một kỹ thuật gần đây đã thay thế RNNs trong xử lý ngôn ngữ tự nhiên. Những phần này sẽ giúp bạn nhanh chóng nắm được những công cụ cơ bản đứng sau hầu hết các ứng dụng hiện đại của học sâu.
- Phần ba thảo luận quy mô mở rộng, hiệu quả và ứng dụng. Đầu tiên, trong chap_optimization, chúng tôi bàn luận một số thuật toán tối ưu phổ biến được sử dụng để huấn luyện các mô hình học sâu. Chương tiếp theo, chap_performance khảo sát những yếu tố chính ảnh hưởng đến chất lượng tính toán của mã nguồn học sâu. Trong chap_cv và chap_nlp, chúng tôi minh họa lần lượt những ứng dụng chính của học sâu trong thị giác máy tính và xử lý ngôn ngữ tự nhiên.

2.1.4 Mã nguồn

Hầu hết các phần của cuốn sách đều bao gồm mã nguồn thực thi được, bởi vì chúng tôi tin rằng trải nghiệm học thông qua tương tác đóng một vai trò quan trọng trong học sâu. Hiện tại, một số kinh nghiệm nhất định chỉ có thể được hình thành thông qua phương pháp thử và sai, thay đổi mã nguồn từng chút một và quan sát kết quả. Lý tưởng nhất là sử dụng một lý thuyết toán học khác biệt nào đó có thể cho chúng ta biết chính xác cách thay đổi mã nguồn để đạt được kết quả mong muốn. Thật đáng tiếc là hiện tại những lý thuyết khác biệt đó vẫn chưa được khám phá ra. Mặc dù chúng tôi đã cố gắng hết sức, vẫn chưa có cách giải thích trọn vẹn nào cho nhiều vấn đề kỹ thuật, bởi vì phần toán học để mô tả những mô hình đó có thể là rất khó và công cuộc tìm hiểu về những chủ đề này mới chỉ tăng cao trong thời gian gần đây. Chúng tôi hy vọng rằng khi mà những lý thuyết về học sâu phát triển, những phiên bản tiếp theo của cuốn sách sẽ có thể cung cấp những cái nhìn sâu sắc hơn mà phiên bản hiện tại chưa làm được.

Hầu hết mã nguồn trong cuốn sách được dựa theo Apache MXNet. MXNet là một framework mã nguồn mở dành cho học sâu và là lựa chọn yêu thích của AWS (Amazon Web Services), và cả ở nhiều trường đại học và công ty. Tất cả mã nguồn trong cuốn sách này đã được kiểm thử trên phiên bản mới nhất của MXNet. Tuy nhiên, bởi vì học sâu phát triển rất nhanh, một vài đoạn mã *trong phiên bản sách in* có thể không hoạt động chuẩn trên những phiên bản MXNet sau này. Dù vậy, chúng tôi dự định sẽ giữ phiên bản trực tuyến luôn được cập nhật. Trong trường hợp bạn gặp phải bất cứ vấn đề nào, hãy tham khảo chap_installation để cập nhật mã nguồn và môi trường thực thi.

Để tránh việc lặp lại không cần thiết, chúng tôi đóng gói những hàm, lớp,... mà thường xuyên được chèn vào và tham khảo đến trong cuốn sách này trong gói thư viện d2l. Đối với bất kỳ đoạn mã nguồn nào như là một hàm, một lớp, hoặc các khai báo thư viện cần được đóng gói, chúng tôi sẽ đánh dấu bằng dòng # Saved in the d2l package for later use (Lưu lại trong gói thư viện d2l để sử dụng sau). Thư viện d2l khá nhẹ và chỉ phụ thuộc vào những gói thư viện và mô-đun sau:

```
# Saved in the d2l package for later use
import collections
from collections import defaultdict
from IPython import display
```

(continues on next page)

```

import math
from matplotlib import pyplot as plt
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
import os
import pandas as pd
import random
import re
import sys
import tarfile
import time
import zipfile

```

Chúng tôi có một bản tổng quan chi tiết về những hàm và lớp này trong `sec_d2l`.

2.1.5 Đối tượng đọc giả

Cuốn sách này dành cho các bạn sinh viên (đại học hoặc sau đại học), các kỹ sư và các nhà nghiên cứu – những người tìm kiếm một nền tảng vững chắc về những kỹ thuật thực tế của học sâu. Bởi vì chúng tôi giải thích mọi khái niệm từ đầu, bạn không bắt buộc phải có nền tảng về học sâu hay học máy. Việc giải thích đầy đủ các phương pháp học sâu đòi hỏi một số kiến thức về toán học và lập trình, nhưng chúng tôi sẽ chỉ giả định rằng bạn nắm được một số kiến thức cơ bản về đại số tuyến tính, giải tích, xác suất, và lập trình Python. Hơn nữa, trong phần Phụ lục, chúng tôi cung cấp thêm về hầu hết các phần toán được đề cập trong cuốn sách này. Phần lớn thời gian, chúng tôi sẽ ưu tiên dùng cách giải thích trực quan và mô tả các ý tưởng hơn là giải thích chặt chẽ bằng toán. Có rất nhiều cuốn sách tuyệt vời có thể thu hút bạn đọc quan tâm sâu hơn nữa. Chẳng hạn, cuốn “Giải tích tuyến tính” (Linear Analysis) của Bela Bollobas (?) bao gồm cả đại số tuyến tính và giải tích hàm ở mức độ rất chi tiết. Cuốn “Tất cả về Thống kê” (All of Statistics) (?) là hướng dẫn tuyệt vời để học thống kê. Và nếu bạn chưa sử dụng Python, bạn có thể muốn xem cuốn [hướng dẫn Python](#)¹⁸.

2.1.6 Diễn đàn

Gắn liền với cuốn sách, chúng tôi đã tạo ra một diễn đàn trực tuyến tại discuss.mxnet.io¹⁹ (và tại [Diễn đàn MXNet](#) d2l nhóm dịch tạo²⁰). Khi có câu hỏi về bất kỳ phần nào của cuốn sách, bạn có thể tìm thấy trang thảo luận liên quan bằng cách quét mã QR ở cuối mỗi chương để tham gia vào các cuộc thảo luận. Các tác giả của cuốn sách này và rộng hơn là cộng đồng phát triển MXNet cũng thường tham gia thảo luận trong diễn đàn.

2.2 Lời cảm ơn

Chúng tôi xin gửi lời cảm ơn chân thành tới hàng trăm người đã đóng góp cho cả hai bản thảo tiếng Anh và tiếng Trung. Mọi người đã giúp cải thiện nội dung và đưa ra những phản hồi rất có giá trị. Cụ thể, chúng tôi cảm ơn tất cả những người đóng góp cho dự thảo tiếng Anh này giúp nó tốt hơn cho tất cả mọi người. Tài khoản GitHub hoặc tên các bạn đóng góp (không theo trình tự cụ thể nào): alxnorden, avinashsingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, vishwesh5,

¹⁸ <http://learnpython.org/>

¹⁹ <https://discuss.mxnet.io/>

²⁰ <https://forum.machinelearningcoban.com/c/d2l>

YaYaB, Yuhong Chen, Evgeniy Smirnov, Igov, Simon Corston-Oliver, IgorDzreyev, Ha Nguyen, pmuens, alukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, prasanthreddy, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargueya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, Igov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, ruslo, Rafael Schlatter, liusy182, Giannis Pappas, ruslo, ati-ozgur, qbaza, dchoi77, Adam Gerson. Notably, Brent Werness (Amazon) và Rachel Hu (Amazon) đồng tác giả chương *Toán học cho Học sâu* trong Phụ lục với chúng tôi và là những người đóng góp chính cho chương đó.

Chúng tôi cảm ơn Amazon Web Services, đặc biệt là Swami Sivasubramanian, Raju Gulabani, Charlie Bell, và Andrew Jassy vì sự hỗ trợ hào phóng của họ trong việc viết cuốn sách này. Nếu không có thời gian, tài nguyên, mọi sự thảo luận cùng các đồng nghiệp, cũng như những khuyến khích liên tục, sự xuất hiện của cuốn sách này sẽ không thể thành hiện thực.

2.3 Tóm tắt

- Học sâu đã cách mạng hóa nhận dạng mẫu, đưa ra công nghệ cốt lõi hiện được sử dụng trong nhiều ứng dụng công nghệ, bao gồm thị giác máy, xử lý ngôn ngữ tự nhiên và nhận dạng giọng nói tự động.
- Để áp dụng thành công kỹ thuật học sâu, bạn phải hiểu được cách biến đổi bài toán, toán học của việc mô hình hóa, các thuật toán để khớp mô hình theo dữ liệu của bạn, và các kỹ thuật để thực hiện tất cả những điều này.
- Cuốn sách này là một nguồn tài liệu toàn diện, bao gồm các diễn giải, hình minh họa, công thức toán và mã nguồn, tất cả trong một.
- Để tìm câu trả lời cho các câu hỏi liên quan đến cuốn sách này, hãy truy cập diễn đàn của chúng tôi tại <https://discuss.mxnet.io/>. (Diễn đàn của nhóm dịch tại <https://forum.machinelearningcoban.com/c/d2l>).
- Apache MXNet là một thư viện mạnh mẽ để lập trình các mô hình học sâu và chạy chúng song song trên các GPU.
- Gluon là một thư viện cấp cao giúp việc viết mã các mô hình học sâu một cách dễ dàng bằng cách sử dụng Apache MXNet.
- Conda là trình quản lý gói Python đảm bảo tất cả các phần mềm phụ thuộc đều được đáp ứng đủ.
- Tất cả các notebook đều có thể tải xuống từ GitHub và các cấu hình conda cần thiết để chạy mã nguồn của cuốn sách này được viết trong tệp môi trường `.yml`.
- Nếu bạn có kế hoạch chạy mã này trên GPU, đừng quên cài đặt các driver cần thiết và cập nhật cấu hình của bạn.

2.4 Bài tập

1. Đăng ký tài khoản diễn đàn của cuốn sách tại [discussion.mxnet.io²¹](https://discuss.mxnet.io) (và của nhóm dịch tại <https://forum.machinelearningcoban.com>).
2. Cài đặt Python trên máy tính.
3. Làm theo hướng dẫn ở các liên kết đến diễn đàn ở cuối phần này, ở các liên kết diễn đàn đó bạn sẽ có thể nhận được giúp đỡ và thảo luận về cuốn sách cũng như tìm ra câu trả lời cho câu hỏi của bạn bằng cách thu hút các tác giả và cộng đồng lớn hơn.
4. Tạo một tài khoản trên diễn đàn và giới thiệu bản thân.

2.5 Thảo luận

- Tiếng Anh²²
- Tiếng Việt²³

2.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Vũ Hữu Tiệp
- Sẩm Thế Hải
- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thượng
- Ngô Thế Anh Khoa
- Trần Thị Hồng Hạnh
- Đoàn Võ Duy Thành

²¹ <https://discuss.mxnet.io/>

²² <https://discuss.mxnet.io/t/2311>

²³ <https://forum.machinelearningcoban.com/c/d2l>

3 | Ký hiệu

Các ký hiệu sử dụng trong cuốn sách này được tổng hợp dưới đây.

3.1 Số

- x : một số vô hướng
- \mathbf{x} : một vector
- \mathbf{X} : một ma trận
- X : một tensor
- \mathbf{I} : một ma trận đồng nhất
- $x_i, [\mathbf{x}]_i$: phần tử thứ i của vector \mathbf{x}
- $x_{ij}, [\mathbf{X}]_{ij}$: phần tử ở hàng thứ i , cột thứ j của ma trận \mathbf{X}

3.2 Lý thuyết Tập hợp

- \mathcal{X} : một tập hợp
- \mathbb{Z} : tập hợp các số nguyên
- \mathbb{R} : tập hợp các số thực
- \mathbb{R}^n : tập các vector thực trong không gian n chiều
- $\mathbb{R}^{a \times b}$: tập hợp các ma trận thực với a hàng và b cột
- $\mathcal{A} \cup \mathcal{B}$: hợp của hai tập hợp \mathcal{A} và \mathcal{B}
- $\mathcal{A} \cap \mathcal{B}$: giao của hai tập hợp \mathcal{A} và \mathcal{B}
- $\mathcal{A} \setminus \mathcal{B}$: hiệu của tập \mathcal{A} và tập \mathcal{B} (là tập hợp gồm các phần tử thuộc \mathcal{A} nhưng không thuộc \mathcal{B})

3.3 Hàm số và các Phép toán

- $f(\cdot)$: một hàm số
- $\log(\cdot)$: logarit tự nhiên
- $\exp(\cdot)$: hàm e mũ
- $\mathbf{1}_{\mathcal{X}}$: hàm đặc trưng (trả về 1 nếu đối số là một phần tử thuộc \mathcal{X} , trả về 0 trong trường hợp còn lại).
- $(\cdot)^\top$: chuyển vị của một vector hoặc một ma trận
- \mathbf{X}^{-1} : nghịch đảo của ma trận \mathbf{X}
- \odot : tích Hadamard (theo từng thành phần)
- $|\mathcal{X}|$: card (số phần tử) của tập \mathcal{X}
- $\|\cdot\|_p$: chuẩn ℓ_p
- $\|\cdot\|$: chuẩn ℓ_2
- $\langle \mathbf{x}, \mathbf{y} \rangle$: tích vô hướng của hai vector \mathbf{x} và \mathbf{y}
- \sum : tổng của một dãy
- \prod : tích của một dãy

3.4 Giải tích

- $\frac{dy}{dx}$: đạo hàm của y theo x
- $\frac{\partial y}{\partial x}$: đạo hàm riêng của y theo x
- $\nabla_{\mathbf{x}} y$: Gradient của y theo vector \mathbf{x}
- $\int_a^b f(x) dx$: tích phân của f từ a đến b theo x
- $\int f(x) dx$: nguyên hàm của f theo x

3.5 Xác suất và Lý thuyết Thông tin

- $P(\cdot)$: phân phối xác suất
- $z \sim P$: biến ngẫu nhiên z tuân theo phân phối xác suất P
- $P(X | Y)$: xác suất của X với điều kiện Y
- $p(x)$: hàm mật độ xác suất
- $E_x[f(x)]$: kỳ vọng của f theo x
- $X \perp Y$: hai biến ngẫu nhiên X và Y là độc lập
- $X \perp Y | Z$: hai biến ngẫu nhiên X và Y là độc lập có điều kiện nếu cho trước biến ngẫu nhiên Z
- $\text{Var}(X)$: phương sai của biến ngẫu nhiên X
- σ_X : độ lệch chuẩn của biến ngẫu nhiên X

- $\text{Cov}(X, Y)$: hiệp phương sai của hai biến ngẫu nhiên X và Y
- $\rho(X, Y)$: độ tương quan của hai biến ngẫu nhiên X và Y
- $H(X)$: Entropy của biến ngẫu nhiên X
- $D_{\text{KL}}(P \| Q)$: phân kỳ KL của hai phân phối P và Q

3.6 Độ phức tạp

- \mathcal{O} : Ký hiệu Big O

3.7 Thảo luận

- Tiếng Anh²⁴
- Tiếng Việt²⁵

3.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Vũ Hữu Tiệp
- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc

²⁴ <https://discuss.mxnet.io/t/4367>

²⁵ <https://forum.machinelearningcoban.com/c/d2l>

4 | Giới thiệu

Cho tới tận gần đây, gần như tất cả mọi chương trình máy tính mà chúng ta tương tác hàng ngày đều được tạo ra bởi lập trình viên phần mềm từ những định đề cơ bản. Giả sử chúng ta muốn viết một ứng dụng quản lý hệ thống thương mại điện tử. Sau khi tạm lại quanh chiếc bảng trắng để suy nghĩ về vấn đề một cách cẩn kẽ, chúng ta có thể phác thảo một giải pháp vận hành được, phần nào sẽ nhìn giống như sau: (i) người dùng tương tác với ứng dụng thông qua một giao diện chạy trên trình duyệt web hoặc ứng dụng trên điện thoại; (ii) ứng dụng tương tác với một hệ thống cơ sở dữ liệu thương mại để theo dõi trạng thái của từng người dùng và duy trì hồ sơ lịch sử các giao dịch; và (iii) (cũng là cốt lõi của ứng dụng) các logic nghiệp vụ (hay cũng có thể nói *bộ não*) mô tả cách thức xử lí cụ thể của ứng dụng trong từng tình huống có thể xảy ra.

Để xây dựng *bộ não* của ứng dụng này, ta phải xem xét tất cả mọi trường hợp mà chúng ta cho rằng sẽ gặp phải, qua đó đặt ra những quy tắc thích hợp. Ví dụ, mỗi lần người dùng nhấn để thêm một món đồ vào giỏ hàng, ta thêm một trường vào bảng giỏ hàng trong cơ sở dữ liệu, liên kết ID của người dùng với ID của món hàng được yêu cầu. Mặc dù hầu như rất ít lập trình viên có thể làm đúng hết trong lần đầu tiên, (sẽ cần vài lần chạy kiểm tra để xử lý hết được những trường hợp hiếm hoi), hầu như phần lớn ta có thể lập trình được từ những định đề cơ bản và tự tin chạy ứng dụng *trước khi được dùng bởi một khách hàng thực sự nào*. Khả năng phát triển những sản phẩm và hệ thống tự động từ những định đề cơ bản, thường là trong những điều kiện mới lạ, là một kỳ công trong suy luận và nhận thức của con người. Và khi bạn có thể tạo ra một giải pháp hoạt động được trong mọi tình huống, *bạn không nên sử dụng học máy*.

May mắn thay cho cộng đồng đang tăng trưởng của các nhà khoa học về học máy, nhiều tác vụ mà chúng ta muốn tự động hoá không dễ dàng bị khuất phục bởi sự tài tình của con người. Thủ tướng tượng bạn đang quây quần bên tấm bảng trắng với những bộ não thông minh nhất mà bạn biết, nhưng lần này bạn đang đương đầu với một trong những vấn đề dưới đây:

- Viết một chương trình dự báo thời tiết ngày mai, cho biết trước thông tin địa lý, hình ảnh vệ tinh, và một chuỗi dữ liệu thời tiết trong quá khứ.
- Viết một chương trình lấy đầu vào là một câu hỏi, được diễn đạt không theo khuôn mẫu nào, và trả lời nó một cách chính xác.
- Viết một chương trình hiển thị ra cho người dùng những sản phẩm mà họ có khả năng cao sẽ thích, nhưng lại ít có khả năng gặp được khi duyệt qua một cách tự nhiên.

Trong mỗi trường hợp trên, cho dù có là lập trình viên thượng thừa cũng không thể lập trình lên được từ con số không. Có nhiều lý do khác nhau. Đôi khi chương trình mà chúng ta cần lại đi theo một khuôn mẫu thay đổi theo thời gian và chương trình của chúng ta cần phải thích ứng với điều đó. Trong trường hợp khác, mối quan hệ (giả dụ như giữa các điểm ảnh và các hạng mục trừu tượng) có thể là quá phức tạp, yêu cầu hàng ngàn hàng triệu phép tính vượt ngoài khả năng thấu hiểu của nhận thức chúng ta (mặc dù mắt của chúng ta có thể xử lý tác vụ này một cách dễ dàng). Học máy (Machine Learning - ML) là lĩnh vực nghiên cứu những kỹ thuật tiên tiến mà có thể *học từ kinh nghiệm*. Khi thuật toán ML tích luỹ thêm nhiều kinh nghiệm, thường là dưới dạng dữ liệu quan sát hoặc tương tác với môi trường, chất lượng của nó sẽ tăng lên. Tương phản với hệ thống thương mại điện tử tất định của chúng ta, khi mà nó luôn tuân theo cùng logic nghiệp vụ đã có, mặc cho đã tích luỹ thêm bao nhiêu kinh nghiệm, tận cho tới khi lập trình viên tự *học* và quyết định rằng đã tới lúc cập nhật

phần mềm này. Trong cuốn sách này, chúng tôi sẽ dạy cho bạn về những điều cần bản nhất trong học máy, và tập trung đặc biệt vào học sâu, một tập hợp hùng mạnh những kỹ thuật đang thúc đẩy sự đổi mới ở nhiều lĩnh vực khác nhau như thị giác máy tính, xử lý ngôn ngữ tự nhiên, chăm sóc y tế và nghiên cứu cấu trúc gen.

4.1 Một ví dụ truyền cảm hứng

Các tác giả của cuốn sách cũng giống như nhiều người lao động khác, cần một tách cà phê trước khi bắt đầu công việc biên soạn của mình. Chúng tôi leo lên xe và bắt đầu lái. Sở hữu chiếc iPhone, Alex gọi “Hey Siri” để đánh thức hệ thống nhận dạng giọng nói của điện thoại. Sau đó Mu ra lệnh “chỉ đường đến quán cà phê Blue Bottle”. Chiếc điện thoại nhanh chóng hiển thị bản ghi thoại (*transcription*) của câu lệnh. Nó cũng nhận ra rằng chúng tôi đang yêu cầu chỉ dẫn đường đi và tự khởi động ứng dụng Bản đồ để hoàn thành yêu cầu đó. Khởi động xong, ứng dụng Bản đồ tự xác định một vài lộ trình tới đích. Đến mỗi tuyến đường, ứng dụng lại cập nhật và hiển thị thời gian di chuyển dự tính mới. Mặc dù đây chỉ là câu chuyện dựng lên cho mục đích giảng dạy, nó cũng cho thấy chỉ trong khoảng vài giây, những tương tác hàng ngày của chúng ta với chiếc điện thoại thông minh có thể liên quan đến nhiều mô hình học máy.

Tưởng tượng rằng ta mới viết một chương trình để phản hồi một *hiệu lệnh đánh thức* như là “Alexa”, “Okay, Google” hoặc “Siri”. Hãy thử tự viết nó chỉ với một chiếc máy tính và một trình soạn thảo mã nguồn như minh họa trong `fig_wake_word`. Bạn sẽ bắt đầu viết một chương trình như vậy như thế nào? Thủ nghĩ xem... vấn đề này khó đấy. Cứ mỗi giây, chiếc mic sẽ thu thập cỡ tầm 44,000 mẫu tín hiệu. Mỗi mẫu là một giá trị biên độ của sóng âm. Quy tắc đáng tin cậy nào có thể từ một đoạn âm thanh thô đưa ra các dự đoán {có, không} để xác định đoạn âm thanh đó có chứa hiệu lệnh đánh thức hay không? Nếu bạn không biết xử lý điều này như thế nào, thì cũng đừng lo lắng. Chúng tôi cũng không biết làm cách nào để viết một chương trình như vậy từ đầu. Đó là lý do vì sao chúng tôi sử dụng học máy.

Fig. 4.1.1: Xác định một hiệu lệnh đánh thức

Thủ thuật là thế này. Ngay cả khi không thể giải thích cụ thể cho một cái máy tính về cách ánh xạ từ đầu vào đến đầu ra như thế nào, thì chúng ta vẫn có khả năng làm việc đó bằng bộ não của mình. Hay nói cách khác, thậm chí nếu chúng ta không biết *cách lập trình một cái máy tính* để nhận dạng từ “Alexa”, chính chúng ta lại có *khả năng nhận thức* được từ “Alexa”. Với khả năng này, chúng ta có thể thu thập một *tập dữ liệu* lớn các mẫu âm thanh kèm nhãn mà *có chứa* hoặc *không chứa* hiệu lệnh đánh thức. Trong cách tiếp cận học máy, chúng ta không thiết kế một hệ thống *rõ ràng* để nhận dạng hiệu lệnh đánh thức. Thay vào đó, chúng ta định nghĩa một chương trình linh hoạt mà hành vi của nó được xác định bởi những *tham số*. Sau đó chúng ta sử dụng *tập dữ liệu* để xác định bộ các tham số tốt nhất có khả năng cải thiện chất lượng của chương trình, cũng như thỏa mãn một số yêu cầu về chất lượng trong nhiệm vụ được giao.

Bạn có thể coi những tham số như các nút vặn có thể điều chỉnh để thay đổi hành vi của chương trình. Sau khi đã cố định các tham số, chúng ta gọi chương trình này là một *mô hình*. Tập hợp của tất cả các chương trình khác nhau (ánh xạ đầu vào–đầu ra) mà chúng ta có thể tạo ra chỉ bằng cách thay đổi các tham số được gọi là một *nhóm* các mô hình. Và *chương trình học tham số* sử dụng *tập dữ liệu* để chọn ra các tham số được gọi là *thuật toán học*.

Trước khi tiếp tục và bắt đầu với các thuật toán học, chúng ta phải định nghĩa rõ ràng vấn đề, hiểu chính xác bản chất của đầu vào và đầu ra và lựa chọn một nhóm mô hình thích hợp. Trong trường hợp này, mô hình của chúng ta nhận *đầu vào* là một đoạn âm thanh và *đầu ra* là một giá trị trong {đúng, sai}. Nếu tất cả diễn ra như kế hoạch, mô hình thông thường sẽ dự đoán chính xác liệu đoạn âm thanh có hay không chứa hiệu lệnh kích hoạt.

Nếu chúng ta lựa chọn đúng nhóm mô hình, sẽ tồn tại một cách thiết lập các nút vặn mà mô hình sẽ đưa ra

đúng mỗi khi nghe thấy từ “Alexa”. Bởi vì việc lựa chọn hiệu lệnh đánh thức nào là tuỳ ý, chúng ta sẽ muốn có một nhóm mô hình đủ mạnh để trong trường hợp với một thiết lập khác của các nút quay, nó sẽ đưa ra kết quả đúng mỗi khi nghe từ “Apricot” (“quả mơ”). Bằng trực giác ta có thể nhận thấy rằng việc *nhận dạng* “Alexa” và *nhận dạng* “Apricot” cũng tương tự nhau và có thể sử dụng chung một nhóm mô hình. Tuy nhiên, trong trường hợp có sự khác biệt về bản chất ở đầu vào và đầu ra, chẳng hạn như việc ánh xạ từ hình ảnh sang chủ thích, hoặc từ câu tiếng Anh sang câu tiếng Trung thì ta có thể sẽ phải sử dụng các nhóm mô hình hoàn toàn khác nhau.

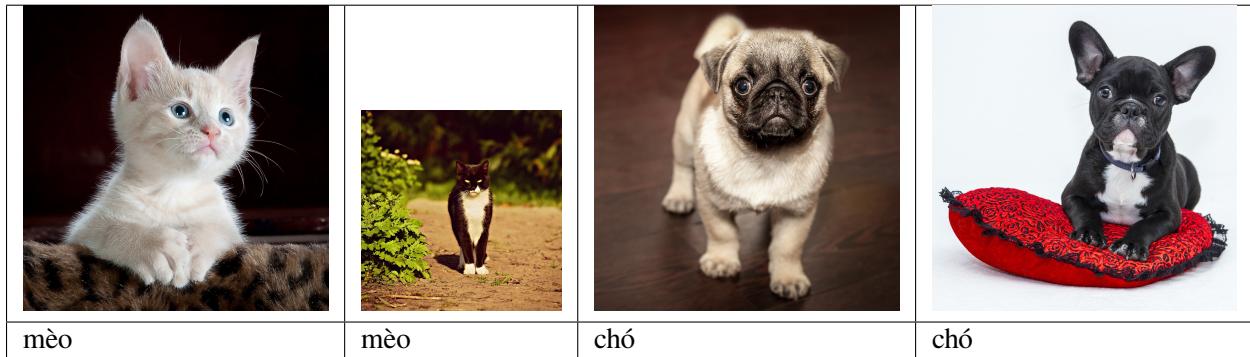
Dễ dàng nhận thấy, nếu như chúng ta chỉ thiết lập một cách ngẫu nhiên các nút vặn, thì mô hình gần như sẽ không có khả năng nhận dạng “Alexa”, “Apricot” hay bất cứ từ tiếng Anh nào khác. Trong học sâu, *học* là quá trình khám phá ra thiết lập đúng của các nút vặn để mô hình có thể hành xử như chúng ta mong muốn.

Quá trình huấn luyện thường giống như mô tả trong hình fig_ml_loop:

1. Khởi tạo mô hình một cách ngẫu nhiên. Lúc này nó vẫn chưa thể thực hiện bất kỳ tác vụ có ích nào.
2. Thu thập một số dữ liệu đã được gán nhãn (ví dụ như đoạn âm thanh kèm nhãn {có, không} tương ứng).
3. Thay đổi các nút vặn để mô hình dự đoán chính xác hơn trên các mẫu.
4. Lặp lại cho đến khi có một mô hình hoạt động tốt.

Fig. 4.1.2: Một quá trình huấn luyện điển hình

Tóm lại, thay vì tự lập trình một chương trình nhận dạng từ đánh thức, ta tạo ra một chương trình có thể *học* cách nhận dạng các từ đánh thức *khi được cho xem một tập lớn những ví dụ đã được gán nhãn*. Ta có thể gọi việc xác định hành vi của một chương trình bằng cách cho nó xem một tập dữ liệu là *lập trình với dữ liệu*. Chúng ta có thể “lập trình” một bộ phát hiện mèo bằng cách cung cấp cho hệ thống học máy nhiều mẫu ảnh chó và mèo, ví dụ như các hình ảnh dưới đây:



Bằng cách này bộ phát hiện sẽ dần học cách trả về một số dương lớn nếu đó là một con mèo, hoặc một số âm lớn nếu đó là một con chó, hoặc một giá trị gần với không nếu nó không chắc chắn. Và đấy mới chỉ là một ví dụ nhỏ về những gì mà học máy có thể làm được.

Học sâu chỉ là một trong nhiều phương pháp phổ biến để giải quyết những bài toán học máy. Tới giờ chúng ta mới chỉ nói tổng quát về học máy chứ chưa nói về học sâu. Để thấy được tại sao học sâu lại quan trọng, ta nên dừng lại một chút để làm rõ một vài điểm thiết yếu.

Thứ nhất, những vấn đề mà chúng ta đã thảo luận–học từ tín hiệu âm thanh thô, từ những giá trị điểm ảnh của tấm ảnh, hoặc dịch những câu có độ dài bất kỳ sang một ngôn ngữ khác–là những vấn đề học sâu có thể xử lý tốt còn học máy truyền thống thì không. Mô hình sâu thực sự *sâu* theo nghĩa nó có thể học nhiều *tầng* tính

toán. Những mô hình đa tầng (hoặc có thứ bậc) này có khả năng xử lý dữ liệu tri giác mức thấp theo cái cách mà những công cụ trước đây không thể. Trước đây, một phần quan trọng trong việc áp dụng học máy vào các bài toán này là tìm thủ công những kỹ thuật biến đổi dữ liệu sang một hình thức mà những mô hình *nông* có khả năng xử lý. Một lợi thế then chốt của học sâu là nó không chỉ thay thế mô hình *nông* ở thành phần cuối cùng của pipeline học tập truyền thống mà còn thay thế quá trình thiết kế đặc trưng tốn nhiều công sức. Thứ hai, bằng cách thay thế các kỹ thuật “tiền xử lý theo từng phân ngành”, học sâu đã loại bỏ ranh giới giữa thị giác máy tính, nhận dạng tiếng nói, xử lý ngôn ngữ tự nhiên, tin học y khoa và các lĩnh vực khác, cung cấp một tập hợp các công cụ xử lý những loại bài toán khác nhau.

4.2 Các thành phần chính: Dữ liệu, Mô hình và Thuật toán

Trong ví dụ về *tù đánh thức*, chúng tôi đã mô tả một bộ dữ liệu bao gồm các đoạn âm thanh và các nhãn nhị phân, giúp các bạn hiểu một cách chung chung về cách *huấn luyện* một mô hình để phân loại các đoạn âm thanh. Với loại bài toán này, ta cố gắng dự đoán một *nhãn* chưa biết với *đầu vào* cho trước, dựa trên tập dữ liệu cho trước bao gồm các mẫu đã được gán nhãn. Đây là ví dụ về bài toán *học có giám sát* và chỉ là một trong số rất nhiều *dạng* bài toán học máy khác nhau mà chúng ta sẽ học trong các chương sau. Trước hết, chúng tôi muốn giải thích rõ hơn về các thành phần cốt lõi sẽ theo chúng ta xuyên suốt tất cả các bài toán học máy:

1. *Dữ liệu* mà chúng ta có thể học.
2. Một *mô hình* về cách biến đổi dữ liệu.
3. Một hàm *mất mát* định lượng *độ lỗi* của mô hình.
4. Một *thuật toán* điều chỉnh các tham số của mô hình để giảm thiểu mất mát.

4.2.1 Dữ liệu

Có một sự thật hiển nhiên là bạn không thể làm khoa học dữ liệu mà không có dữ liệu. Chúng ta sẽ tốn rất nhiều giấy mực để cân nhắc chính xác những gì cấu thành nên dữ liệu, nhưng bây giờ chúng ta sẽ rẽ sang khía cạnh thực tế và tập trung vào các thuộc tính quan trọng cần quan tâm. Thông thường, chúng ta quan tâm đến một bộ *mẫu* (còn được gọi là *điểm dữ liệu*, *ví dụ* hoặc *trường hợp*). Để làm việc với dữ liệu một cách hữu ích, chúng ta thường cần có một cách biểu diễn chúng phù hợp dưới dạng số. Mỗi ví dụ thường bao gồm một bộ thuộc tính số gọi là *đặc trưng*. Trong các bài toán học có giám sát ở trên, một đặc trưng đặc biệt được chọn làm *mục tiêu* dự đoán, (còn được gọi là *nhãn* hoặc *biến phụ thuộc*). Các đặc trưng mà mô hình dựa vào để đưa ra dự đoán có thể được gọi đơn giản là các *đặc trưng*, (hoặc thường là *đầu vào*, *hiệp biến* hoặc *biến độc lập*).

Nếu chúng ta đang làm việc với dữ liệu hình ảnh, mỗi bức ảnh riêng lẻ có thể tạo thành một *mẫu* được biểu diễn bởi một danh sách các giá trị số theo thứ tự tương ứng với độ sáng của từng pixel. Một bức ảnh màu có kích thước 200×200 sẽ bao gồm $200 \times 200 \times 3 = 120000$ giá trị số, tương ứng với độ sáng của các kênh màu đỏ, xanh lá cây và xanh dương cho từng vị trí trong không gian. Trong một tác vụ truyền thống hơn, chúng ta có thể cố gắng dự đoán xem một bệnh nhân liệu có cơ hội sống sót hay không, dựa trên bộ đặc trưng tiêu chuẩn cho trước như tuổi, các triệu chứng quan trọng, thông số chẩn đoán, .v.v.

Khi mỗi mẫu được biểu diễn bởi cùng một số lượng các giá trị, ta nói rằng dữ liệu bao gồm các vector có *độ dài cố định* và ta mô tả độ dài (không đổi) của vector là *chiều* của dữ liệu. Bạn có thể hình dung, chiều dài cố định có thể là một thuộc tính thuận tiện. Nếu ta mong muốn huấn luyện một mô hình để nhận biết ung thư qua hình ảnh từ kính hiển vi, độ dài cố định của đầu vào sẽ giúp ta loại bỏ một vấn đề cần quan tâm.

Tuy nhiên, không phải tất cả dữ liệu có thể được dễ dàng biểu diễn dưới dạng vector có độ dài cố định. Đôi khi ta có thể mong đợi hình ảnh từ kính hiển vi đến từ thiết bị tiêu chuẩn, nhưng ta không thể mong đợi hình ảnh được khai thác từ Internet sẽ hiển thị với cùng độ phân giải hoặc tỉ lệ được. Đối với hình ảnh, ta có thể

tính đến việc cắt xén nhằm đưa chúng về kích thước tiêu chuẩn, nhưng chiến lược này chỉ đưa ta đến đấy mà thôi. Và ta có nguy cơ sẽ mất đi thông tin trong các phần bị cắt bỏ. Hơn nữa, dữ liệu văn bản không thích hợp với cách biểu diễn dưới dạng vector có độ dài cố định. Suy xét một chút về những đánh giá của khách hàng để lại trên các trang Thương mại điện tử như Amazon, IMDB hoặc TripAdvisor. Ta có thể thấy có những bình luận ngắn gọn như: “nó bốc mùi!”, một số khác thì bình luận lan man hàng trang. Một lợi thế lớn của học sâu so với các phương pháp truyền thống đó là các mô hình học sâu hiện đại có thể xử lý dữ liệu có *độ dài biến đổi* một cách uyển chuyển hơn.

Nhìn chung, chúng ta có càng nhiều dữ liệu thì công việc sẽ càng dễ dàng hơn. Khi ta có nhiều dữ liệu hơn, ta có thể huấn luyện ra những mô hình mạnh mẽ hơn và ít phụ thuộc hơn vào các giả định được hình thành từ trước. Việc chuyển từ dữ liệu nhỏ sang dữ liệu lớn là một đóng góp chính cho sự thành công của học sâu hiện đại. Để cho rõ hơn, nhiều mô hình thú vị nhất trong học sâu có thể không hoạt động nếu như không có bộ dữ liệu lớn. Một số người vẫn áp dụng học sâu với số dữ liệu ít ỏi mà mình có được, nhưng trong trường hợp này nó không tốt hơn các cách tiếp cận truyền thống.

Cuối cùng, có nhiều dữ liệu và xử lý dữ liệu một cách khéo léo thôi thì chưa đủ. Ta cần những dữ liệu *đúng*. Nếu dữ liệu mang đầy lỗi, hoặc nếu các đặc trưng được chọn lại không dự đoán được số lượng mục tiêu cần quan tâm, việc học sẽ thất bại. Tình huống trên có thể được khái quát bởi thuật ngữ: *đưa rác vào thì nhận rác ra* (*garbage in, garbage out*). Hơn nữa, chất lượng dự đoán kém không phải hậu quả tiềm tàng duy nhất. Trong các ứng dụng học máy có tính nhạy cảm như: dự đoán hành vi phạm pháp, sàng lọc hồ sơ cá nhân và mô hình rủi ro được sử dụng để cho vay, chúng ta phải đặc biệt cảnh giác với hậu quả của dữ liệu rác. Một dạng lỗi thường thấy xảy ra trong các bộ dữ liệu là khi một nhóm người không tồn tại trong dữ liệu huấn luyện. Hãy hình dung khi áp dụng một hệ thống nhận diện ung thư da trong thực tế mà trước đây nó chưa từng thấy qua da màu đen. Thất bại cũng có thể xảy ra khi dữ liệu không đại diện đầy đủ và chính xác cho một số nhóm người, nhưng lại đánh giá nhóm người này dựa vào định kiến của xã hội. Một ví dụ, nếu như các quyết định tuyển dụng trong quá khứ được sử dụng để huấn luyện một mô hình dự đoán sẽ được sử dụng nhằm sàng lọc sơ yếu lý lịch, thì các mô hình học máy có thể vô tình học được từ những bất công trong quá khứ. Lưu ý rằng tất cả vấn đề trên có thể xảy ra mà không hề có tác động xấu nào của nhà khoa học dữ liệu hoặc thậm chí họ còn không ý thức được về các vấn đề đó.

4.2.2 Mô hình

Phần lớn học máy đều liên quan đến việc *bien đổi* dữ liệu theo một cách nào đó. Có thể ta muốn xây dựng một hệ thống nhận ảnh đầu vào và dự đoán *mức độ cười* của khuôn mặt trong ảnh. Hoặc đó cũng có thể là một hệ thống nhận vào dữ liệu đo đặc từ cảm biến và dự đoán độ *bình thường* hay *bất thường* của chúng. Ở đây chúng ta gọi *mô hình* là một hệ thống tính toán nhận đầu vào là một dạng dữ liệu và sau đó trả về kết quả dự đoán, có thể ở một dạng dữ liệu khác. Cụ thể, ta quan tâm tới các mô hình thống kê mà ta có thể ước lượng được từ dữ liệu. Dù các mô hình đơn giản hoàn toàn có thể giải quyết các bài toán đơn giản phù hợp, những bài toán được đề tâm tới trong cuốn sách này sẽ đầy các phương pháp cổ điển tới giới hạn của chúng. Điểm khác biệt chính của học sâu so với các phương pháp cổ điển là các mô hình mạnh mẽ mà nó nhắm vào. Những mô hình đó bao gồm rất nhiều phép biến đổi dữ liệu liên tiếp, được liên kết với nhau từ trên xuống dưới, và đó cũng là ý nghĩa của cái tên “*học sâu*”. Trong quá trình thảo luận về các mạng nơ-ron sâu, ta cũng sẽ nhắc tới các phương pháp truyền thống.

4.2.3 Hàm mục tiêu

Trước đó, chúng tôi có giới thiệu học máy là việc “học từ kinh nghiệm”. *Học* ở đây tức là việc *tiến bộ* ở một tác vụ nào đó theo thời gian. Nhưng ai biết được như thế nào là tiến bộ? Thủ tướng tượng ta đang đề xuất cập nhật mô hình, nhưng một số người có thể có bất đồng về việc bản cập nhật này có giúp cải thiện mô hình hay không.

Để có thể phát triển một mô hình toán học chính quy cho học máy, chúng ta cần những phép đo chính quy xem mô hình đang tốt (hoặc tệ) như thế nào. Trong học máy, hay rộng hơn là lĩnh vực tối ưu hoá, ta gọi chúng là các hàm mục tiêu (*objective function*). Theo quy ước, ta thường định nghĩa các hàm tối ưu sao cho giá trị càng thấp thì mô hình càng tốt. Nhưng đó cũng chỉ là một quy ước ngầm. Bạn có thể lấy một hàm f sao cho giá trị càng cao thì càng tốt, sau đó đặt một hàm tương đương $f' = -f$, có giá trị càng thấp thì mô hình càng tốt. Chính vì ta mong muốn hàm có giá trị thấp, nó còn được gọi là *hàm mất mát* (*loss function*) và *hàm chi phí* (*cost function*).

Khi muốn dự đoán một giá trị số, hàm mục tiêu phổ biến nhất là hàm bình phương sai số $(y - \hat{y})^2$. Với bài toán phân loại, mục tiêu phổ biến nhất là tối thiểu hóa tỉ lệ lỗi, tức tỉ lệ mẫu mà dự đoán của mô hình lệch với nhãn thực tế. Một vài hàm mục tiêu (ví dụ như bình phương sai số) khá dễ tối ưu hóa. Các hàm khác (như tỉ lệ lỗi) lại khó tối ưu hóa trực tiếp, có thể do các hàm này không khả vi hoặc những vấn đề khác. Trong những trường hợp như vậy, ta thường tối ưu hóa một *hàm mục tiêu thay thế* (*surrogate objective*).

Thông thường, hàm mất mát được định nghĩa theo các tham số mô hình và phụ thuộc vào tập dữ liệu. Những giá trị tham số mô hình tốt nhất được học bằng cách tối thiểu hóa hàm mất mát trên một *tập huấn luyện* bao gồm các *mẫu* được thu thập cho việc huấn luyện. Tuy nhiên, mô hình hoạt động tốt trên tập huấn luyện không có nghĩa là nó sẽ hoạt động tốt trên dữ liệu kiểm tra (mà mô hình chưa nhìn thấy). Bởi vậy, ta thường chia dữ liệu sẵn có thành hai phần: dữ liệu huấn luyện (để khớp các tham số mô hình) và dữ liệu kiểm tra (được giữ lại cho việc đánh giá). Sau đó ta quan sát hai đại lượng:

- **Lỗi huấn luyện:** Lỗi trên dữ liệu được dùng để huấn luyện mô hình. Bạn có thể coi nó như điểm của một sinh viên trên bài thi thử để chuẩn bị cho bài thi thật. Ngay cả khi kết quả thi thử khả quan, không thể đảm bảo rằng bài thi thật sẽ đạt kết quả tốt.
- **Lỗi kiểm tra:** Đây là lỗi trên tập kiểm tra (không dùng để huấn luyện mô hình). Đại lượng này có thể chênh lệch đáng kể so với lỗi huấn luyện. Khi một mô hình hoạt động tốt trên tập huấn luyện nhưng lại không có khả năng tổng quát hóa trên dữ liệu chưa gặp, ta nói rằng mô hình bị *quá khớp* (*overfit*). Theo ngôn ngữ thường ngày, đây là hiện tượng “học lệch tủ” khi kết quả bài thi thật rất kém mặc dù có kết quả cao trong bài thi thử.

4.2.4 Các thuật toán tối ưu

Một khi ta có dữ liệu, một mô hình và một hàm mục tiêu rõ ràng, ta cần một thuật toán có khả năng tìm kiếm các tham số khả dĩ tốt nhất để tối thiểu hóa hàm mất mát. Các thuật toán tối ưu phổ biến nhất cho mạng nơ-ron đều theo một hướng tiếp cận gọi là hạ gradient. Một cách ngắn gọn, tại mỗi bước và với mỗi tham số, ta kiểm tra xem hàm mất mát thay đổi như thế nào nếu ta thay đổi tham số đó bởi một lượng nhỏ. Sau đó các tham số này được cập nhật theo hướng làm giảm hàm mất mát.

4.3 Các dạng Học Máy

Trong các mục tiếp theo, chúng ta sẽ thảo luận chi tiết hơn một số *dạng* bài toán học máy. Cùng bắt đầu với một danh sách *các mục tiêu*, tức một danh sách các tác vụ chúng ta muốn học máy thực hiện. Chú ý rằng các mục tiêu sẽ được gắn liền với một tập các kỹ thuật để đạt được mục tiêu đó, bao gồm các kiểu dữ liệu, mô hình, kỹ thuật huấn luyện, v.v. Danh sách dưới đây là một tuyển tập các bài toán mà học máy có thể xử lý nhằm tạo động lực cho độc giả, đồng thời cung cấp một ngôn ngữ chung khi nói về những bài toán khác xuyên suốt cuốn sách.

4.3.1 Học có giám sát

Học có giám sát giải quyết tác vụ dự đoán *mục tiêu* với *đầu vào* cho trước. Các mục tiêu, thường được gọi là *nhãn*, phần lớn được ký hiệu bằng y . Dữ liệu đầu vào, thường được gọi là *đặc trưng* hoặc *hiệp biến*, thông thường được ký hiệu là \mathbf{x} . Mỗi cặp (đầu vào, mục tiêu) được gọi là một *mẫu*. Thi thoảng, khi văn cảnh rõ ràng hơn, chúng ta có thể sử dụng thuật ngữ *các mẫu* để chỉ một tập các đầu vào, ngay cả khi chưa xác định được mục tiêu tương ứng. Ta ký hiệu một mẫu cụ thể với một chỉ số dưới, thường là i , ví dụ (\mathbf{x}_i, y_i) . Một tập dữ liệu là một tập của n mẫu $\{\mathbf{x}_i, y_i\}_{i=1}^n$. Mục đích của chúng ta là xây dựng một mô hình f_θ ánh xạ đầu vào bất kỳ \mathbf{x}_i tới một dự đoán $f_\theta(\mathbf{x}_i)$.

Một ví dụ cụ thể hơn, trong lĩnh vực chăm sóc sức khoẻ, chúng ta có thể muốn dự đoán liệu một bệnh nhân có bị đau tim hay không. Việc *bị đau tim* hay *không bị đau tim* sẽ là nhãn y . Dữ liệu đầu vào \mathbf{x} có thể là các dấu hiệu quan trọng như nhịp tim, huyết áp tâm trương và tâm thu, v.v.

Sự giám sát xuất hiện ở đây bởi để chọn các tham số θ , chúng ta (các giám sát viên) cung cấp cho mô hình một tập dữ liệu chứa các *mẫu được gán nhãn* (\mathbf{x}_i, y_i) , ở đó mỗi mẫu \mathbf{x}_i tương ứng một nhãn cho trước.

Theo thuật ngữ xác suất, ta thường quan tâm tới việc đánh giá xác suất có điều kiện $P(y|\mathbf{x})$. Dù chỉ là một trong số nhiều mô hình trong học máy, học có giám sát là nhân tố chính đem đến sự thành công cho các ứng dụng của học máy trong công nghiệp. Một phần vì rất nhiều tác vụ có thể được mô tả dưới dạng ước lượng xác suất của một đại lượng chưa biết cho trước trong một tập dữ liệu cụ thể:

- Dự đoán có bị ung thư hay không cho trước một bức ảnh CT.
- Dự đoán bản dịch chính xác trong tiếng Pháp cho trước một câu trong tiếng Anh.
- Dự đoán giá của cổ phiếu trong tháng tới dựa trên dữ liệu báo cáo tài chính của tháng này.

Ngay cả với mô tả đơn giản là “*dự đoán mục tiêu từ đầu vào*”, học có giám sát đã có nhiều hình thái đa dạng và đòi hỏi đưa ra nhiều quyết định mô hình hoá khác nhau, tùy thuộc vào kiểu, kích thước, số lượng của cặp đầu vào và đầu ra cũng như các yếu tố khác. Ví dụ, ta sử dụng các mô hình khác nhau để xử lý các chuỗi (như chuỗi ký tự hay dữ liệu chuỗi thời gian) và các biểu diễn vector với chiều dài cố định. Chúng ta sẽ đào sâu vào rất nhiều bài toán dạng này thông qua phần đầu của cuốn sách.

