

---

# **Dive into Deep Learning**

**Aston Zhang**

**Zachary C. Lipton**

**Mu Li**

**Alexander J. Smola**

**2019년 08월 29일**



This draft is a testing version (draft date: August 29, 2019).

Visit <https://d2l.ai> to obtain a later or release version.



---

## Contents

---

<b>1</b>	<b>서문</b>	<b>1</b>
1.1	코드, 수학, HTML이 모두 하나로 구성됨	2
1.2	구성	3
1.3	직접 하면서 배우기	4
1.4	감사의 글	4
<b>2</b>	<b>이 책의 사용 방법</b>	<b>5</b>
2.1	대상 독자	5
2.2	내용 및 구성	6
2.3	코드	7
2.4	토론(Forum)	8
2.5	문제	8
2.6	Scan the QR Code to Discuss	8
<b>3</b>	<b>딥러닝 소개</b>	<b>9</b>
3.1	데이터를 활용하는 프로그래밍	10
3.2	기원	11
3.3	딥 러닝으로의 길	14
3.4	성공 사례	16
3.5	주요 요소들	18
3.6	요약	20

---

3.7	문제 . . . . .	20
3.8	참고문헌 . . . . .	20
3.9	Scan the QR Code to Discuss . . . . .	22
<b>4</b>	<b>딥러닝 맛보기</b>	<b>23</b>
4.1	소개 . . . . .	23
4.2	Gluon 시작하기 . . . . .	49
4.3	데이터 조작(data manipulation) . . . . .	53
4.4	선형 대수 . . . . .	62
4.5	자동 미분(automatic differentiation) . . . . .	76
4.6	화률과 통계 . . . . .	81
4.7	나이브 베이즈 분류(Naive Bayes Classification) . . . . .	90
4.8	샘플링 . . . . .	97
4.9	문서(documentation) . . . . .	104
<b>5</b>	<b>딥러닝 기초</b>	<b>109</b>
5.1	선형 회귀(Linear Regression) . . . . .	109
5.2	선형 회귀를 처음부터 구현하기 . . . . .	120
5.3	선형 회귀의 간결한 구현 . . . . .	128
5.4	Softmax 회귀(regression) . . . . .	134
5.5	이미지 분류 데이터 (Fashion-MNIST) . . . . .	141
5.6	Softmax 회귀(regression)를 처음부터 구현하기 . . . . .	146
5.7	Softmax 회귀(regression)의 간결한 구현 . . . . .	152
5.8	다층 퍼셉트론 (Multilayer Perceptron) . . . . .	155
5.9	다층 퍼셉트론(multilayer perceptron)을 처음부터 구현하기 . . . . .	165
5.10	다층 퍼셉트론(multilayer perceptron)의 간결한 구현 . . . . .	168
5.11	모델 선택, 언더피팅(underfitting), 오버피팅(overfitting) . . . . .	170
5.12	가중치 감쇠 (weight decay) . . . . .	182
5.13	드롭아웃(dropout) . . . . .	191
5.14	순전파(forward propagation), 역전파(back propagation), 연산 그래프 . . . . .	198
5.15	수치 안정성(numerical stability) 및 초기화 . . . . .	203
5.16	환경 . . . . .	209
5.17	Kaggle의 주택 가격 예측하기 . . . . .	222
<b>6</b>	<b>딥러닝 계산</b>	<b>237</b>
6.1	층(layer)과 블럭(Block) . . . . .	237
6.2	파라미터 관리 . . . . .	246

6.3	초기화 지연(deferred Initialization) . . . . .	257
6.4	커스텀 층(custom layer) . . . . .	261
6.5	NDArray . . . . .	265
6.6	Gluon 모델 파라미터들 . . . . .	267
6.7	요약 . . . . .	268
6.8	문제 . . . . .	268
6.9	Scan the QR Code to Discuss . . . . .	268
6.10	GPU . . . . .	269
<b>7</b>	<b>Appendix</b>	<b>277</b>
7.1	이 책에 기여하는 방법 . . . . .	277
7.2	d2l 패키지 색인 . . . . .	284