Một cách dễ hiểu, quá trình học gồm những bước sau: Lấy một tập mẫu lớn với các hiệp biến đã biết trước. Từ đó chọn ra một tập con ngẫu nhiên và thu thập các nhãn gốc cho chúng. Đôi khi những nhãn này có thể đã

có sẵn trong dữ liệu (ví dụ, liệu bệnh nhân đã qua đời trong năm tiếp theo?). Trong trường hợp khác, chúng ta cần thuê người gán nhãn cho dữ liệu (ví dụ, gán một bức ảnh vào một hạng mục nào đó).

Những đầu vào và nhãn tương ứng này cùng tạo nên tập huấn luyện. Chúng ta đưa tập dữ liệu huấn luyện vào một thuật toán học có giám sát – một hàm số với đầu vào là tập dữ liệu và đầu ra là một hàm số khác thể hiện *mô hình đã học được*. Cuối cùng, ta có thể đưa dữ liệu chưa nhìn thấy vào mô hình đã học được, sử dụng đầu ra của nó như là giá trị dự đoán của các nhãn tương ứng. Toàn bộ quá trình được mô tả trong `fig_supervised_learning`.

Fig. 4.3.1: Học có giám sát.

Hồi quy

Có lẽ tác vụ học có giám sát đơn giản nhất là *hồi quy*. Xét ví dụ một tập dữ liệu thu thập được từ cơ sở dữ liệu buôn bán nhà. Chúng ta có thể xây dựng một bảng dữ liệu, ở đó mỗi hàng tương ứng với một nhà và mỗi cột tương ứng với một thuộc tính liên quan nào đó, chẳng hạn như diện tích nhà, số lượng phòng ngủ, số lượng phòng tắm và thời gian (theo phút) để đi bộ tới trung tâm thành phố. Trong tập dữ liệu này, mỗi *mẫu* là một căn nhà cụ thể và *vector đặc trưng* tương ứng là một hàng trong bảng.

Nếu bạn sống ở New York hoặc San Francisco và bạn không phải là CEO của Amazon, Google, Microsoft hay Facebook, thì vector đặc trưng (diện tích, số phòng ngủ, số phòng tắm, khoảng cách đi bộ) của căn nhà của bạn có thể có dạng [100, 0, 0.5, 60]. Tuy nhiên, nếu bạn sống ở Pittsburgh, vector đó có thể là [3000, 4, 3, 10]. Các vector đặc trưng như vậy là thiết yếu trong hầu hết các thuật toán học máy cổ điển. Chúng ta sẽ tiếp tục ký hiệu vector đặc trưng tương ứng với bất kỳ mẫu i nào bởi \mathbf{x}_i và có thể đặt X là bảng chứa tất cả các vector đặc trưng.

Để xác định một bài toán là *hồi quy* hay không, ta dựa vào đầu ra của nó. Chẳng hạn, bạn đang khảo sát thị trường cho một căn nhà mới. Bạn có thể ước lượng giá thị trường của một căn nhà khi biết những đặc trưng phía trên. Giá trị mục tiêu, hay giá bán của căn nhà, là một *số thực*. Nếu bạn còn nhớ định nghĩa toán học của số thực, bạn có thể đang cảm thấy băn khoăn. Nhà đất có lẽ không bao giờ bán với giá được tính bằng các phần nhỏ hơn cent, chứ đừng nói đến việc giá bán được biểu diễn bằng các số vô tỉ. Trong những trường hợp này, khi mục tiêu thực sự là các số rời rạc, nhưng việc làm tròn có thể chấp nhận được, chúng ta sẽ lạm dụng cách dùng từ một chút để tiếp tục mô tả đầu ra và mục tiêu là các số thực.

Ta ký hiệu mục tiêu bất kỳ là y_i (tương ứng với mẫu \mathbf{x}_i) và tập tất cả các mục tiêu là \mathbf{y} (tương ứng với tất cả các mẫu X). Khi các mục tiêu mang các giá trị bất kỳ trong một khoảng nào đó, chúng ta gọi đây là bài toán hồi quy. Mục đích của chúng ta là tạo ra một mô hình mà các dự đoán của nó xấp xỉ với các giá trị mục tiêu thực sự. Chúng ta ký hiệu dự đoán mục tiêu của một mẫu là \hat{y}_i . Đừng quá lo lắng nếu các ký hiệu đang làm bạn nản chí. Chúng ta sẽ tìm hiểu kỹ từng ký hiệu trong các chương tiếp theo.

Rất nhiều bài toán thực tế là các bài toán hồi quy. Dự đoán điểm đánh giá của một người dùng cho một bộ phim có thể được coi là một bài toán hồi quy và nếu bạn thiết kế một thuật toán vĩ đại để đạt được điều này vào năm 2009, bạn có thể đã giành giải thưởng Netflix một triệu Đô-la²⁶. Dự đoán thời gian nằm viện của một bệnh nhân cũng là một bài toán hồi quy. Một quy tắc dễ nhớ là khi ta phải trả lời câu hỏi *bao nhiêu* (*baō lâú*, *baō xa*, v.v.), khá chắc chắn đó là bài toán hồi quy.

- “Ca phẫu thuật này sẽ mất bao lâu?”: *hồi quy*
- “Có bao nhiêu chú chó trong bức ảnh?”: *hồi quy*

Tuy nhiên, nếu bạn có thể diễn đạt bài toán của bạn bằng câu hỏi “Đây có phải là _?” thì khả năng cao đó là bài toán phân loại, một dạng khác của bài toán học có giám sát mà chúng ta sẽ thảo luận trong phần tiếp theo.

²⁶ https://en.wikipedia.org/wiki/Netflix_Prize

Ngay cả khi bạn chưa từng làm việc với học máy, bạn có thể đã làm việc với các bài toán hồi quy một cách không chính thức. Ví dụ, hãy tưởng tượng bạn cần sửa chữa đường ống cống và người thợ đã dành $x_1 = 3$ giờ để thông cống rồi gửi hoá đơn $y_1 = \$350$. Bây giờ bạn của bạn thuê cũng thuê người thợ đó trong $x_2 = 2$ tiếng và cô ấy nhận được hoá đơn là $y_2 = \$250$. Nếu một người khác sau đó hỏi bạn giá dự tính phải trả cho việc thông cống, bạn có thể có một vài giả sử có lý, chẳng hạn như mất nhiều thời gian sẽ tốn nhiều tiền hơn. Bạn cũng có thể giả sử rằng có một mức phí cơ bản và sau đó người thợ tính tiền theo giờ. Nếu giả sử này là đúng, thì với hai điểm dữ liệu trên, bạn đã có thể tính được cách mà người thợ tính tiền công: \$100 cho mỗi giờ cộng với \$50 để có mặt tại nhà bạn. Nếu bạn theo được logic tới đây thì bạn đã hiểu ý tưởng tổng quan của hồi quy tuyến tính (và bạn đã vô tình thiết kế một mô hình tuyến tính với thành phần điều chỉnh).

Trong trường hợp này, chúng ta có thể tìm được các tham số sao cho mô hình ước tính chính xác được chi phí người thợ sửa ống cống đưa ra. Đôi khi việc này là không khả thi, ví dụ một biến thể nào đó gây ra bởi các yếu tố ngoài hai đặc trưng kể trên. Trong những trường hợp này, ta sẽ cố học các mô hình sao cho khoảng cách giữa các giá trị dự đoán và các giá trị thực sự được tối thiểu hoá. Trong hầu hết các chương, chúng ta sẽ tập trung vào một trong hai hàm mất mát phổ biến nhất: hàm mất mát L1²⁷, ở đó

$$l(y, y') = \sum_i |y_i - y'_i| \quad (4.3.1)$$

và hàm thứ hai là mất mát trung bình bình phương nhỏ nhất, hoặc mất mát L2²⁸, ở đó

$$l(y, y') = \sum_i (y_i - y'_i)^2. \quad (4.3.2)$$

Như chúng ta sẽ thấy về sau, mất mát L_2 tương ứng với giả sử rằng dữ liệu của chúng ta có nhiễu Gauss, trong khi mất mát L_1 tương ứng với giả sử nhiễu đến từ một phân phối Laplace.

Phân loại

Trong khi các mô hình hồi quy hiệu quả trong việc trả lời các câu hỏi *có bao nhiêu?*, rất nhiều bài toán không phù hợp với nhóm câu hỏi này. Ví dụ, một ngân hàng muốn thêm chức năng quét ngân phiếu trong ứng dụng di động của họ. Tác vụ này bao gồm việc khách hàng chụp một tấm ngân phiếu với camera của điện thoại và mô hình học máy cần tự động hiểu nội dung chữ trong bức ảnh. Hiểu được cả chữ viết tay sẽ giúp ứng dụng hoạt động càng mạnh mẽ hơn. Kiểu hệ thống này được gọi là nhận dạng ký tự quang học (*optical character recognition – OCR*), và kiểu bài toán mà nó giải quyết được gọi là *phân loại* (*classification*). Nó được giải quyết với một tập các thuật toán khác với thuật toán dùng trong hồi quy (mặc dù có nhiều kỹ thuật chung).

Trong bài toán phân loại, ta muốn mô hình nhìn vào một vector đặc trưng, ví dụ như các giá trị điểm ảnh trong một bức ảnh, và sau đó dự đoán mẫu đó rơi vào hạng mục (được gọi là *lớp*) nào trong một tập các lựa chọn (rời rạc). Với chữ số viết tay, ta có thể có 10 lớp tương ứng với các chữ số từ 0 tới 9. Dạng đơn giản nhất của phân loại là khi chỉ có hai lớp, khi đó ta gọi bài toán này là phân loại nhị phân. Ví dụ, tập dữ liệu X có thể chứa các bức ảnh động vật và các nhãn Y có thể là các lớp {chó, mèo}. Trong khi với bài toán hồi quy, ta cần tìm một bộ hồi quy để đưa ra một giá trị thực \hat{y} , thì với bài toán phân loại, ta tìm một bộ phân loại để dự đoán lớp \hat{y} .

Khi cuốn sách đi sâu hơn vào các vấn đề kỹ thuật, chúng ta sẽ bàn về các lý do tại sao lại khó hơn để tối ưu hoá một mô hình mà đầu ra là các giá trị hạng mục rời rạc, ví dụ, *mèo* hoặc *chó*. Trong những trường hợp này, thường sẽ dễ hơn khi thay vào đó, ta biểu diễn mô hình dưới ngôn ngữ xác suất. Cho trước một mẫu x , mô hình cần gán một giá trị xác suất \hat{y}_k cho mỗi nhãn k . Vì là các giá trị xác suất, chúng phải là các số dương có tổng bằng 1. Bởi vậy, ta chỉ cần $K - 1$ số để gán xác suất cho K hạng mục. Việc này dễ nhận thấy đối với

²⁷ <http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L1Loss>

²⁸ <http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L2Loss>

phân loại nhị phân. Nếu một đồng xu không đều có xác suất ra mặt ngửa là 0.6 (60%), thì xác suất ra mặt sấp là 0.4 (40%). Trở lại với ví dụ phân loại động vật, một bộ phân loại có thể nhìn một bức ảnh và đưa ra xác suất để bức ảnh đó là mèo $P(y = \text{mèo} | x) = 0.9$. Chúng ta có thể diễn giải giá trị này tương ứng với việc bộ phân loại 90% tin rằng bức ảnh đó chứa một con mèo. Giá trị xác suất của lớp được dự đoán truyền đạt một ý niệm về sự không chắc chắn. Tuy nhiên, đó không phải là ý niệm duy nhất về sự không chắc chắn, chúng ta sẽ thảo luận thêm về những loại khác trong các chương nâng cao.

Khi có nhiều hơn hai lớp, ta gọi bài toán này là *phân loại đa lớp*. Bài toán phân loại chữ viết tay [0, 1, 2, 3 ... 9, a, b, c, ...] là một trong số các ví dụ điển hình. Trong khi các hàm mất mát thường được sử dụng trong các bài toán hồi quy là hàm mất mát L1 hoặc L2, hàm mất mát phổ biến cho bài toán phân loại được gọi là entropy chéo (*cross-entropy*), hàm tương ứng trong MXNet Gluon có thể tìm được [tại đây²⁹](#)

Lưu ý rằng lớp có khả năng xảy ra nhất theo dự đoán của mô hình không nhất thiết là lớp mà ta quyết định sử dụng. Giả sử bạn tìm được một cây nấm rất đẹp trong sân nhà như hình `fig_death_cap`.



Fig. 4.3.2: Nấm độc—đừng ăn!

Bây giờ giả sử ta đã xây dựng một bộ phân loại và huấn luyện nó để dự đoán liệu một cây nấm có độc hay không dựa trên ảnh chụp. Giả sử bộ phân loại phát hiện chất độc đưa ra $P(y = \text{nấm độc} | \text{bức ảnh}) = 0.2$. Nói cách khác, bộ phân loại này chắc chắn rằng 80% cây này *không phải* nấm độc. Dù vậy, đừng dại mà ăn nhé. Vì việc có bữa tối ngon lành không đáng gì so với rủi ro 20% sẽ chết vì nấm độc. Nói cách khác, hậu quả của *rủi ro không chắc chắn* nghiêm trọng hơn nhiều so với lợi ích thu được. Ta có thể nhìn việc này theo khía cạnh lý thuyết. Về cơ bản, ta cần tính toán rủi ro kỳ vọng mà mình sẽ gánh chịu, ví dụ, ta nhân xác suất xảy ra kết quả đó với lợi ích (hoặc hậu quả) đi liền tương ứng:

Do đó, mất mát L do ăn phải nấm là $L(a = \text{ăn} | x) = 0.2 * \infty + 0.8 * 0 = \infty$, mặc dù phí tổn do bỏ nấm đi là $L(a = \text{bỏ đi} | x) = 0.2 * 0 + 0.8 * 1 = 0.8$.

Sự thận trọng của chúng ta là chính đáng: như bất kỳ nhà nghiên cứu nấm nào cũng sẽ nói, cây nấm ở trên thực sự *là* nấm độc. Việc phân loại có thể còn phức tạp hơn nhiều so với phân loại nhị phân, đa lớp, hoặc thậm chí đa nhãn. Ví dụ, có vài biến thể của phân loại để xử lý vấn đề phân cấp bậc (*hierarchy*). Việc phân cấp giả định rằng tồn tại các mối quan hệ giữa các lớp với nhau. Vậy nên không phải tất cả các lối đều như nhau—nếu bắt buộc có lối, ta sẽ mong rằng các mẫu bị phân loại nhầm thành một lớp họ hàng thay vì một lớp khác xa nào đó. Thông thường, việc này được gọi là *phân loại cấp bậc* (*hierarchical classification*). Một trong những ví dụ đầu tiên về việc xây dựng hệ thống phân cấp là từ [Linnaeus³⁰](#), người đã sắp xếp các loại động vật theo một hệ thống phân cấp.

²⁹ <https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.SoftmaxCrossEntropyLoss>

³⁰ https://en.wikipedia.org/wiki/Carl_Linnaeus

Trong trường hợp phân loại động vật, cũng không tệ lầm nếu phân loại nhầm hai giống chó xù poodle và schnauzer với nhau, nhưng sẽ rất nghiêm trọng nếu ta nhầm lẫn chó poodle với một con khủng long. Hệ phân cấp nào là phù hợp phụ thuộc vào việc ta dự định dùng mô hình như thế nào. Ví dụ, rắn đuôi chuông và rắn sọc không độc có thể混淆 nhầm nhau trong cây phả hệ, nhưng phân loại nhầm hai loài này có thể dẫn tới hậu quả chết người.

Gán thẻ

Một vài bài toán phân loại không phù hợp với các mô hình phân loại nhị phân hoặc đa lớp. Ví dụ, chúng ta có thể huấn luyện một bộ phân loại nhị phân thông thường để phân loại mèo và chó. Với khả năng hiện tại của thị giác máy tính, việc này có thể được thực hiện dễ dàng bằng các công cụ sẵn có. Tuy nhiên, bất kể mô hình của bạn chính xác đến đâu, nó có thể gặp rắc rối khi thấy bức ảnh Những Nhạc Sĩ thành Bremen.



Fig. 4.3.3: Mèo, gà trống, chó và lừa

Bạn có thể thấy trong ảnh có một con mèo, một con gà trống, một con chó, một con lừa và một con chim, cùng với một vài cái cây ở hậu cảnh. Tuỳ vào mục đích cuối cùng của mô hình, sẽ không hợp lý nếu coi đây là một bài toán phân loại nhị phân. Thay vào đó, có thể chúng ta muốn cung cấp cho mô hình tuỳ chọn để mô tả bức ảnh gồm một con mèo và một con chó và một con lừa và một con gà trống và một con chim.

Bài toán học để dự đoán các lớp *không xung khắc* được gọi là phân loại đa nhãn. Các bài toán tự động gán thẻ là các ví dụ điển hình của phân loại đa nhãn. Nghĩ về các thẻ mà một người có thể gán cho một blog công nghệ, ví dụ “học máy”, “công nghệ”, “ngôn ngữ lập trình”, “linux”, “diện toán đám mây” hay “AWS”. Một bài báo thông thường có thể có từ 5-10 thẻ bởi các khái niệm này có liên quan với nhau. Các bài về “diện toán đám mây” khả năng cao đề cập “AWS” và các bài về “học máy” cũng có thể dính dáng tới “ngôn ngữ lập trình”.

Ta cũng phải xử lý các vấn đề này trong nghiên cứu y sinh, ở đó việc gán thẻ cho các bài báo một cách chính xác là quan trọng bởi nó cho phép các nhà nghiên cứu tổng hợp đầy đủ các tài liệu liên quan. Tại Thư viện Y khoa Quốc gia, một số chuyên gia gán nhãn duyệt qua tất cả các bài báo được lưu trên PubMed để gán chúng với các thuật ngữ y khoa (*MeSH*) liên quan – một bộ sưu tập với khoảng 28 nghìn thẻ. Đây là một quá trình tốn thời gian và những người gán nhãn thường bị trễ một năm kể từ khi lưu trữ cho tới lúc gán thẻ. Học máy có thể được sử dụng ở đây để cung cấp các thẻ tạm thời cho tới khi được kiểm chứng lại một cách thủ công. Thực vậy, BioASQ đã tổ chức một cuộc thi³¹ dành riêng cho việc này.

Tìm kiếm và xếp hạng

Đôi khi ta không chỉ muốn gán một lớp hoặc một giá trị vào một mẫu. Trong lĩnh vực thu thập thông tin (*information retrieval*), ta muốn gán thứ hạng cho một tập các mẫu. Lấy ví dụ trong tìm kiếm trang web, mục tiêu không chỉ dừng lại ở việc xác định liệu một trang web có liên quan tới từ khoá tìm kiếm, mà xa hơn, trang web nào trong số vô vàn kết quả trả về *liên quan nhất* tới người dùng. Chúng ta rất quan tâm đến thứ tự của các kết quả tìm kiếm và thuật toán cần đưa ra các tập con có thứ tự từ những thành phần trong một tập lớn hơn. Nói cách khác, nếu được hỏi đưa ra năm chữ cái từ bảng chữ cái, hai kết quả A B C D E và C A B E D là khác nhau. Ngay cả khi các phần tử trong hai tập kết quả là như nhau, thứ tự các phần tử trong mỗi tập mới là điều quan trọng.

Một giải pháp khả dĩ cho bài toán này là trước tiên gán cho mỗi phần tử trong tập hợp một số điểm về sự phù hợp và sau đó trả về những phần tử có điểm cao nhất. *PageRank*³², vũ khí bí mật đằng sau cỗ máy tìm kiếm của Google, là một trong những ví dụ đầu tiên của hệ thống tính điểm kiểu này. Tuy nhiên, điều bất thường là nó không phụ thuộc vào từ khoá tìm kiếm. Chúng phụ thuộc vào một bộ lọc đơn giản để xác định tập hợp các trang phù hợp rồi sau đó mới dùng PageRank để sắp xếp các kết quả có chứa cụm tìm kiếm. Có cả những hội thảo khoa học chuyên nghiên cứu về lĩnh vực này.

Hệ thống gợi ý

Hệ thống gợi ý là một bài toán khác liên quan đến tìm kiếm và xếp hạng. Tuy có chung mục đích hiển thị một tập các kết quả liên quan tới người dùng, hệ thống gợi ý nhấn mạnh việc *cá nhân hóa* cho từng người dùng cụ thể. Ví dụ khi gợi ý phim ảnh, kết quả gợi ý cho một fan của phim khoa học viễn tưởng và cho một người sành sỏi hài Peter Sellers có thể khác nhau một cách đáng kể. Các bài toán gợi ý khác có thể bao gồm hệ thống gợi ý sản phẩm bán lẻ, âm nhạc hoặc tin tức.

Trong một vài trường hợp, khách hàng cung cấp phản hồi trực tiếp (*explicit feedback*) thể hiện mức độ yêu thích một sản phẩm cụ thể (ví dụ các đánh giá sản phẩm và phản hồi trên Amazon, IMDB, Goodreads, v.v.). Trong những trường hợp khác, họ cung cấp phản hồi gián tiếp (*implicit feedback*), ví dụ như khi bỏ qua bài hát trong danh sách chơi nhạc. Những bài hát đó có thể đã không làm hài lòng người nghe, hoặc chỉ đơn thuần là không phù hợp với bối cảnh. Diễn giải một cách đơn giản, những hệ thống này được huấn luyện để ước lượng một điểm y_{ij} nào đó, ví dụ như ước lượng điểm đánh giá hoặc ước lượng xác suất mua hàng, của một người dùng u_i tới một sản phẩm p_j .

Với một mô hình như vậy, cho một người dùng bất kỳ, ta có thể thu thập một tập các sản phẩm với điểm y_{ij} lớn nhất để gợi ý cho khách hàng. Các hệ thống đang vận hành trong thương mại còn cao cấp hơn nữa. Chúng sử dụng hành vi của người dùng và các thuộc tính sản phẩm để tính điểm. *fig_deeplearning_amazon* là một ví dụ về các cuốn sách học sâu được gợi ý bởi Amazon dựa trên các thuật toán cá nhân hóa được điều chỉnh phù hợp với sở thích của tác giả cuốn sách này.

³¹ <http://bioasq.org/>

³² <https://en.wikipedia.org/wiki/PageRank>

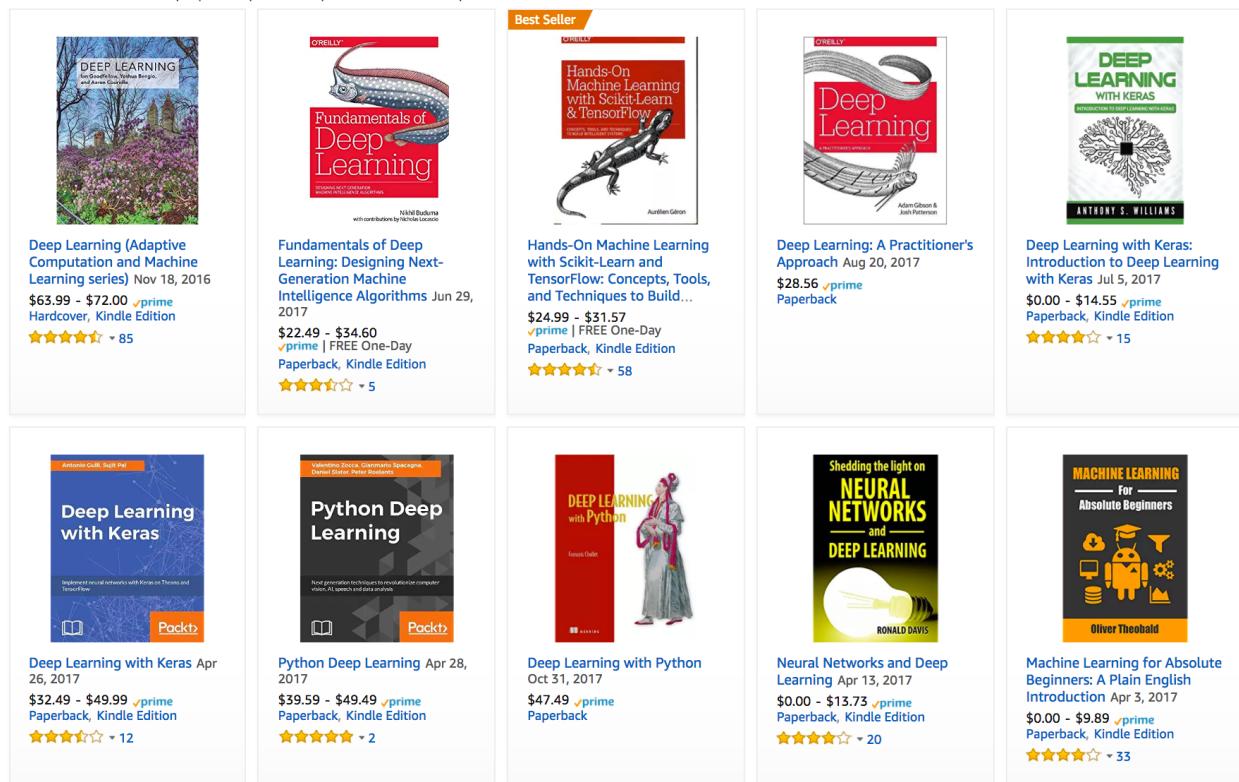


Fig. 4.3.4: Các sách học sâu được gợi ý bởi Amazon.

Mặc dù có giá trị kinh tế lớn, các hệ thống gợi ý được xây dựng đơn thuần theo các mô hình dự đoán về bản chất có những hạn chế nghiêm trọng. Ban đầu, ta chỉ quan sát các *phản hồi được kiểm duyệt*. Người dùng thường có xu hướng đánh giá các bộ phim họ thực sự thích hoặc ghét: bạn có thể để ý rằng các bộ phim nhận được rất nhiều đánh giá 5 và 1 sao nhưng rất ít các bộ phim với 3 sao. Hơn nữa, thói quen mua hàng hiện tại thường là kết quả của thuật toán gợi ý đang được dùng, nhưng các thuật toán gợi ý không luôn để ý đến chi tiết này. Vì vậy, các vòng phản hồi (*feedback loop*) luẩn quẩn có thể xảy ra khi mà hệ thống gợi ý đẩy lên một sản phẩm và cho rằng sản phẩm này tốt hơn (do được mua nhiều hơn), từ đó sản phẩm này lại được hệ thống gợi ý thường xuyên hơn nữa. Rất nhiều trong số các vấn đề về cách xử lý với kiểm duyệt, động cơ của việc đánh giá và vòng phản hồi là các câu hỏi quan trọng cho nghiên cứu.

Học chuỗi

Cho tới giờ, chúng ta đã gặp các bài toán mà ở đó mô hình nhận đầu vào với kích thước cố định và đưa ra kết quả cũng với kích thước cố định. Trước đây chúng ta xem xét dự đoán giá nhà từ một tập các đặc trưng cố định: diện tích, số phòng ngủ, số phòng tắm và thời gian đi bộ tới trung tâm thành phố. Ta cũng đã thảo luận cách ánh xạ từ một bức ảnh (với kích thước cố định) tới các dự đoán xác suất nó thuộc vào một tập cố định các lớp, hoặc lấy một mã người dùng và mã sản phẩm để dự đoán số sao đánh giá. Trong những trường hợp này, một khi chúng ta đưa cho mô hình một đầu vào có độ dài cố định để dự đoán một đầu ra, mô hình ngay lập tức “quên” dữ liệu nó vừa thấy.

Việc này không ảnh hưởng nhiều nếu mọi đầu vào đều có cùng kích thước và nếu các đầu vào liên tiếp thật sự không liên quan đến nhau. Nhưng ta sẽ xử lý các đoạn video như thế nào khi mỗi đoạn có thể có số lượng khung hình khác nhau? Và dự đoán của chúng ta về việc gì đang xảy ra ở mỗi khung hình có thể sẽ chính xác hơn nếu ta xem xét cả các khung hình kè nố. Hiện tượng tương tự xảy ra trong ngôn ngữ. Một bài toán học sâu phổ biến là dịch máy (*machine translation*): tác vụ lấy đầu vào là các câu trong một ngôn ngữ nguồn và trả

về bản dịch của chúng ở một ngôn ngữ khác.

Các bài toán này cũng xảy ra trong y khoa. Với một mô hình theo dõi bệnh nhân trong phòng hồi sức tích cực, ta có thể muốn nó đưa ra cảnh báo nếu nguy cơ tử vong của họ trong 24 giờ tới vượt một ngưỡng nào đó. Dĩ nhiên, ta chắc chắn không muốn mô hình này vứt bỏ mọi số liệu trong quá khứ và chỉ đưa ra dự đoán dựa trên các thông số mới nhất.

Những bài toán này nằm trong những ứng dụng thú vị nhất của học máy và chúng là các ví dụ của *học chuỗi*. Chúng đòi hỏi một mô hình có khả năng nhận chuỗi các đầu vào hoặc dự đoán chuỗi các đầu ra. Thậm chí có mô hình phải thỏa mãn cả hai tiêu chí đó, và những bài toán có cấu trúc như vậy còn được gọi là seq2seq (*sequence to sequence: chuỗi sang chuỗi*). Dịch ngôn ngữ là một bài toán seq2seq. Chuyển một bài nói về dạng văn bản cũng là một bài toán seq2seq. Mặc dù không thể xét hết mọi dạng của biến đổi chuỗi, có một vài trường hợp đặc biệt đáng được lưu tâm:

Gán thẻ và Phân tích cú pháp. Đây là bài toán chú thích cho một chuỗi văn bản. Nói cách khác, số lượng đầu vào và đầu ra là như nhau. Ví dụ, ta có thể muốn biết vị trí của động từ và chủ ngữ. Hoặc ta cũng có thể muốn biết từ nào là danh từ riêng. Mục tiêu tổng quát là phân tích và chú thích văn bản dựa trên các giả định về cấu trúc và ngữ pháp. Việc này nghe có vẻ phức tạp hơn thực tế. Dưới đây là một ví dụ rất đơn giản về việc chú thích một câu bằng các thẻ đánh dấu danh từ riêng.

Tom has dinner in Washington with Sally.
Ent - - - Ent - Ent

Tự động nhận dạng giọng nói. Với nhận dạng giọng nói, chuỗi đầu vào x là một bản thu âm giọng nói của một người (fig_speech) và đầu ra y là một văn bản ghi lại những gì người đó nói. Thủ thách ở đây là việc số lượng các khung âm thanh (âm thanh thường được lấy mẫu ở 8kHz or 16kHz) nhiều hơn hẳn so với số lượng từ, nghĩa là không tồn tại một phép ánh xạ 1:1 nào giữa các khung âm thanh và các từ, bởi một từ có thể tương ứng với hàng ngàn mẫu âm thanh. Có các bài toán seq2seq mà đầu ra ngắn hơn rất nhiều so với đầu vào.

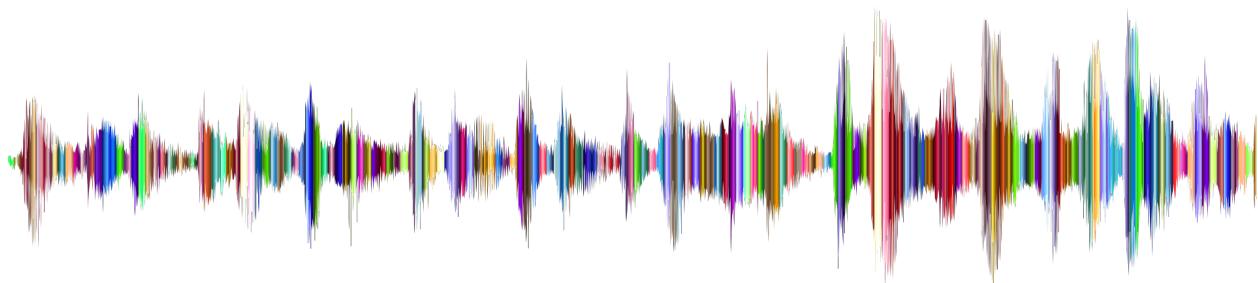


Fig. 4.3.5: -D-e-e-p- L-ea-r-ni-ng-

Chuyển văn bản thành giọng nói (*Text-to-Speech* hay TTS) là bài toán ngược của bài toán nhận dạng giọng nói. Nói cách khác, đầu vào x là văn bản và đầu ra y là tệp tin âm thanh. Trong trường hợp này, đầu ra dài hơn rất nhiều so với đầu vào. Việc nhận dạng các tệp tin âm thanh chất lượng kém không khó với *con người* nhưng lại không hề đơn giản với máy tính.

Dịch máy. Khác với nhận dạng giọng nói, khi các đầu vào và đầu ra tương ứng có cùng thứ tự (sau khi căn chỉnh), trong dịch máy việc đảo thứ tự lại có thể rất quan trọng. Nói cách khác, ta vẫn chuyển đổi từ chuỗi này sang chuỗi khác, nhưng ta không thể giả định số lượng đầu vào và đầu ra cũng như thứ tự của các điểm dữ liệu tương ứng là giống nhau. Xét ví dụ minh họa dưới đây về trật tự từ kỳ lạ khi dịch tiếng Anh sang tiếng Việt.

Tiếng Anh: Did you already check out this excellent tutorial?

Tiếng Việt: Bạn đã xem qua hướng dẫn tuyệt vời này chưa?

Căn chỉnh sai: Đã chưa bạn xem qua này tuyệt vời hướng dẫn?

Rất nhiều bài toán liên quan xuất hiện trong các tác vụ học khác. Chẳng hạn, xác định thứ tự người dùng đọc một trang mạng là một bài toán phân tích bối cảnh hai chiều. Các bài toán hội thoại thì chứa đủ các loại vấn đề phức tạp khác, như việc xác định câu nói tiếp theo đòi hỏi kiến thức thực tế cũng như trạng thái trước đó rất lâu của cuộc hội thoại. Đây là một lĩnh vực nghiên cứu đang phát triển.

4.3.2 Học không giám sát

Tất cả các ví dụ từ trước đến nay đều liên quan tới lĩnh vực *Học có giám sát*, ví dụ, những trường hợp mà ta nạp vào mô hình lượng dữ liệu khổng lồ gồm cả các đặc trưng và giá trị mục tiêu tương ứng. Bạn có thể tưởng tượng người học có giám sát giống như anh nhân viên đang làm một công việc có tính chuyên môn cao cùng với một ông sếp cực kỳ hâm tài. Ông sếp đứng ngay bên cạnh bạn và chỉ bảo chi tiết những điều phải làm trong từng tình huống một cho tới khi bạn học được cách liên kết các tình huống đó với hành động tương ứng. Làm việc với ông sếp kiểu này có vẻ khá là chán. Nhưng ở một khía cạnh khác, thì làm ông sếp này hài lòng rất là dễ dàng. Bạn chỉ việc nhận ra những khuôn mẫu thật nhanh và bắt chước lại những hành động theo khuôn mẫu đó.

Trong một hoàn cảnh hoàn toàn trái ngược, nếu gặp phải người sếp không biết họ muốn bạn làm cái gì thì sẽ rất là khó chịu. Tuy nhiên, nếu định trở thành nhà khoa học dữ liệu, tốt nhất hãy làm quen với điều này. Ông ta có thể chỉ đưa cho bạn một đống dữ liệu to sụ và bảo *sử dụng khoa học dữ liệu với cái này đi!* Nghe có vẻ khá mơ hồ bởi vì đúng là nó mơ hồ thật. Chúng ta gọi những loại vấn đề như thế này là *học không giám sát* (*unsupervised learning*); với chúng, loại câu hỏi và lượng câu hỏi ta có thể đặt ra chỉ bị giới hạn bởi trí tưởng tượng của chính mình. Ta sẽ đề cập tới một số kỹ thuật học không giám sát ở các chương sau. Bây giờ, để gợi cho bạn đọc chút hứng khởi, chúng tôi sẽ diễn giải một vài câu hỏi bạn có thể sẽ hỏi:

- Liệu có thể dùng một lượng nhỏ nguyên mẫu để tóm lược dữ liệu một cách chính xác? Giả sử với một bộ ảnh, liệu có thể nhóm chúng thành các ảnh phong cảnh, chó, trẻ con, mèo, đỉnh núi, v.v.? Tương tự, với bộ dữ liệu duyệt web của người dùng, liệu có thể chia họ thành các nhóm người dùng có hành vi giống nhau? Vấn đề như này thường được gọi là *phân cụm* (*clustering*).
- Liệu ta có tìm được một lượng nhỏ các tham số mà vẫn tóm lược được chính xác những thuộc tính cốt lõi của dữ liệu? Như là quỹ đạo bay của quả bóng được miêu tả tương đối tốt bởi số liệu vận tốc, đường kính và khối lượng của quả bóng. Như một người thợ may chỉ cần một lượng nhỏ các số đo để miêu tả hình dáng cơ thể người tương đối chuẩn xác để may ra quần áo vừa vặn. Những ví dụ trên được gọi là bài toán *ước lượng không gian con* (*subspace estimation*). Nếu mối quan hệ phụ thuộc là tuyến tính, bài toán này được gọi là phép *phân tích phân chính* (*principal component analysis – PCA*).
- Liệu có tồn tại cách biểu diễn các đối tượng (có cấu trúc bất kỳ) trong không gian Euclid (ví dụ: không gian vector \mathbb{R}^n) mà những thuộc tính đặc trưng có thể được ghép khớp với nhau? Phương pháp này được gọi là *học biểu diễn* (*representation learning*) và được dùng để miêu tả các thực thể và mối quan hệ giữa chúng, giống như Rome – Ý + Pháp = Paris.
- Liệu có tồn tại cách miêu tả những nguyên nhân gốc rễ của lượng dữ liệu mà ta đang quan sát được? Ví dụ, nếu chúng ta có dữ liệu nhân khẩu học về giá nhà, mức độ ô nhiễm, tệ nạn, vị trí, trình độ học vấn, mức lương, v.v.. thì liệu ta có thể khám phá ra cách chúng liên hệ với nhau chỉ đơn thuần dựa vào dữ liệu thực nghiệm? Những lĩnh vực liên quan tới *nhân quả và mô hình đồ thị xác suất* (*probabilistic graphical models*) sẽ giải quyết bài toán này.

Một bước phát triển quan trọng và thú vị gần đây của học không giám sát là sự ra đời của *mạng đối sinh* (*generative adversarial network – GAN*). GAN cho ta một quy trình sinh dữ liệu, kể cả những dữ liệu cấu trúc

phức tạp như hình ảnh và âm thanh. Cùng các cơ chế toán thống kê ẩn bên dưới sẽ kiểm tra xem liệu những dữ liệu thật và giả này có giống nhau không. Chúng tôi sẽ viết vài mục về chủ đề này sau.

4.3.3 Tương tác với Môi trường

Cho tới giờ, chúng ta chưa thảo luận về việc dữ liệu tới từ đâu hoặc chuyện gì thực sự sẽ xảy ra khi một mô hình học máy trả về kết quả dự đoán. Điều này là do học có giám sát và học không giám sát chưa giải quyết các vấn đề một cách thấu đáo. Trong cả hai cách học, chúng ta yêu cầu mô hình học từ một lượng dữ liệu lớn đã được cung cấp từ đầu mà không cho nó tương tác trở lại với môi trường trong suốt quá trình học. Bởi vì toàn bộ việc học diễn ra khi thuật toán đã được ngắt kết nối khỏi môi trường, đôi khi ta gọi đó là *học ngoại tuyến* (*offline learning*). Quá trình này cho học có giám sát được mô tả trong `fig_data_collection`.

Fig. 4.3.6: Thu thập dữ liệu từ môi trường cho học có giám sát

Sự đơn giản của học ngoại tuyến có nét đẹp của nó. Ưu điểm là ta chỉ cần quan tâm đến vấn đề nhận dạng mẫu mà không bị phân tâm bởi những vấn đề khác. Nhưng nhược điểm là sự hạn chế trong việc thiết lập các bài toán. Nếu bạn đã đọc loạt truyện ngắn Robots của Asimov hoặc là người có tham vọng, bạn có thể đang tưởng tượng ra trí tuệ nhân tạo không những biết đưa ra dự đoán mà còn có thể tương tác với thế giới. Chúng ta muốn nghĩ tới những *tác nhân* (*agent*) thông minh chứ không chỉ những *mô hình* dự đoán. Tức là ta phải nhắm tới việc chọn *hành động* chứ không chỉ đưa ra những *dự đoán*. Hơn thế nữa, không giống dự đoán, hành động còn tác động đến môi trường. Nếu muốn huấn luyện một tác nhân thông minh, chúng ta phải tính đến cách những hành động của nó có thể tác động đến những gì nó nhận lại trong tương lai.

Xem xét việc tương tác với môi trường mở ra một loạt những câu hỏi về mô hình hoá mới. Liệu môi trường có:

- Nhớ những gì ta đã làm trước đó?
- Muốn giúp đỡ chúng ta, chẳng hạn: một người dùng đọc văn bản vào một bộ nhận dạng giọng nói?
- Muốn đánh bại chúng ta, chẳng hạn: một thiết lập đối kháng giống như bộ lọc thư rác (chống lại những kẻ viết thư rác) hay là chơi game (với đối thủ)?
- Không quan tâm (có rất nhiều trường hợp thế này)?
- Có xu hướng thay đổi (dữ liệu trong tương lai có giống với trong quá khứ không, hay là khuôn mẫu có thay đổi theo thời gian một cách tự nhiên hoặc do phản ứng với những công cụ tự động)?

Câu hỏi cuối cùng nêu lên vấn đề về *dịch chuyển phân phối* (*distribution shift*), khi dữ liệu huấn luyện và dữ liệu kiểm tra khác nhau. Vấn đề này giống như khi chúng ta phải làm bài kiểm tra được cho bởi giảng viên trong khi bài tập về nhà lại do trợ giảng chuẩn bị. Chúng ta sẽ thảo luận sơ qua về học tăng cường (*reinforcement learning*) và học đối kháng (*adversarial learning*), đây là hai thiết lập đặc biệt có xét tới tương tác với môi trường.

4.3.4 Học tăng cường

Nếu bạn muốn dùng học máy để phát triển một tác nhân tương tác với môi trường và đưa ra hành động, khả năng cao là bạn sẽ cần tập trung vào *học tăng cường* (*reinforcement learning* – RL). Học tăng cường có các ứng dụng trong ngành công nghệ robot, hệ thống đối thoại và cả việc phát triển AI cho trò chơi điện tử. *Học sâu tăng cường* (*Deep reinforcement learning* – DRL) áp dụng kỹ thuật học sâu để giải quyết những vấn đề của học tăng cường và đã trở nên phổ biến trong thời gian gần đây. Hai ví dụ tiêu biểu nhất là thành tựu đột phá của *mạng-Q* sâu đánh bại con người trong các trò chơi điện tử Atari chỉ sử dụng đầu vào hình ảnh³³, và chương trình AlphaGo chiếm ngôi vô địch thế giới trong môn Cờ Vây³⁴.

Học tăng cường mô tả bài toán theo cách rất tổng quát, trong đó tác nhân tương tác với môi trường qua một chuỗi các *bước thời gian* (*timesteps*). Tại mỗi bước thời gian t , tác nhân sẽ nhận được một quan sát o_t từ môi trường và phải chọn một hành động a_t để tương tác với môi trường thông qua một cơ chế nào đó (đôi khi còn được gọi là bộ dẫn động). Sau cùng, tác nhân sẽ nhận được một điểm thưởng r_t từ môi trường. Sau đó, tác nhân lại nhận một quan sát khác và chọn ra một hành động, cứ tiếp tục như thế. Hành vi của tác nhân học tăng cường được kiểm soát bởi một *chính sách* (*policy*). Nói ngắn gọn, một *chính sách* là một hàm ánh xạ từ những quan sát (từ môi trường) tới các hành động. Mục tiêu của học tăng cường là tạo ra một chính sách tốt.

Fig. 4.3.7: Sự tương tác giữa học tăng cường và môi trường.

Tính tổng quát của mô hình học tăng cường không phải là một thứ được nói quá lên. Ví dụ, ta có thể chuyển bất cứ bài toán học có giám sát nào thành một bài toán học tăng cường. Chẳng hạn với một bài toán phân loại, ta có thể tạo một tác nhân học tăng cường với một *hành động* tương ứng với mỗi lớp. Sau đó ta có thể tạo một môi trường mà trả về điểm thưởng đúng bằng với giá trị của hàm mất mát từ bài toán học có giám sát ban đầu.

Vì thế, học tăng cường có thể giải quyết nhiều vấn đề mà học có giám sát không thể. Lấy ví dụ ở trong học có giám sát, chúng ta luôn đòi hỏi dữ liệu huấn luyện phải đi kèm với đúng nhãn. Tuy nhiên với học tăng cường, ta không giả định rằng môi trường sẽ chỉ ra hành động nào là tối ưu tại mỗi quan sát (điểm dữ liệu). Nhìn chung, mô hình sẽ chỉ nhận được một điểm thưởng nào đó. Hơn thế nữa, môi trường có thể sẽ không chỉ ra những hành động nào đã dẫn tới điểm thưởng đó.

Lấy cờ vua làm ví dụ. Phần thưởng thật sự sẽ đến vào cuối trò chơi. Khi thắng, ta sẽ được 1 điểm, hoặc khi thua, ta sẽ nhận về -1 điểm. Vì vậy, việc học tăng cường phải giải quyết *bài toán phân bổ công trạng* (*credit assignment problem*): xác định hành động nào sẽ được thưởng hay bị phạt dựa theo kết quả. Tương tự như khi một nhân viên được thăng chức vào ngày 11/10. Việc thăng chức khả năng cao phản ánh những việc làm tốt của nhân viên này trong suốt một năm qua. Để được thăng chức sau này đòi hỏi nhân viên đó phải nhận ra đâu là những hành động dẫn đến việc thăng chức này.

Học tăng cường còn phải đương đầu với vấn đề về những quan sát không hoàn chỉnh. Có nghĩa là quan sát hiện tại có thể không cho bạn biết mọi thứ về tình trạng lúc này. Lấy ví dụ, khi robot hút bụi bị kẹt tại một trong nhiều phòng giống y như nhau trong căn nhà. Việc can thiệp vào vị trí chính xác (cũng là trạng thái) của robot có thể cần đến những quan sát từ trước khi nó đi vào phòng.

Cuối cùng, tại một thời điểm bất kỳ, các thuật toán học tăng cường có thể biết một chính sách tốt, tuy nhiên có thể có những chính sách khác tốt hơn mà tác nhân chưa bao giờ thử tới. Các thuật toán học tăng cường phải luôn lựa chọn giữa việc tiếp tục *khai thác* chính sách tốt nhất hiện thời hay *khám phá* thêm những giải pháp khác, tức bỏ qua những điểm thưởng ngắn hạn để thu về thêm kiến thức.

³³ <https://www.wired.com/2015/02/google-ai-plays-atari-like-pros/>

³⁴ <https://www.wired.com/2017/05/googles-alphago-trounces-humans-also-gives-boost/>

MDPs, những kẻ trộm, và những người bạn

Các bài toán học tăng cường thường có một thiết lập rất tổng quát. Các hành động của tác nhân có ảnh hưởng đến những quan sát về sau. Những điểm thưởng nhận được tương ứng chỉ với các hành động được chọn. Môi trường có thể được quan sát đầy đủ hoặc chỉ một phần. Tính toán tất cả sự phức tạp này cùng lúc có thể cần sự tham gia của quá nhiều nhà nghiên cứu. Hơn nữa, không phải mọi vấn đề thực tế đều thể hiện tất cả sự phức tạp này. Vì vậy, các nhà nghiên cứu đã nghiên cứu một số *trường hợp đặc biệt* về những bài toán học tăng cường.

Khi ở môi trường được quan sát đầy đủ, ta gọi bài toán học tăng cường này là *Quá trình Quyết định Markov* (*Markov Decision Process* – MDP). Khi trạng thái không phụ thuộc vào các hành động trước đó, ta gọi bài toán này là *bài toán trộm ngữ cảnh* (*contextual bandit problem*). Khi không có trạng thái, chỉ có một tập hợp các hành động có sẵn với điểm thưởng chưa biết ban đầu, bài toán kinh điển này là *bài toán trộm đa nhánh* (*multi-armed bandit problem*).

4.4 Nguồn gốc

Mặc dù có nhiều phương pháp học sâu được phát minh gần đây, nhưng mong muốn phân tích dữ liệu để dự đoán kết quả tương lai của con người đã tồn tại trong nhiều thế kỷ. Và trong thực tế, phần lớn khoa học tự nhiên đều có nguồn gốc từ điều này. Ví dụ, phân phối Bernoulli được đặt theo tên của Jacob Bernoulli (1655-1705)³⁵, và bản phân phối Gausian được phát hiện bởi Carl Friedrich Gauss (1777-1855)³⁶. Ông đã phát minh ra thuật toán trung bình bình phương nhỏ nhất, thuật toán này vẫn được sử dụng cho rất nhiều vấn đề từ tính toán bảo hiểm cho đến chẩn đoán y khoa. Những công cụ này đã mở đường cho một cách tiếp cận dựa trên thử nghiệm trong các ngành khoa học tự nhiên – ví dụ, định luật Ohm mô tả rằng dòng điện và điện áp trong một điện trở được diễn tả hoàn hảo bằng một mô hình tuyến tính.

Ngay cả trong thời kỳ trung cổ, các nhà toán học đã có một trực giác nhạy bén trong các ước tính của mình. Chẳng hạn, cuốn sách hình học của Jacob Köbel (1460-1533)³⁷ minh họa việc lấy trung bình chiều dài 16 bàn chân của những người đàn ông trưởng thành để có được chiều dài bàn chân trung bình.

³⁵ https://en.wikipedia.org/wiki/Jacob_Bernoulli

³⁶ https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss

³⁷ <https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry>



Fig. 4.4.1: Ước tính chiều dài của một bàn chân

`fig_koebel` minh họa cách các công cụ ước tính này hoạt động. 16 người đàn ông trưởng thành được yêu cầu xếp hàng nối tiếp nhau khi rời nhà thờ. Tổng chiều dài của các bàn chân sau đó được chia cho 16 để có được ước tính trị số hiện tại tầm 1 feet. “Thuật toán” này đã được cải tiến ngay sau đó nhằm giải quyết trường hợp bàn chân bị biến dạng – 2 người đàn ông có bàn chân ngắn nhất và dài nhất tương ứng được loại ra, trung bình chỉ được tính trong phần còn lại. Đây là một trong những ví dụ đầu tiên về ước tính trung bình cắt ngọn.

Thống kê thật sự khởi sắc với việc thu thập và có sẵn dữ liệu. Một trong số những người phi thường đã đóng góp lớn vào lý thuyết và ứng dụng của nó trong di truyền học, đó là [Ronald Fisher \(1890-1962\)](#)³⁸. Nhiều thuật toán và công thức của ông (như Phân tích biệt thức tuyến tính - Linear Discriminant Analysis hay Ma trận thông tin Fisher - Fisher Information Matrix) vẫn được sử dụng thường xuyên cho đến ngày nay (ngay cả bộ dữ liệu Iris mà ông công bố năm 1936 đôi khi vẫn được sử dụng để minh họa cho các thuật toán học máy). Fisher cũng là một người ủng hộ thuyết ưu sinh. Điều này nhắc chúng ta rằng việc áp dụng khoa học dữ liệu vào những ứng dụng mờ ám trên phương diện đạo đức cũng như các ứng dụng có ích trong công nghiệp và khoa học tự nhiên đều đã có lịch sử tồn tại và phát triển lâu đời.

Ảnh hưởng thứ hai đối với học máy đến từ Lý Thuyết Thông Tin ([Claude Shannon, 1916-2001](#))³⁹ và Lý Thuyết Điện Toán của [Alan Turing \(1912-1954\)](#)⁴⁰. Turing đã đặt ra câu hỏi “Liệu máy móc có thể suy nghĩ?” trong bài báo nổi tiếng của mình [Máy Tính và Trí Thông Minh](#)⁴¹ (Mind, Tháng 10 1950). Tại đó ông ấy đã giới thiệu về phép thử Turing: một máy tính được xem là thông minh nếu như người đánh giá khó có thể phân biệt phản hồi mà anh ta nhận được thông qua tương tác văn bản xuất phát từ máy tính hay con người.

Một ảnh hưởng khác có thể được tìm thấy trong khoa học thần kinh và tâm lý học. Trên hết, loài người thể hiện rất rõ ràng hành vi thông minh của mình. Bởi vậy câu hỏi tự nhiên là liệu một người có thể giải thích và

³⁸ https://en.wikipedia.org/wiki/Ronald_Fisher

³⁹ https://en.wikipedia.org/wiki/Claude_Shannon

⁴⁰ https://en.wikipedia.org/wiki/Alan_Turing

⁴¹ https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence

lần ngược để tận dụng năng lực này. Một trong những thuật toán lâu đời nhất lấy cảm hứng từ câu hỏi trên được giới thiệu bởi Donald Hebb (1904-1985)⁴². Trong cuốn sách đột phá của mình về Tổ Chức Hành Vi (?), ông ấy cho rằng các nơ-ron học bằng cách dựa trên những phản hồi tích cực. Điều này về sau được biết đến với cái tên là luật học Hebbian. Nó là nguyên mẫu cho thuật toán perceptron của Rosenblatt và là nền tảng của rất nhiều thuật toán hạ gradient ngẫu nhiên đã đặt nền móng cho học sâu ngày nay: tăng cường các hành vi mong muốn và giảm bớt các hành vi không mong muốn để đạt được những thông số tốt trong một mạng nơ-ron.

Niềm cảm hứng từ sinh học đã tạo nên cái tên *mạng nơ-ron*. Trong suốt hơn một thế kỷ (từ mô hình của Alexander Bain, 1873 và James Sherrington, 1890), các nhà nghiên cứu đã cố gắng lắp ráp các mạch tính toán tương tự như các mạng tương tác giữa các nơ-ron. Theo thời gian, yếu tố sinh học ngày càng giảm đi nhưng tên gọi của nó thì vẫn ở lại. Những nguyên tắc thứ yếu của mạng nơ-ron vẫn có thể tìm thấy ở hầu hết các mạng ngày nay:

- Sự đan xen giữa các đơn vị xử lý tuyến tính và phi tuyến tính, thường được đề cập tới như là *các tầng*.
- Việc sử dụng quy tắc chuỗi (còn được biết đến là *làn truyền ngược – backpropagation*) để điều chỉnh các tham số trong toàn bộ mạng cùng lúc.

Sau những tiến bộ nhanh chóng ban đầu, các nghiên cứu về mạng nơ-ron giảm dần trong khoảng từ 1995 tới 2005 bởi một vài lí do. Thứ nhất là huấn luyện mạng rất tốt kém tài nguyên tính toán. Mặc dù dung lượng RAM đã dồi dào vào cuối thế kỷ trước, sức mạnh tính toán vẫn còn hạn chế. Thứ hai là tập dữ liệu vẫn còn tương đối nhỏ. Lúc đó tập dữ liệu Fisher's Iris từ năm 1932 vẫn là công cụ phổ biến để kiểm tra tính hiệu quả của các thuật toán. MNIST với 60.000 ký tự viết tay đã được xem là rất lớn.

Với sự khan hiếm của dữ liệu và tài nguyên tính toán, các công cụ thống kê mạnh như Phương Pháp Hạt Nhân, Cây Quyết Định và Mô Hình Đồ Thị đã chứng tỏ sự vượt trội trong thực nghiệm. Khác với mạng nơ-ron, chúng không đòi hỏi nhiều tuần huấn luyện nhưng vẫn đưa ra những kết quả dự đoán với sự đảm bảo vững chắc về lý thuyết.

4.5 Con đường tới Học Sâu

Con đường tới học sâu đã có rất nhiều thay đổi cùng với sự sẵn có của lượng lớn dữ liệu nhờ vào Mạng Lưới Toàn Cầu (World Wide Web)– sự phát triển của các công ty với hàng triệu người dùng trực tuyến, sự phổ biến của các cảm biến giá rẻ với chất lượng cao, bộ lưu trữ dữ liệu giá rẻ (theo quy luật Kryder), và tính toán chi phí thấp (theo định luật Moore)– đặc biệt là các GPU, với thiết kế ban đầu được dành cho việc xử lý các trò chơi máy tính. Và rồi những thuật toán và mô hình tưởng chừng không khả thi về mặt tính toán đã không còn ngoài tầm với. Điều này được minh họa trong tab_intro_decade.

Thập kỷ	Tập dữ liệu	Bộ nhớ	Số lượng phép tính dấu phẩy động trên giây
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (Giá nhà ở Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (Nhận dạng ký tự quang học)	10 MB	10 MF (Intel 80486)
2000	10 M (các trang web)	100 MB	1 GF (Intel Core)
2010	10 G (quảng cáo)	1 GB	1 TF (Nvidia C2050)
2020	1 T (mạng xã hội)	100 GB	1 PF (Nvidia DGX-2)

Table: Liên hệ giữa tập dữ liệu với bộ nhớ máy tính và năng lực tính toán

⁴² https://en.wikipedia.org/wiki/Donald_O._Hebb

Sự thật là RAM đã không theo kịp với tốc độ phát triển của dữ liệu. Đồng thời, sự tiến bộ trong năng lực tính toán đã vượt lên tốc độ phát triển của dữ liệu có sẵn. Vì vậy, mô hình thống kê cần phải trở nên hiệu quả hơn về bộ nhớ (thường đạt được bằng cách thêm các thành phần phi tuyến) đồng thời có thể tập trung thời gian cho việc tối ưu các tham số nhờ sự gia tăng trong khả năng tính toán. Kéo theo đó, tiêu điểm trong học máy và thống kê đã chuyển từ các mô hình tuyến tính (tổng quát) và các phương pháp hạt nhân (*kernel methods*) sang các mạng nơ-ron sâu. Đây cũng là một trong những lý do những kỹ thuật cổ điển trong học sâu như perceptron đa tầng (?), mạng nơ-ron tích chập, (?), bộ nhớ ngắn hạn dài (*Long Short-Term Memory – LSTM*) (?), và học Q (?), đã được “tái khám phá” trong thập kỷ trước sau một khoảng thời gian dài ít được sử dụng.

Những tiến bộ gần đây trong các mô hình thống kê, các ứng dụng và các thuật toán đôi khi được liên hệ với Sự bùng nổ kỷ Cambry: thời điểm phát triển nhanh chóng trong sự tiến hóa của các loài. Thật vậy, các kỹ thuật tiên tiến nhất hiện nay không chỉ đơn thuần chỉ là hệ quả của việc các kỹ thuật cũ được áp dụng với các nguồn tài nguyên hiện tại. Danh sách dưới đây còn chưa thấm vào đâu với số lượng những ý tưởng đã và đang giúp các nhà nghiên cứu đạt được những thành tựu khổng lồ trong thập kỷ vừa qua.

- Các phương pháp tiên tiến trong việc kiểm soát năng lực, như Dropout (?), đã giúp làm giảm sự nguy hiểm của quá khứ. Việc này đạt được bằng cách thêm nhiễu (?) xuyên suốt khắp mạng, thay các trọng số bởi các biến ngẫu nhiên cho mục đích huấn luyện.
- Cơ chế tập trung giải quyết vấn đề thứ hai đã ám ảnh ngành thống kê trong hơn một thế kỷ: làm thế nào để tăng bộ nhớ và độ phức tạp của một hệ thống mà không làm tăng lượng tham số cần học. (?) đã tìm ra một giải pháp tinh tế bằng cách sử dụng một cấu trúc con trỏ có thể học được. Ví dụ trong dịch máy, thay vì phải nhớ toàn bộ câu với cách biểu diễn có số chiều cố định, ta chỉ cần lưu một con trỏ tới trạng thái trung gian của quá trình dịch. Việc này giúp tăng đáng kể độ chính xác của các câu dài vì lúc này mô hình không còn cần nhớ toàn bộ câu trước khi chuyển sang tạo câu tiếp theo.
- Thiết kế đa bước, ví dụ thông qua các Mạng Bộ Nhớ (*MemNets*) (?) và Bộ Lập trình-Phiên dịch Nơ-ron (?) cho phép các nhà nghiên cứu mô hình hóa thống kê mô tả các hướng tiếp cận tới việc suy luận (*reasoning*) qua nhiều chu kỳ. Những công cụ này cho phép các trạng thái nội tại của mạng nơ-ron sâu được biến đổi liên tục, từ đó có thể thực hiện một chuỗi các bước suy luận, tương tự như cách bộ vi xử lý thay đổi bộ nhớ khi thực hiện một phép tính toán.
- Một bước phát triển quan trọng khác là sự ra đời của GAN (?). Trong quá khứ, các phương pháp thống kê để đánh giá hàm mật độ xác suất và các mô hình sinh (*generative models*) tập trung vào việc tìm các phân phối xác suất hợp lý và các thuật toán (thường là xấp xỉ) để lấy mẫu từ các phân phối đó. Vì vậy, những thuật toán này có rất nhiều hạn chế và sự thiếu linh động khi kế thừa chính các mô hình thống kê đó. Phát kiến quan trọng của GANs là thay thế các thuật toán lấy mẫu đó bởi một thuật toán với các tham số khả vi (*differentiable*: có thể tính đạo hàm để áp dụng các thuật toán tối ưu dựa trên đó) bất kỳ. Các phương pháp này sau đó được điều chỉnh sao cho bộ phân loại (có hiệu quả giống bài kiểm tra hai mẫu trong xác suất) không thể phân biệt giữa dữ liệu thật và giả. Khả năng sử dụng các thuật toán bất kỳ để sinh dữ liệu đã thúc đẩy phương pháp đánh giá hàm mật độ xác suất khai sinh một loạt các kỹ thuật. Các ví dụ về biến đổi Ngựa thường thành Ngựa Vằn (?) và tạo giả khuôn mặt người nổi tiếng là các minh chứng của quá trình này.

Trong rất nhiều trường hợp, một GPU là không đủ để xử lý một lượng lớn dữ liệu sẵn có cho huấn luyện. Khả năng xây dựng các thuật toán huấn luyện phân tán song song đã cải tiến đáng kể trong thập kỷ vừa rồi. Một trong những thách thức chính trong việc thiết kế các thuật toán cho quy mô lớn là việc thuật toán tối ưu học sâu – hạ gradient ngẫu nhiên – phụ thuộc vào cách xử lý một lượng nhỏ dữ liệu, được gọi là minibatch. Đồng thời, batch nhỏ lại hạn chế sự hiệu quả của GPU. Bởi vậy, nếu ta huấn luyện trên 1024 GPU với 32 ảnh trong một batch sẽ cấu thành một minibatch lớn với 32 ngàn ảnh. Các công trình gần đây, khởi nguồn bởi Li (?), tiếp tục bởi (?) và (?) đẩy kích thước lên tới 64 ngàn mẫu, giảm thời gian huấn luyện ResNet50 trên ImageNet xuống dưới bảy phút so với thời gian huấn luyện hàng nhiều ngày trước đó.

- Khả năng song song hóa việc tính toán cũng đã góp phần quan trọng cho sự phát triển của học tăng

cường, ít nhất là với ứng dụng mà có thể tạo và sử dụng môi trường giả lập. Việc này đã dẫn tới sự tiến triển đáng kể ở môn cờ vây, các game Atari, Starcraft, và trong giả lập vật lý (ví dụ, sử dụng MuJoCo). Máy tính đạt được chất lượng vượt mức con người ở các ứng dụng này. Xem thêm mô tả về cách đạt được điều này trong AlphaGo tại (?). Tóm lại, học tăng cường làm việc tốt nhất nếu có cực nhiều bộ (trạng thái, hành động, điểm thưởng) để mô hình có thể thử và học cách chúng được liên hệ với nhau như thế nào. Các phần mềm mô phỏng giả lập cung cấp một môi trường như thế.

- Các framework Học Sâu đóng một vai trò thiết yếu trong việc thực hiện hóa các ý tưởng. Các framework thế hệ đầu tiên cho phép dễ dàng mô hình hóa bao gồm Caffe⁴³, Torch⁴⁴, và Theano⁴⁵. Rất nhiều bài báo được viết về cách sử dụng các công cụ này. Hiện tại, chúng bị thay thế bởi TensorFlow⁴⁶, thường được sử dụng thông qua API mức cao Keras⁴⁷, CNTK⁴⁸, Caffe 2⁴⁹ và Apache MxNet⁵⁰. Thế hệ công cụ thứ ba – công cụ học sâu dạng mệnh lệnh – được tiên phong bởi Chainer⁵¹, công cụ này sử dụng cú pháp tương tự như Python NumPy để mô tả các mô hình. Ý tưởng này được áp dụng bởi PyTorch⁵² và Gluon API⁵³ của MXNet. Khóa học này sử dụng nhóm công cụ cuối cùng để dạy về học sâu.

Sự phân chia công việc giữa (i) các nhà nghiên cứu hệ thống xây dựng các công cụ tốt hơn và (ii) các nhà mô hình hóa thống kê xây dựng các mạng tốt hơn đã đơn giản hóa công việc một cách đáng kể. Ví dụ, huấn luyện một mô hình hồi quy logistic tuyến tính từng là một bài tập về nhà không đơn giản cho tân nghiên cứu sinh tiến sĩ ngành học máy tại Đại học Carnegie Mellon năm 2014. Hiện nay, tác vụ này có thể đạt được với ít hơn 10 dòng mã, khiến việc này trở nên đơn giản với các lập trình viên.

4.6 Các câu chuyện thành công

Trí Tuệ Nhân Tạo có một lịch sử lâu dài trong việc mang đến những kết quả mà khó có thể đạt được bằng các phương pháp khác. Ví dụ, sắp xếp thư tín sử dụng công nghệ nhận dạng ký tự quang. Những hệ thống này được triển khai từ những năm 90 (đây là nguồn của các bộ dữ liệu chữ viết tay nổi tiếng MNIST và USPS). Các hệ thống tương tự cũng được áp dụng vào đọc ngân phiếu nộp tiền vào ngân hàng và tính điểm tín dụng cho ứng viên. Các giao dịch tài chính được kiểm tra có phải lừa đảo không một cách tự động. Đây là nền tảng phát triển cho rất nhiều hệ thống thanh toán điện tử như Paypal, Stripe, AliPay, WeChat, Apple, Visa và MasterCard. Các chương trình máy tính cho cờ vua đã phát triển trong hàng thập kỷ. Học máy đứng sau các hệ thống tìm kiếm, gợi ý, cá nhân hóa và xếp hạng trên mạng Internet. Nói cách khác, trí tuệ nhân tạo và học máy xuất hiện mọi nơi tuy nhiên khi ta không để ý thấy.

Chỉ tới gần đây AI mới được chú ý đến, chủ yếu là bởi nó cung cấp giải pháp cho các bài toán mà trước đây được coi là không khả thi.

- Các trợ lý thông minh như: Apple Siri, Amazon Alexa, hay Google Assistant có khả năng trả lời các câu hỏi thoại với độ chính xác chấp nhận được. Việc này cũng bao gồm những tác vụ đơn giản như bật đèn (hữu ích cho người tàn tật) tới đặt lịch hẹn cắt tóc và đưa ra các đoạn hội thoại để hỗ trợ các tổng đài chăm sóc khách hàng. Đây có lẽ là tín hiệu đáng chú ý nhất cho thấy AI đang ảnh hưởng tới cuộc sống thường ngày.

⁴³ <https://github.com/BVLC/caffe>

⁴⁴ <https://github.com/torch>

⁴⁵ <https://github.com/Theano/Theano>

⁴⁶ <https://github.com/tensorflow/tensorflow>

⁴⁷ <https://github.com/keras-team/keras>

⁴⁸ <https://github.com/Microsoft/CNTK>

⁴⁹ <https://github.com/caffe2/caffe2>

⁵⁰ <https://github.com/apache/incubator-mxnet>

⁵¹ <https://github.com/chainer/chainer>

⁵² <https://github.com/pytorch/pytorch>

⁵³ <https://github.com/apache/incubator-mxnet>

- Một thành phần chính trong trợ lý số là khả năng nhận dạng chính xác tiếng nói. Dần dần độ chính xác của những hệ thống này được cải thiện tới mức tương đương con người ((?)) cho vài ứng dụng cụ thể.
- Tương tự, nhận dạng vật thể cũng đã tiến một bước dài. Đánh giá một vật thể trong ảnh là một tác vụ khó trong năm 2010. Trong bảng xếp hạng ImageNet, (?) đạt được tỉ lệ lỗi top-5 là 28%. Tới 2017, (?) giảm tỉ lệ lỗi này xuống còn 2,25%. Các kết quả kinh ngạc tương tự cũng đã đạt được trong việc xác định chim cũng như chẩn đoán ung thư da.
- Các trò chơi từng là một pháo đài của trí tuệ loài người. Bắt đầu từ TDGammon [23], một chương trình chơi Backgammon (một môn cờ) sử dụng học tăng cường sai khác thời gian (*temporal difference* – TD), tiến triển trong giải thuật và tính toán đã dẫn đến các thuật toán cho một loạt các ứng dụng. Không giống Backgammon, cờ vua có một không gian trạng thái và tập các nước đi phức tạp hơn nhiều. DeepBlue chiến thắng Gary Kasparov, Campbell et al. (?), bằng cách sử dụng phần cứng chuyên biệt, đa luồng khổng lồ và thuật toán tìm kiếm hiệu quả trên toàn bộ cây trò chơi. Cờ vây còn khó hơn vì không gian trạng thái khổng lồ của nó. Năm 2015, AlphaGo đạt tới đẳng cấp con người, (?) nhờ sử dụng Học Sâu kết hợp với lấy mẫu cây Monte Carlo (*Monte Carlo tree sampling*). Thách thức trong Poker là không gian của trạng thái lớn và nó không được quan sát đầy đủ (ta không biết các quân bài của đối thủ). Libratus vượt chất lượng con người trong môn Poker sử dụng các chiến thuật có cấu trúc một cách hiệu quả (?). Những điều này thể hiện một sự tiến triển ấn tượng trong các trò chơi và tầm quan trọng của các thuật toán nâng cao trong đó.
- Dấu hiệu khác của tiến triển trong AI là sự phát triển của xe hơi và xe tải tự hành. Trong khi hệ thống tự động hoàn toàn còn xa mới đạt được, tiến triển ấn tượng đã được tạo ra theo hướng này với việc các công ty như Tesla, NVIDIA và Waymo ra mắt các sản phẩm ít nhất hỗ trợ bán tự động. Điều khiến tự động hoàn toàn mang nhiều thách thức là việc lái xe chuẩn mực đòi hỏi khả năng tiếp nhận, suy đoán và tích hợp các quy tắc vào hệ thống. Tại thời điểm hiện tại, học máy được sử dụng chủ yếu trong phần thị giác máy tính của các bài toán này. Phần còn lại vẫn phụ thuộc chủ yếu bởi những điều chỉnh của các kỹ sư.

Danh sách trên đây chỉ lướt qua những ứng dụng mà học máy có ảnh hưởng lớn. Ngoài ra, robot, hậu cần, sinh học điện toán, vật lý hạt và thiên văn học cũng tạo ra những thành quả ấn tượng gần đây phần nào nhờ vào học máy. Bởi vậy, Học Máy đang trở thành một công cụ phổ biến cho các kỹ sư và nhà khoa học.

Gần đây, câu hỏi về ngày tận thế do AI, hay điểm kỳ dị (*singularity*) của AI đã được nhắc tới trong các bài viết phi kỹ thuật về AI. Đã có những nỗi lo sợ về việc các hệ thống học máy bằng cách nào đó sẽ trở nên có cảm xúc và ra quyết định độc lập với những lập trình viên (và chủ nhân) về những điều ảnh hưởng trực tiếp tới sự sống của nhân loại. Trong phạm vi nào đó, AI đã ảnh hưởng tới sự sống của con người một cách trực tiếp, chẳng hạn như điểm tín dụng được tính tự động, tự động điều hướng xe hơi, hay các quyết định về việc liệu có chấp nhận bảo lãnh hay không sử dụng đầu vào là dữ liệu thống kê. Hoặc ít nghiêm trọng hơn, ta có thể yêu cầu Alexa bật máy pha cà phê.

May mắn thay, chúng ta còn xa mới đạt được một hệ thống AI có cảm xúc sẵn sàng điều khiển chủ nhân của nó (hay đốt cháy cà phê của họ). Thứ nhất, các hệ thống AI được thiết kế, huấn luyện và triển khai trong một môi trường cụ thể hướng mục đích. Trong khi hành vi của chúng có thể tạo ra ảo giác về trí tuệ phổ quát, đó vẫn là một tổ hợp của các quy tắc và các mô hình thống kê. Thứ hai, hiện tại các công cụ cho *trí tuệ nhân tạo phổ quát* đơn giản là không tồn tại. Chúng không thể tự cải thiện, hoài nghi bản thân, không thể thay đổi, mở rộng và tự cải thiện cấu trúc trong khi cố gắng giải quyết các tác vụ thông thường.

Một mối lo cấp bách hơn đó là AI có thể được sử dụng trong cuộc sống thường nhật như thế nào. Nhiều khả năng rất nhiều tác vụ đơn giản đang được thực hiện bởi tài xế xe tải và trợ lý cửa hàng có thể và sẽ bị tự động hóa. Các robot nông trại sẽ không những có khả năng làm giảm chi phí của nông nghiệp hữu cơ mà còn tự động hóa quá trình thu hoạch. Thời điểm này của cuộc cách mạng công nghiệp có thể có những hậu quả lan rộng khắp toàn xã hội (tài xế xe tải và trợ lý cửa hàng là một vài trong số những ngành phổ biến nhất ở nhiều địa phương). Đối với các mô hình thống kê khi được áp dụng không cẩn thận, có thể dẫn đến các quyết định

phân biệt chủng tộc, giới tính hoặc tuổi tác và gây nên những nỗi lo có cơ sở về tính công bằng nếu chúng được tự động hóa để đưa ra các quyết định có nhiều hệ lụy. Việc sử dụng các thuật toán này một cách cẩn thận là rất quan trọng. Với những gì ta biết ngày nay, việc này dấy lên một nỗi lo lớn hơn so với khả năng hủy diệt loài người của các siêu trí tuệ độc ác.

4.7 Tóm tắt

- Học máy nghiên cứu cách mà các hệ thống máy tính tận dụng *kinh nghiệm* (thường là dữ liệu) để cải thiện chất lượng trong những tác vụ cụ thể. Lĩnh vực này kết hợp các ý tưởng từ thống kê, khai phá dữ liệu, trí tuệ nhân tạo và tối ưu hóa. Học máy thường được sử dụng như một công cụ để triển khai các giải pháp trí tuệ nhân tạo.
- Là một nhánh của học máy, học biểu diễn (*representational learning*) tập trung vào việc tự động tìm kiếm phương pháp biểu diễn dữ liệu thích hợp, phần lớn thông qua việc học một quá trình biến đổi gồm nhiều bước.
- Hầu hết các tiến triển gần đây trong học sâu đạt được nhờ một lượng lớn dữ liệu thu thập từ các cảm biến giá rẻ và các ứng dụng quy mô Internet, cùng với sự phát triển đáng kể trong điện toán, chủ yếu là GPU.
- Việc tối ưu hóa toàn bộ hệ thống là yếu tố chính để đạt được chất lượng tốt. Sự sẵn có của các framework học sâu hiệu quả giúp cho việc thiết kế và triển khai tối ưu hóa trở nên dễ dàng hơn rất nhiều.

4.8 Bài tập

1. Phần nào của mã nguồn mà bạn đang viết có thể “được học”, tức có thể được cải thiện bằng cách học và tự động xác định các lựa chọn thiết kế? Trong mã nguồn của bạn có hiện diện các lựa chọn thiết kế dựa trên trực giác không?
2. Những bài toán nào bạn từng gặp có nhiều cách giải quyết, nhưng lại không có cách cụ thể nào để tự động hóa? Những bài toán này có thể rất phù hợp để áp dụng học sâu.
3. Nếu xem sự phát triển của trí tuệ nhân tạo như một cuộc cách mạng công nghiệp mới thì mối quan hệ giữa thuật toán và dữ liệu là gì? Nó có tương tự như động cơ hơi nước và than đá không (sự khác nhau căn bản là gì)?
4. Bạn còn có thể áp dụng phương pháp huấn luyện đầu-cuối ở lĩnh vực nào nữa? Vật lý? Kỹ thuật? Kinh tế lượng?

4.9 Thảo luận

- Tiếng Anh⁵⁴
- Tiếng Việt⁵⁵

⁵⁴ <https://discuss.mxnet.io/t/2310>

⁵⁵ <https://forum.machinelearningcoban.com/c/d2l>

4.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Lê Khắc Hồng Phúc
- Vũ Hữu Tiệp
- Sâm Thế Hải
- Hoàng Trọng Tuấn
- Trần Thị Hồng Hạnh
- Đoàn Võ Duy Thanh
- Phạm Chí Thành
- Mai Sơn Hải
- Vũ Đình Quyên
- Nguyễn Cảnh Thượng
- Lê Đàm Hồng Lộc
- Nguyễn Lê Quang Nhật

5 | Sơ bộ

Để bắt đầu với học sâu, ta sẽ cần nắm bắt một vài kỹ năng cơ bản. Tất cả những vấn đề về học máy đều có liên quan đến việc trích xuất thông tin từ dữ liệu. Vì vậy, chúng tôi sẽ bắt đầu bằng cách học các kỹ năng thực tế để lưu trữ, thao tác và xử lý dữ liệu.

Hơn nữa, học máy thường yêu cầu làm việc với các tập dữ liệu lớn, mà chúng ta có thể coi như ở dạng bảng, trong đó các hàng tương ứng với các mẫu và các cột tương ứng với các thuộc tính. Đại số tuyến tính cung cấp cho ta một tập kỹ thuật mạnh mẽ để làm việc với dữ liệu dạng bảng. Chúng ta sẽ không đi quá sâu mà chỉ tập trung cơ bản vào các toán tử ma trận cơ bản và cách thực thi chúng.

Bên cạnh đó, học sâu luôn liên quan đến tối ưu hoá. Chúng ta có một mô hình với bộ tham số và muốn tìm ra các tham số khớp với dữ liệu *nhất*. Việc xác định cách điều chỉnh từng tham số ở mỗi bước trong thuật toán đòi hỏi một chút kiến thức về giải tích, mà sẽ được giới thiệu ngắn gọn dưới đây. May thay, gói autograd sẽ tự động tính đạo hàm cho chúng ta, và sẽ được đề cập ngay sau đó.

Kế tiếp, học máy liên quan đến việc đưa ra những dự đoán như: Xác định giá trị của một số thuộc tính chưa biết dựa trên thông tin quan sát được? Để suy luận chặt chẽ dưới sự bất định, chúng ta sẽ cần tìm đến ngôn ngữ của xác suất.

Cuối cùng, tài liệu tham khảo chính thức cung cấp rất nhiều mô tả và ví dụ nằm ngoài cuốn sách này. Để kết thúc chương này, chúng tôi sẽ chỉ bạn cách tra cứu tài liệu tham khảo cho các thông tin cần thiết.

Cuốn sách này đã cung cấp nội dung toán học ở mức tối thiểu cần có để có được sự hiểu biết đúng đắn về học sâu. Tuy nhiên, điều đó không đồng nghĩa rằng cuốn sách này không cần các kiến thức toán học. Do vậy, chương này sẽ giới thiệu nhanh về các kiến thức toán học cơ bản và thông dụng, cho phép tất cả mọi người tối thiểu là sẽ hiểu được *hầu hết* nội dung toán trong quyển sách này. Nếu bạn muốn hiểu *tất cả* nội dung toán học, hãy tham khảo thêm `chap_appendix_math`.

5.1 Thao tác với Dữ liệu

Muốn thực hiện bất cứ điều gì, chúng ta đều cần một cách nào đó để lưu trữ và thao tác với dữ liệu. Thường sẽ có hai điều quan trọng chúng ta cần làm với dữ liệu: (i) thu thập; và (ii) xử lý sau khi đã có dữ liệu trên máy tính. Sẽ thật vô nghĩa khi thu thập dữ liệu mà không có cách để lưu trữ nó, vậy nên trước tiên hãy cùng làm quen với dữ liệu tổng hợp. Để bắt đầu, chúng tôi giới thiệu mảng n chiều (`ndarray`) – công cụ chính trong MXNet để lưu trữ và biến đổi dữ liệu. Trong MXNet, `ndarray` là một lớp và mỗi thực thể của lớp đó là “một `ndarray`”.

Nếu bạn từng làm việc với NumPy, gói tính toán phổ biến nhất trong Python, bạn sẽ thấy mục này quen thuộc. Việc này là có chủ đích. Chúng tôi thiết kế `ndarray` trong MXNet là một dạng mở rộng của `ndarray` trong NumPy với một vài tính năng đặc biệt. Thứ nhất, `ndarray` trong MXNet hỗ trợ tính toán phi đồng bộ trên CPU, GPU, và các kiến trúc phân tán đám mây, trong khi NumPy chỉ hỗ trợ tính toán trên CPU. Thứ hai, `ndarray` trong MXNet hỗ trợ tính vi phân tự động. Những tính chất này khiến `ndarray` của MXNet phù

hợp với học sâu. Thông qua cuốn sách, nếu không nói gì thêm, chúng ta ngầm hiểu ndarray là ndarray của MXNet.

5.1.1 Bắt đầu

Trong mục này, mục tiêu của chúng tôi là trang bị cho bạn các kiến thức toán cơ bản và cài đặt các công cụ tính toán mà bạn sẽ xây dựng dựa trên nó xuyên suốt cuốn sách này. Đừng lo nếu bạn gặp khó khăn với các khái niệm toán khó hiểu hoặc các hàm trong thư viện tính toán. Các mục tiếp theo sẽ nhắc lại những khái niệm này trong từng ngữ cảnh kèm theo ví dụ thực tiễn. Mặt khác, nếu bạn đã có kiến thức nền tảng và muốn đi sâu hơn vào các nội dung toán, bạn có thể bỏ qua mục này.

Để bắt đầu, ta cần khai báo mô-đun np (numpy) và npx (numpy_extension) từ MXNet. Ở đây, mô-đun np bao gồm các hàm hỗ trợ bởi NumPy, trong khi mô-đun npx chứa một tập các hàm mở rộng được phát triển để hỗ trợ học sâu trong một môi trường giống với NumPy. Khi sử dụng ndarray, ta luôn cần gọi hàm set_np: điều này nhằm đảm bảo sự tương thích của việc xử lý ndarray bằng các thành phần khác của MXNet.

```
from mxnet import np, npx  
npx.set_np()
```

Một ndarray biểu diễn một mảng (có thể đa chiều) các giá trị số. Với một trực, một ndarray tương ứng (trong toán) với một *vector*. Với hai trực, một ndarray tương ứng với một *ma trận*. Các mảng với nhiều hơn hai trực không có tên toán học cụ thể—chúng được gọi chung là *tensor*.

Để bắt đầu, chúng ta sử dụng arange để tạo một vector hàng x chứa 12 số nguyên đầu tiên bắt đầu từ 0, nhưng được khởi tạo mặc định dưới dạng số thực. Mỗi giá trị trong một ndarray được gọi là một *phần tử* của ndarray đó. Như vậy, có 12 phần tử trong ndarray x. Nếu không nói gì thêm, một ndarray mới sẽ được lưu trong bộ nhớ chính và được tính toán trên CPU.

```
x = np.arange(12)  
x
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

Chúng ta có thể lấy *kích thước* (độ dài theo mỗi trực) của ndarray bằng thuộc tính shape.

```
x.shape
```

```
(12,)
```

Nếu chỉ muốn biết tổng số phần tử của một ndarray, nghĩa là tích của tất cả các thành phần trong shape, ta có thể sử dụng thuộc tính size. Vì ta đang làm việc với một vector, cả shape và size của nó đều chứa cùng một phần tử duy nhất.

```
x.size
```

```
12
```

Để thay đổi kích thước của một ndarray mà không làm thay đổi số lượng phần tử cũng như giá trị của chúng, ta có thể gọi hàm reshape. Ví dụ, ta có thể biến đổi ndarray x trong ví dụ trên, từ một vector

hàng với kích thước (12,) sang một ma trận với kích thước (3, 4). ndarray mới này chứa 12 phần tử y hệt, nhưng được xem như một ma trận với 3 hàng và 4 cột. Mặc dù kích thước thay đổi, các phần tử của x vẫn giữ nguyên. Chú ý rằng size giữ nguyên khi thay đổi kích thước.

```
x = x.reshape(3, 4)  
x
```

```
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.]])
```

Việc chỉ định cụ thể mọi chiều khi thay đổi kích thước là không cần thiết. Nếu kích thước mong muốn là một ma trận với kích thước (chiều_cao, chiều_rộng), thì sau khi biết chiều_rộng, chiều_cao có thể được ngầm suy ra. Tại sao ta lại cần phải tự làm phép tính chia? Trong ví dụ trên, để có được một ma trận với 3 hàng, chúng ta phải chỉ định rõ ràng nó có 3 hàng và 4 cột. May mắn thay, ndarray có thể tự động tính một chiều từ các chiều còn lại. Ta có thể dùng chức năng này bằng cách đặt -1 cho chiều mà ta muốn ndarray tự suy ra. Trong trường hợp vừa rồi, thay vì gọi `x.reshape(3, 4)`, ta có thể gọi `x.reshape(-1, 4)` hoặc `x.reshape(3, -1)`.

Phương thức `empty` lấy một đoạn bộ nhớ và trả về một ma trận mà không thay đổi các giá trị sẵn có tại đoạn bộ nhớ đó. Việc này có hiệu quả tính toán đáng kể nhưng ta phải cẩn trọng bởi các phần tử đó có thể chứa bất kỳ giá trị nào, kể cả các số rất lớn!

```
np.empty((3, 4))
```

```
array([[2.553881e-08, 3.062538e-41, 0.000000e+00, 0.000000e+00],  
       [0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00],  
       [0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00]])
```

Thông thường ta muốn khởi tạo các ma trận với các giá trị bằng không, bằng một, bằng hằng số nào đó hoặc bằng các mẫu ngẫu nhiên lấy từ một phân phối cụ thể. Ta có thể tạo một ndarray biểu diễn một tensor với tất cả các phần tử bằng 0 và có kích thước (2, 3, 4) như sau:

```
np.zeros((2, 3, 4))
```

```
array([[[0., 0., 0., 0.],  
        [0., 0., 0., 0.],  
        [0., 0., 0., 0.]],  
  
       [[[0., 0., 0., 0.],  
        [0., 0., 0., 0.],  
        [0., 0., 0., 0.]]])
```

Tương tự, ta có thể tạo các tensor với các phần tử bằng 1 như sau:

```
np.ones((2, 3, 4))
```

```
array([[[1., 1., 1., 1.],  
        [1., 1., 1., 1.],  
        [1., 1., 1., 1.]],  
       [[[1., 1., 1., 1.],  
        [1., 1., 1., 1.],  
        [1., 1., 1., 1.]]])
```

(continues on next page)

```
[ [1., 1., 1., 1.],
  [1., 1., 1., 1.],
  [1., 1., 1., 1.]] )
```

Ta thường muốn lấy mẫu ngẫu nhiên cho mỗi phần tử trong một ndarray từ một phân phối xác suất. Ví dụ, khi xây dựng các mảng để chứa các tham số của một mạng nơ-ron, ta thường khởi tạo chúng với các giá trị ngẫu nhiên. Đoạn mã dưới đây tạo một ndarray có kích thước (3, 4) với các phần tử được lấy mẫu ngẫu nhiên từ một phân phối Gauss (phân phối chuẩn) với trung bình bằng 0 và độ lệch chuẩn 1.

```
np.random.normal(0, 1, size=(3, 4))
```

```
array([[ 2.2122064 ,  1.1630787 ,  0.7740038 ,  0.4838046 ],
       [ 1.0434403 ,  0.29956347,  1.1839255 ,  0.15302546],
       [ 1.8917114 , -1.1688148 , -1.2347414 ,  1.5580711 ]])
```

Ta cũng có thể khởi tạo giá trị cụ thể cho mỗi phần tử trong ndarray mong muốn bằng cách đưa vào một mảng Python (hoặc mảng của mảng) chứa các giá trị số. Ở đây, mảng ngoài cùng tương ứng với trục 0, và mảng bên trong tương ứng với trục 1.

```
np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
array([[2., 1., 4., 3.],
       [1., 2., 3., 4.],
       [4., 3., 2., 1.]])
```

5.1.2 Phép toán

Cuốn sách này không nói về kỹ thuật phần mềm. Chúng tôi không chỉ hứng thú với việc đơn giản đọc và ghi dữ liệu vào/từ các mảng mà còn muốn thực hiện các phép toán trên các mảng này. Một vài phép toán đơn giản và hữu ích nhất là các phép toán tác động lên *từng phần tử* (*elementwise*). Các phép toán này hoạt động như những phép toán chuẩn trên số vô hướng áp dụng lên từng phần tử của mảng. Với những hàm nhận hai mảng đầu vào, phép toán theo từng thành phần được áp dụng trên từng cặp phần tử tương ứng của hai mảng. Ta có thể tạo một hàm theo từng phần tử từ một hàm bất kỳ ánh xạ từ một số vô hướng tới một số vô hướng.

Trong toán học, ta ký hiệu một toán tử *đơn ngôi* vô hướng (lấy một đầu vào) bởi $f : \mathbb{R} \rightarrow \mathbb{R}$. Điều này nghĩa là hàm số ánh xạ từ một số thực bất kỳ (\mathbb{R}) sang một số thực khác. Tương tự, ta ký hiệu một toán tử *hai ngôi* vô hướng (lấy hai đầu vào, trả về một đầu ra) bởi $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Cho trước hai vector bất kỳ \mathbf{u} và \mathbf{v} với *cùng kích thước*, và một toán tử hai ngôi f , ta có thể tính được một vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ bằng cách tính $c_i \leftarrow f(u_i, v_i)$ cho mọi i với c_i, u_i , và v_i là các phần tử thứ i của vector \mathbf{c}, \mathbf{u} , và \mathbf{v} . Ở đây, chúng ta tạo một hàm trả về vector $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ bằng cách áp dụng hàm f lên từng phần tử.

Trong MXNet, các phép toán tiêu chuẩn (+, -, *, /, và **) là các phép toán theo từng phần tử trên các tensor đồng kích thước bất kỳ. Ta có thể gọi những phép toán theo từng phần tử lên hai tensor đồng kích thước. Trong ví dụ dưới đây, các dấu phẩy được sử dụng để tạo một tuple 5 phần tử với mỗi phần tử là kết quả của một phép toán theo từng phần tử.

```
x = np.array([1, 2, 4, 8])
y = np.array([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y # The ** operator is exponentiation
```

```
(array([ 3.,  4.,  6., 10.]),
 array([-1.,  0.,  2.,  6.]),
 array([ 2.,  4.,  8., 16.]),
 array([ 0.5,  1.,  2.,  4.]),
 array([ 1.,  4., 16., 64.]))
```

Rất nhiều các phép toán khác có thể được áp dụng theo từng phần tử, bao gồm các phép toán đơn ngôi như hàm mũ cơ số e .

```
np.exp(x)
```

```
array([2.7182817e+00, 7.3890562e+00, 5.4598148e+01, 2.9809580e+03])
```

Ngoài các phép tính theo từng phần tử, ta cũng có thể thực hiện các phép toán đại số tuyến tính, bao gồm tích vô hướng của hai vector và phép nhân ma trận. Chúng ta sẽ giải thích những điểm quan trọng của đại số tuyến tính (mà không cần kiến thức nền tảng) trong `sec_linear-algebra`.

Ta cũng có thể *nối* nhiều ndarray với nhau, xếp chồng chúng lên nhau để tạo ra một ndarray lớn hơn. Ta chỉ cần cung cấp một danh sách các ndarray và khai báo chúng được nối theo trực nào. Ví dụ dưới đây thể hiện cách nối hai ma trận theo hàng (trục 0, phần tử đầu tiên của kích thước) và theo cột (trục 1, phần tử thứ hai của kích thước). Ta có thể thấy rằng, cách thứ nhất tạo một ndarray với độ dài trục 0 (6) bằng tổng các độ dài trục 0 của hai ndarray đầu vào ($3 + 3$); trong khi cách thứ hai tạo một ndarray với độ dài trục 1 (8) bằng tổng các độ dài trục 1 của hai ndarray đầu vào ($4 + 4$).

```
x = np.arange(12).reshape(3, 4)
y = np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
np.concatenate([x, y], axis=0), np.concatenate([x, y], axis=1)
```

```
(array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [ 2.,  1.,  4.,  3.],
       [ 1.,  2.,  3.,  4.],
       [ 4.,  3.,  2.,  1.]]),
 array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
       [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
       [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

Đôi khi, ta muốn tạo một ndarray nhị phân thông qua các *mệnh đề logic*. Lấy $x == y$ làm ví dụ. Với mỗi vị trí, nếu giá trị của x và y tại vị trí đó bằng nhau thì phần tử tương ứng trong ndarray mới lấy giá trị 1, nghĩa là mệnh đề logic $x == y$ là đúng tại vị trí đó; ngược lại vị trí đó lấy giá trị 0.

```
x == y
```

```
array([[False,  True, False,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

Lấy tổng mọi phần tử trong một ndarray tạo ra một ndarray với chỉ một phần tử.

```
x.sum()
```

```
array(66.)
```

Ta cũng có thể thay `x.sum()` bởi `np.sum(x)`.

5.1.3 Cơ chế Lan truyền

Trong mục trên, ta đã thấy cách thực hiện các phép toán theo từng phần tử với hai ndarray đồng kích thước. Trong những điều kiện nhất định, thậm chí khi kích thước khác nhau, ta vẫn có thể thực hiện các phép toán theo từng phần tử bằng cách sử dụng *cơ chế lan truyền* (*broadcasting mechanism*). Cơ chế này hoạt động như sau: Thứ nhất, mở rộng một hoặc cả hai mảng bằng cách lặp lại các phần tử một cách hợp lý sao cho sau phép biến đổi này, hai ndarray có cùng kích thước. Thứ hai, thực hiện các phép toán theo từng phần tử với hai mảng mới này.

Trong hầu hết các trường hợp, chúng ta lan truyền một mảng theo trực có độ dài ban đầu là 1, như ví dụ dưới đây:

```
a = np.arange(3).reshape(3, 1)
b = np.arange(2).reshape(1, 2)
a, b
```

```
(array([[0.],
       [1.],
       [2.]]),
 array([[0., 1.]]))
```

Vì `a` và `b` là các ma trận có kích thước lần lượt là 3×1 và 1×2 , kích thước của chúng không khớp nếu ta muốn thực hiện phép cộng. Ta *lan truyền* các phần tử của cả hai ma trận thành các ma trận 3×2 như sau: lặp lại các cột trong ma trận `a` và các hàng trong ma trận `b` trước khi cộng chúng theo từng phần tử.

```
a + b
```

```
array([[0., 1.],
       [1., 2.],
       [2., 3.]])
```

5.1.4 Chỉ số và Cắt chọn mảng

Cũng giống như trong bất kỳ mảng Python khác, các phần tử trong một ndarray có thể được truy cập theo chỉ số. Tương tự, phần tử đầu tiên có chỉ số 0 và khoảng được cắt chọn bao gồm phần tử đầu tiên nhưng *không* phần tử cuối cùng. Và trong các danh sách Python tiêu chuẩn, chúng ta có thể truy cập các phần tử theo vị trí đếm ngược từ cuối danh sách bằng cách sử dụng các chỉ số âm.

Vì vậy, `[-1]` chọn phần tử cuối cùng và `[1:3]` chọn phần tử thứ hai và phần tử thứ ba như sau:

```
x[-1], x[1:3]
```

```
(array([ 8.,  9., 10., 11.]),
 array([[ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

Ngoài việc đọc, chúng ta cũng có thể viết các phần tử của ma trận bằng cách chỉ định các chỉ số.

```
x[1, 2] = 9
x
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  9.,  7.],
       [ 8.,  9., 10., 11.]])
```

Nếu chúng ta muốn gán cùng một giá trị cho nhiều phần tử, chúng ta chỉ cần trỏ đến tất cả các phần tử đó và gán giá trị cho chúng. Chẳng hạn, `[0:2, :]` truy cập vào hàng thứ nhất và thứ hai, trong đó : lấy tất cả các phần tử đọc theo trục 1 (cột). Ở đây chúng ta đã thảo luận về cách truy cập vào ma trận, nhưng tất nhiên phương thức này cũng áp dụng cho các vector và tensor với nhiều hơn 2 chiều.

```
x[0:2, :] = 12
x
```

```
array([[12., 12., 12., 12.],
       [12., 12., 12., 12.],
       [ 8.,  9., 10., 11.]])
```

5.1.5 Tiết kiệm Bộ nhớ

Ở ví dụ trước, mỗi khi chạy một phép tính, chúng ta sẽ cấp phát bộ nhớ mới để lưu trữ kết quả của lượt chạy đó. Cụ thể hơn, nếu viết `y = x + y`, ta sẽ ngừng tham chiếu đến ndarray mà `y` đã chỉ đến trước đó và thay vào đó gán `y` vào bộ nhớ được cấp phát mới. Trong ví dụ tiếp theo, chúng ta sẽ minh họa việc này với hàm `id()` của Python - hàm cung cấp địa chỉ chính xác của một đối tượng được tham chiếu trong bộ nhớ. Sau khi chạy `y = y + x`, chúng ta nhận ra rằng `id(y)` chỉ đến một địa chỉ khác. Đó là bởi vì Python trước hết sẽ tính `y + x`, cấp phát bộ nhớ mới cho kết quả trả về và gán `y` vào địa chỉ mới này trong bộ nhớ.

```
before = id(y)
y = y + x
id(y) == before
```

```
False
```

Đây có thể là điều không mong muốn vì hai lý do. Thứ nhất, không phải lúc nào chúng ta cũng muốn cấp phát bộ nhớ không cần thiết. Trong học máy, ta có thể có đến hàng trăm megabytes tham số và cập nhật tất cả chúng nhiều lần mỗi giây, và thường thì ta muốn thực thi các cập nhật này *tại chỗ*. Thứ hai, chúng ta có thể trỏ đến cùng tham số từ nhiều biến khác nhau. Nếu không cập nhật tại chỗ, các bộ nhớ đã bị loại bỏ sẽ không được giải phóng, dẫn đến khả năng một số chỗ trong mã nguồn sẽ vô tình tham chiếu lại các tham số cũ.