---

## 서문

---

불과 몇 년 전만 해도, 주요 기업과 스타트업에서 지능형 제품과 서비스를 개발하는 다수의 딥러닝 과학자들은 존재하지 않았습니다. 우리 중 막내 (저자)가 현장에 들어갔을 때는 머신 러닝은 일간지의 헤드라인을 장식하지도 않았습니다. 우리의 부모님들은 머신 러닝이 무엇인지 전혀 몰랐습니다. 왜 우리가 의학이나 법률 분야의 직업보다 그것을 선호하는지 말할 필요도 없었습니다. 머신 러닝은 실제 응용의 좁은 분야에 해당하는 미래지향적인 분야였습니다. 그리고 음성 인식 및 컴퓨터 비전과 같은 응용들은 많은 도메인 지식을 필요로 했기 때문에, 머신 러닝은 종종 별도로 분리된 하나의 작은 컴퓨포넌트로 간주 되었습니다. 이 책에서 다루고 있는, 딥러닝의 선행 모델인 뉴럴 네트워크는 구식 도구로 간주되었습니다.

지난 5년간 딥러닝은 컴퓨터 비전, 자연어 처리, 자동 음성 인식, 강화 학습, 통계 모델링 등 다양한 분야의 급속한 발전을 이룩하여 세상을 놀라게 했습니다. 이러한 진보를 통해 우리는 이제 (자율성이 증가하면서) 스스로를 운전하는 자동차, 평범한 응답을 예상하는 스마트 응답 시스템, 사람들이 이메일의 산에서 발굴하도록 돋고, 바둑을 세계에서 제일 잘 두는 사람보다 우월한 소프트웨어 에이전트, 수십 년의 노력이 필요한 기술들을 만들 수 있습니다. 이미 이러한 도구들은 영화 제작 방식, 질병 진단 방식, 천체 물리학에서 생물학 등까지 기초 과학에서 점점 더 많은 역할을 하고 있습니다. 이 책은 딥러닝을 친숙하게, 개념, 문맥 및 코드를 이해 시키려는 우리의 노력입니다.

## 1.1 코드, 수학, HTML이 모두 하나로 구성됨

컴퓨팅 기술이 완전히 영향을 미치려면, 잘 이해하고, 잘 문서화되며, 잘 관리되고 완성된 도구를 통해 지원되어야 합니다. 핵심 아이디어를 명확하게 뽑아내어, 새로운 실무자를 최신 상태로 유지해야 데 필요한 온보딩 시간을 최소화해야 합니다. 성숙한 라이브러리는 일반적인 작업을 자동화해야하며, 예제 코드는 실무자가 필요에 맞게 공통 응용 프로그램을 쉽게 수정, 적용 및 확장 할 수 있도록 해야 합니다. 동적 웹 응용 프로그램을 예로 들어 보겠습니다. Amazon과 같은 많은 수의 회사들이, 1990년대에 데이터베이스 기반 웹 애플리케이션을 성공적으로 개발했음에도 불구하고, 창조적 기업가를 돋기 위한 이러한 기술의 완전한 가능성은 강력하고, 잘 문서화된 프레임워크의 개발로 겨우 지난 10년간 현실화 되었습니다.

딥러닝의 실현은, 모든 어플리케이션은 다양한 분야의 통합을 가져왔기 때문에, 고유한 과제를 제시합니다. 딥러닝을 적용하기 위해서는, (i) 특정 방식으로 문제제기를 위한 동기, (ii) 주어진 모델링 접근을 위한 수학, (iii) 모델을 데이터에 맞추기 위한 최적화 알고리즘 (iv) 모델을 효율적으로 훈련하는데 필요한 엔지니어링들을 이해하고 수치 연산의 위험(Pitfalls)을 탐색하고 가용한 하드웨어의 최대한 활용하세요. 문제를 수식으로 만들기 위한 비판적 사고 기술, 그것을 풀 수 있는 수학, 그리고 이러한 솔루션들을 모두 구현하기 위한 소프트웨어 도구, 이 모두를 한 곳에서 가리치는 것은 매우 어려운 과제입니다. 이 책에서 우리의 목표는 독자들이 최대한 빠르게 실무자가 될 수 있도록, 통합된 자원을 제시하는 것입니다.