May mắn thay, ta có thể dễ dàng thực hiện các phép tính tại chỗ với MXNet. Chúng ta có thể gán kết quả của một phép tính cho một mảng đã được cấp phát trước đó bằng ký hiệu cắt chọn (*slice notation*), ví dụ, `y[:] = <expression>`. Để minh họa khái niệm này, đầu tiên chúng ta tạo một ma trận mới `z` có cùng kích thước với ma trận `y`, sử dụng `zeros_like` để gán giá trị khởi tạo bằng 0.

```
z = np.zeros_like(y)
print('id(z):', id(z))
z[:] = x + y
print('id(z):', id(z))
```

```
id(z): 139833398823744
id(z): 139833398823744
```

Nếu các tính toán tiếp theo không tái sử dụng giá trị của `x`, chúng ta có thể viết `x[:] = x + y` hoặc `x += y` để giảm thiểu việc sử dụng bộ nhớ không cần thiết trong quá trình tính toán.

```
before = id(x)
x += y
id(x) == before
```

```
True
```

5.1.6 Chuyển đổi sang các Đối Tượng Python Khác

Chuyển đổi một MXNet `ndarray` sang NumPy `ndarray` hoặc ngược lại là khá đơn giản. Tuy nhiên, kết quả của phép chuyển đổi này không chia sẻ bộ nhớ với đối tượng cũ. Điểm bất tiện này tuy nhỏ nhưng lại khá quan trọng: khi bạn thực hiện các phép tính trên CPU hoặc GPUs, bạn không muốn MXNet dừng việc tính toán để chờ xem liệu gói Numpy của Python có sử dụng cùng bộ nhớ đó để làm việc khác không. Hàm `array` và `asnumpy` sẽ giúp bạn giải quyết vấn đề này.

```
a = x.asnumpy()
b = np.array(a)
type(a), type(b)
```

```
(numpy.ndarray, mxnet.numpy.ndarray)
```

Để chuyển đổi một mảng `ndarray` chứa một phần tử sang số vô hướng Python, ta có thể gọi hàm `item` hoặc các hàm có sẵn trong Python.

```
a = np.array([3.5])
a, a.item(), float(a), int(a)
```

```
(array([3.5]), 3.5, 3.5, 3)
```

5.1.7 Tổng kết

- MXNet ndarray là phần mở rộng của NumPy ndarray với một số ưu thế vượt trội giúp cho nó phù hợp với học sâu.
- MXNet ndarray cung cấp nhiều chức năng bao gồm các phép toán cơ bản, cơ chế lan truyền (*broadcasting*), chỉ số (*indexing*), cắt chọn (*slicing*), tiết kiệm bộ nhớ và khả năng chuyển đổi sang các đối tượng Python khác.

5.1.8 Bài tập

1. Chạy đoạn mã nguồn trong mục này. Thay đổi điều kiện mệnh đề $x == y$ sang $x < y$ hoặc $x > y$, sau đó kiểm tra dạng của ndarray nhận được.
2. Thay hai ndarray trong phép tính theo từng phần tử ở phần cơ chế lan truyền (*broadcasting mechanism*) với các ndarray có kích thước khác, ví dụ như tensor ba chiều. Kết quả có giống như bạn mong đợi hay không?

5.1.9 Thảo luận

- Tiếng Anh⁵⁶
- Tiếng Việt⁵⁷

5.1.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Trần Thị Hồng Hạnh
- Phạm Minh Đức
- Lê Đàm Hồng Lộc
- Nguyễn Lê Quang Nhật

⁵⁶ <https://discuss.mxnet.io/t/2315>

⁵⁷ <https://forum.machinelearningcoban.com/c/d2l>

5.2 Tiền xử lý dữ liệu

Cho đến giờ chúng tôi đã đề cập tới rất nhiều kỹ thuật thao tác dữ liệu được lưu trong dạng ndarray. Nhưng để áp dụng học sâu vào giải quyết các vấn đề thực tế, ta thường phải bắt đầu bằng việc xử lý dữ liệu thô, chứ không có luôn dữ liệu ngăn nắp được chuẩn bị sẵn trong định dạng ndarray. Trong số các công cụ phân tích dữ liệu phổ biến của Python, gói pandas khá được ưa chuộng. Cũng như nhiều gói khác trong hệ sinh thái rộng lớn của Python, ‘pandas’ có thể được sử dụng kết hợp với định dạng ndarray. Vì vậy, chúng ta sẽ đi nhanh qua các bước để tiền xử lý dữ liệu thô bằng pandas rồi đổi chúng sang dạng ndarray. Nhiều kỹ thuật tiền xử lý dữ liệu khác sẽ được giới thiệu trong các chương sau.

5.2.1 Đọc tập dữ liệu

Để lấy ví dụ, ta bắt đầu bằng việc tạo một tập dữ liệu nhân tạo lưu trong file csv `../data/house_tiny.csv` (csv - comma-separated values - giá trị tách nhau bằng dấu phẩy). Dữ liệu lưu ở các định dạng khác cũng có thể được xử lý tương tự. Hàm `mkdir_if_not_exist` dưới đây để đảm bảo rằng thư mục `../data` tồn tại. Chú thích `# Saved in the d2l package for later use` (*Lưu lại trong gói d2l để dùng sau*) là kí hiệu đánh dấu các hàm, lớp hoặc các lệnh import được lưu trong gói d2l, để sau này ta có thể trực tiếp gọi hàm `d2l.mkdir_if_not_exist()`.

```
import os

# Saved in the d2l package for later use
def mkdir_if_not_exist(path):
    if not isinstance(path, str):
        path = os.path.join(*path)
    if not os.path.exists(path):
        os.makedirs(path)
```

Sau đây ta ghi tệp dữ liệu vào file csv theo từng hàng một.

```
data_file = '../data/house_tiny.csv'
mkdir_if_not_exist('../data')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n') # Column names
    f.write('NA,Pave,127500\n') # Each row is a data point
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

Để nạp tập dữ liệu thô từ tệp csv vừa được tạo ra, ta dùng gói thư viện pandas và gọi hàm `read_csv`. Bộ dữ liệu này có 4 hàng và 3 cột, trong đó mỗi hàng biểu thị số phòng (“NumRooms”), kiểu lối đi (“Alley”), và giá (“Price”) của căn nhà.

```
# If pandas is not installed, just uncomment the following line:
# !pip install pandas
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

	NumRooms	Alley	Price
0	NaN	Pave	127500
1	2.0	NaN	106000
2	4.0	NaN	178100
3	NaN	NaN	140000

5.2.2 Xử lý dữ liệu thiếu

Để ý rằng giá trị “NaN” là các giá trị bị thiếu. Để xử lý dữ liệu thiếu, các cách thường được áp dụng là *quy buộc* (*imputation*) và *xoá bỏ* (*deletion*), trong đó quy buộc thay thế giá trị bị thiếu bằng giá trị khác, trong khi xoá bỏ sẽ bỏ qua các giá trị bị thiếu. Dưới đây chúng ta xem xét phương pháp quy buộc.

Bằng phương pháp đánh chỉ số theo số nguyên (`iloc`), chúng ta tách data thành `inputs` (tương ứng với hai cột đầu) và `outputs` (tương ứng với cột cuối cùng). Với các giá trị số bị thiếu trong `inputs`, ta thay thế phần tử “NaN” bằng giá trị trung bình cộng của cùng cột đó.

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

	NumRooms	Alley
0	3.0	Pave
1	2.0	NaN
2	4.0	NaN
3	3.0	NaN

Với các giá trị dạng hạng mục hoặc số rời rạc trong `inputs`, ta coi “NaN” là một mục riêng. Vì cột “Alley” chỉ nhận 2 giá trị riêng lẻ là “Pave” (được lát gạch) và “NaN”, pandas có thể tự động chuyển cột này thành 2 cột “Alley_Pave” và “Alley_nan”. Những hàng có kiểu lối đi là “Pave” sẽ có giá trị của cột “Alley_Pave” và cột “Alley_nan” tương ứng là 1 và 0. Hàng mà không có giá trị cho kiểu lối đi sẽ có giá trị cột “Alley_Pave” và cột “Alley_nan” lần lượt là 0 và 1.

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

	NumRooms	Alley_Pave	Alley_nan
0	3.0	1	0
1	2.0	0	1
2	4.0	0	1
3	3.0	0	1

5.2.3 Chuyển sang định dạng ndarray

Giờ thì toàn bộ các giá trị trong inputs và outputs đã ở dạng số, chúng đã có thể được chuyển sang định dạng ndarray. Khi đã ở định dạng này, chúng có thể được biến đổi và xử lý với những chức năng của ndarray đã được giới thiệu ở sec_ndarray.

```
from mxnet import np

X, y = np.array(inputs.values), np.array(outputs.values)
X, y

(array([[3., 1., 0.],
       [2., 0., 1.],
       [4., 0., 1.],
       [3., 0., 1.]], dtype=float64),
 array([127500, 106000, 178100, 140000], dtype=int64))
```

5.2.4 Tóm tắt

- Cũng như nhiều gói mở rộng trong hệ sinh thái khổng lồ của Python, pandas có thể làm việc được với ndarray.
- Phương pháp quy buộc hoặc xoá bỏ có thể dùng để xử lý dữ liệu bị thiếu.

5.2.5 Bài tập

Tạo một tập dữ liệu với nhiều hàng và cột hơn.

1. Xoá cột có nhiều giá trị bị thiếu nhất.
2. Chuyển bộ dữ liệu đã được xử lý sang định dạng ndarray.

5.2.6 Thảo luận

- Tiếng Anh⁵⁸
- Tiếng Việt⁵⁹

5.2.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thượng
- Phạm Hồng Vinh
- Đoàn Võ Duy Thành

⁵⁸ <https://discuss.mxnet.io/t/2315>

⁵⁹ <https://forum.machinelearningcoban.com/c/d2l>

- Vũ Hữu Tiệp
- Mai Sơn Hải

5.3 Đại số tuyến tính

Bây giờ bạn đã có thể lưu trữ và xử lý dữ liệu, hãy cùng ôn qua những kiến thức đại số tuyến tính cần thiết để hiểu và lập trình hầu hết các mô hình được nhắc tới trong quyển sách này. Dưới đây, chúng tôi giới thiệu các đối tượng toán học, số học và phép tính cơ bản trong đại số tuyến tính, biểu diễn chúng bằng cả ký hiệu toán học và cách triển khai lập trình tương ứng.

5.3.1 Số vô hướng

Nếu bạn chưa từng học đại số tuyến tính hay học máy, có lẽ bạn mới chỉ có kinh nghiệm làm toán với từng con số riêng lẻ. Và nếu bạn đã từng phải cân bằng sổ thu chi hoặc đơn giản là trả tiền cho bữa ăn, thì hẳn bạn đã biết cách thực hiện các phép tính cơ bản như cộng trừ nhân chia các cặp số. Ví dụ, nhiệt độ tại Palo Alto là 52 độ Fahrenheit. Chúng ta gọi các giá trị mà chỉ bao gồm một số duy nhất là *số vô hướng* (*scalar*). Nếu bạn muốn chuyển giá trị nhiệt độ trên sang độ Celsius (thang đo nhiệt độ hợp lý hơn theo hệ mét), bạn sẽ phải tính biểu thức $c = \frac{5}{9}(f - 32)$ với giá trị f bằng 52. Trong phương trình trên, mỗi số hạng — 5, 9 và 32 — là các số vô hướng. Các ký hiệu c và f được gọi là *biến* và chúng biểu diễn các giá trị số vô hướng chưa biết.

Trong quyển sách này, chúng tôi sẽ tuân theo quy ước ký hiệu các biến vô hướng bằng các chữ cái viết thường (chẳng hạn x , y và z). Chúng tôi ký hiệu không gian (liên tục) của tất cả các số thực vô hướng là \mathbb{R} . Vì tính thiết thực, chúng tôi sẽ bỏ qua định nghĩa chính xác của *không gian*. Nhưng bạn cần nhớ $x \in \mathbb{R}$ là cách toán học để thể hiện x là một số thực vô hướng. Ký hiệu \in đọc là “thuộc” và đơn thuần biểu diễn việc phần tử thuộc một tập hợp. Tương tự, ta có thể viết $x, y \in \{0, 1\}$ để ký hiệu cho việc các số x và y chỉ có thể nhận giá trị 0 hoặc 1.

Trong mã nguồn MXNet, một số vô hướng được biểu diễn bằng một `ndarray` với chỉ một phần tử. Trong đoạn mã dưới đây, chúng ta khởi tạo hai số vô hướng và thực hiện các phép tính quen thuộc như cộng, trừ, nhân, chia và lũy thừa với chúng.

```
from mxnet import np, npx
npx.set_np()

x = np.array(3.0)
y = np.array(2.0)

x + y, x * y, x / y, x ** y
```

```
(array(5.), array(6.), array(1.5), array(9.))
```

5.3.2 Vector

Bạn có thể xem vector đơn thuần như một dãy các số vô hướng. Chúng ta gọi các giá trị đó là *phần tử* (*thành phần*) của vector. Khi dùng vector để biểu diễn các mẫu trong tập dữ liệu, giá trị của chúng thường mang ý nghĩa liên quan tới đời thực. Ví dụ, nếu chúng ta huấn luyện một mô hình dự đoán rủi ro vỡ nợ, chúng ta có thể gán cho mỗi ứng viên một vector gồm các thành phần tương ứng với thu nhập, thời gian làm việc, số lần vỡ nợ trước đó của họ và các yếu tố khác. Nếu chúng ta đang tìm hiểu về rủi ro bị đau tim của bệnh nhân, ta có thể biểu diễn mỗi bệnh nhân bằng một vector gồm các phần tử mang thông tin về dấu hiệu sinh tồn gần nhất, nồng độ cholesterol, số phút tập thể dục mỗi ngày, v.v. Trong ký hiệu toán học, chúng ta thường biểu diễn vector bằng chữ cái in đậm viết thường (ví dụ **x**, **y**, và **z**).

Trong MXNet, chúng ta làm việc với vector thông qua các ndarray 1-chiều. Thường thì ndarray có thể có chiều dài bất kỳ, tùy thuộc vào giới hạn bộ nhớ máy tính.

```
x = np.arange(4)  
x
```

```
array([0., 1., 2., 3.])
```

Một phần tử bất kỳ trong vector có thể được ký hiệu sử dụng chỉ số dưới. Ví dụ ta có thể viết x_i để ám chỉ phần tử thứ i của **x**. Lưu ý rằng phần tử x_i là một số vô hướng nên nó không được in đậm. Có rất nhiều tài liệu tham khảo xem vector cột là chiều mặc định của vector, và quyển sách này cũng vậy. Trong toán học, một vector có thể được viết như sau

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad (5.3.1)$$

trong đó x_1, \dots, x_n là các phần tử của vector. Trong mã nguồn, chúng ta sử dụng chỉ số để truy cập các phần tử trong ndarray.

```
x[3]
```

```
array(3.)
```

Độ dài, Chiều, và Kích thước

Hãy quay lại với những khái niệm từ `sec_ndarray`. Một vector đơn thuần là một dãy các số. Mỗi vector, tương tự như dãy, đều có một độ dài. Trong ký hiệu toán học, nếu ta muốn nói rằng một vector **x** chứa n các số thực vô hướng, ta có thể biểu diễn nó bằng $\mathbf{x} \in \mathbb{R}^n$. Độ dài của một vector còn được gọi là số **chiều** của vector.

Cũng giống như một dãy thông thường trong Python, chúng ta có thể xem độ dài của một ndarray bằng cách gọi hàm `len()` có sẵn của Python.

```
len(x)
```

```
4
```

Khi một ndarray biểu diễn một vector (với chính xác một trục), ta cũng có thể xem độ dài của nó qua thuộc tính `.shape` (kích thước). Kích thước là một tuple liệt kê độ dài (số chiều) đọc theo mỗi trục của ndarray. Với các ndarray có duy nhất một trục, kích thước của nó chỉ có một phần tử.

```
x.shape
```

```
(4, )
```

Ở đây cần lưu ý rằng, từ “chiều” là một từ đa nghĩa và khi đặt vào nhiều ngữ cảnh thường dễ làm ta bị nhầm lẫn. Để làm rõ, chúng ta dùng số chiều của một *vector* hoặc của một *trục* để chỉ độ dài của nó, tức là số phần tử trong một vector hay một trục. Tuy nhiên, chúng ta sử dụng số chiều của một ndarray để chỉ số trục của ndarray đó. Theo nghĩa này, chiều của một trục của một ndarray là độ dài của trục đó.

5.3.3 Ma trận

Giống như vector khái quát số vô hướng từ bậc 0 sang bậc 1, ma trận sẽ khái quát những vector từ bậc 1 sang bậc 2. Ma trận thường được ký hiệu với ký tự hoa và được in đậm (ví dụ: **X**, **Y**, và **Z**); và được biểu diễn bằng các ndarray với 2 trục khi lập trình.

Trong ký hiệu toán học, ta dùng $\mathbf{A} \in \mathbb{R}^{m \times n}$ để biểu thị một ma trận \mathbf{A} gồm m hàng và n cột các giá trị số thực. Về mặt hình ảnh, ta có thể minh họa bất kỳ ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ như một bảng biểu mà mỗi phần tử a_{ij} nằm ở dòng thứ i và cột thứ j của bảng:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \quad (5.3.2)$$

Với bất kỳ ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ nào, kích thước của ma trận \mathbf{A} là (m, n) hay $m \times n$. Trong trường hợp đặc biệt, khi một ma trận có số dòng bằng số cột, dạng của nó là một hình vuông; như vậy, nó được gọi là một *ma trận vuông* (*square matrix*).

Ta có thể tạo một ma trận $m \times n$ trong MXNet bằng cách khai báo kích thước của nó với hai thành phần m và n khi sử dụng bất kỳ hàm khởi tạo ndarray nào mà ta thích.

```
A = np.arange(20).reshape(5, 4)
A
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]])
```

Ta có thể truy cập phần tử vô hướng a_{ij} của ma trận \mathbf{A} trong (??) bằng cách khai báo chỉ số dòng (i) và chỉ số cột (j), như là $[A]_{ij}$. Khi những thành phần vô hướng của ma trận \mathbf{A} , như trong (??) chưa được đưa ra, ta có

thể sử dụng ký tự viết thường của ma trận \mathbf{A} với các chỉ số ghi dưới, a_{ij} , để chỉ thành phần $[\mathbf{A}]_{ij}$. Nhằm giữ sự đơn giản cho các ký hiệu, dấu phẩy chỉ được thêm vào để phân tách các chỉ số khi cần thiết, như $a_{2,3j}$ và $[\mathbf{A}]_{2i-1,3}$.

Đôi khi, ta muốn hoán đổi các trục. Khi ta hoán đổi các dòng với các cột của ma trận, kết quả có được là *chuyển vị (transpose)* của ma trận đó. Về lý thuyết, chuyển vị của ma trận \mathbf{A} được ký hiệu là \mathbf{A}^\top và nếu $\mathbf{B} = \mathbf{A}^\top$ thì $b_{ij} = a_{ji}$ với mọi i và j . Do đó, chuyển vị của \mathbf{A} trong (??) là một ma trận $n \times m$:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \quad (5.3.3)$$

Trong mã nguồn, ta lấy chuyển vị của một ma trận thông qua thuộc tính T.

```
A.T
```

```
array([[ 0.,  4.,  8., 12., 16.],
       [ 1.,  5.,  9., 13., 17.],
       [ 2.,  6., 10., 14., 18.],
       [ 3.,  7., 11., 15., 19.]])
```

Là một biến thể đặc biệt của ma trận vuông, *ma trận đối xứng (symmetric matrix)* \mathbf{A} có chuyển vị bằng chính nó: $\mathbf{A} = \mathbf{A}^\top$.

```
B = np.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
```

```
array([[1., 2., 3.],
       [2., 0., 4.],
       [3., 4., 5.]])
```

```
B == B.T
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Ma trận là một cấu trúc dữ liệu hữu ích: chúng cho phép ta tổ chức dữ liệu có nhiều phương thức biến thể khác nhau. Ví dụ, các dòng trong ma trận của chúng ta có thể tương trưng cho các căn nhà khác nhau (các điểm dữ liệu), còn các cột có thể tương trưng cho những thuộc tính khác nhau của ngôi nhà. Bạn có thể thấy quen thuộc với điều này nếu đã từng sử dụng các phần mềm lập bảng tính hoặc đã đọc `sec_pandas`. Do đó, mặc dù một vector đơn lẻ có hướng mặc định là một vector cột, trong một ma trận biểu thị một tập dữ liệu bằng biểu, sẽ tốt hơn nếu ta xem mỗi điểm dữ liệu như một vector dòng trong ma trận. Chúng ta sẽ thấy ở những chương sau, quy ước này sẽ giúp dễ dàng áp dụng các kỹ thuật học sâu thông dụng. Ví dụ, với trục ngoài cùng của `ndarray`, ta có thể truy cập hay duyệt qua các batch nhỏ của những điểm dữ liệu hoặc chỉ đơn thuần là các điểm dữ liệu nếu không có batch nhỏ nào cả.

5.3.4 Tensor

Giống như vector khái quát hoá số vô hướng và ma trận khái quát hoá vector, ta có thể xây dựng những cấu trúc dữ liệu với thậm chí nhiều trục hơn. Tensor cho chúng ta một phương pháp tổng quát để miêu tả các ndarray với số trục bất kỳ. Ví dụ, vector là các tensor bậc một còn ma trận là các tensor bậc hai. Tensor được ký hiệu với ký tự viết hoa sử dụng một font chữ đặc biệt (ví dụ: X, Y, và Z) và có cơ chế truy vấn (ví dụ: x_{ijk} and $[X]_{1,2i-1,3}$) giống như ma trận.

Tensor sẽ trở nên quan trọng hơn khi ta bắt đầu làm việc với hình ảnh, thường được biểu diễn dưới dạng ndarray với 3 trục tương ứng với chiều cao, chiều rộng và một trục *kênh* (*channel*) để xếp chồng các kênh màu (đỏ, xanh lá và xanh dương). Tạm thời, ta sẽ bỏ qua các tensor bậc cao hơn và tập trung vào những điểm cơ bản trước.

```
X = np.arange(24).reshape(2, 3, 4)
X
```

```
array([[[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]],
      [[12., 13., 14., 15.],
       [16., 17., 18., 19.],
       [20., 21., 22., 23.]])
```

5.3.5 Các thuộc tính Cơ bản của Phép toán Tensor

Số vô hướng, vector, ma trận và tensor với một số trục bất kỳ có một vài thuộc tính rất hữu dụng. Ví dụ, bạn có thể để ý từ định nghĩa của phép toán theo từng phần tử (*elementwise*), bất kỳ phép toán theo từng phần tử một ngôi nào cũng không làm thay đổi kích thước của toán hạng của nó. Tương tự, cho hai tensor bất kỳ có cùng kích thước, kết quả của bất kỳ phép toán theo từng phần tử hai ngôi sẽ là một tensor có cùng kích thước. Ví dụ, cộng hai ma trận có cùng kích thước sẽ thực hiện phép cộng theo từng phần tử giữa hai ma trận này.

```
A = np.arange(20).reshape(5, 4)
B = A.copy() # Assign a copy of A to B by allocating new memory
A, A + B
```

```
(array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]]),
 array([[ 0.,  2.,  4.,  6.],
       [ 8., 10., 12., 14.],
       [16., 18., 20., 22.],
       [24., 26., 28., 30.],
       [32., 34., 36., 38.]]))
```

Đặc biệt, phép nhân theo phần tử của hai ma trận được gọi là *phép nhân Hadamard* (*Hadamard product* – ký hiệu toán học là \odot). Xét ma trận $\mathbf{B} \in \mathbb{R}^{m \times n}$ có phần tử dòng i và cột j là b_{ij} . Phép nhân Hadamard giữa ma

trận **A** (khai báo ở (??)) và **B** là

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \quad (5.3.4)$$

```
A * B
```

```
array([[ 0.,  1.,  4.,  9.],
       [ 16., 25., 36., 49.],
       [ 64., 81., 100., 121.],
       [144., 169., 196., 225.],
       [256., 289., 324., 361.]])
```

Nhân hoặc cộng một tensor với một số vô hướng cũng sẽ không thay đổi kích thước của tensor, mỗi phần tử của tensor sẽ được cộng hoặc nhân cho số vô hướng đó.

```
a = 2
X = np.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(array([[[ 2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9.],
         [10., 11., 12., 13.]],
        [[[14., 15., 16., 17.],
          [18., 19., 20., 21.],
          [22., 23., 24., 25.]]]),
       (2, 3, 4))
```

5.3.6 Rút gọn

Một phép toán hữu ích mà ta có thể thực hiện trên bất kỳ tensor nào là phép tính tổng các phần tử của nó. Ký hiệu toán học của phép tính tổng là \sum . Ta biểu diễn phép tính tổng các phần tử của một vector x với độ dài d dưới dạng $\sum_{i=1}^d x_i$. Trong mã nguồn, ta chỉ cần gọi hàm `sum`.

```
x = np.arange(4)
x, x.sum()
```

```
(array([0., 1., 2., 3.]), array(6.))
```

Ta có thể biểu diễn phép tính tổng các phần tử của tensor có kích thước tùy ý. Ví dụ, tổng các phần tử của một ma trận $m \times n$ có thể được viết là $\sum_{i=1}^m \sum_{j=1}^n a_{ij}$.

```
A.shape, A.sum()
```

```
((5, 4), array(190.))
```

Theo mặc định, hàm `sum` sẽ rút gọn tensor dọc theo tất cả các trục của nó và trả về kết quả là một số vô hướng. Ta cũng có thể chỉ định các trục được rút gọn bằng phép tổng. Lấy ma trận làm ví dụ, để rút gọn theo chiều hàng (trục 0) bằng việc tính tổng tất cả các hàng, ta đặt `axis=0` khi gọi hàm `sum`.

```
A_sum_axis0 = A.sum(axis=0)  
A_sum_axis0, A_sum_axis0.shape
```

```
(array([40., 45., 50., 55.]), (4,))
```

Việc đặt `axis=1` sẽ rút gọn theo cột (trục 1) bằng việc tính tổng tất cả các cột. Do đó, kích thước trục 1 của đầu vào sẽ không còn trong kích thước của đầu ra.

```
A_sum_axis1 = A.sum(axis=1)  
A_sum_axis1, A_sum_axis1.shape
```

```
(array([ 6., 22., 38., 54., 70.]), (5,))
```

Việc rút gọn ma trận dọc theo cả hàng và cột bằng phép tổng tương đương với việc cộng tất cả các phần tử trong ma trận đó lại.

```
A.sum(axis=[0, 1]) # Same as A.sum()
```

```
array(190.)
```

Một đại lượng liên quan là *trung bình cộng*. Ta tính trung bình cộng bằng cách chia tổng các phần tử cho số lượng phần tử. Trong mã nguồn, ta chỉ cần gọi hàm `mean` với đầu vào là các tensor có kích thước tùy ý.

```
A.mean(), A.sum() / A.size
```

```
(array(9.5), array(9.5))
```

Giống như `sum`, hàm `mean` cũng có thể rút gọn tensor dọc theo các trục được chỉ định.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(array([ 8., 9., 10., 11.]), array([ 8., 9., 10., 11.]))
```

Tổng không rút gọn

Tuy nhiên, việc giữ lại số các trục đôi khi là cần thiết khi gọi hàm `sum` hoặc `mean`, bằng cách đặt `keepdims=True`.

```
sum_A = A.sum(axis=1, keepdims=True)  
sum_A
```

```
array([[ 6.,
       [22.],
       [38.],
       [54.],
       [70.]])
```

Ví dụ, vì `sum_A` vẫn giữ lại 2 trục sau khi tính tổng của mỗi hàng, chúng ta có thể chia `A` cho `sum_A` thông qua cơ chế lan truyền.

```
A / sum_A
```

```
array([[ 0.          ,  0.16666667,  0.33333334,  0.5         ],
       [0.18181819,  0.22727273,  0.27272728,  0.3181818 ],
       [0.21052632,  0.23684211,  0.2631579 ,  0.28947368],
       [0.22222222,  0.24074075,  0.25925925,  0.27777778],
       [0.22857143,  0.24285714,  0.25714287,  0.27142859]])
```

Nếu chúng ta muốn tính tổng tích lũy các phần tử của `A` dọc theo các trục, giả sử `axis=0` (từng hàng một), ta có thể gọi hàm `cumsum`. Hàm này không rút gọn chiều của tensor đầu vào theo bất cứ trục nào.

```
A.cumsum(axis=0)
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  6.,  8., 10.],
       [12., 15., 18., 21.],
       [24., 28., 32., 36.],
       [40., 45., 50., 55.]])
```

5.3.7 Tích vô hướng

Cho đến giờ, chúng ta mới chỉ thực hiện những phép tính từng phần tử tương ứng, như tổng và trung bình. Nếu đây là tất những gì chúng ta có thể làm, đại số tuyến tính có lẽ không xứng đáng để có nguyên một mục. Tuy nhiên, một trong những phép tính căn bản nhất của đại số tuyến tính là tích vô hướng. Với hai vector $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ cho trước, *tích vô hướng* (*dot product*) $\mathbf{x}^\top \mathbf{y}$ (hoặc $\langle \mathbf{x}, \mathbf{y} \rangle$) là tổng các tích của những phần tử có cùng vị trí: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^d x_i y_i$.

```
y = np.ones(4)
x, y, np.dot(x, y)
```

```
(array([0., 1., 2., 3.]), array([1., 1., 1., 1.]), array(6.))
```

Lưu ý rằng chúng ta có thể thể hiện tích vô hướng của hai vector một cách tương tự bằng việc thực hiện tích từng phần tử tương ứng rồi lấy tổng:

```
np.sum(x * y)
```

```
array(6.)
```

Tích vô hướng sẽ hữu dụng trong rất nhiều trường hợp. Ví dụ, với một tập các giá trị cho trước, biểu thị bởi vector $\mathbf{x} \in \mathbb{R}^d$, và một tập các trọng số được biểu thị bởi $\mathbf{w} \in \mathbb{R}^d$, tổng trọng số của các giá trị trong \mathbf{x} theo các trọng số trong \mathbf{w} có thể được thể hiện bởi tích vô hướng $\mathbf{x}^\top \mathbf{w}$. Khi các trọng số không âm và có tổng bằng một ($\sum_{i=1}^d w_i = 1$), tích vô hướng thể hiện phép tính *trung bình trọng số* (*weighted average*). Sau khi được chuẩn hoá thành hai vector đơn vị, tích vô hướng của hai vector đó là giá trị cos của góc giữa hai vector đó. Chúng tôi sẽ giới thiệu khái niệm về *độ dài* ở các phần sau trong mục này.

5.3.8 Tích giữa Ma trận và Vector

Giờ đây, khi đã biết cách tính toán tích vô hướng, chúng ta có thể bắt đầu hiểu *tích giữa ma trận và vector*. Bạn có thể xem lại cách ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ và vector $\mathbf{x} \in \mathbb{R}^n$ được định nghĩa và biểu diễn trong (??) và (??). Ta sẽ bắt đầu bằng việc biểu diễn ma trận \mathbf{A} qua các vector hàng của nó.

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \quad (5.3.5)$$

Mỗi $\mathbf{a}_i^\top \in \mathbb{R}^n$ là một vector hàng thể hiện hàng thứ i của ma trận \mathbf{A} . Tích giữa ma trận và vector \mathbf{Ax} đơn giản chỉ là một vector cột với chiều dài m , với phần tử thứ i là kết quả của phép tích vô hướng $\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \quad (5.3.6)$$

Chúng ta có thể nghĩ đến việc nhân một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ với một vector như một phép biến hình, chiếu vector từ không gian \mathbb{R}^n thành \mathbb{R}^m . Những phép biến hình này hóa ra lại trở nên rất hữu dụng. Ví dụ, chúng ta có thể biểu diễn phép xoay là tích với một ma trận vuông. Bạn sẽ thấy ở những chương tiếp theo, chúng ta cũng có thể sử dụng tích giữa ma trận và vector để thực hiện hầu hết những tính toán cần thiết khi tính các tầng trong một mạng nơ-ron dựa theo kết quả của tầng trước đó.

Khi lập trình, để thực hiện nhân ma trận với vector ndarray, chúng ta cũng sử dụng hàm `dot` giống như tích vô hướng. Việc gọi `np.dot(A, x)` với ma trận \mathbf{A} và một vector \mathbf{x} sẽ thực hiện phép nhân vô hướng giữa ma trận và vector. Lưu ý rằng chiều của cột \mathbf{A} (chiều dài theo trục 1) phải bằng với chiều của vector \mathbf{x} (chiều dài của nó).

```
A.shape, x.shape, np.dot(A, x)
```

```
((5, 4), (4,), array([ 14., 38., 62., 86., 110.]))
```

5.3.9 Phép nhân Ma trận

Nếu bạn đã quen với tích vô hướng và tích ma trận-vector, tích *ma trận-ma trận* cũng tương tự như thế.

Giả sử ta có hai ma trận $\mathbf{A} \in \mathbb{R}^{n \times k}$ và $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \quad (5.3.7)$$

Đặt $\mathbf{a}_i^\top \in \mathbb{R}^k$ là vector hàng biểu diễn hàng thứ i của ma trận \mathbf{A} và $\mathbf{b}_j \in \mathbb{R}^k$ là vector cột thứ j của ma trận \mathbf{B} . Để tính ma trận tích $\mathbf{C} = \mathbf{AB}$, cách đơn giản nhất là viết các hàng của ma trận \mathbf{A} và các cột của ma trận \mathbf{B} :

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_m]. \quad (5.3.8)$$

Khi đó ma trận tích $\mathbf{C} \in \mathbb{R}^{n \times m}$ được tạo với phần tử c_{ij} bằng tích vô hướng $\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_m] = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \quad (5.3.9)$$

Ta có thể coi tích hai ma trận \mathbf{AB} như việc tính m phép nhân ma trận và vector, sau đó ghép các kết quả với nhau để tạo ra một ma trận $n \times m$. Giống như tích vô hướng và phép nhân ma trận-vector, ta có thể tính phép nhân hai ma trận bằng cách sử dụng hàm `dot`. Trong đoạn mã dưới đây, chúng ta tính phép nhân giữa \mathbf{A} và \mathbf{B} . Ở đây, \mathbf{A} là một ma trận với 5 hàng 4 cột và \mathbf{B} là một ma trận với 4 hàng 3 cột. Sau phép nhân này, ta thu được một ma trận với 5 hàng 3 cột.

```
B = np.ones(shape=(4, 3))
np.dot(A, B)
```

```
array([[ 6.,  6.,  6.],
       [22., 22., 22.],
       [38., 38., 38.],
       [54., 54., 54.],
       [70., 70., 70.]])
```

Phép nhân hai ma trận có thể được gọi đơn giản là *phép nhân ma trận* và không nên nhầm lẫn với phép nhân Hadamard.

5.3.10 Chuẩn

Một trong những toán tử hữu dụng nhất của đại số tuyến tính là *chuẩn* (*norm*). Nói dân dã thì, các chuẩn của một vector cho ta biết một vector *lớn* tầm nào. Thuật ngữ *kích thước* đang xét ở đây không nói tới số chiều không gian mà đúng hơn là về độ lớn của các thành phần.

Trong đại số tuyến tính, chuẩn của một vector là hàm số f ánh xạ một vector đến một số vô hướng, thỏa mãn các tính chất sau. Cho vector \mathbf{x} bất kỳ, tính chất đầu tiên phát biểu rằng nếu chúng ta co giãn toàn bộ các phần tử của một vector bằng một hằng số α , chuẩn của vector đó cũng co giãn theo *giá trị tuyệt đối* của hằng số đó:

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \quad (5.3.10)$$

Tính chất thứ hai cũng giống như bất đẳng thức tam giác:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \quad (5.3.11)$$

Tính chất thứ ba phát biểu rằng chuẩn phải không âm:

$$f(\mathbf{x}) \geq 0. \quad (5.3.12)$$

Điều này là hợp lý vì trong hầu hết các trường hợp thì *kích thước* nhỏ nhất cho các vật đều bằng 0. Tính chất cuối cùng yêu cầu chuẩn nhỏ nhất thu được khi và chỉ khi toàn bộ thành phần của vector đó bằng 0.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \quad (5.3.13)$$

Bạn chắc sẽ để ý là các chuẩn có vẻ giống như một phép đo khoảng cách. Và nếu còn nhớ khái niệm khoảng cách Euclid (định lý Pythagoras) được học ở phổ thông, thì khái niệm không âm và bất đẳng thức tam giác có thể gợi nhắc lại một chút. Thực tế là, khoảng cách Euclid cũng là một chuẩn: cụ thể là ℓ_2 . Giả sử rằng các thành phần trong vector n chiều \mathbf{x} là x_1, \dots, x_n . Chuẩn ℓ_2 của \mathbf{x} là căn bậc hai của tổng các bình phương của các thành phần trong vector:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \quad (5.3.14)$$

Ở đó, chỉ số dưới 2 thường được lược đi khi viết chuẩn ℓ_2 , ví dụ, $\|\mathbf{x}\|$ cũng tương đương với $\|\mathbf{x}\|_2$. Khi lập trình, ta có thể tính chuẩn ℓ_2 của một vector bằng cách gọi hàm `linalg.norm`.

```
u = np.array([3, -4])
np.linalg.norm(u)
```

```
array(5.)
```

Trong học sâu, chúng ta thường gặp chuẩn ℓ_2 bình phương hơn. Bạn cũng sẽ thường xuyên gặp chuẩn ℓ_1 , chuẩn được biểu diễn bằng tổng các giá trị tuyệt đối của các thành phần trong vector:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|. \quad (5.3.15)$$

So với chuẩn ℓ_2 , nó ít bị ảnh hưởng bởi các giá trị ngoại biên hơn. Để tính chuẩn ℓ_1 , chúng ta dùng hàm giá trị tuyệt đối rồi lấy tổng các thành phần.

```
np.abs(u).sum()
```

```
array(7.)
```

Cả hai chuẩn ℓ_2 và ℓ_1 đều là trường hợp riêng của một chuẩn tổng quát hơn, *chuẩn* ℓ_p :

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}. \quad (5.3.16)$$

Tương tự với chuẩn ℓ_2 của vector, *chuẩn Frobenius* của một ma trận $\mathbf{X} \in \mathbb{R}^{m \times n}$ là căn bậc hai của tổng các bình phương của các thành phần trong ma trận:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}. \quad (5.3.17)$$

Chuẩn Frobenius thỏa mãn tất cả các tính chất của một chuẩn vector. Nó giống chuẩn ℓ_2 của một vector nhưng ở dạng của ma trận. Ta dùng hàm `linalg.norm` để tính toán chuẩn Frobenius của ma trận.

```
np.linalg.norm(np.ones((4, 9)))
```

```
array(6.)
```

Chuẩn và Mục tiêu

Tuy không muốn đi quá nhanh nhưng chúng ta có thể xây dựng phần nào trực giác để hiểu tại sao những khái niệm này lại hữu dụng. Trong học sâu, ta thường cố giải các bài toán tối ưu: *tối đa hóa* xác suất xảy ra của dữ liệu quan sát được; *tối thiểu hóa* khoảng cách giữa dự đoán và nhãn gốc. Gán các biểu diễn vector cho các đối tượng (như từ, sản phẩm hay các bài báo) để tối thiểu khoảng cách giữa các đối tượng tương tự nhau và tối đa khoảng cách giữa các đối tượng khác nhau. Mục tiêu, thành phần quan trọng nhất của một thuật toán học sâu (bên cạnh dữ liệu), thường được biểu diễn theo *chuẩn* (*norm*).

5.3.11 Bàn thêm về Đại số Tuyến tính

Chỉ trong mục này, chúng tôi đã trang bị cho bạn tất cả những kiến thức đại số tuyến tính cần thiết để hiểu một lượng lớn các mô hình học máy hiện đại. Vẫn còn rất nhiều kiến thức đại số tuyến tính, phần lớn đều hữu dụng cho học máy. Một ví dụ là phép phân tích ma trận ra các thành phần, các phép phân tích này có thể tạo ra các cấu trúc thấp chiều trong các tập dữ liệu thực tế. Có cả một nhánh của học máy tập trung vào sử dụng các phép phân tích ma trận và tổng quát chúng lên cho các tensor bậc cao để khám phá cấu trúc trong các tập dữ liệu và giải quyết các bài toán dự đoán. Tuy nhiên, cuốn sách này chỉ tập trung vào học sâu. Và chúng tôi tin rằng bạn sẽ muốn học thêm nhiều về toán một khi đã có thể triển khai được các mô hình học máy hữu dụng cho các tập dữ liệu thực tế. Bởi vậy, trong khi vẫn còn nhiều kiến thức toán cần bàn thêm ở phần sau, chúng tôi sẽ kết thúc mục này ở đây.

Nếu bạn muốn học thêm về đại số tuyến tính, bạn có thể tham khảo `sec_geometry-linear-algebraic-ops` hoặc các nguồn tài liệu xuất sắc tại (???).

5.3.12 Tóm tắt

- Số vô hướng, vector, ma trận, và tensor là các đối tượng toán học cơ bản trong đại số tuyến tính.
- Vector là dạng tổng quát của số vô hướng và ma trận là dạng tổng quát của vector.
- Trong cách biểu diễn ndarray, các số vô hướng, vector, ma trận và tensor lần lượt có 0, 1, 2 và một số lượng tùy ý các trục.
- Một tensor có thể thu gọn theo một số trục bằng sum và mean.
- Phép nhân theo từng phần tử của hai ma trận được gọi là tích Hadamard của chúng. Phép toán này khác với phép nhân ma trận.
- Trong học sâu, chúng ta thường làm việc với các chuẩn như chuẩn ℓ_1 , chuẩn ℓ_2 và chuẩn Frobenius.
- Chúng ta có thể thực hiện một số lượng lớn các toán tử trên số vô hướng, vector, ma trận và tensor với các hàm của ndarray.

5.3.13 Bài tập

1. Chứng minh rằng chuyển vị của một ma trận chuyển vị là chính nó: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Cho hai ma trận \mathbf{A} và \mathbf{B} , chứng minh rằng tổng của chuyển vị bằng chuyển vị của tổng: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Cho một ma trận vuông \mathbf{A} , liệu rằng $\mathbf{A} + \mathbf{A}^\top$ có luôn đổi xứng? Tại sao?
4. Chúng ta đã định nghĩa tensor X với kích thước (2, 3, 4) trong mục này. Kết quả của len(X) là gì?
5. Cho một tensor X với kích thước bất kỳ, liệu len(X) có luôn tương ứng với độ dài của một trục nhất định của X hay không? Đó là trục nào?
6. Chạy `A / A.sum(axis=1)` và xem điều gì xảy ra. Bạn có phân tích được nguyên nhân không?
7. Khi di chuyển giữa hai điểm ở Manhattan (đường phố hình bàn cờ), khoảng cách tính bằng tọa độ (tức độ dài các đại lộ và phố) mà bạn cần di chuyển là bao nhiêu? Bạn có thể đi theo đường chéo không? (Xem thêm bản đồ Manhattan, New York để trả lời câu hỏi này)
8. Xét một tensor với kích thước (2, 3, 4). Kích thước của kết quả sau khi tính tổng theo trục 0, 1 và 2 sẽ như thế nào?
9. Đưa một tensor với 3 trục hoặc hơn vào hàm `linalg.norm` và quan sát kết quả. Hàm này thực hiện việc gì cho các ndarray với kích thước bất kỳ?

5.3.14 Thảo luận

- Tiếng Anh⁶⁰
- Tiếng Việt⁶¹

⁶⁰ <https://discuss.mxnet.io/t/2317>

⁶¹ <https://forum.machinelearningcoban.com/c/d2l>

5.3.15 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Ngô Thế Anh Khoa
- Nguyễn Lê Quang Nhật
- Vũ Hữu Tiệp
- Mai Sơn Hải
- Phạm Hồng Vinh

5.4 Giải tích

Tìm diện tích của một đa giác vẫn là một bí ẩn cho tới ít nhất 2.500 năm trước, khi người Hy Lạp cổ đại chia đa giác thành các tam giác và cộng diện tích của chúng lại. Để tìm diện tích của các hình cong, như hình tròn, người Hy Lạp cổ đại đặt các đa giác nội tiếp bên trong các hình cong đó. Như trong `fig_circle_area`, một đa giác nội tiếp với càng nhiều cạnh bằng nhau thì càng xấp xỉ đúng hình tròn. Quy trình này còn được biết đến như *phương pháp vét kiệt*.

Fig. 5.4.1: Tìm diện tích hình tròn bằng phương pháp vét kiệt.

Phương pháp vét kiệt chính là khởi nguồn của *giải tích tích phân* (sẽ được miêu tả trong `sec_integral_calculus`). Hơn 2.000 năm sau, nhánh còn lại của giải tích, *giải tích vi phân*, ra đời. Trong những ứng dụng quan trọng nhất của giải tích vi phân, các bài toán tối ưu hoá sẽ tìm *cách tốt nhất* để thực hiện một công việc nào đó. Như đã bàn đến trong `subsec_norms_and_objectives`, các bài toán như vậy vô cùng phổ biến trong học sâu.

Trong học sâu, chúng ta *huấn luyện* các mô hình, cập nhật chúng liên tục để chúng ngày càng tốt hơn khi học với nhiều dữ liệu hơn. Thông thường, trở nên tốt hơn tương đương với tối thiểu hoá một *hàm mất mát*, một điểm số sẽ trả lời câu hỏi “mô hình của ta đang tệ tới mức nào?” Câu hỏi này lắt léo hơn ta tưởng nhiều. Mục đích cuối cùng mà ta muốn là mô hình sẽ hoạt động tốt trên dữ liệu mà nó chưa từng nhìn thấy. Nhưng chúng ta chỉ có thể khớp mô hình trên dữ liệu mà ta đang có thể thấy. Do đó ta có thể chia việc huấn luyện mô hình thành hai vấn đề chính: i) *tối ưu hoá*: quy trình huấn luyện mô hình trên dữ liệu đã thấy. ii) *tổng quát hoá*: dựa trên các nguyên tắc toán học và sự uyên thâm của người huấn luyện để tạo ra các mô hình mà tính hiệu quả của nó vượt ra khỏi tập dữ liệu huấn luyện.

Để giúp bạn hiểu các bài toán tối ưu hóa và các phương pháp tối ưu hóa trong các chương sau, ở đây chúng tôi sẽ cung cấp một chương ngắn vỡ lòng về các kỹ thuật giải tích vi phân thông dụng trong học sâu.

5.4.1 Đạo hàm và Vi phân

Chúng ta bắt đầu bằng việc đề cập tới khái niệm đạo hàm, một bước quan trọng của hầu hết các thuật toán tối ưu trong học sâu. Trong học sâu, ta thường chọn những hàm mượt mà khả vi theo các tham số của mô hình. Nói đơn giản, với mỗi tham số, ta có thể xác định hàm mượt tăng hoặc giảm nhanh như thế nào khi tham số đó *tăng* hoặc *giảm* chỉ một lượng cực nhỏ.

Giả sử ta có một hàm $f : \mathbb{R} \rightarrow \mathbb{R}$ có đầu vào và đầu ra đều là số vô hướng. *Đạo hàm* của f được định nghĩa là

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \quad (5.4.1)$$

nếu giới hạn này tồn tại. Nếu $f'(a)$ tồn tại, f được gọi là *khả vi (differentiable)* tại a . Nếu f khả vi tại mọi điểm trong một khoảng, thì hàm này được gọi là khả vi trong khoảng đó. Ta có thể giải nghĩa đạo hàm $f'(x)$ trong (??) như là tốc độ thay đổi *tức thời* của hàm f theo biến x . Cái gọi là tốc độ thay đổi tức thời được dựa trên độ biến thiên h trong x khi h tiến về 0.

Để minh họa cho khái niệm đạo hàm, hãy thử với một ví dụ. Định nghĩa $u = f(x) = 3x^2 - 4x$.

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

def f(x):
    return 3 * x ** 2 - 4 * x
```

Cho $x = 1$ và h tiến về 0, kết quả của phương trình $\frac{f(x+h)-f(x)}{h}$ trong (??) tiến về 2. Dù thử nghiệm này không phải là một dạng chứng minh toán học, lát nữa ta sẽ thấy rằng quả thật đạo hàm của u' là 2 khi $x = 1$.

```
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
    print('h=% .5f, numerical limit=% .5f' % (h, numerical_lim(f, 1, h)))
    h *= 0.1
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Hãy làm quen với một vài ký hiệu cùng được dùng để biểu diễn đạo hàm. Cho $y = f(x)$ với x và y lần lượt là biến độc lập và biến phụ thuộc của hàm f . Những biểu diễn sau đây là tương đương nhau:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = D_x f(x), \quad (5.4.2)$$

với ký hiệu $\frac{d}{dx}$ và D là các *toán tử vi phân (differentiation operator)* để chỉ các phép toán *vi phân*. Ta có thể sử dụng các quy tắc lấy đạo hàm của các hàm thông dụng sau đây:

- $DC = 0$ (C là một hằng số),

- $Dx^n = nx^{n-1}$ (quy tắc số mũ, n là số thực bất kỳ),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

Để lấy đạo hàm của một hàm được tạo từ vài hàm đơn giản hơn, ví dụ như từ những hàm thông dụng ở trên, có thể dùng các quy tắc hữu dụng dưới đây. Giả sử hàm f và g đều khả vi và C là một hằng số, ta có *quy tắc nhân hằng số*

$$\frac{d}{dx}[Cf(x)] = C \frac{d}{dx}f(x), \quad (5.4.3)$$

quy tắc tổng

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \quad (5.4.4)$$

quy tắc nhân

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \quad (5.4.5)$$

và *quy tắc đạo hàm phân thức*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \quad (5.4.6)$$

Bây giờ ta có thể áp dụng các quy tắc ở trên để tìm đạo hàm $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$. Vậy nên, với $x = 1$, ta có $u' = 2$: điều này đã được kiểm chứng với thử nghiệm lúc trước ở mục này khi kết quả có được cũng tiến tới 2. Giá trị đạo hàm này cũng đồng thời là độ dốc của đường tiếp tuyến với đường cong $u = f(x)$ tại $x = 1$.

Để minh họa cách hiểu này của đạo hàm, ta sẽ dùng `matplotlib`, một thư viện vẽ biểu đồ thông dụng trong Python. Ta cần định nghĩa một số hàm để cấu hình thuộc tính của các biểu đồ được tạo ra bởi `matplotlib`. Trong đoạn mã sau, hàm `use_svg_display` chỉ định `matplotlib` tạo các biểu đồ ở dạng `svg` để có được chất lượng ảnh sắc nét hơn.

```
# Saved in the d2l package for later use
def use_svg_display():
    """Use the svg format to display a plot in Jupyter."""
    display.set_matplotlib_formats('svg')
```

Ta định nghĩa hàm `set_figsize` để chỉ định kích thước của biểu đồ. Lưu ý rằng ở đây ta đang dùng trực tiếp `d2l=plt` do câu lệnh `from matplotlib import pyplot as plt` đã được đánh dấu để lưu vào gói `d2l` trong phần Lời nói đầu.

```
# Saved in the d2l package for later use
def set_figsize(figsize=(3.5, 2.5)):
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

Hàm `set_axes` sau cấu hình thuộc tính của các trục biểu đồ tạo bởi `matplotlib`.

```

# Saved in the d2l package for later use
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()

```

Với ba hàm cấu hình biểu đồ trên, ta định nghĩa hàm plot để vẽ nhiều đồ thị một cách nhanh chóng vì ta sẽ cần minh họa khá nhiều đồ thị xuyên suốt cuốn sách.

```

# Saved in the d2l package for later use
def plot(X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=['-', '--', '-.', ':'], figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    d2l.set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if X (ndarray or list) has 1 axis
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)

```

Giờ ta có thể vẽ đồ thị của hàm số $u = f(x)$ và đường tiếp tuyến của nó $y = 2x - 3$ tại $x = 1$, với hệ số 2 là độ dốc của tiếp tuyến.

```

x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])

```

5.4.2 Đạo hàm riêng

Cho tới giờ, ta đã làm việc với đạo hàm của các hàm một biến. Trong học sâu, các hàm lại thường phụ thuộc vào *nhiều biến*. Do đó, ta cần mở rộng ý tưởng của đạo hàm cho các hàm *nhiều biến* đó.

Cho $y = f(x_1, x_2, \dots, x_n)$ là một hàm với n biến. *Đạo hàm riêng* của y theo tham số thứ i , x_i , là

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}. \quad (5.4.7)$$

Để tính $\frac{\partial y}{\partial x_i}$, ta chỉ cần coi $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ là các hằng số và tính đạo hàm của y theo x_i . Để biểu diễn đạo hàm riêng, các ký hiệu sau đây đều có ý nghĩa tương đương:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \quad (5.4.8)$$

5.4.3 Gradient

Chúng ta có thể ghép các đạo hàm riêng của mọi biến trong một hàm nhiều biến để thu được vector *gradient* của hàm số đó. Giả sử rằng đầu vào của hàm $f : \mathbb{R}^n \rightarrow \mathbb{R}$ là một vector n chiều $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top$ và đầu ra là một số vô hướng. Gradient của hàm $f(\mathbf{x})$ theo \mathbf{x} là một vector gồm n đạo hàm riêng đó:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^\top. \quad (5.4.9)$$

Biểu thức $\nabla_{\mathbf{x}} f(\mathbf{x})$ thường được viết gọn thành $\nabla f(\mathbf{x})$ trong trường hợp không sợ nhầm lẫn.

Cho \mathbf{x} là một vector n -chiều, các quy tắc sau thường được dùng khi tính vi phân hàm đa biến:

- Với mọi $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
- Với mọi $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- Với mọi $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Tương tự, với bất kỳ ma trận \mathbf{X} nào, ta đều có $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$. Sau này ta sẽ thấy, gradient sẽ rất hữu ích khi thiết kế thuật toán tối ưu trong học sâu.

5.4.4 Quy tắc dây chuyền

Tuy nhiên, những gradient như thế có thể khó để tính toán. Đó là bởi vì các hàm nhiều biến trong học sâu đa phần là những *hàm hợp*, nên ta không thể áp dụng các quy tắc đề cập ở trên để lấy vi phân cho những hàm này. May mắn thay, *quy tắc dây chuyền* cho phép chúng ta lấy vi phân của các hàm hợp.

Trước tiên, chúng ta hãy xem xét các hàm một biến. Giả sử hai hàm $y = f(u)$ và $u = g(x)$ đều khả vi, quy tắc dây chuyền được mô tả như sau

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}. \quad (5.4.10)$$

Giờ ta sẽ xét trường hợp tổng quát hơn đối với các hàm nhiều biến. Giả sử một hàm khả vi y có các biến số u_1, u_2, \dots, u_m , trong đó mỗi biến u_i là một hàm khả vi của các biến x_1, x_2, \dots, x_n . Lưu ý rằng y cũng là hàm của các biến x_1, x_2, \dots, x_n . Quy tắc dây chuyền cho ta

$$\frac{dy}{dx_i} = \frac{dy}{du_1} \frac{du_1}{dx_i} + \frac{dy}{du_2} \frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m} \frac{du_m}{dx_i} \quad (5.4.11)$$

cho mỗi $i = 1, 2, \dots, n$.

5.4.5 Tóm tắt

- Vi phân và tích phân là hai nhánh con của giải tích, trong đó vi phân được ứng dụng rộng rãi trong các bài toán tối ưu hóa của học sâu.
- Đạo hàm có thể được hiểu như là tốc độ thay đổi tức thì của một hàm số đối với các biến số. Nó cũng là độ dốc của đường tiếp tuyến với đường cong của hàm.
- Gradient là một vector có các phần tử là đạo hàm riêng của một hàm nhiều biến theo tất cả các biến số của nó.
- Quy tắc dây chuyền cho phép chúng ta lấy vi phân của các hàm hợp.

5.4.6 Bài tập

dịch đoạn phía trên 1. Vẽ đồ thị của hàm số $y = f(x) = x^3 - \frac{1}{x}$ và đường tiếp tuyến của nó tại $x = 1$. 1. Tìm gradient của hàm số $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$. 1. Gradient của hàm $f(\mathbf{x}) = \|\mathbf{x}\|_2$ là gì? 1. Có thể dùng quy tắc dây chuyền cho trường hợp sau đây không: $u = f(x, y, z)$, với $x = x(a, b)$, $y = y(a, b)$ và $z = z(a, b)$?

5.4.7 Thảo luận

- Tiếng Anh⁶²
- Tiếng Việt⁶³

5.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Nguyễn Cảnh Thượng
- Phạm Minh Đức
- Tạ H. Duy Nguyên

⁶² <https://discuss.mxnet.io/t/5008>

⁶³ <https://forum.machinelearningcoban.com/c/d2l>

5.5 Tính vi phân Tự động

Như đã giải thích trong `sec_calculus`, vi phân là phép tính thiết yếu trong hầu như tất cả mọi thuật toán học sâu. Mặc dù các phép toán trong việc tính đạo hàm khá trực quan và chỉ yêu cầu một chút kiến thức giải tích, nhưng với các mô hình phức tạp, việc tự tính rõ ràng từng bước khá là mệt (và thường rất dễ sai).

Gói thư viện `autograd` giải quyết vấn đề này một cách nhanh chóng và hiệu quả bằng cách tự động hóa các phép tính đạo hàm (*automatic differentiation*). Trong khi nhiều thư viện yêu cầu ta phải biên dịch một *dồ thị biểu tượng* (*symbolic graph*) để có thể tự động tính đạo hàm, `autograd` cho phép ta tính đạo hàm ngay lập tức thông qua các dòng lệnh thông thường. Mỗi khi đưa dữ liệu chạy qua mô hình, `autograd` xây dựng một đồ thị và theo dõi xem dữ liệu nào kết hợp với các phép tính nào để tạo ra kết quả. Với đồ thị này `autograd` sau đó có thể lan truyền ngược gradient lại theo ý muốn. *Lan truyền ngược* ở đây chỉ đơn thuần là truy ngược lại *dồ thị tính toán* và điền vào đó các giá trị đạo hàm riêng theo từng tham số.

```
from mxnet import autograd, np, npx  
npx.set_np()
```

5.5.1 Một ví dụ đơn giản

Lấy ví dụ đơn giản, giả sử chúng ta muốn tính vi phân của hàm số $y = 2\mathbf{x}^\top \mathbf{x}$ theo vector cột \mathbf{x} . Để bắt đầu, ta sẽ tạo biến \mathbf{x} và gán cho nó một giá trị ban đầu.

```
x = np.arange(4)  
x  
  
array([0., 1., 2., 3.])
```

Lưu ý rằng trước khi có thể tính gradient của y theo \mathbf{x} , chúng ta cần một nơi để lưu giữ nó. Điều quan trọng là ta không được cấp phát thêm bộ nhớ mới mỗi khi tính đạo hàm theo một biến xác định, vì ta thường cập nhật cùng một tham số hàng ngàn vạn lần và sẽ nhanh chóng dùng hết bộ nhớ.

Cũng lưu ý rằng, bản thân giá trị gradient của hàm số đơn trị theo một vector \mathbf{x} cũng là một vector với cùng kích thước. Do vậy trong mã nguồn sẽ trực quan hơn nếu chúng ta lưu giá trị gradient tính theo \mathbf{x} dưới dạng một thuộc tính của chính `ndarray` \mathbf{x} . Chúng ta cấp bộ nhớ cho gradient của một `ndarray` bằng cách gọi phương thức `attach_grad`.

```
x.attach_grad()
```

Sau khi đã tính toán gradient theo biến \mathbf{x} , ta có thể truy cập nó thông qua thuộc tính `grad`. Để an toàn, \mathbf{x} .`grad` được khởi tạo là một mảng chứa các giá trị không. Điều này hợp lý vì trong học sâu, việc lấy gradient thường là để cập nhật các tham số bằng cách cộng (hoặc trừ) gradient của một hàm để tối đa (hoặc tối thiểu) hóa hàm đó. Bằng cách khởi tạo gradient bằng mảng chứa giá trị không, ta đảm bảo rằng bất kỳ cập nhật vô tình nào trước khi gradient được tính toán sẽ không làm thay đổi giá trị các tham số.

```
x.grad  
  
array([0., 0., 0., 0.])
```

Giờ hãy tính y . Bởi vì mục đích sau cùng là tính gradient, ta muốn MXNet tạo đồ thị tính toán một cách nhanh chóng. Ta có thể tưởng tượng rằng MXNet sẽ bật một thiết bị ghi hình để thu lại chính xác đường đi mà mỗi biến được tạo.

Chú ý rằng ta cần một số lượng phép tính không hề nhỏ để xây dựng đồ thị tính toán. Vậy nên MXNet sẽ chỉ dựng đồ thị khi được ra lệnh rõ ràng. Ta có thể thực hiện việc này bằng cách đặt đoạn mã trong phạm vi `autograd.record`.

```
with autograd.record():
    y = 2 * np.dot(x, x)
```

```
Y
```

```
array(28.)
```

Bởi vì x là một `ndarray` có độ dài bằng 4, `np.dot` sẽ tính toán tích vô hướng của x và x , trả về một số vô hướng mà sẽ được gán cho y . Tiếp theo, ta có thể tính toán gradient của y theo mỗi thành phần của x một cách tự động bằng cách gọi hàm `backward` của y .

```
y.backward()
```

Nếu kiểm tra lại giá trị của $x.grad$, ta sẽ thấy nó đã được ghi đè bằng gradient mới được tính toán.

```
x.grad
```

```
array([ 0.,  4.,  8., 12.])
```

Gradient của hàm $y = 2x^T x$ theo x phải là $4x$. Hãy kiểm tra một cách nhanh chóng rằng giá trị gradient mong muốn được tính toán đúng. Nếu hai `ndarray` là giống nhau, thì mọi cặp phần tử tương ứng cũng bằng nhau.

```
x.grad == 4 * x
```

```
array([ True,  True,  True,  True])
```

Nếu ta tiếp tục tính gradient của một biến khác mà giá trị của nó là kết quả của một hàm theo biến x , thì nội dung trong $x.grad$ sẽ bị ghi đè.

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad
```

```
array([1., 1., 1., 1.])
```

5.5.2 Truyền ngược cho các biến không phải Số vô hướng

Về mặt kỹ thuật, khi y không phải một số vô hướng, cách diễn giải tự nhiên nhất cho vi phân của một vector y theo vector x đó là một ma trận. Với các bậc và chiều cao hơn của y và x , kết quả của phép vi phân có thể là một tensor bậc cao.

Tuy nhiên, trong khi những đối tượng như trên xuất hiện trong học máy nâng cao (bao gồm học sâu), thường thì khi ta gọi lan truyền ngược trên một vector, ta đang cố tính toán đạo hàm của hàm mất mát theo mỗi *batch* bao gồm một vài mẫu huấn luyện. Ở đây, ý định của ta không phải là tính toán ma trận vi phân mà là tổng của các đạo hàm riêng được tính toán một cách độc lập cho mỗi mẫu trong batch.

Vậy nên khi ta gọi `backward` lên một biến vector y – là một hàm của x , MXNet sẽ cho rằng ta muốn tính tổng của các gradient. Nói ngắn gọn, MXNet sẽ tạo một biến mới có giá trị là số vô hướng bằng cách cộng lại các phần tử trong y và tính gradient theo x của biến mới này.

```
with autograd.record():
    y = x * x # y is a vector
    y.backward()

    u = x.copy()
    u.attach_grad()
    with autograd.record():
        v = (u * u).sum() # v is a scalar
        v.backward()

    x.grad == u.grad
```

```
array([ True,  True,  True,  True])
```

5.5.3 Tách rời Tính toán

Đôi khi chúng ta muốn chuyển một số phép tính ra khỏi đồ thị tính toán. Ví dụ, giả sử y đã được tính như một hàm của x , rồi sau đó z được tính như một hàm của cả y và x . Bây giờ, giả sử ta muốn tính gradient của z theo x , nhưng vì lý do nào đó ta lại muốn xem y như là một hằng số và chỉ xét đến vai trò của x như là biến số của z sau khi giá trị của y đã được tính.

Trong trường hợp này, ta có thể gọi $u = y.detach()$ để trả về một biến u mới có cùng giá trị như y nhưng không còn chứa các thông tin về cách mà y đã được tính trong đồ thị tính toán. Nói cách khác, gradient sẽ không thể chảy ngược qua u về x được. Bằng cách này, ta đã tính u như một hàm của x ở ngoài phạm vi của `autograd.record`, dẫn đến việc biến u sẽ được xem như là một hằng số mỗi khi ta gọi `backward`. Chính vì vậy, hàm `backward` sau đây sẽ tính đạo hàm riêng của $z = u * x$ theo x khi xem u như là một hằng số, thay vì đạo hàm riêng của $z = x * x * x$ theo x .

```
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
    z.backward()
    x.grad == u
```

```
array([ True,  True,  True,  True])
```

Bởi vì sự tính toán của y đã được ghi lại, chúng ta có thể gọi $y.backward()$ sau đó để lấy đạo hàm của $y = x * x$ theo x , tức là $2 * x$.

```
y.backward()  
x.grad == 2 * x
```

```
array([ True,  True,  True,  True])
```

Lưu ý rằng khi ta gắn gradient vào một biến x , $x = x.detach()$ sẽ được gọi ngầm. Nếu x được tính dựa trên các biến khác, phần tính toán này sẽ không được sử dụng trong hàm backward.

```
y = np.ones(4) * 2  
y.attach_grad()  
with autograd.record():  
    u = x * y  
    u.attach_grad() # Implicitly run u = u.detach()  
    z = 5 * u - x  
z.backward()  
x.grad, u.grad, y.grad
```

```
(array([-1., -1., -1., -1.]), array([5., 5., 5., 5.]), array([0., 0., 0., 0.  
→]))
```

5.5.4 Tính gradient của Luồng điều khiển Python

Một lợi thế của việc sử dụng vi phân tự động là khi việc xây dựng đồ thị tính toán đòi hỏi trải qua một loạt các câu lệnh điều khiển luồng Python, (ví dụ như câu lệnh điều kiện, vòng lặp và các lệnh gọi hàm tùy ý), ta vẫn có thể tính gradient của biến kết quả. Trong đoạn mã sau, hãy lưu ý rằng số lần lặp của vòng lặp while và kết quả của câu lệnh if đều phụ thuộc vào giá trị của đầu vào a .

```
def f(a):  
    b = a * 2  
    while np.linalg.norm(b) < 1000:  
        b = b * 2  
    if b.sum() > 0:  
        c = b  
    else:  
        c = 100 * b  
    return c
```

Một lần nữa, để tính gradient ta chỉ cần “ghi lại” các phép tính (bằng cách gọi hàm record) và sau đó gọi hàm backward.

```
a = np.random.normal()  
a.attach_grad()  
with autograd.record():  
    d = f(a)  
d.backward()
```

Giờ ta có thể phân tích hàm f được định nghĩa ở phía trên. Hãy để ý rằng hàm này tuyến tính từng khúc theo đầu vào a . Nói cách khác, với mọi giá trị của a tồn tại một hằng số k sao cho $f(a) = k * a$, ở đó giá trị của k phụ thuộc vào đầu vào a . Do đó, ta có thể kiểm tra giá trị của gradient bằng cách tính d / a .

```
a.grad == d / a
```

```
array (True)
```

5.5.5 Chế độ huấn luyện và Chế độ dự đoán

Như đã thấy, sau khi gọi `autograd.record`, MXNet sẽ ghi lại những tính toán xảy ra trong khối mã nguồn theo sau. Có một chi tiết tinh tế nữa mà ta cần để ý. `autograd.record` sẽ thay đổi chế độ chạy từ *chế độ dự đoán* sang *chế độ huấn luyện*. Ta có thể kiểm chứng hành vi này bằng cách gọi hàm `is_training`.

```
print(autograd.is_training())
with autograd.record():
    print(autograd.is_training())
```

```
False
True
```

Khi ta tìm hiểu tới các mô hình học sâu phức tạp, ta sẽ gặp một vài thuật toán mà mô hình hoạt động khác nhau khi huấn luyện và khi được sử dụng sau đó để dự đoán. Những khác biệt này sẽ được đề cập chi tiết trong các chương sau.

5.5.6 Tóm tắt

- MXNet cung cấp gói `autograd` để tự động hóa việc tính toán đạo hàm. Để sử dụng nó, đầu tiên ta gắn gradient cho các biến mà ta muốn lấy đạo hàm riêng theo nó. Sau đó ta ghi lại tính toán của giá trị mục tiêu, thực thi hàm `backward` của nó và truy cập kết quả gradient thông qua thuộc tính `grad` của các biến.
- Ta có thể tách rời gradient để kiểm soát những phần tính toán được sử dụng trong hàm `backward`.
- Các chế độ chạy của MXNet bao gồm chế độ huấn luyện và chế độ dự đoán. Ta có thể kiểm tra chế độ đang chạy bằng cách gọi hàm `is_training`.

5.5.7 Bài tập

- Tại sao đạo hàm bậc hai lại mất thêm rất nhiều tài nguyên để tính toán hơn đạo hàm bậc một?
- Sau khi chạy `y.backward()`, lập tức chạy lại lần nữa và xem chuyện gì sẽ xảy ra.
- Trong ví dụ về luồng điều khiển khi ta tính toán đạo hàm của `d` theo `a`, điều gì sẽ xảy ra nếu ta thay đổi biến `a` thành một vector hay ma trận ngẫu nhiên. Lúc này, kết quả của tính toán `f(a)` sẽ không còn là số vô hướng nữa. Điều gì sẽ xảy ra với kết quả? Ta có thể phân tích nó như thế nào?
- Hãy tái thiết kế một ví dụ về việc tìm gradient của luồng điều khiển. Chạy ví dụ và phân tích kết quả.
- Cho $f(x) = \sin(x)$. Vẽ đồ thị của $f(x)$ và $\frac{df(x)}{dx}$ với điều kiện không được tính trực tiếp đạo hàm $f'(x) = \cos(x)$.
- Trong một cuộc đấu giá kín theo giá thứ hai (ví dụ như trong eBay hay trong quảng cáo điện toán), người thắng cuộc đấu giá chỉ trả mức giá cao thứ hai. Hãy tính gradient của mức giá cuối cùng theo mức đặt

của người thắng cuộc bằng cách sử dụng autograd. Kết quả cho bạn biết điều gì về cơ chế đấu giá này? Nếu bạn tò mò muốn tìm hiểu thêm về các cuộc đấu giá kín theo giá thứ hai, hãy đọc bài báo nghiên cứu của Edelman et al. (?).

5.5.8 Thảo luận

- Tiếng Anh⁶⁴
- Tiếng Việt⁶⁵

5.5.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thượng
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Tạ H. Duy Nguyên
- Phạm Minh Đức

5.6 Xác suất

Theo cách này hay cách khác, học máy đơn thuần là đưa ra các dự đoán. Chúng ta có thể muốn dự đoán xác suất của một bệnh nhân có thể bị đau tim vào năm sau, khi đã biết tiền sử lâm sàng của họ. Trong tác vụ phát hiện điều bất thường, chúng ta có thể muốn đánh giá khả năng các thông số động cơ máy bay ở mức nào, liệu có ở mức hoạt động bình thường không. Trong học tăng cường, chúng ta muốn có một tác nhân hoạt động thông minh trong một môi trường. Nghĩa là chúng ta cần tính tới xác suất đạt điểm thưởng cao nhất cho từng hành động có thể thực hiện. Và khi xây dựng một hệ thống gợi ý chúng ta cũng cần quan tâm tới xác suất. Ví dụ, giả thiết rằng chúng ta làm việc cho một hãng bán sách trực tuyến lớn. Chúng ta có thể muốn ước lượng xác suất một khách hàng cụ thể muốn mua một cuốn sách cụ thể nào đó. Để làm được điều này, chúng ta cần dùng tới ngôn ngữ xác suất. Có những khóa học, chuyên ngành, luận văn, sự nghiệp, và cả các ban ngành đều dành toàn bộ cho xác suất. Vì thế đương nhiên mục tiêu của chúng tôi trong chương này không phải để dạy toàn bộ môn xác suất. Thay vào đó, chúng tôi hy vọng đưa tới cho bạn đọc các kiến thức nền tảng, đủ để bạn đọc có thể bắt đầu xây dựng mô hình học sâu đầu tiên của chính mình, và truyền cảm hứng cho bạn thêm yêu thích xác suất để có thể bắt đầu tự khám phá nếu muốn.

Chúng tôi đã nhắc tới xác suất trong các chương trước mà không nói rõ chính xác nó là gì hay là đưa ra một ví dụ cụ thể nào. Giờ hãy cùng bắt đầu nghiêm túc hơn bằng cách xem xét trường hợp đầu tiên: phân biệt mèo và chó dựa trên các bức ảnh. Điều này tưởng chừng đơn giản nhưng thực ra là một thách thức. Để bắt đầu, độ phức tạp của bài toán này có thể phụ thuộc vào độ phân giải của ảnh.

⁶⁴ <https://discuss.mxnet.io/t/2318>

⁶⁵ <https://forum.machinelearningcoban.com/c/d2l>



Fig. 5.6.1: Ảnh ở các độ phân giải khác nhau (10×10 , 20×20 , 40×40 , 80×80 , and 160×160 điểm ảnh).

Như thể hiện trong `fig_cat_dog`, con người phân biệt mèo và chó dễ dàng ở độ phân giải 160×160 điểm ảnh, có chút thử thách hơn ở 40×40 điểm ảnh, và gần như không thể ở 10×10 điểm ảnh. Nói cách khác, khả năng phân biệt mèo và chó của chúng ta ở khoảng cách càng xa (đồng nghĩa với độ phân giải thấp) càng giống đoán mò. Xác suất trang bị cho ta một cách suy luận hìn thức về mức độ chắc chắn. Nếu chúng ta hoàn toàn chắc chắn rằng bức ảnh mô tả một con mèo, ta có thể nói rằng xác suất nhãn tương ứng y là “mèo”, ký hiệu là $P(y = \text{“mèo”})$ equals 1. Nếu chúng ta không có manh mối nào để đoán rằng $y = \text{“mèo”}$ hoặc $y = \text{“chó”}$, thì ta có thể nói rằng hai xác suất này có *khả năng* bằng nhau, biểu diễn bởi $P(y = \text{“mèo”}) = P(y = \text{“chó”}) = 0.5$. Nếu ta khá tự tin, nhưng không thực sự chắc chắn bức ảnh mô tả một con mèo, ta có thể gán cho nó một xác suất $0.5 < P(y = \text{“mèo”}) < 1$.

Giờ hãy xem xét trường hợp thứ hai: cho dữ liệu theo dõi khí tượng, chúng ta muốn dự đoán xác suất ngày mai trời sẽ mưa ở Đài Bắc. Nếu vào mùa hè, xác suất trời mưa có thể là 0.5.

Trong cả hai trường hợp, chúng ta đều quan tâm tới một đại lượng nào đó và cùng không chắc chắn về giá trị đầu ra. Nhưng có một khía cạnh quan trọng giữa hai trường hợp. Trong trường hợp đầu tiên, bức ảnh chỉ có thể là chó hoặc mèo, và chúng ta chỉ không biết là loài nào. Trong trường hợp thứ hai, đầu ra thực sự có thể là một sự kiện ngẫu nhiên, nếu bạn tin vào những thứ như vậy (và hầu hết các nhà vật lý tin vậy). Như vậy xác suất là một ngôn ngữ linh hoạt để suy đoán về mức độ chắc chắn của chúng ta, và nó có thể được áp dụng hiệu quả trong vô vàn ngữ cảnh khác nhau.

5.6.1 Lý thuyết Xác suất cơ bản

Giả sử, ta tung xúc xắc và muốn biết cơ hội để thấy mặt số 1 so với các mặt khác là bao nhiêu? Nếu chiếc xúc xắc có chất liệu đồng nhất, thì cả 6 mặt $\{1, \dots, 6\}$ đều có khả năng xuất hiện như nhau, nên ta sẽ thấy mặt 1 xuất hiện một lần trong mỗi sáu lần tung xúc xắc như trên. Ta có thể nói rằng mặt 1 xuất hiện với xác suất là $\frac{1}{6}$.

Với một chiếc xúc xắc thật, ta có thể không biết được tỷ lệ này và cần kiểm tra liệu xúc xắc có bị hư hỏng gì không. Cách duy nhất để kiểm tra là tung thật nhiều lần rồi ghi lại kết quả. Mỗi lần tung, ta quan sát thấy một số trong $\{1, \dots, 6\}$ xuất hiện. Với kết quả này, ta muốn kiểm chứng xác suất xuất hiện của từng mặt số.

Cách tính trực quan nhất là lấy số lần xuất hiện của mỗi mặt số chia cho tổng số lần tung. Cách này cho ta một *ước lượng* của xác suất ứng với một *sự kiện* cho trước. *Luật số lớn* cho ta biết rằng số lần tung xúc xắc càng

tăng thì ước lượng này càng gần hơn với xác suất thực. Trước khi giải thích chi tiết hơn, hãy cùng lập trình thí nghiệm này.

Bắt đầu, ta nhập các gói lệnh cần thiết.

```
%matplotlib inline
import d2l
from mxnet import np, npx
import random
npx.set_np()
```

Tiếp theo, ta sẽ cần tung xúc xắc. Trong thống kê, ta gọi quá trình thu các mẫu từ phân phối xác suất là quá trình *lấy mẫu*. Phân phối mà gán các xác suất cho các lựa chọn rời rạc (*discrete choices*) được gọi là *phân phối đa thức* (*multinomial distribution*). Sau này, ta sẽ đưa ra định nghĩa chính quy *phân phối* là gì; nhưng để hình dung, hãy xem nó như phép gán xác suất xảy ra cho các sự kiện. Trong MXNet, ta có thể lấy mẫu từ phân phối đa thức với hàm `np.random.multinomial`. Có nhiều cách sử dụng hàm này, nhưng ta tập trung vào cách dùng đơn giản nhất. Muốn lấy một mẫu đơn, ta chỉ cần đưa vào hàm này một vector chứa các xác suất. Hàm `np.random.multinomial` sẽ cho kết quả là một vector có chiều dài tương tự: trong vector này, giá trị tại chỉ số i là số lần kết quả i xuất hiện.

```
fair_probs = [1.0 / 6] * 6
np.random.multinomial(1, fair_probs)

array([0, 0, 0, 1, 0, 0], dtype=int64)
```

Nếu chạy hàm lấy mẫu vài lần, bạn sẽ thấy rằng mỗi lần các giá trị trả về đều là ngẫu nhiên. Giống với việc đánh giá một con xúc xắc có đều hay không, chúng ta thường muốn tạo nhiều mẫu từ cùng một phân phối. Tạo dữ liệu như trên với vòng lặp `for` trong Python là rất chậm, vì vậy hàm `random.multinomial` hỗ trợ sinh nhiều mẫu trong một lần gọi, trả về một mảng chứa các mẫu độc lập với kích thước bất kỳ.

```
np.random.multinomial(10, fair_probs)

array([1, 1, 5, 1, 1, 1], dtype=int64)
```

Chúng ta cũng có thể giả sử làm 3 thí nghiệm, trong đó mỗi thí nghiệm cùng lúc lấy ra 10 mẫu.

```
counts = np.random.multinomial(10, fair_probs, size=3)
counts

array([[1, 2, 1, 2, 4, 0],
       [3, 2, 2, 1, 0, 2],
       [1, 2, 1, 3, 1, 2]], dtype=int64)
```

Giờ chúng ta đã biết cách lấy mẫu các lần tung của một con xúc xắc, ta có thể giả lập 1000 lần tung. Sau đó, chúng ta có thể đếm xem mỗi mặt xuất hiện bao nhiêu lần. Cụ thể, chúng ta tính toán tần suất tương đối như là một ước lượng của xác suất thực.

```
# Store the results as 32-bit floats for division
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000 # Reletive frequency as the estimate
```

```
array([0.164, 0.153, 0.181, 0.163, 0.163, 0.176])
```

Do dữ liệu được sinh bởi một con xúc xắc đều, ta biết mỗi lần tung đều có xác suất thực bằng $\frac{1}{6}$, cỡ 0.167, do đó kết quả ước lượng bên trên trông khá ổn.

Chúng ta cũng có thể minh họa những xác suất này hội tụ tới xác suất thực như thế nào. Hãy cũng làm 500 thí nghiệm trong đó mỗi thí nghiệm lấy ra 10 mẫu.

```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

d21.set_figsize((6, 4.5))
for i in range(6):
    d21.plt.plot(estimates[:, i].asnumpy(),
                  label="P(die=" + str(i + 1) + ")")
d21.plt.axhline(y=0.167, color='black', linestyle='dashed')
d21.plt.gca().set_xlabel('Groups of experiments')
d21.plt.gca().set_ylabel('Estimated probability')
d21.plt.legend();
```

Mỗi đường cong liền tương ứng với một trong sáu giá trị của xúc xắc và chỉ ra xác suất ước lượng của sự kiện xúc xắc ra mặt tương ứng sau mỗi thí nghiệm. Đường đứt đoạn màu đen tương ứng với xác suất thực. Khi ta lấy thêm dữ liệu bằng cách thực hiện thêm các thí nghiệm, thì 6 đường cong liền sẽ hội tụ tiến tới xác suất thực.

Các Tiên đề của Lý thuyết Xác suất

Khi thực hiện tung một con xúc xắc, chúng ta gọi tập hợp $S = \{1, 2, 3, 4, 5, 6\}$ là *không gian mẫu* hoặc *không gian kết quả*, trong đó mỗi phần tử là một *kết quả*. Một *sự kiện* là một tập hợp các kết quả của không gian mẫu. Ví dụ, “tung được một số 5” ($\{5\}$) và “tung được một số lẻ” ($\{1, 3, 5\}$) đều là những sự kiện hợp lệ khi tung một con xúc xắc. Chú ý rằng nếu kết quả của một phép tung ngẫu nhiên nằm trong sự kiện \mathcal{A} , sự kiện \mathcal{A} đã xảy ra. Như vậy, nếu mặt 3 chấm ngửa lên sau khi xúc xắc được tung, chúng ta nói sự kiện “tung được một số lẻ” đã xảy ra bởi vì $3 \in \{1, 3, 5\}$.

Một cách chính thống hơn, *xác suất* có thể được xem là một hàm số ánh xạ một tập hợp các sự kiện tới một số thực. Xác suất của sự kiện \mathcal{A} trong không gian mẫu S , được kí hiệu là $P(\mathcal{A})$, phải thoả mãn những tính chất sau:

- Với mọi sự kiện \mathcal{A} , xác suất của nó là không âm, tức là: $P(\mathcal{A}) \geq 0$;
- Xác suất của toàn không gian mẫu luôn bằng 1, tức: $P(S) = 1$;
- Đối với mọi dãy sự kiện có thể đếm được $\mathcal{A}_1, \mathcal{A}_2, \dots$ *xung khắc lẫn nhau* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ với mọi $i \neq j$), xác suất có ít nhất một sự kiện xảy ra sẽ là tổng của những giá trị xác suất riêng lẻ, hay: $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

Đây cũng là những tiên đề của lý thuyết xác suất, được đề xuất bởi Kolmogorov năm 1933. Nhờ vào hệ thống tiên đề này, ta có thể tránh được những tranh luận chủ quan về sự ngẫu nhiên; và ta có thể có được những suy luận chặt chẽ sử dụng ngôn ngữ toán học. Ví dụ, cho sự kiện \mathcal{A}_1 là toàn bộ không gian mẫu và $\mathcal{A}_i = \emptyset$ với mọi $i > 1$, chúng ta có thể chứng minh rằng $P(\emptyset) = 0$, nghĩa là xác suất của sự kiện không thể xảy ra bằng 0.

Biến ngẫu nhiên

Trong thí nghiệm tung xúc xắc ngẫu nhiên, chúng ta đã giới thiệu khái niệm của một *biến ngẫu nhiên*. Một biến ngẫu nhiên có thể dùng để biểu diễn cho hầu như bất kỳ đại lượng nào và giá trị của nó không cố định. Nó có thể nhận một giá trị trong tập các giá trị khả dĩ từ một thí nghiệm ngẫu nhiên. Hãy xét một biến ngẫu nhiên X có thể nhận một trong những giá trị từ tập không gian mẫu $S = \{1, 2, 3, 4, 5, 6\}$ của thí nghiệm tung xúc xắc. Chúng ta có thể biểu diễn sự kiện “trông thấy mặt 5” là $\{X = 5\}$ hoặc $X = 5$, và xác suất của nó là $P(\{X = 5\})$ hoặc $P(X = 5)$. Khi viết $P(X = a)$, chúng ta đã phân biệt giữa biến ngẫu nhiên X và các giá trị (ví dụ như a) mà X có thể nhận. Tuy nhiên, ký hiệu như vậy khá là rườm rà. Để đơn giản hóa ký hiệu, một mặt, chúng ta có thể chỉ cần dùng $P(X)$ để biểu diễn *phân phối* của biến ngẫu nhiên X : phân phối này cho chúng ta biết xác xuất mà X có thể nhận cho bất kỳ giá trị nào. Mặt khác, chúng ta có thể đơn thuần viết $P(a)$ để biểu diễn xác suất mà một biến ngẫu nhiên nhận giá trị a . Bởi vì một sự kiện trong lý thuyết xác suất là một tập các kết quả từ không gian mẫu, chúng ta có thể xác định rõ một khoảng các giá trị mà một biến ngẫu nhiên có thể nhận. Ví dụ, $P(1 \leq X \leq 3)$ diễn tả xác suất của sự kiện $\{1 \leq X \leq 3\}$, nghĩa là $\{X = 1, 2, \text{ hoặc }, 3\}$. Tương tự, $P(1 \leq X \leq 3)$ biểu diễn xác suất mà biến ngẫu nhiên X có thể nhận giá trị trong tập $\{1, 2, 3\}$.

Lưu ý rằng có một sự khác biệt tinh tế giữa các biến ngẫu nhiên *rời rạc*, ví dụ như các mặt của xúc xắc, và các biến ngẫu nhiên *liên tục*, ví dụ như cân nặng và chiều cao của một người. Việc hỏi rằng hai người có cùng chính xác chiều cao hay không khá là vô nghĩa. Nếu ta đo với độ chính xác, ta sẽ thấy rằng không có hai người nào trên hành tinh này mà có cùng chính xác chiều cao cả. Thật vậy, nếu đo đủ chính xác, chiều cao của bạn lúc mới thức dậy và khi đi ngủ sẽ khác nhau. Cho nên không có lý do gì để tìm xác suất một người nào đó cao 1.80139278291028719210196740527486202 mét cả. Trong toàn bộ dân số trên thế giới, xác suất này gần như bằng 0. Sẽ có lý hơn nếu ta hỏi chiều cao của một người nào đó có rơi vào một khoảng cho trước hay không, ví dụ như giữa 1.79 và 1.81 mét. Trong các trường hợp này, ta có thể định lượng khả năng mà ta thấy một giá trị nào đó theo một *mật độ xác suất*. Xác suất để có chiều cao chính xác 1.80 mét không tồn tại, nhưng mật độ của sự kiện này khác không. Trong bất kỳ khoảng nào giữa hai chiều cao khác nhau ta đều có xác suất khác không. Trong phần còn lại của mục này, ta sẽ xem xét xác suất trong không gian rời rạc. Về xác suất của biến ngẫu nhiên liên tục, bạn có thể xem ở `sec_random_variables`.

5.6.2 Làm việc với Nhiều Biến Ngẫu nhiên

Chúng ta sẽ thường xuyên phải làm việc với nhiều hơn một biến ngẫu nhiên cùng lúc. Ví dụ, chúng ta có thể muốn mô hình hóa mối quan hệ giữa các loại bệnh và các triệu chứng bệnh. Cho một loại bệnh và một triệu chứng bệnh, giả sử “cảm cúm” và “ho”, chúng có thể xuất hiện hoặc không trên một bệnh nhân với xác suất nào đó. Mặc dù chúng ta hy vọng xác suất cả hai xảy ra gần bằng không, ta có thể vẫn muốn ước lượng các xác suất này và mối quan hệ giữa chúng để có thể thực hiện các biện pháp chăm sóc y tế tốt hơn.

Xét một ví dụ phức tạp hơn: các bức ảnh chứa hàng triệu điểm ảnh, tương ứng với hàng triệu biến ngẫu nhiên. Và trong nhiều trường hợp các bức ảnh sẽ được gắn một nhãn chứa tên các vật xuất hiện trong ảnh. Chúng ta cũng có thể xem nhãn này như một biến ngẫu nhiên. Thậm chí, ta còn có thể xem tất cả các siêu dữ liệu như địa điểm, thời gian, khẩu độ, tiêu cự, ISO, khoảng lấy nét và loại máy ảnh, là các biến ngẫu nhiên. Tất cả các những biến ngẫu nhiên này xảy ra đồng thời. Khi làm việc với nhiều biến ngẫu nhiên, có một số đại lượng đáng được quan tâm.

Xác suất Đồng thời

Đầu tiên là xác suất đồng thời $P(A = a, B = b)$. Cho hai biến a và b bất kỳ, xác suất đồng thời cho ta biết xác suất để cả $A = a$ và $B = b$ đều xảy ra. Ta có thể thấy rằng với mọi giá trị a và b , $P(A = a, B = b) \leq P(A = a)$. Bởi để $A = a$ và $B = b$ xảy ra thì $A = a$ phải xảy ra và $B = b$ cũng phải xảy ra (và ngược lại). Do đó, khả năng $A = a$ và $B = b$ xảy ra đồng thời không thể lớn hơn khả năng $A = a$ hoặc $B = b$ xảy ra một cách độc lập được.

Xác suất có điều kiện

Điều này giúp ta thu được một tỉ lệ thú vị: $0 \leq \frac{P(A=a, B=b)}{P(A=a)} \leq 1$. Chúng ta gọi tỉ lệ này là *xác suất có điều kiện* và ký hiệu là $P(B = b | A = a)$: xác suất để $B = b$, với điều kiện $A = a$ đã xảy ra.

Định lý Bayes

Sử dụng định nghĩa của xác suất có điều kiện, chúng ta có thể thu được một trong những phương trình nổi tiếng và hữu dụng nhất trong thống kê: *định lý Bayes*. Cụ thể như sau: Theo định nghĩa chúng ta có *quy tắc nhân* $P(A, B) = P(B | A)P(A)$. Tương tự, ta cũng có $P(A, B) = P(A | B)P(B)$. Giả sử $P(B) > 0$. Kết hợp các điều kiện trên ta có:

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)}. \quad (5.6.1)$$

Lưu ý rằng ở đây chúng ta sử dụng ký hiệu ngắn gọn hơn, với $P(A, B)$ là *xác suất đồng thời* và $P(A | B)$ là *xác suất có điều kiện*. Các phân phối này có thể được tính tại các giá trị cụ thể $A = a, B = b$.

Phép biến hóa

Định lý Bayes rất hữu ích nếu chúng ta muốn suy luận một điều gì đó từ một điều khác, như là nguyên nhân và kết quả, nhưng ta chỉ biết các đặc tính theo chiều ngược lại, như ta sẽ thấy trong phần sau của chương này. Chúng ta cần làm một thao tác quan trọng để đạt được điều này, đó là *phép biến hóa*. Có thể hiểu là việc xác định $P(B)$ từ $P(A, B)$. Chúng ta có thể tính được xác suất của B bằng tổng xác suất kết hợp của A và B tại mọi giá trị có thể của A :

$$P(B) = \sum_A P(A, B), \quad (5.6.2)$$

Công thức này cũng được biết đến với tên gọi *quy tắc tổng*. Xác suất hay phân phối thu được từ thao tác biến hóa được gọi là *xác suất biến* hoặc *phân phối biến*.

Tính độc lập

Một tính chất hữu ích khác cần kiểm tra là *tính phụ thuộc* và *tính độc lập*. Hai biến ngẫu nhiên A và B độc lập nghĩa là việc một sự kiện của A xảy ra không tiết lộ bất kỳ thông tin nào về việc xảy ra một sự kiện của B . Trong trường hợp này $P(B | A) = P(B)$. Các nhà thống kê thường biểu diễn điều này bằng ký hiệu $A \perp B$. Từ định lý Bayes, ta có $P(A | B) = P(A)$. Trong tất cả các trường hợp khác, chúng ta gọi A và B là hai biến phụ thuộc. Ví dụ, hai lần đổ liên tiếp của một con xúc xắc là độc lập. Ngược lại, vị trí của công tắc đèn

và độ sáng trong phòng là không độc lập (tuy nhiên chúng không hoàn toàn xác định, vì bóng đèn luôn có thể bị hỏng, mất điện hoặc công tắc bị hỏng).

Vì $P(A | B) = \frac{P(A, B)}{P(B)} = P(A)$ tương đương với $P(A, B) = P(A)P(B)$, hai biến ngẫu nhiên là độc lập khi và chỉ khi phân phối đồng thời của chúng là tích các phân phối riêng lẻ của chúng. Tương tự, cho một biến ngẫu nhiên C khác, hai biến ngẫu nhiên A và B là *độc lập có điều kiện* khi và chỉ khi $P(A, B | C) = P(A | C)P(B | C)$. Điều này được ký hiệu là $A \perp B | C$.

Ứng dụng

Hãy thử nghiệm các kiến thức chúng ta vừa học. Giả sử rằng một bác sĩ phụ trách xét nghiệm AIDS cho một bệnh nhân. Việc xét nghiệm này khá chính xác và nó chỉ thất bại với xác suất 1%, khi nó cho kết quả dương tính dù bệnh nhân khỏe mạnh. Hơn nữa, nó không bao giờ thất bại trong việc phát hiện HIV nếu bệnh nhân thực sự bị nhiễm bệnh. Ta sử dụng D_1 để biểu diễn kết quả chẩn đoán (1 nếu dương tính và 0 nếu âm tính) và H để biểu thị tình trạng nhiễm HIV (1 nếu dương tính và 0 nếu âm tính). `conditional_prob_D1` liệt kê xác suất có điều kiện đó.

Xác suất có điều kiện	$H = 1$	$H = 0$
$P(D_1 = 1 H)$	1	0.01
$P(D_1 = 0 H)$	0	0.99

Table: Xác suất có điều kiện của $P(D_1 | H)$.

Lưu ý rằng tổng của từng cột đều bằng 1 (nhưng tổng từng hàng thì không), vì xác suất có điều kiện cần có tổng bằng 1, giống như xác suất. Hãy cùng tìm xác suất bệnh nhân bị AIDS nếu xét nghiệm trả về kết quả dương tính, tức $P(H = 1 | D_1 = 1)$. Rõ ràng điều này sẽ phụ thuộc vào mức độ phổ biến của bệnh, bởi vì nó ảnh hưởng đến số lượng dương tính giả. Giả sử rằng dân số khá khỏe mạnh, ví dụ: $P(H = 1) = 0.0015$. Để áp dụng Định lý Bayes, chúng ta cần áp dụng phép biến hóa và quy tắc nhân để xác định

$$\begin{aligned} & P(D_1 = 1) \\ &= P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ &= P(D_1 = 1 | H = 0)P(H = 0) + P(D_1 = 1 | H = 1)P(H = 1) \\ &= 0.011485. \end{aligned} \tag{5.6.3}$$

Do đó, ta có

$$\begin{aligned} & P(H = 1 | D_1 = 1) \\ &= \frac{P(D_1 = 1 | H = 1)P(H = 1)}{P(D_1 = 1)} \\ &= 0.1306 \end{aligned} \tag{5.6.4}$$

Nói cách khác, chỉ có 13,06% khả năng bệnh nhân thực sự mắc bệnh AIDS, dù ta dùng một bài kiểm tra rất chính xác. Như ta có thể thấy, xác suất có thể trở nên khá phản trực giác.

Một bệnh nhân phải làm gì nếu nhận được tin dữ như vậy? Nhiều khả năng họ sẽ yêu cầu bác sĩ thực hiện một xét nghiệm khác để làm rõ sự việc. Bài kiểm tra thứ hai có những đặc điểm khác và không tốt bằng bài thứ nhất, như ta có thể thấy trong `conditional_prob_D2`.

Xác xuất có điều kiện	$H = 1$	$H = 0$
$P(D_2 = 1 H)$	0.98	0.03
$P(D_2 = 0 H)$	0.02	0.97

Table: Xác suất có điều kiện của $P(D_2 | H)$.

Không may thay, bài kiểm tra thứ hai cũng có kết quả dương tính. Hãy cùng tính các xác suất cần thiết để sử dụng định lý Bayes bằng cách giả định tính độc lập có điều kiện:

$$\begin{aligned} & P(D_1 = 1, D_2 = 1 | H = 0) \\ &= P(D_1 = 1 | H = 0)P(D_2 = 1 | H = 0) \\ &= 0.0003, \end{aligned} \tag{5.6.5}$$

$$\begin{aligned} & P(D_1 = 1, D_2 = 1 | H = 1) \\ &= P(D_1 = 1 | H = 1)P(D_2 = 1 | H = 1) \\ &= 0.98. \end{aligned} \tag{5.6.6}$$

Bây giờ chúng ta có thể áp dụng phép biến hóa và quy tắc nhân xác suất:

$$\begin{aligned} & P(D_1 = 1, D_2 = 1) \\ &= P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\ &= P(D_1 = 1, D_2 = 1 | H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 | H = 1)P(H = 1) \\ &= 0.00176955. \end{aligned} \tag{5.6.7}$$

Cuối cùng xác suất bệnh nhân mắc bệnh AIDS qua hai lần dương tính là

$$\begin{aligned} & P(H = 1 | D_1 = 1, D_2 = 1) \\ &= \frac{P(D_1 = 1, D_2 = 1 | H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)} \\ &= 0.8307. \end{aligned} \tag{5.6.8}$$

Cụ thể hơn, thử nghiệm thứ hai mang lại độ tin cậy cao hơn rằng không phải mọi chuyện đều ổn. Mặc dù bài kiểm tra thứ hai kém chính xác hơn bài đầu, nó vẫn cải thiện đáng kể dự đoán.

5.6.3 Kỳ vọng và Phương sai

Để tóm tắt những đặc tính then chốt của các phân phối xác suất, chúng ta cần một vài phép đo. *Kỳ vọng* (hay trung bình) của một biến ngẫu nhiên X , được ký hiệu là

$$E[X] = \sum_x xP(X = x). \tag{5.6.9}$$

Khi giá trị đầu vào của phương trình $f(x)$ là một biến ngẫu nhiên cho trước theo phân phối P với các giá trị x khác nhau, kỳ vọng của $f(x)$ sẽ được tính theo phương trình:

$$E_{x \sim P}[f(x)] = \sum_x f(x)P(x). \tag{5.6.10}$$

Trong nhiều trường hợp, chúng ta muốn đo độ lệch của biến ngẫu nhiên X so với kỳ vọng của nó. Đại lượng này có thể được đo bằng phương sai

$$\text{Var}[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2. \tag{5.6.11}$$

Nếu lấy căn bậc hai của kết quả ta sẽ được độ lệch chuẩn. Phương sai của một hàm của một biến ngẫu nhiên đo độ lệch của hàm số đó từ kỳ vọng của nó khi các giá trị x khác nhau được lấy mẫu từ phân phối của biến ngẫu nhiên đó:

$$\text{Var}[f(x)] = E[(f(x) - E[f(x)])^2]. \tag{5.6.12}$$

5.6.4 Tóm tắt

- Chúng ta có thể sử dụng MXNet để lấy mẫu từ phân phối xác suất.
- Các biến ngẫu nhiên có thể được phân tích bằng các phương pháp như phân phối đồng thời (*joint distribution*), phân phối có điều kiện (*conditional distribution*), định lý Bayes, phép biên hóa (*marginalization*) và giả định độc lập (*independence assumptions*).
- Kỳ vọng và phương sai là các phép đo hữu ích để tóm tắt các đặc điểm chính của phân phối xác suất.

5.6.5 Bài tập

1. Tiến hành $m = 500$ nhóm thí nghiệm với mỗi nhóm lấy ra $n = 10$ mẫu. Thay đổi m và n . Quan sát và phân tích kết quả của thí nghiệm.
2. Cho hai sự kiện với xác suất $P(\mathcal{A})$ và $P(\mathcal{B})$, tính giới hạn trên và dưới của $P(\mathcal{A} \cup \mathcal{B})$ và $P(\mathcal{A} \cap \mathcal{B})$. (Gợi ý: sử dụng biểu đồ Venn⁶⁶.)
3. Giả sử chúng ta có các biến ngẫu nhiên A , B và C , với B chỉ phụ thuộc A , và C chỉ phụ thuộc vào B . Làm thế nào để đơn giản hóa xác suất đồng thời của $P(A, B, C)$? (Gợi ý: đây là một Chuỗi Markov⁶⁷.)
4. Trong subsec_probability_hiv_app, bài xét nghiệm đầu tiên có độ chính xác cao hơn. Vậy tại sao chúng ta không sử dụng bài xét nghiệm đầu tiên cho lần thử tiếp theo?

5.6.6 Thảo luận

- Tiếng Anh⁶⁸
- Tiếng Việt⁶⁹

5.6.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm
- Vũ Hữu Tiệp
- Nguyễn Cảnh Thượng
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Mai Sơn Hải
- Trần Kiến An
- Tạ H. Duy Nguyên
- Phạm Minh Đức

⁶⁶ https://en.wikipedia.org/wiki/Venn_diagram

⁶⁷ https://en.wikipedia.org/wiki/Markov_chain

⁶⁸ <https://discuss.mxnet.io/t/2319>

⁶⁹ <https://forum.machinelearningcoban.com/c/d2l>

- Trần Thị Hồng Hạnh
- Lê Thành Vinh
- Nguyễn Minh Thư

5.7 Tài liệu

Vì độ dài cuốn sách này có giới hạn, chúng tôi không thể giới thiệu hết tất cả các hàm và lớp của MXNet (và tốt nhất nên như vậy). Tài liệu API, các hướng dẫn và ví dụ sẽ cung cấp nhiều thông tin vượt ra khỏi nội dung cuốn sách. Trong chương này, chúng tôi sẽ cung cấp một vài chỉ dẫn để bạn có thể khám phá MXNet API.

5.7.1 Tra cứu tất cả các hàm và lớp trong một Mô-đun

Để biết những hàm/lớp nào có thể được gọi trong một mô-đun, chúng ta dùng hàm `dir`. Ví dụ, ta có thể lấy tất cả thuộc tính của mô-đun `np.random` bằng cách:

```
from mxnet import np
print(dir(np.random))

['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', '_mx_nd_np', 'absolute_import',
 'choice', 'multinomial', 'normal', 'rand', 'randint', 'shuffle', 'uniform']
```

Thông thường, ta có thể bỏ qua những hàm bắt đầu và kết thúc với `__` (các đối tượng đặc biệt trong Python) hoặc những hàm bắt đầu bằng `_` (thường là các hàm địa phương). Dựa trên tên của những hàm và thuộc tính còn lại, ta có thể dự đoán rằng mô-đun này cung cấp những phương thức sinh số ngẫu nhiên, bao gồm lấy mẫu từ phân phối đều liên tục (`uniform`), phân phối chuẩn (`normal`) và phân phối đa thức (`multinomial`).

5.7.2 Tra cứu cách sử dụng một hàm hoặc một lớp cụ thể

Để tra cứu chi tiết cách sử dụng một hàm hoặc lớp nhất định, ta dùng hàm `help`. Ví dụ, để tra cứu cách sử dụng hàm `ones_like` với `ndarray`:

```
help(np.ones_like)
```

Help on function `ones_like` in module `mxnet.numpy`:

```
ones_like(a)
    Return an array of ones with the same shape and type as a given_
array.
```

```
Parameters
-----
a : ndarray
    The shape and data-type of a define these same attributes of
    the returned array.
```

```
Returns
```

```
-----  
out : ndarray  
      Array of ones with the same shape and type as a.
```

Examples

```
-----  
>>> x = np.arange(6)  
>>> x = x.reshape((2, 3))  
>>> x  
array([[0., 1., 2.],  
       [3., 4., 5.]])  
>>> np.ones_like(x)  
array([[1., 1., 1.],  
       [1., 1., 1.]])  
  
>>> y = np.arange(3, dtype=float)  
>>> y  
array([0., 1., 2.], dtype=float64)  
>>>  
>>> np.ones_like(y)  
array([1., 1., 1.], dtype=float64)
```

Từ tài liệu, ta có thể thấy hàm `ones_like` tạo một mảng mới có cùng kích thước với `ndarray` nhưng tất cả các phần tử của nó đều chứa giá trị 1. Nếu có thể, bạn nên chạy thử để xác nhận rằng mình hiểu đúng.

```
x = np.array([[0, 0, 0], [2, 2, 2]])  
np.ones_like(x)
```

```
array([[1., 1., 1.],  
       [1., 1., 1.]])
```

Trong Jupyter notebook, ta có thể dùng `?` để mở tài liệu trong một cửa sổ khác. Ví dụ, `np.random.uniform?` sẽ in ra nội dung y hệt `help(np.random.uniform)` trong một cửa sổ trình duyệt mới. Ngoài ra, nếu chúng ta dùng dấu `?` hai lần như `np.random.uniform??` thì đoạn mã định nghĩa hàm cũng sẽ được in ra.

5.7.3 Tài liệu API

Chi tiết cụ thể về các API của MXNet có thể được tìm thấy tại trang <http://mxnet.apache.org/>. Chi tiết từng phần được tìm thấy tại các đề mục tương ứng (cho cả các ngôn ngữ lập trình khác ngoài Python).

5.7.4 Tóm tắt

- Tài liệu chính thức cung cấp rất nhiều các mô tả và ví dụ ngoài cuốn sách này.
- Chúng ta có thể tra cứu tài liệu về cách sử dụng MXNet API bằng cách gọi hàm `dir` và `help`, hoặc kiểm tra tại trang web của MXNet.

5.7.5 Bài tập

1. Tra cứu `ones_like` và `autograd` trên trang MXNet.
2. Tất cả các kết quả khả dĩ sau khi chạy `np.random.choice(4, 2)` là gì?
3. Bạn có thể viết lại `np.random.choice(4, 2)` bằng cách sử dụng hàm `np.random.randint` không?

5.7.6 Thảo luận

- Tiếng Anh⁷⁰
- Tiếng Việt⁷¹

5.7.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Hoàng Quân
- Vũ Hữu Tiệp

5.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Cảnh Thượng
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

⁷⁰ <https://discuss.mxnet.io/t/2322>

⁷¹ <https://forum.machinelearningcoban.com/c/d2l>

6 | Mạng nơ-ron tuyến tính

6.1 Chương này đang trong quá trình hiệu đính, dự tính hoàn thiện 20/03/2020

Trước khi tìm hiểu chi tiết về mạng nơ-ron sâu, chúng ta cần đề cập đến căn bản của việc huấn luyện mạng nơ-ron. Chương này sẽ đề cập đến toàn bộ quá trình huấn luyện, bao gồm xác định kiến trúc mạng nơ-ron đơn giản, xử lý dữ liệu, chỉ rõ hàm mất mát và huấn luyện mô hình. Để mọi thứ dễ nắm bắt hơn, ta sẽ bắt đầu với một số khái niệm cơ bản nhất. Rất may mắn, một số phương pháp học thống kê cổ điển như hồi quy tuyến tính, hồi quy logistic có thể được xem như những mạng nơ-ron *nồng*. Bắt đầu bằng những thuật toán cổ điển này, chúng tôi sẽ giới thiệu những thứ cơ bản, tạo nền tảng cho cho những kỹ thuật phức tạp hơn như hồi quy Softmax (sẽ giới thiệu ở cuối chương này) và Mạng nơ-ron truyền thẳng nhiều lớp (sẽ giới thiệu ở chương sau).

6.1.1 Hồi quy Tuyến tính

Hồi quy ám chỉ các phương pháp để xây dựng mối quan hệ giữa điểm dữ liệu x và mục tiêu với giá trị số thực y . Trong khoa học tự nhiên và khoa học xã hội, mục tiêu của hồi quy thường là *đặc trưng hóa* mối quan hệ của đầu vào và đầu ra. Mặt khác, học máy lại thường quan tâm đến việc *dự đoán*.

Bài toán hồi quy xuất hiện mỗi khi chúng ta muốn dự đoán một giá trị số. Các ví dụ phổ biến bao gồm dự đoán giá cả (nhà, cổ phiếu, ...), thời gian bệnh nhân nằm viện, nhu cầu trong ngành bán lẻ và vô vàn thứ khác. Không phải mọi bài toán dự đoán đều là bài toán *hồi quy* cổ điển. Trong các phần tiếp theo, chúng tôi sẽ giới thiệu bài toán phân loại, khi mục tiêu là dự đoán lớp đúng trong một tập các lớp cho trước.

Các Thành phần Cơ bản của Hồi quy Tuyến tính

Hồi quy tuyến tính có lẽ là công cụ tiêu chuẩn đơn giản và phổ biến nhất được sử dụng cho bài toán hồi quy. Xuất hiện từ đầu thế kỷ 19, hồi quy tuyến tính được phát triển từ một vài giả thuyết đơn giản. Đầu tiên, ta giả sử quan hệ giữa các *đặc trưng* x và mục tiêu y là tuyến tính, do đó y có thể được biểu diễn bằng tổng trọng số của đầu vào x , cộng hoặc trừ thêm nhiễu của các quan sát. Thứ hai, ta giả sử nhiễu có quy tắc (tuân theo phân phối Gauss). Để tạo động lực, hãy bắt đầu với một ví dụ. Giả sử ta muốn ước lượng giá nhà (bằng đô la) dựa vào diện tích (đơn vị feet vuông) và tuổi đời (theo năm).

Để khớp một mô hình dự đoán giá nhà, chúng ta cần một tập dữ liệu các giao dịch mà trong đó ta biết giá bán, diện tích, tuổi đời cho từng căn nhà. Trong thuật ngữ của học máy, tập dữ liệu này được gọi là *dữ liệu huấn luyện* hoặc *tập huấn luyện*, và mỗi hàng (tương ứng với dữ liệu của một giao dịch) được gọi là một *ví dụ* hoặc *mẫu*. Thứ mà chúng ta muốn dự đoán (giá nhà) được gọi là *mục tiêu* hoặc *nhãn*. Các biến (*tuổi đời* và *diện tích*) mà những dự đoán dựa vào được gọi là các *đặc trưng* hoặc *hiệp biến*.

Thông thường, chúng ta sẽ dùng n để ký hiệu số lượng mẫu trong tập dữ liệu. Chỉ số i được dùng để xác định một mẫu cụ thể. Ta ký hiệu mỗi điểm dữ liệu đầu vào là $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$ và nhãn tương ứng là $y^{(i)}$.

Mô hình Tuyến tính

Giả định tuyến tính trên cho thấy rằng mục tiêu (giá nhả) có thể được biểu diễn bởi tổng có trọng số của các đặc trưng (diện tích và tuổi đồi):

Ở đây, w

Lưu ý rằng khi hiệu giữa giá trị ước lượng $\hat{y}^{(i)}$ và giá trị quan sát $y^{(i)}$ lớn, giá trị hàm mất mát sẽ tăng một lượng còn lớn hơn thế do sự phụ thuộc bậc hai. Để đo chất lượng của mô hình trên toàn bộ tập dữ liệu, ta đơn thuần lấy trung bình (hay tương đương là lấy tổng) các giá trị mất mát của từng mẫu trong tập huấn luyện.

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2. \quad (6.1.2)$$

Khi huấn luyện mô hình, ta muốn tìm các tham số (\mathbf{w}^*, b^*) sao cho tổng độ mất mát trên toàn bộ các mẫu huấn luyện được tối thiểu hóa:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} L(\mathbf{w}, b). \quad (6.1.3)$$

Nghiệm theo công thức

Hóa ra hồi quy tuyến tính chỉ là một bài toán tối ưu hóa đơn giản. Khác với hầu hết các mô hình được giới thiệu trong cuốn sách này, hồi quy tuyến tính có thể được giải bằng cách áp dụng một công thức đơn giản, cho một nghiệm tối ưu toàn cục. Để bắt đầu, chúng ta có thể gộp hệ số điều chỉnh b vào tham số \mathbf{w} bằng cách thêm một cột toàn 1 vào ma trận dữ liệu. Khi đó bài toán dự đoán trở thành bài toán tối thiểu hóa $\|\mathbf{y} - X\mathbf{w}\|$. Bởi vì biểu thức này có dạng toàn phương, nó là một hàm số lồi, và miễn là bài toán này không suy biến (các đặc trưng độc lập tuyến tính), nó là một hàm số lồi chặt.

Bởi vậy chỉ có một điểm cực trị trên mặt mất mát và nó tương ứng với giá trị mất mát nhỏ nhất. Lấy đạo hàm của hàm mất mát theo \mathbf{w} và giải phương trình đạo hàm này bằng 0, ta sẽ được nghiệm theo công thức:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \quad (6.1.4)$$

Tuy những bài toán đơn giản như hồi quy tuyến tính có thể có nghiệm theo công thức, bạn không nên làm quen với sự may mắn này. Mặc dù các nghiệm theo công thức giúp ta phân tích toán học một cách thuận tiện, các điều kiện để có được nghiệm này chặt chẽ đến nỗi không có phương pháp học sâu nào thoả mãn được.

Hạ Gradient

Trong nhiều trường hợp ở đó ta không thể giải quyết các mô hình theo phép phân tích, và thậm chí khi mặt măt măt là các mặt bậc cao và không lồi, trên thực tế ta vẫn có thể huấn luyện các mô hình này một cách hiệu quả. Hơn nữa, trong nhiều tác vụ, những mô hình khó để tối ưu hóa này hoà ra lại tốt hơn các phương pháp khác nhiều, vậy nên việc bỏ công sức để tìm cách tối ưu chúng là hoàn toàn xứng đáng.

Kỹ thuật chính để tối ưu hóa gần như bất kỳ mô hình học sâu nào, sẽ được sử dụng xuyên suốt cuốn sách này, bao gồm việc giảm thiểu lỗi qua các vòng lặp bằng cách cập nhật tham số theo hướng làm giảm dần hàm măt măt. Thuật toán này được gọi là *hạ gradient*. Trên các mặt măt măt lồi, giá trị măt măt cuối cùng sẽ hội tụ về giá trị nhỏ nhất. Tuy điều tương tự không thể áp dụng cho các mặt không lồi, ít nhất thuật toán sẽ dẫn tới một cực tiểu (hy vọng là tốt).

Ứng dụng đơn giản nhất của hạ gradient bao gồm việc tính đạo hàm của hàm măt măt, tức trung bình của các giá trị măt măt được tính trên mỗi mẫu của tập dữ liệu. Trong thực tế, việc này có thể cực kì chậm. Chúng ta phải duyệt qua toàn bộ tập dữ liệu trước khi thực hiện một lần cập nhật. Vì thế, thường ta chỉ muốn lấy một minibatch ngẫu nhiên các mẫu mỗi khi ta cần tính bước cập nhật. Phương pháp biến thể này được gọi là *hạ gradient ngẫu nhiên*.

Trong mỗi vòng lặp, đầu tiên chúng ta lấy ngẫu nhiên một minibatch \mathcal{B} dữ liệu huấn luyện với kích thước cố định. Sau đó, chúng ta tính đạo hàm (gradient) của hàm măt măt trên minibatch đó theo các tham số của mô hình. Cuối cùng, gradient này được nhân với tốc độ học $\eta > 0$ và kết quả này được trừ đi từ các giá trị tham số hiện tại.

Chúng ta có thể biểu diễn việc cập nhật bằng công thức toán như sau (∂ là ký hiệu đạo hàm riêng của hàm số) :

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b). \quad (6.1.5)$$

Tổng kết lại, các bước của thuật toán như sau: (i) khởi tạo các giá trị tham số của mô hình, thường thì sẽ được chọn ngẫu nhiên. (ii) tại mỗi vòng lặp, ta lấy ngẫu nhiên từng batch từ tập dữ liệu (nhiều lần), rồi tiến hành cập nhật các tham số của mô hình theo hướng ngược với gradient.

Khi sử dụng hàm măt măt bậc hai và mô hình tuyến tính, chúng ta có thể biểu diễn bước này một cách tường minh như sau: Lưu ý rằng \mathbf{w} và \mathbf{x} là các vector. Ở đây, việc ký hiệu bằng các vector giúp công thức dễ đọc hơn nhiều so với việc biểu diễn bằng các hệ số như w_1, w_2, \dots, w_d .

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \quad (6.1.6)$$

Trong phương trình trên, $|\mathcal{B}|$ là số ví dụ trong mỗi minibatch (kích thước batch) và η là *tốc độ học*. Cũng cần phải nhấn mạnh rằng các giá trị của kích thước batch và tốc độ học được lựa chọn trước một cách thủ công và thường không được học thông qua quá trình huấn luyện mô hình. Các tham số này tuy điều chỉnh được nhưng không được cập nhật trong vòng huấn luyện, và được gọi là *siêu tham số*. *Điều chỉnh siêu tham số* là quá trình lựa chọn chúng, thường dựa trên kết quả của vòng lặp huấn luyện được đánh giá trên một tập *kiểm định* riêng biệt.

Sau khi huấn luyện đủ số vòng lặp được xác định trước (hoặc đạt được một tiêu chí dừng khác), ta sẽ ghi lại các tham số mô hình đã được ước lượng, ký hiệu là $\hat{\mathbf{w}}, \hat{b}$ (ký hiệu “mũ” thường thể hiện các giá trị ước lượng). Lưu ý rằng cả khi hàm số thực sự tuyến tính và không có nhiễu, các tham số này sẽ không tối thiểu hóa

được hàm mất mát. Mặc dù thuật toán dần dần hội tụ đến một điểm cực tiểu, nó vẫn không thể tối chính xác được cực tiểu đó với số bước hữu hạn.

Hồi quy tuyến tính thực ra là một bài toán tối ưu lồi, do đó chỉ có một cực tiểu (toàn cục). Tuy nhiên, đối với các mô hình phức tạp hơn, như mạng sâu, mặt của hàm mất mát sẽ có nhiều cực tiểu. May mắn thay, vì một lý do nào đó mà những người làm về học sâu hiếm khi phải vật lộn để tìm ra các tham số tối thiểu hóa hàm mất mát trên dữ liệu huấn luyện. Nhiệm vụ khó khăn hơn là tìm ra các tham số dẫn đến giá trị mất mát thấp trên dữ liệu mà mô hình chưa từng thấy trước đây, một thử thách được gọi là *sự khai quát hóa*. Chúng ta sẽ gặp lại chủ đề này xuyên suốt cuốn sách.

Dự đoán bằng Mô hình đã được Huấn luyện

Với mô hình hồi quy tuyến tính đã được huấn luyện $\hat{w}^\top x + \hat{b}$, ta có thể ước lượng giá của một căn nhà mới (ngoài bộ dữ liệu dùng để huấn luyện) với diện tích x_1 và tuổi đời x_2 của nó. Việc ước lượng mục tiêu khi biết trước những đặc trưng thường được gọi là *dự đoán* hay *suy luận* (*inference*).

Ở đây ta sẽ dùng từ *dự đoán* thay vì *suy luận*, dù *suy luận* là một thuật ngữ khá phổ biến trong học sâu, áp dụng thuật ngữ này ở đây lại không phù hợp. Trong thống kê, *suy luận* thường được dùng cho việc ước lượng thông số dựa trên tập dữ liệu. Việc dùng sai thuật ngữ này là nguyên nhân gây ra sự hiểu nhầm giữa những người làm học sâu và các nhà thống kê.

Vector hóa để tăng Tốc độ Tính toán

Khi huấn luyện mô hình, chúng ta thường muốn xử lý đồng thời các mẫu dữ liệu trong minibatch. Để làm được điều này một cách hiệu quả, chúng ta phải vector hóa việc tính toán bằng cách sử dụng các thư viện đại số tuyến tính thay vì sử dụng các vòng lặp `for` trong Python.

Chúng ta sẽ sử dụng hai phương pháp cộng vector dưới đây để hiểu được tại sao vector hóa là cần thiết trong học máy. Đầu tiên, ta khởi tạo hai vector 10000 chiều chứa toàn giá trị một. Chúng ta sẽ sử dụng vòng lặp `for` trong Python ở phương pháp thứ nhất và một hàm trong thư viện `np` ở phương pháp thứ hai.

```
%matplotlib inline
import d2l
import math
from mxnet import np
import time

n = 10000
a = np.ones(n)
b = np.ones(n)
```

Vì ta sẽ cần đánh giá xếp hạng thời gian xử lý một cách thường xuyên trong cuốn sách này, ta sẽ định nghĩa một bộ tính giờ (sau đó có thể truy cập được thông qua gói `d2l` để theo dõi thời gian chạy).

```
# Saved in the d2l package for later use
class Timer(object):
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()
```

(continues on next page)

```

def start(self):
    # Start the timer
    self.start_time = time.time()

def stop(self):
    # Stop the timer and record the time in a list
    self.times.append(time.time() - self.start_time)
    return self.times[-1]

def avg(self):
    # Return the average time
    return sum(self.times)/len(self.times)

def sum(self):
    # Return the sum of time
    return sum(self.times)

def cumsum(self):
    # Return the accumulated times
    return np.array(self.times).cumsum().tolist()

```

Bây giờ, ta có thể đánh giá xếp hạng hai phương pháp cộng vector. Đầu tiên, ta sử dụng vòng lặp `for` để cộng các tọa độ tương ứng.

```

timer = Timer()
c = np.zeros(n)
for i in range(n):
    c[i] = a[i] + b[i]
'%.5f sec' % timer.stop()

```

```
'1.89710 sec'
```

Trong phương pháp hai, ta dựa vào thư viện `np` để tính tổng hai vector theo từng phần tử.

```

timer.start()
d = a + b
'%.5f sec' % timer.stop()

```

```
'0.00012 sec'
```

Bạn có thể nhận thấy rằng, phương pháp thứ hai nhanh hơn rất nhiều lần so với phương pháp thứ nhất. Việc vector hóa thường tăng tốc độ tính toán lên nhiều bậc. Ngoài ra, giao phó công việc tính toán cho thư viện để tránh phải tự viết lại sẽ giảm thiểu khả năng phát sinh lỗi.

Phân phối Chuẩn và Hàm mất mát Bình phương

Mặc dù bạn đã có thể thực hành mà chỉ sử dụng thông tin phía trên, trong phần tiếp theo chúng ta có thể hiểu rõ hơn nguồn gốc của hàm mất mát bình phương thông qua các giả định về phân phối của nhiễu.

Nhớ lại ở trên rằng hàm mất mát bình phương $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ có nhiều thuộc tính tiện lợi. Một trong số đó là đạo hàm đơn giản $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$.

Như được đề cập trước đó, hồi quy tuyến tính được phát minh bởi Gauss vào năm 1795, ông cũng là người khám phá ra phân phối chuẩn (còn được gọi là *Gaussian*). Hóa ra là mối liên hệ giữa phân phối chuẩn và hồi quy tuyến tính sâu hơn chỉ đơn thuần là có chung cha đẻ. Để gợi nhớ lại cho bạn, mật độ xác suất của phân phối chuẩn với trung bình μ và phương sai σ^2 được đưa ra như sau:

$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(z - \mu)^2\right). \quad (6.1.7)$$

Dưới đây ta định nghĩa một hàm Python để tính toán phân phối chuẩn.

```
x = np.arange(-7, 7, 0.01)

def normal(z, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 / sigma**2 * (z - mu)**2)
```

Giờ ta có thể biểu diễn các phân phối chuẩn.

```
# Mean and variance pairs
parameters = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in parameters], xlabel='z',
          ylabel='p(z)', figsize=(4.5, 2.5),
          legend=['mean %d, var %d' % (mu, sigma) for mu, sigma in parameters])
```

Ta có thể thấy rằng, thay đổi giá trị trung bình tương ứng với việc dịch chuyển phân phối dọc theo *trục x*, đồng thời tăng giá trị phương sai sẽ trai rộng phân phối và làm giảm giá trị đỉnh của nó.

Để thấy rõ hơn mối quan hệ giữa hồi quy tuyến tính và hàm mất mát trung bình bình phương sai số (MSE), ta có thể giả định rằng các quan sát bắt nguồn từ những quan sát nhiễu, giá trị nhiễu này tuân theo phân phối chuẩn như sau:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \quad (6.1.8)$$

Do đó, chúng ta có thể viết *khả năng* thu được giá trị cụ thể của y khi biết trước \mathbf{x} thông qua:

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \quad (6.1.9)$$

Dựa vào *nguyên lý hợp lý cực đại*, giá trị tốt nhất của b và \mathbf{w} là những điểm giúp cực đại hóa *sự hợp lý* của bộ dữ liệu:

$$P(Y | X) = \prod_{i=1}^n p(y^{(i)} | \mathbf{x}^{(i)}). \quad (6.1.10)$$

Bộ ước lượng được chọn theo *nguyên lý hợp lý cực đại* được gọi là *bộ ước lượng hợp lý cực đại* (*Maximum Likelihood Estimators – MLE*). Trong khi việc tối đa hóa tích của nhiều hàm mũ có thể gặp khó khăn, chúng ta có thể đơn giản hóa phép tính mà không làm ảnh hưởng tới mục đích đề ra bằng cách tối đa hóa log của hàm hợp lý. Vì lý do lịch sử, các bài toán tối ưu thường được biểu diễn dưới dạng bài toán tối thiểu hóa thay vì tối đa hóa. Vì vậy chúng ta có thể tối thiểu hóa *hàm đối log hợp lý* (*Negative Log-Likelihood - NLL*) – $-\log p(\mathbf{y}|\mathbf{X})$ mà không cần thay đổi gì. Ta có:

$$-\log p(\mathbf{y}|\mathbf{X}) = \sum_{i=1}^n \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b \right)^2. \quad (6.1.11)$$

Bây giờ, ta chỉ cần thêm một giả định rằng: σ là một hằng số cố định. Do đó, ta có thể bỏ qua cụm đầu tiên bởi nó không phụ thuộc vào \mathbf{w} hoặc b . Khi đó, cụm thứ hai giống hệt hàm bình phương sai số đã được giới thiệu trên đây, nhưng với nhân tử hằng $\frac{1}{\sigma^2}$. May mắn là, lời giải trên không phụ thuộc vào σ . Điều này dẫn tới việc cực tiểu hóa bình phương sai số tương đương với việc ước lượng cực đại hợp lý cho mô hình dưới giả định có nhiều cộng Gaussian.

Từ Hồi quy Tuyến tính tới Mạng Học sâu

Cho đến nay, chúng ta mới chỉ đề cập về các hàm tuyến tính. Trong khi mạng nơ-ron có thể xấp xỉ rất nhiều họ mô hình, ta có thể bắt đầu coi mô hình tuyến tính như một mạng nơ-ron và biểu diễn nó theo ngôn ngữ của mạng nơ-ron. Để bắt đầu, hãy cùng viết lại mọi thứ theo ký hiệu ‘tầng’ (*layer*).

Giản đồ Mạng Nơ-ron

Những người làm học sâu thích vẽ giản đồ để trực quan hóa những gì đang xảy ra trong mô hình của họ. Trong *fig_single_neuron*, mô hình tuyến tính được minh họa như một mạng nơ-ron. Những giản đồ này chỉ ra cách kết nối (ở đây, mỗi đầu vào được kết nối tới đầu ra) nhưng không có giá trị của các trọng số và các hệ số điều chỉnh.

Fig. 6.1.0: Hồi quy tuyến tính là một mạng nơ-ron đơn tầng.

Vì chỉ có một nơ-ron tính toán (một nút) trong đồ thị (các giá trị đầu vào không cần tính mà được cho trước), chúng ta có thể coi mô hình tuyến tính như mạng nơ-ron với chỉ một nơ-ron nhân tạo duy nhất. Với mô hình này, mọi đầu vào đều được kết nối tới mọi đầu ra (trong trường hợp này chỉ có một đầu ra!), ta có thể coi phép biến đổi này là một *tầng kết nối đầy đủ*, hay còn gọi là *tầng kết nối dày đặc*. Chúng ta sẽ nói nhiều hơn về các mạng nơ-ron cấu tạo từ những tầng như vậy trong chương kế tiếp về mạng perceptron đa tầng.

Sinh vật học

Vì hồi quy tuyến tính (được phát minh vào năm 1795) được phát triển trước ngành khoa học thần kinh tính toán, nên việc mô tả hồi quy tuyến tính như một mạng nơ-ron có vẻ hơi ngược thời. Để hiểu tại sao nhà nghiên cứu sinh vật học/thần kinh học Warren McCulloch và Walter Pitts tìm đến các mô hình tuyến tính để làm điểm khởi đầu nghiên cứu và phát triển các mô hình nơ-ron nhân tạo, hãy xem ảnh của một nơ-ron sinh học tại *fig_Neuron*. Mô hình này bao gồm *sợi nhánh* (cổng đầu vào), *nhân tế bào* (bộ xử lý trung tâm), *sợi trực* (dây đầu ra), và *đầu cuối sợi trực* (cổng đầu ra), cho phép kết nối với các tế bào thần kinh khác thông qua *synapses*.

Fig. 6.1.1: Nơ-ron trong thực tế

Thông tin x_i đến từ các nơ-ron khác (hoặc các cảm biến môi trường như võng mạc) được tiếp nhận tại các sợi nhánh. Cụ thể, thông tin đó được nhân với các *trọng số* của synapses w_i để xác định mức ảnh hưởng của từng đầu vào (ví dụ: kích hoạt hoặc ức chế thông qua tích $x_i w_i$). Các đầu vào có trọng số đến từ nhiều nguồn được tổng hợp trong nhân tế bào dưới dạng tổng có trọng số $y = \sum_i x_i w_i + b$ và thông tin này sau đó được gửi đi để xử lý thêm trong sợi trục y , thường là sau một vài xử lý phi tuyến tính qua $\sigma(y)$. Từ đó, nó có thể được gửi đến đích (ví dụ, cơ bắp) hoặc được đưa vào một tế bào thần kinh khác thông qua các sợi nhánh.

Dựa trên các nghiên cứu thực tế về các hệ thống thần kinh sinh học, ta chắc chắn một điều rằng nhiều đơn vị như vậy khi được kết hợp với nhau theo đúng cách, cùng với thuật toán học phù hợp, sẽ tạo ra các hành vi thú vị và phức tạp hơn nhiều so với bất kỳ nơ-ron đơn lẻ nào có thể làm được.

Đồng thời, hầu hết các nghiên cứu trong học sâu ngày nay chỉ lấy một phần cảm hứng nhỏ từ ngành thần kinh học. Như trong cuốn sách kinh điển về AI *Trí tuệ Nhân tạo: Một hướng Tiếp cận Hiện đại* (?) của Stuart Russell và Peter Norvig, họ đã chỉ ra rằng: mặc dù máy bay có thể được lấy cảm hứng từ loài chim, ngành điều học không phải động lực chính làm đổi mới ngành hàng không trong nhiều thế kỷ qua. Tương tự, cảm hứng trong học sâu hiện nay chủ yếu đến từ ngành toán học, thống kê và khoa học máy tính.

Tổng kết

- Nguyên liệu của một mô hình học máy bao gồm dữ liệu huấn luyện, một hàm mất mát, một thuật toán tối ưu, và tất nhiên là cả chính mô hình đó.
- Vector hóa giúp mọi thứ trở nên dễ hiểu hơn (về mặt toán học) và nhanh hơn (về mặt lập trình).
- Tối thiểu hóa hàm mục tiêu và thực hiện tối đa hóa hàm hợp lý có ý nghĩa giống nhau.
- Các mô hình tuyến tính cũng là các mạng nơ-ron.

Bài tập

1. Giả sử ta có dữ liệu $x_1, \dots, x_n \in \mathbb{R}$. Mục tiêu của ta là đi tìm một hằng số b để tối thiểu hóa $\sum_i (x_i - b)^2$.
 - Tìm một công thức nghiệm cho giá trị tối ưu của b .
 - Bài toán và nghiệm của nó có liên hệ như thế nào tới phân phối chuẩn?
2. Xây dựng công thức nghiệm cho bài toán tối ưu hóa hồi quy tuyến tính với bình phương sai số. Để đơn giản hơn, bạn có thể bỏ qua hệ số điều chỉnh b ra khỏi bài toán (chúng ta có thể thực hiện việc này bằng cách thêm vào một cột toàn giá trị một vào X).
 - Viết bài toán tối ưu hóa theo ký hiệu ma trận-vector (xem tất cả các điểm dữ liệu như một ma trận và tất cả các giá trị mục tiêu như một vector).
 - Tính gradient của hàm mất mát theo w .
 - Tìm công thức nghiệm bằng cách giải phương trình gradient bằng không.
 - Khi nào phương pháp làm này tốt hơn so với sử dụng hạ gradient ngẫu nhiên? Khi nào phương pháp này không hoạt động?
3. Giả sử rằng mô hình nhiều điều khiển nhiều cộng ϵ là phân phối mũ, nghĩa là $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
 - Viết hàm đối log hợp lý của dữ liệu theo mô hình $-\log P(Y | X)$.

- Bạn có thể tìm ra nghiệm theo công thức không?
- Gợi ý là thuật toán hạ gradient ngẫu nhiên có thể giải quyết vấn đề này.
- Điều gì có thể sai ở đây (gợi ý - điều gì xảy ra gần điểm dừng khi chúng ta tiếp tục cập nhật các tham số). Bạn có thể sửa nó không?

Thảo luận

- Tiếng Anh⁷²
- Tiếng Việt⁷³

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Phạm Ngọc Bảo Anh
- Nguyễn Văn Tâm
- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Nguyễn Phan Hùng Thuận
- Vũ Hữu Tiệp
- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Tạ H. Duy Nguyên
- Bùi Nhật Quân
- Lê Gia Thiên Bửu
- Lý Phi Long
- Nguyễn Minh Thư
- Tạ Đức Huy
- Phạm Hồng Vinh
- Minh Trí Nguyễn
- Đoàn Võ Duy Thanh
- Vũ Hữu Tiệp
- Phạm Hồng Vinh

⁷² <https://discuss.mxnet.io/t/2331>

⁷³ <https://forum.machinelearningcoban.com/c/d2l>

- Trần Thị Hồng Hạnh
- Nguyễn Quang Hải
- Nguyễn Văn Tâm
-
- Nguyễn Phan Hùng Thuận
- Phạm Hồng Vinh
- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm
- Bùi Nhật Quân

6.1.2 Lập trình Hồi quy Tuyến tính từ đầu

Bây giờ bạn đã hiểu được điểm mấu chốt dằng sau thuật toán hồi quy tuyến tính, chúng ta đã có thể bắt đầu thực hành viết mã. Trong phần này, ta sẽ thực hiện toàn bộ phương pháp từ đầu, bao gồm: pipeline dữ liệu, mô hình, hàm mất mát và phương pháp tối ưu hạ gradient. Trong khi các framework học sâu hiện đại có thể tự động hóa gần như tất cả các công việc ở trên, thì việc lập trình mọi thứ lại từ đầu dường như chỉ để đảm bảo rằng bạn thực sự biết những gì bạn đang làm. Hơn nữa, việc hiểu rõ mọi thứ hoạt động như thế nào sẽ giúp ta rất nhiều trong những lúc cần tùy chỉnh các mô hình, tự định nghĩa lại các tầng riêng hay các hàm mất mát, v.v. Trong phần này, chúng ta chỉ dựa vào `ndarray` và `autograd`. Sau đó, chúng tôi sẽ giới thiệu một phương pháp triển khai chặt chẽ hơn, tận dụng các tính năng tuyệt vời của Gluon. Để bắt đầu, chúng ta cần nhập một vài gói thư viện cần thiết.

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
import random
npx.set_np()
```

Tạo tập dữ liệu

Để giữ cho mọi thứ đơn giản, chúng ta sẽ xây dựng một tập dữ liệu nhân tạo theo một mô hình tuyến tính với nhiễu cộng. Nhiệm vụ của chúng ta là khôi phục các tham số của mô hình này bằng cách sử dụng một tập hợp hữu hạn các mẫu có trong tập dữ liệu đó. Chúng ta sẽ sử dụng dữ liệu ít chiều để thuận tiện cho việc minh họa. Trong đoạn mã sau, chúng ta đã tạo một tập dữ liệu chứa 1000 mẫu, mỗi mẫu bao gồm 2 đặc trưng theo phân phối chuẩn hóa. Do đó, tập dữ liệu tổng hợp của chúng ta sẽ là một đối tượng $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

Các tham số đúng để tạo tập dữ liệu sẽ là $\mathbf{w} = [2, -3.4]^\top$ và $b = 4.2$ và các nhãn tổng hợp sẽ được tính dựa theo mô hình tuyến tính với nhiễu ϵ :

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \quad (6.1.12)$$

Bạn đọc có thể xem ϵ như là sai số tiềm ẩn của phép đo trên các đặc trưng và các nhãn. Chúng ta sẽ mặc định các giả định tiêu chuẩn đều thỏa mãn và vì thế ϵ tuân theo phân phối chuẩn với trung bình bằng 0. Để đơn giản, ta sẽ thiết lập độ lệch chuẩn của nó bằng 0.1. Đoạn mã nguồn sau sẽ sinh ra tập dữ liệu tổng hợp:

```
# Saved in the d2l package for later use
def synthetic_data(w, b, num_examples):
    """Generate y = X w + b + noise."""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

Lưu ý rằng mỗi hàng trong `features` chứa một điểm dữ liệu hai chiều và mỗi hàng trong `labels` chứa một giá trị mục tiêu một chiều (một số vô hướng).

```
print('features:', features[0], '\nlabel:', labels[0])
```

```
features: [2.2122064 1.1630787]
label: 4.662078
```

Bằng cách vẽ đồ thị phân tán với chiều thứ hai `features[:, 1]` và `labels`, ta có thể quan sát rõ mối tương quan giữa chúng.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```

Đọc từ tập dữ liệu

Nhắc lại rằng việc huấn luyện mô hình bao gồm tách tập dữ liệu thành nhiều phần (các minibatch), lân lượt đọc từng phần của tập dữ liệu mẫu, và sử dụng chúng để cập nhật mô hình của chúng ta. Vì quá trình này là cơ sở để huấn luyện các giải thuật học máy, ta nên định nghĩa một hàm để trộn và truy xuất dữ liệu trong các minibatch một cách tiện lợi.

Một hiện thực khả dĩ của chức năng này được minh họa qua hàm `data_iter` dưới đây. Hàm này lấy kích thước một batch, một ma trận đặc trưng và một vector các nhãn rồi sinh ra các minibatch có kích thước `batch_size`. Mỗi minibatch gồm một tuple các đặc trưng và nhãn.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

Lưu ý rằng thông thường chúng ta muốn dùng các minibatch có kích thước phù hợp để tận dụng tài nguyên phần cứng từ GPU cho việc thực hiện xử lý song song hiệu quả nhất. Vì mỗi mẫu có thể được mô hình xử lý

và tính đạo hàm riêng của hàm mất mát song song với nhau, GPUs cho phép ta xử lý hàng trăm mẫu cùng lúc với thời gian chỉ nhỉnh hơn một chút so với thời gian xử lý cho một mẫu duy nhất.

Để hiểu hơn, chúng ta hãy chạy đoạn chương trình để đọc và in ra batch đầu tiên của mẫu dữ liệu. Kích thước của các đặc trưng trong mỗi minibatch cho ta biết kích thước của batch lẫn kích thước của các đặc trưng đầu vào. Tương tự, tập minibatch của các nhãn sẽ có kích thước theo batch_size.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
[ [-0.6918596 -0.7476888 ]
 [ 0.2863085 -0.74571455]
 [ 1.4326776  1.3821784 ]
 [-0.26001498  1.8115263 ]
 [ 0.2444218 -0.68106437]
 [-0.594405   0.7975923 ]
 [ 1.2208143  0.34541664]
 [-1.4749662  1.5194443 ]
 [-1.0103831  0.8281874 ]
 [ 1.6323917 -0.96297354]]
 [ 5.3732486  7.2932053  2.3762138 -2.4844196  7.012023      0.30207413
  5.472156   -3.913631   -0.6321874  10.743488 ]
```

Khi chạy bộ duyệt, ta lấy từng minibatch cho đến khi đã lấy hết bộ dữ liệu. Mặc dù sử dụng bộ duyệt như trên phục vụ tốt cho công tác giảng dạy, nó lại không phải là cách hiệu quả và có thể khiến chúng ta gặp nhiều rắc rối trong thực tế. Ví dụ, nó buộc ta phải nạp toàn bộ dữ liệu vào bộ nhớ, do đó phải thực thi rất nhiều thao tác truy cập bộ nhớ ngẫu nhiên. Các bộ duyệt trong Apache MXNet lại khá hiệu quả khi chúng có thể xử lý cả dữ liệu lưu trữ trên tập tin lẫn các luồng dữ liệu.

Khởi tạo các tham số mô hình

Để tối ưu các tham số của dữ liệu bằng gradient, đầu tiên ta cần khởi tạo chúng. Trong đoạn mã dưới đây, ta khởi tạo các trọng số bằng cách lấy ngẫu nhiên các mẫu từ một phân phối chuẩn với giá trị trung bình bằng 0 và độ lệch chuẩn là 0.01, tiếp đó gán hệ số điều chỉnh b bằng 0.

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
```

Sau khi khởi tạo các tham số, bước tiếp theo là cập nhật chúng cho đến khi chúng ăn khớp với dữ liệu của ta đủ tốt. Mỗi lần cập nhật, ta tính gradient (đạo hàm nhiều biến) của hàm mất mát theo các tham số. Với gradient này, chúng ta có thể cập nhật mỗi tham số theo hướng giảm dần giá trị mất mát.

Bởi vì không ai muốn tính gradient bằng tay (việc này rất chán và dễ sai), ta sử dụng chương trình để tính gradient (autograd). Xem `sec_autograd` để có thêm chi tiết. Nhắc lại mục tính vi phân tự động, để autograd có thể lưu gradient vào một biến, ta cần gọi hàm `attach_grad`, khai báo và truyền biến để lưu trữ các gradients đó vào.

```
w.attach_grad()
b.attach_grad()
```

Định nghĩa mô hình

Tiếp theo, chúng ta cần xác định mô hình của mình dựa trên đầu vào và đầu ra của các tham số. Nhắc lại rằng để tính đầu ra của một mô hình tuyến tính, chúng ta có thể đơn giản là tính tích vô hướng ma trận-vector của các mẫu X và trọng số mô hình w , sau đó thêm vào hệ số điều chỉnh b với mỗi mẫu. Chú ý rằng $\text{np.dot}(X, w)$ dưới đây là một vector trong khi b là một số vô hướng. Cần nhớ rằng khi chúng ta tính tổng vector và số vô hướng, thì số vô hướng sẽ được thêm vào mỗi phần tử của vector.

```
# Saved in the d2l package for later use
def linreg(X, w, b):
    return np.dot(X, w) + b
```

Định nghĩa Hàm Mất mát

Để cập nhật mô hình ta cần tính gradient của hàm mất mát, vậy nên ta phải định nghĩa hàm mất mát trước tiên. Chúng ta sẽ sử dụng hàm mất mát bình phương (SE) như đã trình bày ở phần trước. Trong thực tế, chúng ta cần chuyển đổi giá trị nhãn đúng y sang kích thước của giá trị dự đoán y_{hat} . Kết quả trả về bởi hàm dưới đây cũng sẽ có kích thước như kích thước của y_{hat} .

```
# Saved in the d2l package for later use
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

Định nghĩa Thuật toán Tối ưu

Như đã thảo luận ở mục trước, hồi quy tuyến tính có một nghiệm (dạng đóng)[https://vi.wikipedia.org/wiki/Bi%E1%BB%83u_th%E1%BB%A9c_d%E1%BA%A1ng_%C4%91%C3%B3ng]. Tuy nhiên, đây không phải là một cuốn sách về hồi quy tuyến tính, mà là cuốn sách về học sâu. Vì không một mô hình nào khác được trình bày trong cuốn sách này có thể giải được bằng phương pháp phân tích, chúng tôi sẽ nhân đó giới thiệu với các bạn ví dụ đầu tiên về hạ gradient ngẫu nhiên (*stochastic gradient descent – SGD*)

Tại mỗi bước, sử dụng một batch được rút ngẫu nhiên từ mẫu, chúng ta sẽ ước tính được gradient của mất mát theo các tham số. Tiếp theo đó, chúng ta sẽ cập nhật các tham số (với một lượng nhỏ) theo chiều hướng làm giảm sự mất mát. Nhớ lại từ `sec_autograd` rằng sau khi chúng ta gọi ‘backward’, mỗi tham số (`param`) sẽ có gradient của nó lưu ở `param.grad`. Đoạn mã sau áp dụng cho việc cập nhật SGD, đưa ra một bộ các tham số, tốc độ học và kích cỡ batch. Kích cỡ của bước cập nhật được xác định bởi tốc độ học `lr`. Bởi vì các mất mát được tính dựa trên tổng các mẫu của batch, chúng ta chuẩn hóa kích cỡ bước cập nhật theo kích cỡ của batch (`batch_size`), sao cho độ lớn của một bước cập nhật thông thường không phụ thuộc nhiều vào kích cỡ batch.

```
# Saved in the d2l package for later use
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

Huấn luyện

Bây giờ, sau khi đã có tất cả các thành phần, chúng ta đã sẵn sàng để viết vòng lặp huấn luyện. Quan trọng nhất là bạn phải hiểu được rõ đoạn mã này bởi vì việc huấn luyện gần như tương tự thế này sẽ được lặp lại nhiều lần trong suốt quá trình chúng ta tìm hiểu và lập trình các thuật toán học sâu.

Trong mỗi vòng lặp, đầu tiên chúng ta sẽ lấy ra các minibatch dữ liệu và chạy nó qua mô hình để lấy ra tập kết quả dự đoán. Sau khi tính toán sự mất mát, chúng ta dùng hàm backward để bắt đầu lan truyền ngược qua mạng lưới, lưu trữ các gradient tương ứng với mỗi tham số trong từng thuộc tính .grad của chúng. Cuối cùng, chúng ta sẽ dùng thuật toán tối ưu sgd để cập nhật các tham số của mô hình. Từ đầu chúng ta đã đặt kích thước batch batch_size là 10, vậy nên mất mát \mathbb{I} cho mỗi minibatch có kích thước là (10,1).

Tóm lại, chúng ta sẽ thực thi vòng lặp sau:

- Khởi tạo bộ tham số (\mathbf{w}, b)
- Lặp lại cho tới khi hoàn thành
 - Tính gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w}, b)} \frac{1}{B} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$
 - Cập nhật bộ tham số $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Trong đoạn mã dưới đây, \mathbf{l} là một vector của các mất mát của từng mẫu trong minibatch. Vì \mathbf{l} không phải là biến vô hướng, chạy $\mathbf{l}.\text{backward}()$ sẽ cộng các phần tử trong \mathbf{l} để tạo ra một biến mới và sau đó mới tính gradient.

Với mỗi epoch (một lần chạy qua tập dữ liệu), chúng ta sẽ lặp qua toàn bộ tập dữ liệu (sử dụng hàm data_iter) cho đến khi đi qua toàn bộ mọi mẫu trong tập huấn luyện (giả định rằng số mẫu chia hết cho kích thước batch). Số epoch num_epochs và tốc độ học lr đều là siêu tham số, mà chúng ta đặt ở đây là tương ứng 3 và 0.03. Không may thay, việc lựa chọn siêu tham số thường không đơn giản và đòi hỏi một vài sự điều chỉnh bằng cách thử và sai. Bây giờ chúng ta sẽ bỏ qua những chi tiết này nhưng chúng ta sẽ xem lại chúng sau qua chap_optimization.

```
lr = 0.03 # Learning rate
num_epochs = 3 # Number of iterations
net = linreg # Our fancy linear model
loss = squared_loss # 0.5 (y-y')^2

for epoch in range(num_epochs):
    # Assuming the number of examples can be divided by the batch size, all
    # the examples in the training dataset are used once in one epoch
    # iteration. The features and tags of minibatch examples are given by X
    # and y respectively
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y) # Minibatch loss in X and y
            l.backward() # Compute gradient on l with respect to [w, b]
            sgd([w, b], lr, batch_size) # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

```
epoch 1, loss 0.025017
epoch 2, loss 0.000089
epoch 3, loss 0.000051
```

Trong trường hợp này, bởi vì chúng ta đã tự tổng hợp dữ liệu nên chúng ta biết chính xác giá trị đúng của các tham số. Vì vậy, chúng ta có thể đánh giá sự thành công của việc huấn luyện bằng cách so sánh giá trị đúng

của các tham số với những giá trị học được thông qua quá trình huấn luyện. Quả thật giá trị các tham số học được và giá trị chính xác của các tham số rất gần với nhau.

```
print('Error in estimating w', true_w - w.reshape(true_w.shape))
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [0.00030553 0.00045419]
Error in estimating b [0.00037384]
```

Lưu ý rằng chúng ta không nên thừa nhận có thể khôi phục giá trị của các tham số một cách chính xác. Điều này chỉ xảy ra với một số bài toán đặc biệt: các bài toán tối ưu lồi chặt với lượng dữ liệu “đủ” để đảm bảo rằng các mẫu nhiễu vẫn cho phép chúng ta khôi phục được các tham số. Trong hầu hết các trường hợp, điều này *không* xảy ra. Trên thực tế, các tham số của một mạng học sâu hiếm khi giống nhau (hoặc thậm chí gần nhau) giữa hai lần chạy khác nhau, trừ khi tất cả các điều kiện đều giống hệt nhau, bao gồm cả thứ tự mà dữ liệu được duyệt qua. Tuy nhiên, trong học máy, chúng ta thường ít quan tâm đến việc khôi phục chính xác giá trị của các tham số, mà quan tâm nhiều hơn đến bộ tham số nào sẽ dẫn tới việc dự đoán chính xác hơn. May mắn thay, thậm chí với những bài toán tối ưu khó, giải thuật hạ gradient ngẫu nhiên thường có thể tìm ra các lời giải đủ tốt, do một thực tế là, đối với các mạng học sâu, có thể tồn tại nhiều bộ tham số dẫn đến việc dự đoán chính xác.

Tóm tắt

Chúng ta đã thấy cách một mạng sâu có thể được triển khai và tối ưu hóa từ đầu chỉ với `ndarray` và `auto_grad`, mà không cần phải định nghĩa các tầng, các thuật toán tối ưu đặc biệt, v.v. Điều này chỉ mới chạm đến bề mặt của những gì mà ta có thể làm. Trong các phần sau, chúng tôi sẽ mô tả các mô hình bổ sung dựa trên các khái niệm vừa được giới thiệu và học cách triển khai chúng một cách chính xác hơn.

Bài tập

- Điều gì sẽ xảy ra nếu chúng ta khởi tạo các trọng số $w = 0$. Liệu thuật toán sẽ vẫn hoạt động chứ?
- Giả sử rằng bạn là [Georg Simon Ohm](#)⁷⁴ và bạn đang cố gắng nghĩ ra một mô hình giữa điện áp và dòng điện. Bạn có thể sử dụng `autograd` để học các tham số cho mô hình của bạn không?
- Bạn có thể sử dụng [Planck's Law](#)⁷⁵ để xác định nhiệt độ của một vật thể sử dụng mật độ năng lượng quang phổ không?
- Bạn có thể gặp phải những vấn đề gì nếu bạn muốn mở rộng `autograd` đến các đạo hàm bậc hai? Bạn sẽ sửa chúng bằng cách nào?
- Tại sao hàm `reshape` lại cần thiết trong hàm `squared_loss`?
- Thử nghiệm các tốc độ học khác nhau để xem giá trị hàm mất mát giảm nhanh đến mức nào.
- Nếu số lượng mẫu không thể chia hết cho kích thước batch, điều gì sẽ xảy ra với hành vi của hàm `data_iter`?

⁷⁴ https://en.wikipedia.org/wiki/Georg_Ohm

⁷⁵ https://en.wikipedia.org/wiki/Planck%27s_law

Thảo luận

- Tiếng Anh⁷⁶
- Tiếng Việt⁷⁷

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lý Phi Long
- Vũ Hữu Tiệp
- Phạm Hồng Vinh
- Nguyễn Văn Tâm
- Nguyễn Cảnh Thượng
- Phạm Hồng Vinh
-
- Nguyễn Lê Quang Nhật
- Dương Nhật Tân
- Vũ Hữu Tiệp
- Phạm Hồng Vinh
- Nguyễn Lê Quang Nhật
- Dương Nhật Tân
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Nguyễn Lê Quang Nhật
- Nguyễn Minh Thư
- Nguyễn Trường Phát
- Đinh Minh Tân
- Trần Thị Hồng Hạnh

⁷⁶ <https://discuss.mxnet.io/t/2332>

⁷⁷ <https://forum.machinelearningcoban.com/c/d2l>

6.1.3 Triển khai súc tích của Hồi quy Tuyến tính

Sự quan tâm nhiệt thành và rộng khắp với học sâu trong những năm gần đây đã tạo cảm hứng cho các công ty, học viện và những người đam mê tới học sâu phát triển nhiều framework mã nguồn mở hoàn thiện, giúp tự động hóa các công việc lặp đi lặp lại trong quá trình triển khai các thuật toán học dựa trên gradient. Trong chương trước, chúng ta chỉ dựa vào (i) ndarray để lưu dữ liệu và thực hiện tính toán đại số tuyến tính; và (ii) autograd để thực hiện tính đạo hàm. Trên thực tế, do các iterator dữ liệu, các hàm mượt mát, các bộ tối ưu và các tầng của mạng nơ-ron (thậm chí là toàn bộ kiến trúc) rất phổ biến, các thư viện hiện đại đã cài đặt sẵn những thành phần này cho chúng ta.

Mục này sẽ hướng dẫn bạn cách để xây dựng mô hình hồi quy tuyến tính trong phần sec_linear_scratch một cách súc tích với Gluon.

Tạo Tập dữ liệu

Chúng ta bắt đầu bằng việc tạo một tập dữ liệu như ở mục trước.

```
import d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

Đọc tập dữ liệu

Thay vì tự viết iterator riêng để đọc dữ liệu thì ta có thể gọi mô-đun data của Gluon để xử lý việc này. Bước đầu tiên sẽ là khởi tạo một `ArrayDataset`. Hàm tạo của đối tượng này sẽ lấy một hoặc nhiều `ndarray` làm đối số. Tại đây, ta truyền vào hàm hai đối số là `features` và `labels`. Kế tiếp, ta sử dụng `ArrayDataset` để khởi tạo một `DataLoader`, lớp này yêu cầu ta truyền vào một giá trị `batch_size` và giá trị Boolean `shuffle` để cho biết chúng ta có muốn `DataLoader` xáo trộn dữ liệu trên mỗi epoch (một lần duyệt qua toàn bộ tập dữ liệu) hay không.

```
# Saved in the d2l package for later use
def load_array(data_arrays, batch_size, is_train=True):
    """Construct a Gluon data loader"""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Bây giờ, ta có thể sử dụng `data_iter` theo cách tương tự như cách ta gọi hàm `data_iter` trong phần trước. Để biết rằng nó có hoạt động được hay không, ta có thể thử đọc và in ra minibatch đầu tiên.

```
for X, y in data_iter:
    print(X, '\n', y)
    break
```

```

[[ -0.9712193  0.5035904 ]
 [ 1.9984016  0.30198845]
 [-0.22556685 0.27800548]
 [-0.5836186 -0.07530449]
 [-0.44584066 -0.6373412 ]
 [-1.2106425 -1.006717 ]
 [ 0.16556863 -0.45846123]
 [ 0.41196275 0.18814288]
 [ 2.0431964 -0.97231627]
 [-1.7030034 -1.2475805 ]]
[ 0.53360987 7.185257    2.7957761   3.2921002   5.4724007   5.2098837
 6.0865664   4.3656664  11.580438   5.0381503 ]

```

Định nghĩa Mô hình

Khi ta lập trình hồi quy tuyến tính từ đầu (trong :numref:`sec_linear_scratch`), ta đã định nghĩa rõ ràng các tham số của mô hình và lập trình các tính toán cho giá trị đầu ra sử dụng các phép toán đại số tuyến tính cơ bản. Bạn *nên* biết cách để làm được điều này. Nhưng một khi mô hình trở nên phức tạp hơn và đồng thời khi bạn phải làm điều này gần như hàng ngày, bạn sẽ thấy vui mừng khi có sự hỗ trợ từ các thư viện. Tình huống này tương tự như việc lập trình blog của riêng bạn lại từ đầu. Làm điều này một hoặc hai lần thì sẽ bổ ích và mang tính hướng dẫn, nhưng bạn sẽ trở thành một nhà phát triển web “khó ố” nếu mỗi khi cần một trang blog bạn lại phải dành ra cả một tháng chỉ để phát triển lại từ đầu.

Đối với những tác vụ tiêu chuẩn, chúng ta có thể sử dụng các tầng đã được định nghĩa trước trong Gluon, điều này cho phép chúng ta tập trung vào những tầng được dùng để xây dựng mô hình hơn là việc phải tập trung vào cách lập trình các tầng đó. Để định nghĩa một mô hình tuyến tính, đầu tiên chúng ta cần nhập vào mô-đun `nn`, giúp ta định nghĩa một lượng lớn các tầng trong mạng nơ-ron (lưu ý rằng “`nn`” là chữ viết tắt của “neural network”). Đầu tiên ta sẽ định nghĩa một biến mô hình là `net`, tham chiếu đến một thực thể của lớp `Sequential`. Trong Gluon, `Sequential` định nghĩa một lớp chứa nhiều tầng được liên kết với nhau. Khi nhận được dữ liệu đầu vào, `Sequential` sẽ truyền dữ liệu vào tầng đầu tiên, kết quả đầu ra từ đó trở thành đầu vào của tầng thứ hai và cứ tiếp tục như thế ở các tầng kế tiếp. Trong ví dụ tiếp theo, mô hình chúng ta chỉ có duy nhất một tầng, vì vậy không nhất thiết phải sử dụng `Sequential`. Tuy nhiên vì hầu hết các mô hình chúng ta gặp phải trong tương lai đều có nhiều tầng, do đó dù sao cũng nên dùng để làm quen với quy trình làm việc tiêu chuẩn nhất.

```

from mxnet.gluon import nn
net = nn.Sequential()

```

Hãy cùng nhớ lại kiến trúc của mạng đơn tầng như đã trình bày tại `fig_singleneuron`. Tầng đó được gọi là *kết nối đầy đủ* bởi vì mỗi đầu vào được kết nối lần lượt với từng đầu ra bằng một phép nhân ma trận với vector. Trong Gluon, tầng kết nối đầy đủ được định nghĩa trong lớp `Dense`. Bởi vì chúng ta chỉ mong xuất ra một số vô hướng duy nhất, nên ta gán giá trị là 1.

Fig. 6.1.2: Hồi quy tuyến tính là một mạng nơ-ron đơn tầng.

```

net.add(nn.Dense(1))

```

Để thuận tiện, điều đáng chú ý là Gluon không yêu cầu chúng ta chỉ định kích thước đầu vào mỗi tầng. Nên tại đây, chúng ta không cần thiết cho Gluon biết có bao nhiêu đầu vào cho mỗi tầng tuyến tính. Khi chúng ta

cố gắng truyền dữ liệu qua mô hình lần đầu tiên, ví dụ: khi chúng ta thực hiện `net(X)` sau đó, Gluon sẽ tự động suy ra số lượng đầu vào cho mỗi tầng. Chúng ta sẽ mô tả cách hoạt động của cơ chế này một cách chi tiết hơn trong chương “Tính toán trong Học sâu”.

Khởi tạo tham số mô hình

Trước khi sử dụng `net`, chúng ta cần phải khởi tạo tham số cho mô hình, chẳng hạn như trọng số và hệ số điều chỉnh trong mô hình hồi quy tuyến tính. Chúng ta sẽ nhập mô-đun `initializer` từ MXNet. Mô-đun này cung cấp nhiều phương thức khác nhau để khởi tạo tham số cho mô hình. Gluon cho phép dùng `init` như một cách ngắn gọn (viết tắt) để truy cập đến gói `initializer`. Bằng cách gọi `init.Normal(sigma=0.01)`, chúng ta sẽ khởi tạo ngẫu nhiên các `trọng số` từ một phân phối chuẩn với trung bình bằng 0 và độ lệch chuẩn bằng 0.01. Mặc định, tham số `hệ số điều chỉnh` sẽ được khởi tạo bằng không. Cả hai vector trọng số và hệ số điều chỉnh sẽ có gradient kèm theo.

```
from mxnet import init  
net.initialize(init.Normal(sigma=0.01))
```

Đoạn mã nguồn trên trông khá đơn giản nhưng bạn đọc hãy chú ý một vài điểm khác thường ở đây. Chúng ta khởi tạo các tham số cho một mạng mà thậm chí Gluon chưa hề biết số chiều của đầu vào là bao nhiêu! Nó có thể là 2 trong trường hợp của chúng ta nhưng cũng có thể là 2000. Gluon khiến chúng ta không cần bận tâm về điều này bởi ở hậu trường, quá trình khởi tạo thực sự vẫn đang bị trì hoãn. Quá trình khởi tạo thực sự chỉ bắt đầu khi chúng ta truyền dữ liệu vào mạng lần đầu tiên. Hãy ghi nhớ rằng, do các tham số chưa thực sự được khởi tạo, chúng ta không thể truy cập hoặc thao tác với chúng.

Định nghĩa Hàm mất mát

Trong Gluon, mô-đun `loss` định nghĩa các hàm mất mát khác nhau. Chúng ta sẽ sử dụng mô-đun `loss` được thêm vào dưới tên gọi là `gloss`, để tránh nhầm lẫn nó với biến đang giữ hàm mất mát mà ta đã chọn. Trong ví dụ này, chúng ta sẽ sử dụng triển khai trong Gluon của mất mát bình phương (`L2Loss`).

```
from mxnet.gluon import loss as gloss  
loss = gloss.L2Loss() # The squared loss is also known as the L2 norm loss
```

Định nghĩa Thuật toán Tối ưu hóa

Minibatch SGD và các biến thể liên quan đều là các công cụ chuẩn cho việc tối ưu hóa những mạng nơ-ron và vì vậy Gluon hỗ trợ SGD cùng với một số biến thể của thuật toán này thông qua lớp `Trainers` của nó. Khi chúng ta khởi tạo lớp `Trainer`, ta sẽ chỉ định các tham số để tối ưu hóa (có thể lấy từ các mạng của chúng ta thông qua `net.collect_params()`), thuật toán tối ưu hóa mà chúng ta muốn sử dụng (`sgd`), và một từ điển siêu tham số theo yêu cầu bởi thuật toán tối ưu hóa của chúng ta. SGD chỉ yêu cầu rằng chúng ta sẽ đặt giá trị `learning_rate`, (ở đây chúng ta đặt nó bằng 0.03).

```
from mxnet import gluon  
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

Huấn Luyện

Bạn có thể thấy rằng việc thể hiện mô hình của chúng ta thông qua Gluon đòi hỏi tương đối ít các dòng lệnh. Chúng ta không cần phải phân bổ các tham số một cách riêng lẻ, định nghĩa hàm số mất mát, hay triển khai hạ gradient ngẫu nhiên. Lợi ích Gluon mang lại càng nhiều khi chúng ta làm với những mô hình càng phức tạp. Tuy nhiên, một khi ta nắm được rõ các điều cơ bản, vòng huấn luyện tự thân nó sẽ trông rất giống với những gì chúng ta tự thực hiện từ đầu.

Để nhắc lại: cho một vài epoch, chúng ta sẽ duyệt qua toàn bộ tập dữ liệu (train_data) bằng cách lấy theo từng minibatch của dữ liệu đầu vào cộng với các nhãn gốc tương ứng. Đối với mỗi minibatch, chúng ta cần tuân thủ theo trình tự:

- Đưa ra dự đoán bằng cách gọi net (X) và tính giá trị mất mát l (theo chiều thuận).
- Tính gradient bằng cách gọi l.backward () (theo chiều ngược).
- Cập nhật các tham số của mô hình bằng cách gọi bộ tối ưu hoá SGD (chú ý rằng trainer đã biết được các tham số nào cần tối ưu, nên chúng ta chỉ cần truyền vô thêr của minibatch).

Để có được độ đo lường chính xác, chúng ta tính sự mất mát sau mỗi epoch và in nó ra màn hình trong lúc thực thi.

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    l = loss(net(features), labels)
    print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
```

```
epoch 1, loss: 0.024927
epoch 2, loss: 0.000090
epoch 3, loss: 0.000051
```

Dưới đây, chúng ta so sánh các tham số của mô hình đã học thông qua việc huấn luyện trên tập dữ liệu hữu hạn và các tham số thực sự tạo ra tập dữ liệu. Để truy cập các tham số bằng Gluon, trước hết ta lấy tầng ta cần từ net, sau đó truy cập trọng số (weight) và hệ số điều chỉnh (bias) của tầng đó. Để truy cập giá trị mỗi tham số dưới dạng một mảng ndarray, ta sử dụng phương thức data. Giống như cách xây dựng từ đầu, các tham số ước lượng tìm được gần với giá trị chính xác của chúng.

```
w = net[0].weight.data()
print('Error in estimating w', true_w.reshape(w.shape) - w)
b = net[0].bias.data()
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [[0.00035763 0.00011468]]
Error in estimating b [0.000453]
```

Tóm tắt

- Sử dụng Gluon giúp việc mô tả các mô hình ngắn gọn hơn nhiều.
- Trong Gluon, mô-đun `data` cung cấp các công cụ để xử lý dữ liệu, mô-đun `nn` định nghĩa một lượng lớn các tầng cho mạng nơ-ron, và mô-đun `loss` cho phép ta thiết lập nhiều hàm mất mát phổ biến.
- Mô-đun `initializer` của MXNet cung cấp nhiều phương thức khác nhau để khởi tạo tham số cho mô hình.
- Kích thước và dung lượng lưu trữ sẽ được suy ra tự động (nhưng cần thận đừng thử truy xuất các tham số trước khi chúng được khởi tạo).

Bài tập

1. Nếu thay thế `l = loss(output, y)` bằng `l = loss(output, y).mean()`, chúng ta cần đổi `trainer.step(batch_size)` thành `trainer.step(1)` để phần mã nguồn này hoạt động giống như trước. Tại sao lại thế?
2. Bạn hãy xem lại tài liệu về MXNet để biết các hàm mất mát và các phương thức khởi tạo được cung cấp trong hai mô-đun `gluon.loss` và `init`. Thay thế hàm mất mát đang sử dụng bằng hàm mất mát Huber.
3. Làm thế nào để truy cập gradient của `dense.weight`?

Thảo luận

- Tiếng Anh⁷⁸
- Tiếng Việt⁷⁹

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Lý Phi Long
- Phạm Đăng Khoa
- Lê Khắc Hồng Phúc
- Dương Nhật Tân
- Nguyễn Văn Tâm
- Lê Khắc Hồng Phúc

⁷⁸ <https://discuss.mxnet.io/t/2333>

⁷⁹ <https://forum.machinelearningcoban.com/c/d2l>

- Đoàn Võ Duy Thanh
- Bùi Nhật Quân
- Bùi Nhật Quân
-

6.1.4 Hồi quy Softmax

Trong `sec_linear_regression`, chúng ta đã giới thiệu về hồi quy tuyến tính, tự xây dựng mô hình hồi quy tuyến tính từ đầu trong `sec_linear_scratch` và xây dựng mô hình hồi quy tuyến tính một lần nữa sử dụng Gluon trong `sec_linear_gluon` để thực hiện phần việc nặng nhọc.

Hồi quy là công cụ đắc lực có thể sử dụng khi ta muốn trả lời câu hỏi *bao nhiêu?*. Nếu bạn muốn dự đoán một ngôi nhà sẽ được bán với giá bao nhiêu tiền (*Đô la*), hay số trận thắng mà một đội bóng có thể đạt được, hoặc số ngày một bệnh nhân phải điều trị nội trú trước khi được xuất viện, thì có lẽ bạn đang cần một mô hình hồi quy.

Trong thực tế, chúng ta thường quan tâm đến việc phân loại hơn: không phải câu hỏi *bao nhiêu?* mà là *loại nào?*

- Email này có phải thư rác/lừa đảo hay không?
- Khách hàng này nhiều khả năng *đăng ký* hay *không đăng ký* một dịch vụ thuê bao?
- Hình ảnh này mô tả một con lừa, một con cún, một con mèo hay một con gà trống?
- Aston có khả năng xem bộ phim nào tiếp theo nhất?

Thông thường, những người làm về học máy dùng từ *phân loại* để mô tả hai bài toán khác nhau đôi chút: (i) ta chỉ quan tâm đến việc gán *cứng* một danh mục cho mỗi ví dụ: là cún, là gà, hay là mèo?; và (ii) ta muốn *gán mềm* tất cả các danh mục cho mỗi ví dụ, tức đánh giá xác suất một ví dụ rơi vào từng danh mục khả dĩ: là cún (92%), là gà (1%), là mèo (7%). Sự khác biệt này thường không rõ ràng, một phần bởi vì thông thường, ngay cả khi chúng ta chỉ quan tâm đến việc gán cứng, chúng ta vẫn sử dụng các mô hình có khả năng thực hiện các phép gán mềm.

Bài toán Phân loại

Để khởi động, ta hãy bắt đầu với bài toán phân loại hình ảnh đơn giản. Ở đây, mỗi đầu vào bao gồm một ảnh xám 2×2 . Bằng cách biểu diễn mỗi giá trị điểm ảnh bởi một số vô hướng, ta thu được bốn đặc trưng x_1, x_2, x_3, x_4 . Hơn nữa, hãy giả sử rằng mỗi hình ảnh đều thuộc về một trong các danh mục “mèo”, “gà” và “chó”.

Tiếp theo, ta cần phải chọn cách biểu diễn nhãn. Ta có hai cách làm hiển nhiên. Cách tự nhiên nhất có lẽ là chọn $y \in \{1, 2, 3\}$ lần lượt ứng với {chó, mèo, gà}. Đây là một cách *lưu trữ* thông tin tuyệt vời trên máy tính. Nếu các danh mục có một thứ tự tự nhiên giữa chúng, chẳng hạn như {trẻ sơ sinh, trẻ tập đi, thiếu niên, thanh niên, người trưởng thành, người cao tuổi}, sẽ là tự nhiên hơn nếu coi bài toán này là một bài toán hồi quy và nhãn sẽ được giữ nguyên dưới dạng số.

Nhưng nhìn chung các bài toán phân loại không có các lớp tuân theo một trật tự tự nhiên nào. May mắn thay, các nhà thông kê từ lâu đã tìm ra một cách đơn giản để có thể biểu diễn dữ liệu danh mục: *biểu diễn One-hot*. Biểu diễn One-hot là một vector với số lượng thành phần bằng số danh mục mà ta có. Thành phần tương ứng với từng danh mục cụ thể sẽ được gán là 1 và tất cả các thành phần khác sẽ được gán là 0.

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \quad (6.1.13)$$

Trong trường hợp này, y sẽ là một vector 3 chiều, với $(1, 0, 0)$ tương ứng với “mèo”, $(0, 1, 0)$ ứng với “gà” và $(0, 0, 1)$ ứng với “chó”.

Kiến trúc mạng

Để tính xác suất có điều kiện ứng với mỗi lớp, chúng ta cần một mô hình có nhiều đầu ra, với một đầu ra cho mỗi lớp. Để phân loại với các mô hình tuyến tính, chúng ta cần số hàm tuyến tính nhiều như số đầu ra. Mỗi đầu ra sẽ tương ứng với hàm tuyến tính của chính nó. Trong trường hợp này, vì có 4 đặc trưng và 3 đầu ra, chúng ta sẽ cần 12 số vô hướng để thể hiện các trọng số, (w với các chỉ số dưới) và 3 số vô hướng để thể hiện các hệ số điều chỉnh (b với các chỉ số dưới). Chúng ta sẽ tính ba *logits*, o_1, o_2 , và o_3 , cho mỗi đầu vào:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \quad (6.1.14)$$

Chúng ta có thể mô tả phép tính này với biểu đồ mạng nơ-ron được thể hiện trong : num-ref:fig_softmaxreg. Như hồi quy tuyến tính, hồi quy softmax cũng là một mạng nơ-ron đơn tầng. Vì vì sự tính toán của mỗi đầu ra, o_1, o_2 , và o_3 , phụ thuộc vào tất cả đầu vào, x_1, x_2, x_3 , và x_4 , tầng đầu ra của hồi quy softmax cũng có thể được xem như một tầng kết nối đầy đủ.

Fig. 6.1.3: Hồi quy softmax là một mạng nơ-ron đơn tầng

Để biểu diễn mô hình gọn hơn, chúng ta có thể sử dụng ký hiệu đại số tuyến tính. Ở dạng vector, ta có $\mathbf{o} = \mathbf{Wx} + \mathbf{b}$, một dạng phù hợp hơn cho cả toán và lập trình. Chú ý rằng chúng ta đã tập hợp tất cả các trọng số vào một ma trận 3×4 và với một mảng cho trước \mathbf{x} , các đầu ra được tính bởi tích ma trận-vector của các trọng số và đầu vào cộng với vector hệ số điều chỉnh \mathbf{b} .

Hàm Softmax

Chúng ta sẽ xem các giá trị đầu ra của mô hình là các giá trị xác suất. Ta sẽ tối ưu hóa các tham số của mô hình sao cho khả năng xuất hiện dữ liệu quan sát được là cao nhất. Sau đó, ta sẽ đưa ra dự đoán bằng cách đặt ngưỡng xác suất, ví dụ dự đoán nhãn đúng là nhãn có xác suất cao nhất (dùng hàm *argmax*).

Nói một cách chính quy hơn, ta mong muốn diễn dịch kết quả \hat{y}_k là xác suất để một điểm dữ liệu cho trước thuộc về một lớp k nào đó. Sau đó, ta có thể chọn lớp cho điểm đó tương ứng với giá trị lớn nhất mà mô hình dự đoán $\text{argmax}_k y_k$. Ví dụ, nếu \hat{y}_1, \hat{y}_2 và \hat{y}_3 lần lượt là 0.1, 0.8, and 0.1, thì ta có thể dự đoán điểm đó thuộc về lớp số 2 là “gà” (ứng với trong ví dụ trước).

Bạn có thể muốn đề xuất rằng ta có thể lấy trực tiếp logit o làm đầu ra để tiến hành dự đoán. Tuy nhiên, tại đây ta có một vài vấn đề khi lấy kết quả trực tiếp của tầng tuyến tính như một kết quả cho xác suất. Bởi vì, không có bất cứ điều kiện nào để ràng buộc tổng của những con số này bằng 1. Hơn nữa, tùy thuộc vào đầu vào mà ta có thể nhận được giá trị âm. Các điều kể trên đã vi phạm vào các tiêu đề cơ bản của xác suất đã được nhắc đến trong *sec_prob*

Để có thể diễn dịch kết quả đầu ra là xác xuất, ta phải đảm bảo rằng các kết quả không âm và tổng của chúng phải bằng 1 (điều này phải đúng trên cả dữ liệu mới). Hơn nữa, ta cần một hàm mục tiêu trong quá trình huấn luyện để cho mô hình có thể ước lượng xác suất một cách chính xác. Trong tất cả các trường hợp, khi kết quả phân lớp cho ra xác suất là 0.5 thì ta hy vọng phân nửa số mẫu đó *thực sự* thuộc về đúng lớp được dự đoán. Đây được gọi là *hiệu chuẩn*.

Hàm softmax, được phát minh vào năm 1959 bởi nhà khoa học xã hội R Duncan Luce trong nhánh *mô hình lựa chọn*, thỏa mãn chính xác những điều trên. Để biến đổi kết quả logit thành kết quả không âm và có tổng là 1, trong khi vẫn giữ tính chất khả vi, đầu tiên ta cần lấy hàm mũ cho từng logit (để chắc chắn chúng không âm) và sau đó ta chia cho tổng của chúng (để chắc rằng tổng của chúng luôn bằng 1).

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \quad (6.1.15)$$

Để thấy rằng $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ với $0 \leq \hat{y}_i \leq 1$ với mọi i . Do đó, $\hat{\mathbf{y}}$ là phân phối xác suất phù hợp và các giá trị của $\hat{\mathbf{y}}$ có thể được hiểu theo đó. Lưu ý rằng hàm softmax không thay đổi thứ tự giữa các logit và do đó ta vẫn có thể chọn ra lớp phù hợp nhất bằng cách:

$$\hat{i}(\mathbf{o}) = \underset{i}{\operatorname{argmax}} o_i = \underset{i}{\operatorname{argmax}} \hat{y}_i. \quad (6.1.16)$$

Các logit \mathbf{o} đơn giản chỉ là các giá trị trước khi cho qua hàm softmax để xác định xác xuất thuộc về mỗi danh mục. Tóm tắt lại, ta có ký hiệu dưới dạng vector như sau: $\mathbf{o}^{(i)} = \mathbf{Wx}^{(i)} + \mathbf{b}$, với $\hat{\mathbf{y}}^{(i)} = \text{softmax}(\mathbf{o}^{(i)})$.

Vector hóa Minibatch

Để cải thiện hiệu suất tính toán và tận dụng GPU, ta thường phải thực hiện các phép tính vector cho các minibatch dữ liệu. Giả sử, ta có một minibatch \mathbf{X} của mẫu với số chiều d và kích cỡ batch là n . Thêm vào đó, chúng ta có q lớp đầu ra. Như vậy, minibatch đặc trưng \mathbf{X} sẽ thuộc $\mathbb{R}^{n \times d}$, trọng số $\mathbf{W} \in \mathbb{R}^{d \times q}$, và độ chêch sẽ thỏa mãn $\mathbf{b} \in \mathbb{R}^q$.

$$\begin{aligned} \mathbf{O} &= \mathbf{WX} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}). \end{aligned} \quad (6.1.17)$$

Việc tăng tốc diễn ra chủ yếu tại tích ma trận - ma trận \mathbf{WX} so với tích ma trận - vector nếu chúng ta xử lý từng mẫu một. Bản thân softmax có thể được tính bằng lũy thừa tất cả các mục trong \mathbf{O} và sau đó chuẩn hóa chúng theo tổng.

Hàm mất mát

Tiếp theo, chúng ta cần một *hàm mất mát* để đánh giá chất lượng dự đoán xác suất của mình. Chúng ta sẽ dựa trên *hợp lý cực đại*, khái niệm tương tự đã gặp phải khi đưa ra biện minh xác suất cho mục tiêu bình phương nhỏ nhất trong hồi quy tuyến tính (`sec_linear_regression`).

Log hợp lý (Log-likelihood)

Hàm softmax cho chúng ta một vector $\hat{\mathbf{y}}$, có thể được hiểu như các xác suất có điều kiện của từng lớp biết đầu vào là x . Ví dụ: $\hat{y}_1 = \hat{P}(y = \text{cat} | \mathbf{x})$. Để biết các ước lượng có sát với thực tế hay không, ta kiểm tra xác suất mà mô hình gán cho lớp *thật sự* khi biết các đặc trưng.

$$P(Y | X) = \prod_{i=1}^n P(y^{(i)} | x^{(i)}) \text{ and thus } -\log P(Y | X) = \sum_{i=1}^n -\log P(y^{(i)} | x^{(i)}). \quad (6.1.18)$$

Cực đại hoá $P(Y | X)$ (và vì vậy tương đương với cực tiểu hóa $-\log P(Y | X)$) giúp việc dự đoán nhẫn tốt hơn. Điều này dẫn đến hàm mất mát (chúng tôi lược bỏ chỉ số trên (i) để tránh sự rối rắm về kí hiệu):

$$l = -\log P(y | x) = -\sum_j y_j \log \hat{y}_j. \quad (6.1.19)$$

Bởi vì những lí do sẽ được giải thích trong chốc lát, hàm mất mát này thường được gọi là mất mát *entropy chéo*. Ở đây, chúng ta đã sử dụng nó bằng cách xây dựng \hat{y} giống như một phân phối xác suất rời rạc và vector y là một vector one-hot. Vì thế, tổng các số hạng với chỉ số j sẽ biến mất ngoại trừ duy nhất một số hạng. Bởi mọi \hat{y}_j đều là xác suất, log của chúng không bao giờ lớn hơn 0. Vì vậy, hàm mất mát sẽ không thể giảm thêm được nữa nếu chúng ta dự đoán chính xác y với độ chắc chắn tuyệt đối, tức $P(y | x) = 1$ cho nhẫn đúng. Chú ý rằng điều này thường không khả thi. Ví dụ, nhẫn bị nhiễu sẽ xuất hiện trong tập dữ liệu (một vài mẫu bị dán nhầm nhẫn). Điều này cũng khó xảy ra khi những đặc trưng đầu vào không chứa đủ thông tin để phân loại các mẫu một cách hoàn hảo.

Softmax và Đạo hàm

Vì softmax và hàm mất mát softmax rất phổ biến, nên việc hiểu cách tính giá trị các hàm này sẽ có ích về sau. Thay o vào định nghĩa của hàm mất mát l và dùng định nghĩa của softmax, ta được:

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j. \quad (6.1.20)$$

Để hiểu rõ hơn, hãy cùng xét đạo hàm riêng của l theo o . Ta có:

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = P(y = j | x) - y_j. \quad (6.1.21)$$

Nói cách khác, gradient chính là hiệu giữa xác suất mô hình gán cho lớp đúng $P(y | x)$, và nhẫn của dữ liệu y . Điều này cũng khá giống như trong bài toán hồi quy, khi gradient là hiệu giữa dữ liệu quan sát được y và kết quả ước lượng \hat{y} . Điều này không phải ngẫu nhiên. Trong mọi mô hình [họ lũy thừa](#)⁸⁰, gradient của hàm log hợp lý đều có dạng như thế này. Điều này giúp cho việc tính toán gradient trong thực tế trở nên dễ dàng hơn.

Hàm mất mát Entropy Chéo

Giờ hãy xem xét trường hợp mà ta quan sát được toàn bộ phân phối của đầu ra thay vì chỉ một giá trị đầu ra duy nhất. Ta có thể biểu diễn y giống hệt như trước. Sự khác biệt duy nhất là thay vì có một vector chỉ chứa các phần tử nhị phân, giả sử như $(0, 0, 1)$, giờ ta có một vector xác suất tổng quát, ví dụ như $(0.1, 0.2, 0.7)$. Các công thức toán học ta dùng trước đó để định nghĩa hàm mất mát l vẫn áp dụng tốt ở đây, chẳng qua ý tưởng của nó bây giờ khái quát hơn một chút. Nó là giá trị kỳ vọng của hàm mất mát trên phân phối của nhẫn.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_j y_j \log \hat{y}_j. \quad (6.1.22)$$

Hàm mất mát này được gọi là hàm mất mát entropy chéo và nó là một trong những hàm mất mát phổ biến nhất dùng cho bài toán phân loại đa lớp. Ta có thể làm sáng tỏ cái tên entropy chéo bằng việc giới thiệu các kiến thức cơ bản trong lý thuyết thông tin.

⁸⁰ https://en.wikipedia.org/wiki/Exponential_family

Lý thuyết Thông tin Cơ bản

Lý thuyết thông tin giải quyết các bài toán mã hóa, giải mã, truyền tải và xử lý thông tin (hay còn được gọi là dữ liệu) dưới dạng súc tích nhất có thể.

Entropy

Ý tưởng cốt lõi trong lý thuyết thông tin chính là việc định lượng lượng thông tin chứa trong dữ liệu. Giá trị định lượng này chỉ ra giới hạn tối đa cho khả năng cô đọng dữ liệu của chúng ta (khi tìm biểu diễn ngắn gọn nhất mà không mất thông tin). Giá trị định lượng này gọi là *entropy*⁸¹, xác định trên phân phối p của bộ dữ liệu, được định nghĩa bằng phương trình dưới đây:

$$H[p] = \sum_j -p(j) \log p(j). \quad (6.1.23)$$

Một định lý căn bản của lý thuyết thông tin là để có thể biểu diễn dữ liệu thu thập ngẫu nhiên từ phân phối p , chúng ta cần sử dụng ít nhất $H[p]$ “nat”. “nat” là đơn vị biểu diễn dữ liệu sử dụng cơ số e , tương tự với bit biểu diễn dữ liệu sử dụng cơ số 2. Một nat bằng $\frac{1}{\log(2)} \approx 1.44$ bit. $H[p]/2$ thường được gọi là entropy nhị phân.

Tự lượng thông tin

Có lẽ bạn sẽ tự hỏi việc cô đọng dữ liệu thì liên quan gì với việc đưa ra dự đoán? Hãy tưởng tượng chúng ta có một dòng chảy (stream) dữ liệu mà ta muốn nén lại. Nếu chúng ta luôn có thể dễ dàng đoán được đơn vị dữ liệu (token) kế tiếp thì dữ liệu này rất dễ nén! Ví như tất cả các đơn vị dữ liệu trong dòng dữ liệu luôn có một giá trị cố định thì đây là một dòng dữ liệu tẻ nhạt! Không những tẻ nhạt, mà nó còn dễ đoán nữa. Bởi vì chúng luôn có cùng giá trị, ta sẽ không phải truyền bất cứ thông tin nào để trao đổi nội dung của dòng dữ liệu này. Dễ đoán thì cũng dễ nén là vậy.

Tuy nhiên, nếu ta không thể dự đoán một cách hoàn hảo cho mỗi sự kiện, thì thi thoảng ta sẽ thấy ngạc nhiên. Sự ngạc nhiên trong chúng ta sẽ lớn hơn khi ta gán một xác suất thấp hơn cho sự kiện. Vì nhiều lý do mà chúng ta sẽ nghiên cứu trong phần phụ lục, Claude Shannon đã đưa ra giải pháp $\log(1/p(j)) = -\log p(j)$ để định lượng *sự ngạc nhiên* của một người lúc quan sát sự kiện j được gán cho một xác suất (chủ quan) $p(j)$. Entropy sau đó là *ngạc nhiên kỳ vọng* khi ai đó gán xác suất chính xác (mà thực sự khớp với quá trình sinh dữ liệu). Entropy của dữ liệu sau đó là điều ít ngạc nhiên nhất mà nó có thể trở thành (trong kỳ vọng).

Xem xét lại Entropy chéo

Vậy, nếu entropy là mức độ ngạc nhiên trải nghiệm bởi một người nắm rõ xác suất thật, thế thì bạn có thể băn khoăn rằng *entropy chéo là gì?* Entropy chéo từ p đến q , ký hiệu $H(p, q)$, là sự ngạc nhiên kỳ vọng của một người quan sát với các xác suất chủ quan q đối với việc nhìn thấy dữ liệu mà đã được thật sự sinh ra dựa trên các xác suất p . Giá trị entropy chéo thấp nhất có thể đạt được khi $p = q$. Trong trường hợp này, entropy chéo từ p đến q là $H(p, p) = H(p)$. Liên hệ điều này lại với mục tiêu phân loại của chúng ta, thậm chí khi ta có khả năng dự đoán tốt nhất có thể và cho rằng việc này là khả thi, thì ta sẽ không bao giờ đạt đến mức hoàn hảo. Mất mát của ta bị cản dưới bởi entropy được cho bởi các phân phối thực tế có điều kiện $P(\mathbf{y} | \mathbf{x})$.

⁸¹ <https://en.wikipedia.org/wiki/Entropy>

Phân kì Kullback Leibler

Có lẽ cách thông thường nhất để đo lường khoảng cách giữa hai phân phối là tính toán *phân kì Kullback Leibler* $D(p\|q)$. Phân kì Kullback Leibler đơn giản là sự khác nhau giữa entropy chéo và entropy, có nghĩa là giá trị entropy chéo bổ sung phát sinh so với giá trị tối thiểu không thể giảm được mà nó có thể nhận:

$$D(p\|q) = H(p, q) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)}. \quad (6.1.24)$$

Lưu ý rằng trong bài toán phân loại, ta không biết giá trị thật của p , vì thế mà ta không thể tính toán entropy trực tiếp được. Tuy nhiên, bởi vì entropy nằm ngoài tầm kiểm soát của chúng ta, việc giảm thiểu $D(p\|q)$ so với q là tương đương với việc giảm thiểu mất mát entropy chéo.

Tóm lại, chúng ta có thể nghĩ đến mục tiêu của phân loại entropy chéo theo hai hướng: (i) tối đa hóa khả năng xảy ra của dữ liệu được quan sát; và (ii) giảm thiểu sự ngạc nhiên của ta (cũng như số lượng các bit) cần thiết để truyền đạt các nhãn.

Sử dụng Mô hình để Dự đoán và Đánh giá

Sau khi huấn luyện mô hình hồi quy softmax, với các đặc trưng đầu vào bất kì, chúng ta có thể dự đoán xác suất đầu ra ứng với mỗi lớp. Thông thường, chúng ta sử dụng lớp với xác suất dự đoán cao nhất làm lớp đầu ra. Một dự đoán được xem là chính xác nếu nó trùng khớp hay tương thích với lớp thật sự (nhãn). Ở phần tiếp theo của thí nghiệm, chúng ta sẽ sử dụng độ chính xác để đánh giá chất lượng của mô hình. Giá trị này là tỉ lệ giữa số mẫu được dự đoán chính xác so với tổng số mẫu được dự đoán.

Tóm tắt

- Chúng tôi đã giới thiệu về hàm softmax giúp ánh xạ một vector đầu vào sang các giá trị xác suất.
- Hồi quy softmax được áp dụng cho các bài toán phân loại. Nó sử dụng phân phối xác suất của các lớp đầu ra thông qua hàm softmax.
- Entropy chéo là một phép đánh giá tốt cho sự khác nhau giữa 2 phân phối xác suất. Nó đo lường số lượng bit cần để biểu diễn dữ liệu cho mô hình.

Bài tập

- Chứng minh rằng độ phân kì Kullback-Leibler $D(p\|q)$ không âm với mọi phân phối p và q . Gợi ý: sử dụng bất đẳng thức Jensen hay nghĩa là sử dụng bổ đề $-\log x$ là một hàm lồi.
- Chứng minh rằng $\log \sum_j \exp(o_j)$ là một hàm lồi với o .
- Chúng ta có thể tìm hiểu sâu hơn về sự liên kết giữa các họ hàm mũ và softmax.
 - Tính đạo hàm cấp hai của hàm mất mát entropy chéo $l(y, \hat{y})$ cho softmax.
 - Tính phương sai của phân phối được cho bởi softmax(o) và chứng minh rằng nó khớp với đạo hàm cấp hai được tính ở trên.
- Giả sử rằng chúng ta có 3 lớp, xác suất xảy ra cho mỗi lớp bằng nhau, nói cách khác vector xác suất là $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

- Vấn đề là gì nếu chúng ta cố gắng thiết kế một mã nhị phân cho nó? Chúng ta có thể làm khớp cận dưới của entropy trên số lượng bits hay không?
 - Bạn có thể thiết kế một mã tốt hơn không? Gợi ý: Điều gì xảy ra nếu chúng ta cố gắng biểu diễn 2 quan sát độc lập và nếu chúng ta biểu diễn n quan sát đồng thời?
5. Softmax là một cách gọi sai cho phép ánh xạ đã được giới thiệu ở trên (nhưng cộng đồng học sâu vẫn sử dụng nó). Công thức thật sự của softmax là $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
- Chứng minh rằng $\text{RealSoftMax}(a, b) > \max(a, b)$.
 - Chứng minh rằng $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) > \max(a, b)$ với $\lambda > 0$.
 - Chứng minh rằng khi $\lambda \rightarrow \infty$, chúng ta có $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \rightarrow \max(a, b)$.
 - Soft-min sẽ trông như thế nào?
 - Mở rộng nó cho nhiều hơn 2 số.

Thảo luận

- Tiếng Anh⁸²
- Tiếng Việt⁸³

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Thị Hồng Hạnh
- Lý Phi Long
- Lê Khắc Hồng Phúc

*Bùi Nhật Quân

- Lý Phi Long
- Nguyễn Minh Thư
- Trần Kiến An
- Lý Phi Long
- Lý Phi Long
- Vũ Hữu Tiệp
- Dương Nhật Tân
- Nguyễn Văn Tâm
- Trần Yến Thy
- Đinh Minh Tân

⁸² <https://discuss.mxnet.io/t/2334>

⁸³ <https://forum.machinelearningcoban.com/c/d2l>

- Phạm Hồng Vinh

6.1.5 Bộ dữ liệu Phân loại Ánh (Fashion-MNIST)

Ở sec_naive_bayes, chúng ta đã huấn luyện bộ phân loại Naive Bayes, sử dụng bộ dữ liệu MNIST được giới thiệu vào năm 1998 (?). Mặc dù MNIST hoạt động tốt như bộ dữ liệu đánh giá xếp hạng (*benchmark*), các mô hình đơn giản dựa trên tiêu chuẩn ngày nay cũng có thể đạt được độ chính xác phân loại lên tới 95%. Điều này khiến cho việc phân biệt độ mạnh yếu của các mô hình trở nên không chắc chắn. Ngày nay, MNIST đóng vai trò như là một bài kiểm tra sơ bộ hơn là bài kiểm tra đánh giá xếp hạng.

Để cải thiện vấn đề này, chúng ta sẽ cùng tập trung thảo luận ở các mục tiếp theo về một bộ dữ liệu tương tự nhưng phức tạp hơn, đó là bộ dữ liệu Fashion-MNIST (?) được giới thiệu vào năm 2017.

```
%matplotlib inline
import d2l
from mxnet import gluon
import sys

d2l.use_svg_display()
```

Tải về Bộ dữ liệu

Cũng giống như với MNIST, Gluons giúp việc tải và nạp bộ dữ liệu FashionMNIST vào bộ nhớ với lớp FashionMNIST tại gluon.data.vision trở nên dễ dàng. Các cơ chế của việc nạp và khám phá bộ dữ liệu sẽ được hướng dẫn ngắn gọn bên dưới. Vui lòng tham khảo sec_naive_bayes để biết thêm chi tiết về việc nạp dữ liệu.

```
mnist_train = gluon.data.vision.FashionMNIST(train=True)
mnist_test = gluon.data.vision.FashionMNIST(train=False)
```

FashionMNIST chứa các hình ảnh từ 10 lớp, mỗi danh mục được thể hiện bằng sáu nghìn tấm ảnh trong tập huấn luyện và một nghìn tấm ảnh trong tập kiểm tra. Do đó, tập huấn luyện và tập kiểm tra sẽ chứa tổng lần lượt 60 nghìn và 10 nghìn tấm ảnh.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

Hình ảnh trong Fashion-MNIST tương ứng với các danh mục sau: áo phông, quần dài, áo thun, váy, áo khoác, dép, áo sơ-mi, giày thể thao, túi và giày cao gót. Hàm dưới đây giúp chuyển đổi các nhãn giá trị số sang tên của chúng dưới dạng văn bản.

```
# Saved in the d2l package for later use
def get_fashion_mnist_labels(labels):
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                  'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

Chúng ta có thể tạo một hàm để minh họa các mẫu này.

```

# Saved in the d2l package for later use
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.asnumpy())
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes

```

Dưới đây là các hình ảnh và nhãn tương ứng của chúng (ở dạng văn bản) từ một vài mẫu đầu tiên trong tập huấn luyện.

```

X, y = mnist_train[:18]
show_images(X.squeeze(axis=-1), 2, 9, titles=get_fashion_mnist_labels(y));

```

Đọc một Minibatch

Để dễ dàng hơn trong việc đọc dữ liệu từ tập huấn luyện và tập kiểm tra, chúng ta sử dụng một `DataLoader` có sẵn thay vì tạo từ đầu như đã làm ở `sec_linear_scratch`. Nhắc lại là ở mỗi vòng lặp, một `DataLoader` sẽ đọc một minibatch của bộ dữ liệu với kích thước `batch_size`.

Trong quá trình huấn luyện, việc đọc dữ liệu có thể trở thành một nút thắt cổ chai đáng kể về hiệu năng, trừ khi chúng ta sử dụng một mô hình đơn giản hoặc là máy tính của chúng ta rất nhanh. Một tính năng tiện dụng của `DataLoader` là khả năng sử dụng đa luồng (*multiple processes*) để tăng tốc việc đọc dữ liệu. Ví dụ, chúng ta có thể dùng 4 luồng để đọc dữ liệu (thông qua `num_workers`). Vì tính năng này hiện tại không được hỗ trợ trên Windows, đoạn mã lập trình dưới đây sẽ kiểm tra nền tảng hệ điều hành để đảm bảo rằng chúng ta không làm phiền những người dùng Windows với các thông báo lỗi sau này.

```

# Saved in the d2l package for later use
def get_dataloader_workers(num_workers=4):
    # 0 means no additional process is used to speed up the reading of data.
    if sys.platform.startswith('win'):
        return 0
    else:
        return num_workers

```

Dưới đây, chúng ta chuyển đổi dữ liệu hình ảnh từ `uint8` sang số thực dấu phẩy động (*floating point number*) có 32-bit với lớp `ToTensor`. Ngoài ra, bộ chuyển đổi sẽ chia tất cả các số với 255 để tất cả các điểm ảnh sẽ có giá trị từ 0 đến 1. Lớp `ToTensor` cũng chuyển kênh hình ảnh từ chiều cuối cùng sang chiều thứ nhất để tạo điều kiện cho các tính toán mạng nơ-ron sẽ được giới thiệu sau này. Thông qua hàm `transform_first` của bộ dữ liệu, chúng ta có thể áp dụng phép biến đổi `ToTensor` cho phần tử đầu tiên của mỗi ví dụ (ảnh và nhãn).

```
batch_size = 256
transformer = gluon.data.vision.transforms.ToTensor()
train_iter = gluon.data.DataLoader(mnist_train.transform_first(transformer),
                                   batch_size, shuffle=True,
                                   num_workers=get_dataloader_workers())
```

Hãy cùng xem thời gian cần thiết để hoàn tất việc đọc dữ liệu huấn luyện.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
'%.2f sec' % timer.stop()
```

```
'0.87 sec'
```

Kết hợp Tất cả lại với nhau

Bây giờ, chúng ta sẽ định nghĩa hàm `load_data_fashion_mnist` để thu thập và đọc bộ dữ liệu Fashion-MNIST. Hàm này sẽ trả về các iterator cho dữ liệu của cả tập huấn luyện và tập kiểm định. Thêm nữa, nó chấp nhận một tham số tùy chọn để thay đổi kích thước hình ảnh đầu vào.

```
# Saved in the d2l package for later use
def load_data_fashion_mnist(batch_size, resize=None):
    """Download the Fashion-MNIST dataset and then load into memory."""
    dataset = gluon.data.vision
    trans = [dataset.transforms.Resize(resize)] if resize else []
    trans.append(dataset.transforms.ToTensor())
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))
```

Dưới đây, chúng ta xác nhận lại kích thước hình ảnh sau khi thay đổi.

```
train_iter, test_iter = load_data_fashion_mnist(32, (64, 64))
for X, y in train_iter:
    print(X.shape)
    break
```

```
(32, 1, 64, 64)
```

Giờ đây chúng ta đã sẵn sàng để làm việc với bộ dữ liệu FashionMNIST trong các mục tiếp theo.

Tóm tắt

- Fashion-MNIST là một bộ dữ liệu phân loại trang phục bao gồm các hình ảnh đại diện cho 10 danh mục.
- Chúng ta sẽ sử dụng bộ dữ liệu này trong các mục và chương tiếp theo để đánh giá các thuật toán phân loại khác nhau.
- Chúng ta lưu trữ kích thước của mỗi hình ảnh với chiều cao h (*height*) và chiều rộng w (*width*) điểm ảnh (*pixel*) dưới dạng $h \times w$ hoặc (h , w).
- Iterator dữ liệu là nhân tố chính để đạt được hiệu suất cao. Dựa vào các iterator được lập trình tốt bằng kỹ thuật khai thác đa luồng sẽ tránh làm chậm các vòng lặp huấn luyện của bạn.

Bài tập

1. Việc giảm `batch_size` (ví dụ xuống 1) có ảnh hưởng việc đọc dữ liệu hay không?
2. Với các người dùng không sử dụng Windows, hãy thử thay đổi `num_workers` để xem nó ảnh hưởng đến hiệu năng đọc dữ liệu như thế nào. Vẽ đồ thị hiệu năng tương ứng với số luồng được sử dụng.
3. Sử dụng tài liệu MXNet để xem các bộ dữ liệu có sẵn khác trong `mxnet.gluon.data.vision`.
4. Sử dụng tài liệu MXNet để xem những phép biến đổi nào có sẵn trong `mxnet.gluon.data.vision.transforms`.

Thảo luận

- Tiếng Anh⁸⁴
- Tiếng Việt⁸⁵

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- @tiepvupsu
- @lkhphuc
- @rootonchair
- @duythanhvn
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Nguyễn Lê Quang Nhật
- Nguyễn Lê Quang Nhật

⁸⁴ <https://discuss.mxnet.io/t/2335>

⁸⁵ <https://forum.machinelearningcoban.com/c/d21>

6.1.6 Lập trình Hồi quy Softmax từ đầu

Như ta đã lập trình hồi quy tuyến tính từ đầu, hồi quy logistic (softmax) đa lớp cũng sẽ tương tự và bạn nên tự biết cách làm thế nào để xây dựng nó một cách chi tiết nhất. Tương tự hồi quy tuyến tính, sau khi thực hiện mọi thứ bằng tay thì ta sẽ dùng Gluon để lập trình và đưa ra sự so sánh. Để bắt đầu, chúng ta nhập các thư viện quen thuộc vào.

```
import d2l
from mxnet import autograd, np, npx, gluon
from IPython import display
npx.set_np()
```

Ta sẽ làm việc trên tập dữ liệu Fashion-MNIST, vừa được giới thiệu trong : numref:sec_fashion_mnist, thiết lập một vòng lặp với kích cỡ batch là 256.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

Khởi tạo các tham số của Mô hình

Giống như ví dụ về hồi quy tuyến tính, mỗi mẫu sẽ được biểu diễn bằng một vector có chiều dài cố định. Mỗi mẫu trong tập dữ liệu thô là một ảnh 28×28 . Trong phần này, chúng ta sẽ trải phẳng mỗi tấm ảnh thành một vector một chiều có kích thước là 784. Sau này ta sẽ bàn về các chiến lược tinh vi hơn có khả năng khai thác cấu trúc không gian giữa các điểm ảnh, còn bây giờ ta hãy xem mỗi điểm ảnh là một đặc trưng.

Nhắc lại trong hồi quy softmax, mỗi lớp sẽ có một đầu ra. Vì tập dữ liệu của chúng ta có 10 lớp, mạng của chúng ta sẽ có 10 đầu ra. Do đó, các trọng số sẽ tạo thành một ma trận 784×10 và các hệ số điều chỉnh sẽ tạo thành một vector 1×10 . Cũng như hồi quy tuyến tính, ta sẽ khởi tạo các trọng số W bằng nhiễu Gauss và các hệ số điều chỉnh sẽ được khởi tạo bằng 0.

```
num_inputs = 784
num_outputs = 10

W = np.random.normal(0, 0.01, (num_inputs, num_outputs))
b = np.zeros(num_outputs)
```

Hãy nhớ rằng ta cần *đính kèm gradient* vào các tham số của mô hình. Cụ thể hơn, ta đang phân bổ bộ nhớ để lưu trữ các gradient trong tương lai và cho MXNet biết rằng ta sẽ muốn tính các gradient theo các tham số này trong tương lai.

```
W.attach_grad()
b.attach_grad()
```

Softmax

Trước khi xây dựng mô hình hồi quy softmax, hãy ôn nhanh tác dụng của các toán tử như `sum` trên những chiều cụ thể của một `ndarray`. Cho một ma trận `X`, chúng ta có thể tính tổng tất cả các phần tử (mặc định) hoặc chỉ trên các phần tử trong cùng một trục, ví dụ, cột (`axis=0`) hoặc cùng một hàng (`axis=1`). Lưu ý rằng nếu `X` là một mảng có kích thước $(2, 3)$, chúng ta tính tổng các cột (`X.sum(axis=0)`), kết quả sẽ là một vector (một chiều) có kích thước là $(3,)$. Nếu chúng ta muốn giữ số lượng trục trong mảng ban đầu (dẫn đến một mảng 2 chiều có kích thước $(1, 3)$), thay vì thu gọn kích thước mà chúng ta đã tính toán, chúng ta có thể gán `keepdims=True` khi gọi hàm `sum`.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
print(X.sum(axis=0, keepdims=True), '\n', X.sum(axis=1, keepdims=True))
```

```
[[5. 7. 9.]]
[[ 6.]
 [15.]]
```

Bây giờ chúng ta có thể bắt đầu xây dựng hàm softmax. Lưu ý rằng việc thực thi hàm softmax bao gồm hai bước: Đầu tiên, chúng ta lũy thừa từng giá trị ma trận (sử dụng `exp`). Sau đó, chúng ta tính tổng trên mỗi hàng (chúng ta có một hàng cho mỗi ví dụ trong batch) để lấy các hằng số chuẩn hóa cho mỗi ví dụ. Cuối cùng, chúng ta chia mỗi hàng theo hằng số chuẩn hóa của nó, đảm bảo rằng kết quả có tổng bằng 1. Trước khi xem đoạn mã, chúng ta hãy nhớ lại các bước này được thể hiện trong phương trình sau:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}. \quad (6.1.25)$$

Mẫu số hoặc hằng số chuẩn hóa đôi khi cũng được gọi là hàm phân hoạch (*partition function*) (và logarit của nó được gọi là hàm log phân hoạch (*log-partition function*)). Tên gốc của hàm được định nghĩa trong [vật lý thống kê⁸⁶](#) với phương trình liên quan mô hình hóa phân phối trên một tập hợp các phần tử.

```
def softmax(X):
    X_exp = np.exp(X)
    partition = X_exp.sum(axis=1, keepdims=True)
    return X_exp / partition # The broadcast mechanism is applied here
```

Chúng ta có thể thấy rằng với bất kỳ đầu vào ngẫu nhiên nào thì mỗi phần tử được biến đổi thành một số không âm. Hơn nữa, theo định nghĩa xác suất thì mỗi hàng có tổng là 1. Chú ý rằng đoạn mã trên tuy đúng về mặt toán học nhưng nó được xây dựng hơi cầu thả, không giống với cách ta thực hiện tại `sec_naive_bayes`, vì ta không kiểm tra vấn đề tràn số trên và dưới gây ra bởi các giá trị vô cùng lớn hoặc vô cùng nhỏ trong ma trận.

```
X = np.random.normal(size=(2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(axis=1)
```

```
(array([[0.22376052, 0.06659239, 0.06583703, 0.29964197, 0.3441681 ],
       [0.63209665, 0.03179282, 0.194987, 0.09209415, 0.04902935]]),
 array([1. , 0.9999994]))
```

⁸⁶ [https://en.wikipedia.org/wiki/Partition_function_\(statistical_mechanics\)](https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics))

Mô hình

Bây giờ chúng ta đã định nghĩa hàm softmax, chúng ta có thể bắt đầu lập trình mô hình hồi quy softmax. Đoạn mã sau định nghĩa lượt truyền xuôi thông qua mạng. Chú ý rằng chúng ta làm phẳng mỗi ảnh gốc trên tập lưu trữ bằng một vector có độ dài num_inputs bằng hàm reshape trước khi truyền dữ liệu sang mô hình đã khởi tạo.

```
def net(X):
    return softmax(np.dot(X.reshape(-1, num_inputs), W) + b)
```

Hàm mất mát

Tiếp đến chúng ta cần lập trình hàm mất mát entropy chéo đã được giới thiệu ở sec_softmax. Đây có lẽ là hàm mất mát thông dụng nhất trong nghiên cứu về học sâu vì hiện nay số lượng bài toán phân loại vượt trội hơn số lượng bài toán hồi quy.

Nhắc lại rằng entropy chéo lấy kết quả là hàm đối log hợp lý của xác suất dự đoán được gán cho nhãn thực $-\log P(y | x)$. Thay vì lặp qua các dự đoán của mô hình bằng vòng lặp for trong Python (có xu hướng kém hiệu quả), chúng ta có thể sử dụng hàm pick mà cho phép ta chọn lựa dễ dàng các phần tử thích hợp từ ma trận của các biến softmax đầu vào. Dưới đây, hàm pick được sử dụng như một ví dụ đơn giản với ma trận 2 hàng 3 cột.

```
y_hat = np.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], [0, 2]]
```

```
array([0.1, 0.5])
```

Bây giờ chúng ta có thể lập trình hàm mất mát entropy chéo hiệu quả hơn với một dòng lệnh.

```
def cross_entropy(y_hat, y):
    return -np.log(y_hat[range(len(y_hat)), y])
```

Độ chính xác cho bài toán Phân loại

Với phân phối xác suất dự đoán y_hat, ta thường chọn lớp có xác suất dự đoán cao nhất khi cần đưa ra một dự đoán cụ thể vì nhiều ứng dụng trong thực tế có yêu cầu như vậy. Ví dụ Gmail phải phân loại một email vào một trong các mục: Chính (Primary), Mạng xã hội (Social), Nội dung cập nhật (Updates) hoặc Diễn đàn (Forums). Có thể các xác suất được tính toán bên trong nội bộ hệ thống, nhưng cuối cùng kết quả vẫn chỉ là một trong các danh mục.

Các dự đoán được coi là chính xác khi chúng khớp với lớp thực tế y. Độ chính xác phân loại được tính bởi tỉ lệ các dự đoán chính xác trên tất cả các dự đoán đã đưa ra. Dù ta có thể gặp khó khăn khi tối ưu hóa trực tiếp độ chính xác (chúng không khả vi), đây thường là phép đo chất lượng được quan tâm tối nhiều nhất và sẽ luôn được tính khi huấn luyện các bộ phân loại.

Độ chính xác được tính toán như sau: Đầu tiên, lệnh y_hat.argmax(axis=1) được thực thi nhằm lấy ra các lớp được dự đoán (cho bởi chỉ số của các phần tử lớn nhất của mỗi hàng). Kết quả trả về sẽ có cùng kích thước với biến y và bây giờ ta chỉ cần so sánh hai vector này. Vì toán tử == so khớp cả kiểu dữ liệu của biến (ví dụ một biến int và một biến float32 không thể bằng nhau), ta cần phải ép kiểu chúng một cách thống nhất (ở đây ta chọn kiểu float32). Kết quả sẽ là một ndarray chứa các giá trị 0 (false) và 1 (true).

```
# Saved in the d2l package for later use
def accuracy(y_hat, y):
    if y_hat.shape[1] > 1:
        return float((y_hat.argmax(axis=1) == y.astype('float32')).sum())
    else:
        return float((y_hat.astype('int32') == y.astype('int32')).sum())
```

Ta sẽ tiếp tục sử dụng biến `y_hat` và `y` đã được định nghĩa trong hàm `pick`, lần lượt tương ứng với phân phối xác suất được dự đoán và nhãn. Có thể thấy rằng kết quả dự đoán của ví dụ đầu tiên là 2 (phần tử lớn nhất trong hàng là 0.6 với chỉ số tương ứng là 2) không khớp với nhãn thực tế là 0. Dự đoán ở ví dụ thứ hai là 2 (phần tử lớn nhất hàng là 0.5 với chỉ số tương ứng là 2) khớp với nhãn thực tế là 2. Do đó độ chính xác phân loại cho hai ví dụ này là 0.5.

```
y = np.array([0, 2])
accuracy(y_hat, y) / len(y)
```

0.5

Tương tự, ta có thể đánh giá độ chính xác của mô hình `net` trên tập dữ liệu (được truy xuất thông qua `data_iter`).

```
# Saved in the d2l package for later use
def evaluate_accuracy(net, data_iter):
    metric = Accumulator(2) # num_corrected_examples, num_examples
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

`Accumulator` ở đây là một lớp đa tiện ích, có tác dụng tính tổng tích lũy của nhiều số.

```
# Saved in the d2l package for later use
class Accumulator(object):
    """Sum a list of numbers over time."""

    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a+float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0] * len(self.data)

    def __getitem__(self, i):
        return self.data[i]
```

Vì ta đã khởi tạo mô hình `net` với trọng số ngẫu nhiên nên độ chính xác của mô hình lúc này sẽ ngang với việc đoán mò, tức độ chính xác bằng 0.1 với 10 lớp.

```
evaluate_accuracy(net, test_iter)
```

Huấn luyện mô hình

Vòng lặp huấn luyện cho hồi quy softmax trông khá quen thuộc nếu bạn đã đọc qua cách lập trình cho hồi quy tuyến tính tại `sec_linear_scratch`. Ở đây, chúng ta tái cấu trúc lại đoạn mã để giúp nó có thể được tái sử dụng. Đầu tiên, chúng ta định nghĩa một hàm để huấn luyện cho 1 epoch dữ liệu. Lưu ý rằng `updater` là một hàm tổng quát để cập nhật các tham số của mô hình và sẽ nhận giá trị kích thước batch làm thông số. Nó có thể là một wrapper của `d2l.sgd` hoặc là một đối tượng huấn luyện Gluon.

```
# Saved in the d2l package for later use
def train_epoch_ch3(net, train_iter, loss, updater):
    metric = Accumulator(3) # train_loss_sum, train_acc_sum, num_examples
    if isinstance(updater, gluon.Trainer):
        updater = updater.step
    for X, y in train_iter:
        # Compute gradients and update parameters
        with autograd.record():
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            updater(X.shape[0])
            metric.add(float(l.sum()), accuracy(y_hat, y), y.size)
    # Return training loss and training accuracy
    return metric[0]/metric[2], metric[1]/metric[2]
```

Trước khi xem đoạn mã thực hiện hàm huấn luyện, chúng ta định nghĩa 1 lớp phụ trợ để biểu diễn trực quan dữ liệu. Nhắc lại, mục đích của nó là giúp làm đơn giản hơn các đoạn mã sẽ xuất hiện trong các chương sau này.

```
# Saved in the d2l package for later use
class Animator(object):
    def __init__(self, xlabel=None, ylabel=None, legend=[], xlim=None,
                 ylim=None, xscale='linear', yscale='linear', fmts=None,
                 nrows=1, ncols=1, figsize=(3.5, 2.5)):
        """Incrementally plot multiple lines."""
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda to capture arguments
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        """Add multiple data points into the figure."""
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
```

(continues on next page)

```

if not self.X:
    self.X = [[] for _ in range(n)]
if not self.Y:
    self.Y = [[] for _ in range(n)]
if not self.fmts:
    self.fmts = ['-' * n
for i, (a, b) in enumerate(zip(x, y)):
    if a is not None and b is not None:
        self.X[i].append(a)
        self.Y[i].append(b)
self.axes[0].cla()
for x, y, fmt in zip(self.X, self.Y, self.fmts):
    self.axes[0].plot(x, y, fmt)
self.config_axes()
display.display(self.fig)
display.clear_output(wait=True)

```

Hàm huấn luyện sau đó sẽ chạy qua nhiều epochs và trực quan hóa quá trình huấn luyện.

```

# Saved in the d2l package for later use
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs],
                         ylim=[0.3, 0.9],
                         legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch+1, train_metrics+(test_acc,))

```

Nhắc lại, chúng ta sử dụng giải thuật hạ gradient ngẫu nhiên theo minibatch để tối ưu hàm mất mát của mô hình. Lưu ý rằng số lượng epochs (num_epochs), và hệ số học (lr) là 2 siêu tham số được hiệu chỉnh. Bằng cách thay đổi các giá trị này, chúng ta có thể tăng độ chính xác khi phân loại của mô hình. Trong thực tế, chúng ta thường sẽ chia tập dữ liệu thành 3 phần, đó là: dữ liệu huấn luyện, dữ liệu kiểm thử và dữ liệu kiểm tra, sử dụng dữ liệu kiểm thử để chọn ra những giá trị tốt nhất cho các siêu tham số.

```

num_epochs, lr = 10, 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)

train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)

```

Dự đoán

Giờ thì việc huấn luyện đã hoàn thành, mô hình của chúng ta đã sẵn sàng để phân loại các ảnh. Cho một loạt các ảnh, chúng ta sẽ so sánh các nhãn thực của chúng (dòng đầu tiên của văn bản đầu ra) với những dự đoán của mô hình (dòng thứ hai của văn bản đầu ra).

```
# Saved in the d2l package for later use
def predict_ch3(net, test_iter, n=6):
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true+'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(X[0:n].reshape(n, 28, 28), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```

Tóm tắt

Với hồi quy softmax, chúng ta có thể huấn luyện các mô hình cho bài toán phân loại đa lớp. Vòng lặp huấn luyện rất giống với vòng lặp huấn luyện của hồi quy tuyến tính: truy xuất và đọc dữ liệu, định nghĩa mô hình và hàm mất mát, và rồi huấn luyện mô hình sử dụng các giải thuật tối ưu. Rồi bạn sẽ thấy rằng hầu hết các mô hình học sâu phổ biến đều có thủ tục huấn luyện tương tự như vậy.

Bài tập

- Trong mục này, chúng ta đã lập trình hàm softmax dựa vào định nghĩa toán học của phép toán softmax. Điều này có thể gây ra những vấn đề gì (gợi ý: thử tính $\exp(50)$)?
- Hàm cross_entropy trong mục này được lập trình dựa vào định nghĩa của hàm mất mát entropy chéo. Vấn đề gì có thể xảy ra với cách lập trình như vậy (gợi ý: xem xét miền của hàm log)?
- Bạn có thể nghĩ ra các giải pháp để giải quyết hai vấn đề trên không?
- Việc trả về nhãn có khả năng nhất có phải lúc nào cũng là ý tưởng tốt không? Ví dụ, bạn có dùng phương pháp này cho chẩn đoán bệnh hay không?
- Giả sử rằng chúng ta muốn sử dụng hồi quy softmax để dự đoán từ tiếp theo dựa vào một số đặc trưng. Những vấn đề gì có thể xảy ra nếu dùng một tập từ vựng lớn?

Thảo luận

- Tiếng Anh⁸⁷
- Tiếng Việt⁸⁸

⁸⁷ <https://discuss.mxnet.io/t/2336>

⁸⁸ <https://forum.machinelearningcoban.com/c/d2l>

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Bùi Nhật Quân
- Lý Phi Long
- Phạm Hồng Vinh
- Lâm Ngọc Tâm
- Vũ Hữu Tiệp
- Phạm Minh Đức
- Lâm Ngọc Tâm
- Phạm Hồng Vinh
- Nguyễn Quang Hải
- Đinh Minh Tân
- Lê Cao Thăng

6.1.7 Triển khai súc tích của Hồi quy Softmax

Giống như cách Gluon giúp việc thực hiện hồi quy tuyến tính ở `sec_linear_gluon` trở nên dễ dàng hơn, chúng ta sẽ thấy nó cũng mang đến sự tiện lợi tương tự (hoặc có thể hơn) cho việc triển khai các mô hình phân loại. Một lần nữa, chúng ta bắt đầu bằng việc nhập gói thư viện.

```
import d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

Chúng ta tiếp tục làm việc với bộ dữ liệu Fashion-MNIST và giữ kích cỡ batch bằng 256 như ở phần trước.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

Khởi tạo Tham số Mô hình

Như đã đề cập trong `sec_softmax`, tầng ra của hồi quy softmax là một tầng kết nối đầy đủ (`Dense`). Do đó, để xây dựng mô hình, ta chỉ cần thêm một tầng `Dense` với 10 đầu ra vào đối tượng `Sequential`. Ở đây, việc sử dụng `Sequential` không thực sự cần thiết, nhưng ta nên hình thành thói quen sử dụng vì nó sẽ luôn hiện diện khi ta lập trình các mô hình học sâu. Một lần nữa, chúng ta khởi tạo các trọng số một cách ngẫu nhiên với trung bình bằng không và độ lệch chuẩn bằng 0.01.

```
net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Hàm Softmax

Ở ví dụ trước, ta đã tính toán kết quả đầu ra của mô hình và sau đó đã đưa các kết quả này qua hàm mất mát entropy chéo. Về mặt toán học, cách làm này hoàn toàn có lý. Tuy nhiên, từ góc độ tính toán, sử dụng hàm `softmax` có thể là nguồn gốc của các vấn đề về ổn định số (được bàn trong `sec_naive_bayes`). Hãy nhớ rằng, hàm softmax tính $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}$, trong đó \hat{y}_j là phần tử thứ j^{th} của `yhat` và z_j là phần tử thứ j^{th} của biến đầu vào `y_linear`.

Nếu một phần tử z_i quá lớn, e^{z_i} có thể sẽ lớn hơn giá trị cực đại mà kiểu `float` có thể biểu diễn được (đây là hiện tượng tràn số trên). Điều này dẫn đến mấu số (và/hoặc tử số) sẽ tiến tới `inf` và ta sẽ gặp phải trường hợp \hat{y}_i bằng 0, `inf` hoặc `nan`. Trong những tình huống này, giá trị trả về của `cross_entropy` có thể không xác định một cách rõ ràng. Có một mẹo để khắc phục điều này, đầu tiên ta lấy tất cả z_i trừ cho $\max(z_i)$, sau đó mới đưa qua hàm `softmax`. Bạn có thể nhận thấy rằng việc tịnh tiến mỗi z_i theo một hệ số không đổi sẽ không làm ảnh hưởng đến giá trị trả về của hàm `softmax`.

Sau khi thực hiện phép trừ và chuẩn hóa, một vài z_j có thể có giá trị âm lớn và do đó e^{z_j} sẽ xấp xỉ 0. Điều này có thể dẫn đến việc làm tròn thành 0 (tức tràn số dưới) do khả năng biểu diễn chính xác là hữu hạn, tức \hat{y}_j tiến về không và giá trị $\log(\hat{y}_j)$ tiến về `-inf`. Thực hiện vài bước lan truyền ngược với lối trên, ta sẽ đối mặt với một loạt giá trị `nan` (*not-a-number: Không phải số*) đáng sợ.

May mắn thay, mặc dù ta tính toán với các hàm `softmax` nhưng trong thực tế kết quả cuối cùng ta muốn là giá trị `log` của nó (ví dụ khi tính hàm mất mát entropy chéo). Bằng cách kết hợp cả hai hàm (`softmax` và `cross_entropy`) lại với nhau, ta có thể khắc phục vấn đề bất ổn trong tính toán mà có thể gây khó khăn trong quá trình lan truyền ngược. Như sẽ thấy trong chương trình bên dưới, ta đã không tính e^{z_j} mà thay vào đó ta tính trực tiếp z_j do việc khử trực tiếp trong $\log(\exp(\cdot))$.

$$\begin{aligned}\log(\hat{y}_j) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}\right) \\ &= \log(e^{z_j}) - \log\left(\sum_{i=1}^n e^{z_i}\right) \\ &= z_j - \log\left(\sum_{i=1}^n e^{z_i}\right).\end{aligned}\tag{6.1.26}$$

Ta sẽ muốn giữ nguyên chức năng của `softmax` thông thường trong trường hợp ta muốn đánh giá xác xuất của đầu ra theo mô hình. Nhưng thay vì truyền xác suất `softmax` vào hàm mất mát mới, ta sẽ chỉ truyền các giá trị `logit` và tính `softmax` cùng giá trị `log` của nó trong hàm mất mát `softmax_cross_entropy`. Hàm này sẽ tự động thực hiện các mẹo thông minh `log-sum-exp` (xem thêm Wikipedia⁸⁹).

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

⁸⁹ <https://en.wikipedia.org/wiki/LogSumExp>

Thuật toán tối ưu

Ở đây, chúng ta sử dụng thuật toán tối ưu hạ gradient ngẫu nhiên theo minibatch với tốc độ học bằng 0.1. Lưu ý rằng cách làm này giống hệt cách làm ở ví dụ về hồi quy tuyến tính, minh chứng cho tính tổng quát của bộ tối ưu hạ gradient ngẫu nhiên.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

Huấn luyện

Tiếp theo, chúng ta sẽ gọi hàm huấn luyện đã được khai báo ở mục trước để huấn luyện mô hình.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

Giống lần trước, thuật toán hội tụ và nghiệm có độ chính xác 83.7%, chỉ khác một điều là cần ít dòng mã hơn. Lưu ý rằng trong nhiều trường hợp, Gluon không chỉ dùng các mánh phổ biến mà còn sử dụng các kỹ thuật khác để tránh các lỗi kỹ thuật tính toán mà ta dễ gặp phải nếu tự xây dựng mô hình từ đầu.

Bài tập

- Thử thay đổi các siêu tham số, như là kích thước batch, epoch, và tốc độ học, để xem kết quả như thế nào.
- Tại sao độ chính xác trên tập kiểm tra lại giảm sau một khoảng thời gian? Chúng ta giải quyết việc này thế nào?

Thảo luận

- Tiếng Anh⁹⁰
- Tiếng Việt⁹¹

Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Lý Phi Long

⁹⁰ <https://discuss.mxnet.io/t/2337>

⁹¹ <https://forum.machinelearningcoban.com/c/d2l>

- Phạm Minh Đức
- Dương Nhật Tân
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Nguyễn Văn Tâm

6.2 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Hoàng Quân

7 | Bảng thuật ngữ

Các thuật ngữ cần được dịch theo chuẩn trong file này.

Nếu một từ chưa có trong bảng Anh - Việt, bạn có thể tạo một Pull Request mới để thêm từ đó vào bảng dưới đây. Nếu bạn cho rằng một từ không nên dịch ra tiếng Việt, bạn có thể giữ nguyên bản tiếng Anh.

Chú ý giữ thứ tự theo bảng chữ cái để tiện tra cứu.

A (page ??) B (page ??) C (page ??) D (page ??) E (page ??) F (page ??) G (page ??) H (page ??) I (page ??)
J (page ??) K (page ??) L (page ??) M (page ??) N (page ??) O (page ??) P (page ??) Q (page ??) R (page ??)
S (page ??) T (page ??) U (page ??) V (page ??) W (page ??) X (page ??) Y (page ??) Z (page ??)

7.1 A

English	Tiếng Việt	Thảo luận tại
accuracy	độ chính xác	
activation function	hàm kích hoạt	
adversarial learning	học đối kháng	http://bit.ly/2MhRgnP
agent	tác nhân	
algorithm's performance	chất lượng thuật toán	
avoidable bias	độ chêch tránh được	
artificial data synthesis	tổng hợp dữ liệu nhân tạo	
artificial general intelligence (AGI)	trí tuệ nhân tạo phổ quát	
attention mechanisms	cơ chế tập trung	
alternative hypothesis	giả thuyết đối	
automatic differentiation	tính vi phân tự động	

7.2 B

English	Tiếng Việt	Thảo luận tại
background noise	nhiều nền	http://bit.ly/31ObyKI
back-propagation	làn truyền ngược	
backward pass	lượt truyền ngược	
batch	batch	
benchmark	đánh giá xếp hạng	http://bit.ly/2BvfPYA
bias (bias as variance)	độ chêch	http://bit.ly/32HJI3S
bias (tham số mô hình)	hệ số điều chỉnh	
big data	big data	
binomial distribution	phân phối nhị thức	
Blackbox dev set	tập phát triển Blackbox	http://bit.ly/2MVHcl7
bounding box	khung chứa	http://bit.ly/2sbhDVj
broadcast	làn truyền	

7.3 C

English	Tiếng Việt	Thảo luận tại
category (trong bài toán phân loại)	lớp	
chain rule	quy tắc dây chuyền	
classifier	bộ phân loại	
clustering	phân cụm	
code (danh từ)	mã nguồn	
code (động từ)	viết mã	
computer vision	thị giác máy tính	
computing (trong Khoa Học Máy Tính)	điện toán	
computational graph	đồ thị tính toán	
conditional distribution	phân phối có điều kiện	
confidence interval	khoảng tin cậy	
constrain	ràng buộc	
(strictly) convex function	hàm lồi (chặt)	
convex optimization	tối ưu lồi	
convex set	tập lồi	
convolution neural networks	mạng nơ-ron tích chập	
cost function	hàm chi phí	
covariate	hiệp biến	Thảo luận ⁹³
cross entropy	entropy chéo	
cross validation	kiểm định chéo	

⁹³ <https://bit.ly/2r5QcfB>

7.4 D

English	Tiếng Việt	Thảo luận tại
data	dữ liệu	
data science	khoa học dữ liệu	
data scientist	nà khoa học dữ liệu	
datapoint (data point)	điểm dữ liệu	
data mismatch	dữ liệu không tương đồng	
dataset (data set)	tập dữ liệu	
data manipulation	thao tác với dữ liệu	
deep learning	học sâu	
dev set	tập phát triển	
dev set performance	chất lượng trên tập phát triển	
development set	tập phát triển	
differentiable	khả vi	
distribution	phân phối	
domain adaptation	thích ứng miền	
dot product	tích vô hướng (hoặc tích trong)	
dropout	dropout	

7.5 E

English	Tiếng Việt	Thảo luận tại
early stopping	dừng sớm	
effect size	hệ số ảnh hưởng	
eigen-decomposition	phân tích trị riêng	
eigenvalue	trị riêng	
eigenvector	vector riêng	
elementwise	(theo) từng phần tử	
embedding	embedding	
end-to-end	đầu-cuối	http://bit.ly/2OyYuEf
epoch (in training)	epoch (khi huấn luyện)	
error analysis	phân tích lỗi	
error rate	tỉ lệ lỗi	
estimator	bộ ước lượng	
evaluation metric	phép đánh giá	
example	mẫu	
expectation	kỳ vọng	
explicit feedback	phản hồi trực tiếp	
exponential distribution	phân phối mũ	
expression (math)	biểu thức (toán học)	
Eyeball dev set	Tập phát triển Eyeball	http://bit.ly/2MVHcl7

7.6 F

English	Tiếng Việt	Thảo luận tại
F1 score	chỉ số F1	
false negative	âm tính giả	
false positive	dương tính giả	
feature	đặc trưng	
fit	khớp	
first principle	định đê cơ bản	
flatten	trải	
forward pass	lượt truyền xuôi	
framework	framework	
functional analysis	giải tích hàm	
fully-connected	kết nối đầy đủ	

7.7 G

English	Tiếng Việt	Thảo luận tại
Gaussian distribution	phân phối Gauss (phân phối chuẩn)	
Gaussian noise	nhiễu Gauss	
generalization error	lỗi khái quát	
generative model	mô hình sinh	
generative adversarial network	mạng đối sinh	
global maximum	giá trị lớn nhất	
global minimum	giá trị nhỏ nhất	
gradient descent	hạ gradient	http://bit.ly/2BvfPYA , http://bit.ly/2rCiYEz
graphical model	mô hình đồ thị	
ground truth	nhãn gốc	http://bit.ly/34TljJ0

7.8 H

English	Tiếng Việt	Thảo luận tại
hand-engineering	thiết kế thủ công	
heuristic	thực nghiệm	
hidden unit	nút ẩn	
human-level performance	chất lượng mức con người	http://bit.ly/36IzQcB , http://bit.ly/33CJfjX
hyperparameter	siêu tham số	
hyperplane	siêu phẳng	
hypothesis test	kiểm định giả thuyết	

7.9 I

English	Tiếng Việt	Thảo luận tại
implement	lập trình	
implementation	cách lập trình	
implicit feedback	phản hồi gián tiếp	
imputation (Preprocessing)	quy buộc	
independence assumption	giả định độc lập	
iteration	vòng lặp	
iterator	iterator	

7.10 J

English	Tiếng Việt	Thảo luận tại
joint distribution	phân phối đồng thời	

7.11 L

English	Tiếng Việt	Thảo luận tại
layer	tầng	
law of large numbers	luật số lớn	
learning curve	đồ thị quá trình học	http://bit.ly/2BvfPYA
learning algorithm	thuật toán học	
linear	tuyến tính	
linear algebra	đại số tuyến tính	
linear dependence	phụ thuộc tuyến tính	
linear form	dạng tuyến tính	
linear independence	độc lập tuyến tính	
linear programming	quy hoạch tuyến tính	
linear regression	hồi quy tuyến tính	
linear discriminant analysis (LDA)	phân tích biệt thức tuyến tính	
local maximum	cực đại	
local minimum	cực tiểu	
Long Short-term Memory (LSTM)	bộ nhớ ngắn hạn dài	
logistic regression	hồi quy logistic	
logit (trong softmax)	logit	
log-likelihood function	hàm log hợp lý	
loss function	hàm mất mát	

7.12 M

English	Tiếng Việt	Thảo luận tại
machine learning	học máy	
marginalization	phép biên hóa	
maximum likelihood estimator	bộ ước lượng hợp lý cực đại	
mean squared error (MSE)	trung bình bình phương sai số	
metric	phép đo	
minibatch	minibatch	
misclassified	bị phân loại nhầm	
mislabeled	bị gán nhãn nhầm	
model	mô hình	
module	mô-đun	
multi-class classification	phân loại đa lớp	
multinomial distribution	phân phối đa thức	
multitask learning	học đa nhiệm	

7.13 N

English	Tiếng Việt	Thảo luận tại
named entity	danh từ riêng	
natural language processing (NLP)	xử lý ngôn ngữ tự nhiên	
negative log-likelihood function	hàm đối log hợp lý	
negative sample/example	mẫu âm	
neural network	mạng nơ-ron	http://bit.ly/2BvfPYA http://bit.ly/2MAkizG
node (trong mạng nơ-ron)	nút	
norm	chuẩn	
normal distribution	phân phối chuẩn (phân phối Gauss)	
null hypothesis	giả thuyết gốc	

7.14 O

English	Tiếng Việt	Thảo luận tại
objective function	hàm mục tiêu	
offline learning	học ngoại tuyến	
optimizing metric	phép đo để tối ưu	http://bit.ly/2BvfPYA
orthogonal	trực giao	
orthonormal	trực chuẩn	
overfit	quá khớp	http://bit.ly/2BvfPYA
overflow (numerical)	tràn (số) trên	
one-hot encoding	biểu diễn one-hot	
one-sided test	kiểm định một phía	
one-tailed test	kiểm định một đuôi	

7.15 P

English	Tiếng Việt	Thảo luận tại
partition function	hàm phân hoạch	http://bit.ly/2T0dY7F
pattern recognition	nhận dạng mẫu	
perceptron	perceptron	
performance	chất lượng	http://bit.ly/36IzQcB
plateau (danh từ)	vùng nằm ngang	
plateau (động từ)	nằm ngang	
pipeline	pipeline	http://bit.ly/2OyYuEf
policy (trong Học Tăng cường)	chính sách	
positive sample/example	mẫu dương	
precision	precision	
principal component analysis (PCA)	phân tích thành phần chính	
probability theory	lý thuyết xác suất	
population	tổng thể	
p-value	trị số p	

7.16 Q

English	Tiếng Việt	Thảo luận tại
quadratic	toàn phương	
quadratic form	dạng toàn phương	
quadratic programming	quy hoạch toàn phương	

7.17 R

English	Tiếng Việt	Thảo luận tại
random variable	biến ngẫu nhiên	
recall	recall	
recognition	nhận dạng	
recurrent neural network	mạng nơ-ron truy hồi	
regressor	bộ hồi quy	
regularization	điều chuẩn	
reinforcement learning	học tăng cường	
representation learning	học biểu diễn	
reward function	hàm điểm thưởng	
root mean squared error (RMSE)	căn bậc hai trung bình bình phương sai số	
running time	thời gian chạy	
region of rejection	miền bác bỏ	

7.18 S

English	Tiếng Việt	Thảo luận tại
sampling with replacement	lấy mẫu có hoàn lại	http://bit.ly/34wQuKr
sampling without replacement	lấy mẫu không hoàn lại	http://bit.ly/34wQuKr
satisficing metric	phép đo thỏa mãn	http://bit.ly/2BvfPYA
scalar	số vô hướng	
scoring function	hàm tính điểm	
sentiment classification	phân loại cảm xúc	
sequence learning	học chuỗi	http://bit.ly/2SsUity
sensitivity	độ nhạy	
shape (trong Đại số Tuyến tính)	kích thước	
significance test	kiểm định ý nghĩa	
slicing (array)	cắt chọn (mảng)	
spam email	email rác	
speech recognition	nhận dạng giọng nói	
standard deviation	độ lệch chuẩn	
stationary point	điểm dừng	
statistical power	năng lực thống kê	
statistical significance	ý nghĩa thống kê	
statistical significant	có ý nghĩa thống kê	
stochastic gradient descent	hạ gradient ngẫu nhiên	
subscript	chỉ số dưới	
subspace estimation	ước lượng không gian con	
superscript	chỉ số trên	
supervised learning	học có giám sát	
surrogate objective	mục tiêu thay thế	http://bit.ly/2PIxkN1
symbolic graph	đồ thị biểu tượng	

7.19 T

English	Tiếng Việt	Thảo luận tại
tensor contraction	phép co tensor	
test set	tập kiểm tra	
test set performance	chất lượng trên tập kiểm tra	
test statistic	tiêu chuẩn kiểm định	
timestep	bước thời gian	
training set	tập huấn luyện	
training dev set	tập phát triển huấn luyện	
training set performance	chất lượng trên tập huấn luyện	
transcribe	phiên thoại	
transcription	bản ghi thoại	
true negative	âm tính thật	
true positive	dương tính thật	
tune parameters	điều chỉnh tham số	
two-sided test	kiểm định hai phía	
two-tailed test	kiểm định hai đuôi	

7.20 U

English	Tiếng Việt	Thảo luận tại
unavoidable bias	độ chêch không tránh được	
underfit	dưới khớp	http://bit.ly/2BvfPYA
underflow (numerical)	tràn (số) dưới	
unit (trong mạng nơ-ron)	nút	
unsupervised learning	học không giám sát	

7.21 V

English	Tiếng Việt	Thảo luận tại
validation set	tập kiểm định	
variance (bias as variance)	phương sai	http://bit.ly/32HJI3S
vector	vector	

7.22 W

English	Tiếng Việt	Thảo luận tại
well-behaved function (analytic function)	hàm khả vi vô hạn	
wrapper function (trong lập trình)	hàm wrapper	

Bibliography

- Ahmed, A., Aly, M., Gonzalez, J., Narayananamurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining* (pp. 123–132).
- Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE transactions on Information Theory*, 46(2), 325–343.
- Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.
- Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1), 108–116.
- Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.
- Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5, 135–146.
- Bollobás, B. (1999). *Linear analysis*. Cambridge University Press, Cambridge.
- Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.
- Brown, N., & Sandholm, T. (2017). Libratus: the superhuman ai for no-limit poker. *IJCAI* (pp. 5226–5228).
- Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1-2), 57–83.
- Canny, J. (1987). A computational approach to edge detection. *Readings in computer vision* (pp. 184–203). Elsevier.
- Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.
- Chowdhury, G. G. (2010). *Introduction to modern information retrieval*. Facet publishing.
- Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.
- Csiszár, I. (2008). Axiomatic characterizations of information measures. *Entropy*, 10(3), 261–273.
- De Cock, D. (2011). Ames, iowa: alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3).
- DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* (pp. 205–220).
- Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. *Sequential Monte Carlo methods in practice* (pp. 3–14). Springer.

- Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul), 2121–2159.
- Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.
- Edelman, B., Ostrovsky, M., & Schwarz, M. (2007). Internet advertising and the generalized second-price auction: selling billions of dollars worth of keywords. *American economic review*, 97(1), 242–259.
- Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a step-size. *Conference on Learning Theory* (pp. 658–695).
- Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2414–2423).
- Ginibre, J. (1965). Statistical ensembles of complex, quaternion, and real matrices. *Journal of Mathematical Physics*, 6(3), 440–449.
- Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).
- Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).
- Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).
- Goh, G. (2017). Why momentum really works. *Distill*. URL: <http://distill.pub/2017/momentum>, doi:10.23915/distill.00006⁹²
- Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12), 61–71.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearning-book.org>.
- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems* (pp. 2672–2680).
- Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*.
- Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, 18(5-6), 602–610.
- Gunawardana, A., & Shani, G. (2015). Evaluating recommender systems. *Recommender systems handbook* (pp. 265–308). Springer.
- Guo, H., Tang, R., Ye, Y., Li, Z., & He, X. (2017). Deepfm: a factorization-machine based neural network for ctr prediction. *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (pp. 1725–1731).
- Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*.
- Hazan, E., Rakhlin, A., & Bartlett, P. L. (2008). Adaptive online gradient descent. *Advances in Neural Information Processing Systems* (pp. 65–72).

⁹² <https://doi.org/10.23915/distill.00006>

- He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *European conference on computer vision* (pp. 630–645).
- He, X., & Chua, T.-S. (2017). Neural factorization machines for sparse predictive analytics. *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval* (pp. 355–364).
- He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T.-S. (2017). Neural collaborative filtering. *Proceedings of the 26th international conference on world wide web* (pp. 173–182).
- Hebb, D. O., & Hebb, D. (1949). *The organization of behavior*. Vol. 65. Wiley New York.
- Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.
- Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. *22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1999* (pp. 230–237).
- Hidasi, B., Karatzoglou, A., Baltrunas, L., & Tikk, D. (2015). Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939*.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.
- Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in neural information processing systems* (pp. 689–696).
- Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7132–7141).
- Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. *2008 Eighth IEEE International Conference on Data Mining* (pp. 263–272).
- Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).
- Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* (pp. 1945–1953).
- Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.
- Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.
- Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ... others. (2018). Highly scalable deep learning training system with mixed-precision: training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*.
- Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... others. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).
- Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.
- Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

- Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.
- Kolter, Z. (2008). Linear algebra review and reference. Available online: <http://>
- Koren, Y. (2009). Collaborative filtering with temporal dynamics. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 447–456).
- Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, pp. 30–37.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (pp. 1097–1105).
- Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy*.
- LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., & others. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.
- Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.
- Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., ... Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th USENIX Symposium on Operating Systems Design and Implementation (\$OSDI\$'14)* (pp. 583–598).
- Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.
- Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).
- Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ... others. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*.
- Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*.
- Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: single shot multibox detector. *European conference on computer vision* (pp. 21–37).
- Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431–3440).
- Loshchilov, I., & Hutter, F. (2016). Sgdr: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 91–110.
- Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint*.
- Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1* (pp. 142–150).
- McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115–133.

- McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., ... others. (2013). Ad click prediction: a view from the trenches. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1222–1230).
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* (pp. 3111–3119).
- Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., ... Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning- Volume 70* (pp. 2430–2439).
- Morey, R. D., Hoekstra, R., Rouder, J. N., Lee, M. D., & Wagenmakers, E.-J. (2016). The fallacy of placing confidence in confidence intervals. *Psychonomic bulletin & review*, 23(1), 103–123.
- Nesterov, Y., & Vial, J.-P. (2000). *Confidence level solutions for stochastic programming, Stochastic Programming E-Print Series*.
- Nesterov, Y. (2018). *Lectures on convex optimization*. Vol. 137. Springer.
- Neyman, J. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 236(767), 333–380.
- Pennington, J., Schoenholz, S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in neural information processing systems* (pp. 4785–4795).
- Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543).
- Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.
- Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark*, 7(15), 510.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1–17.
- Quadrana, M., Cremonesi, P., & Jannach, D. (2018). Sequence-aware recommender systems. *ACM Computing Surveys (CSUR)*, 51(4), 66.
- Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.
- Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.
- Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.
- Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99).
- Rendle, S. (2010). Factorization machines. *2010 IEEE International Conference on Data Mining* (pp. 995–1000).

- Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2009). Bpr: bayesian personalized ranking from implicit feedback. *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence* (pp. 452–461).
- Rumelhart, D. E., Hinton, G. E., Williams, R. J., & others. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3), 1.
- Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.
- Salton, G., Wong, A., & Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11), 613–620.
- Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).
- Sarwar, B. M., Karypis, G., Konstan, J. A., Riedl, J., & others. (2001). Item-based collaborative filtering recommendation algorithms. *Www*, 1, 285–295.
- Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 253–260).
- Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 2673–2681.
- Sedhain, S., Menon, A. K., Sanner, S., & Xie, L. (2015). Autorec: autoencoders meet collaborative filtering. *Proceedings of the 24th International Conference on World Wide Web* (pp. 111–112).
- Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.
- Shannon, C. E. (1948 , 7). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3), 379–423.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587), 484.
- Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
- Smola, A., & Narayananamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2), 703–710.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929–1958.
- Strang, G. (1993). *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA.
- Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009.
- Sukhbaatar, S., Weston, J., Fergus, R., & others. (2015). End-to-end memory networks. *Advances in neural information processing systems* (pp. 2440–2448).
- Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning* (pp. 1139–1147).
- Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.

- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).
- Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).
- Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*.
- Tang, J., & Wang, K. (2018). Personalized top-n sequential recommendation via convolutional sequence embedding. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (pp. 565–573).
- Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*.
- Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 26–31.
- Treisman, A. M., & Gelade, G. (1980). A feature-integration theory of attention. *Cognitive psychology*, 12(1), 97–136.
- Töscher, A., Jahrer, M., & Bell, R. M. (2009). The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, pp. 1–52.
- Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, 104(2), 154–171.
- Van Loan, C. F., & Golub, G. H. (1983). *Matrix computations*. Johns Hopkins University Press.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems* (pp. 5998–6008).
- Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *NETWORKS*, 2(3), 2–3.
- Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the gpu. *ACM SIGPLAN Notices* (p. 11).
- Wasserman, L. (2013). *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.
- Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4), 279–292.
- Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681–688).
- Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math* (pp. 325–327).
- Williams, S., Waterman, A., & Patterson, D. (2009). *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).
- Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. *Communications of the ACM*, 54(2), 91–98.
- Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. *Proceedings of the tenth ACM international conference on web search and data mining* (pp. 495–503).

- Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.
- Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).
- Ye, M., Yin, P., Lee, W.-C., & Lee, D.-L. (2011). Exploiting geographical influence for collaborative point-of-interest recommendation. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (pp. 325–334).
- You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.
- Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).
- Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.
- Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1), 5.
- Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE international conference on computer vision* (pp. 2223–2232).