우리는 MXNet의 새로운 Gluon 인터페이스를 사용자에게 설명해야 했던, 2017년 7월에 이 책 프로젝트를 시작했습니다. 동시에 당시에는, (1) 최신 상태이고, (2) 기술적 깊이와 유사한 것으로 현대의 머신 러닝의 전체 폭을 다루고 (3) 실행 가능한 코드가 있는 교재부터 활발한 튜토리얼 같은 것들이 없었습니다. 우리는 딥러닝과 관련된 프레임워크를 사용하는 방법(예: TensorFlow에서 행렬을 사용하여 기본 수치 계산을 수행하는 방법) 또는 특정 기술(예: LeNet, AlexNet, ResNet 등의 코드 일부들)을 구현해 내는 방법 혹은 예제 코드들을 블로그 게시물 형태 또는 GitHub에서 많이 발견했습니다. 그러나 이 예제는 일반적으로 주어진 접근 방식을 구현하는 방법에 초점을 맞추었지만, 왜 그런 특정 알고리즘이 선택되었는지에 대한 논의를 생략했습니다. 웹 사이트 [Distill](#) 또는 개인 블로그와 같은 곳에서 산발적인 주제들이 논의 되는 동안, 딥러닝에서 선택한 주제만을 다루거나, 종종 관련된 코드가 부족한 경우가 많았습니다. 다른 한편으로는, 딥러닝의 개념에 대한 훌륭한 자료를 제공하는 [Goodfellow, Bengio and Courville, 2016](#)에 여러 교과서가 등장하지만 이러한 자료는 설명과 코드로 그 개념을 실현하는 방법을 잘 녹아내지 못했습니다. 결국 독자들은 그것을 구현하기 위한 단서들을 스스로 찾도록 방치되었습니다. 또한 너무 많은 자료가 유료 교육과정 제공 업체에 숨겨져 있습니다.

우리는 그래서 자료를 만들기 시작했습니다. 이 자료는

- (1) 모든 사람이 자유롭게 이용할 수 있고,

- (2) 실제로 응용 머신 러닝 과학자가 될때 필요한, 충분한 기술적 깊이를 제공하고,
- (3) 실행 가능한 코드를 포함시키고, 실제로 문제를 해결하는 방법을 독자에게 보여 주며,
- (4) 우리뿐만 아니라 많은 커뮤니티에 의해 신속하게 업데이트 할 수 있으며,
- (5) 기술 세부 사항에 대한 상호 토론과 질문에 답하기 위해 [forum](#) 를 이용하여 보완됩니다.

이러한 목표는 종종 충돌했습니다. 방정식, 이론 및 인용식은 LaTeX에서 가장 잘 관리되고 배치됩니다. 코드는 파이썬을 이용해야 가장 잘 설명 됩니다. 그리고 웹페이지들은 기본적으로 HTML과 Javascript로 구성됩니다. 더 나아가, 우리는 코드가 물리적인 책, 다운로드 가능한 PDF, 인터넷 웹페이지로 모두 접근 가능하지만 동시에 실행 가능하길 바랍니다. 현재 이러한 이러한 요구에 완벽하게 부합하는 도구나 워크 플로우가 존재하지 않으므로 직접 만들어야 했습니다. 우리는 우리의 접근 방식을 [부록](#)에 상세하게 기록하였습니다. 우리는 이를 위해, 소스를 공유하고 편집을 허용하기 위해 Github를, 다양한 코드, 방적식과 문장들을 다루기위한 Jupyter 노트북, 다양한 출력을 생산하기 위한 렌더링 엔진으로 Sphinx, 포럼을 위해 Discourse를 각각 설정하였습니다. 아직 우리의 시스템이 완벽하지는 않지만, 이러한 선택은 경쟁적인 상황에서 좋은 절충안을 제공합니다. 우리는 이러한 통합 워크 플로우를 사용하여 출판된 첫 번째 책이 될 거라고 생각합니다.

## 1.2 구성

기본적인 수학적 배경에 대한 특별 과정을 제공하는 몇 가지 예비 노트북을 제외하고, 각각의 후속 노트북은 합리적인 수의 새로운 개념을 소개하고 실제 데이터셋을 사용하여 하나의 독립적인 작업 예제를 제공합니다. 이것은 구조적인 과제를 제시한다. 일부 모델은 논리적으로 단일 노트북으로 그룹화될 수 있습니다. 그리고 어떤 아이디어들은 여러 모델을 연속적으로 실행하여 가장 잘 가르쳐 질 수 있습니다. 반면에 *I* 작업 예제, *I* 노트북의 정책을 준수하면 큰 이점이 있습니다. 이렇게 하면 우리의 코드를 활용하여 여러분의 연구 프로젝트를 가능한 한 쉽게 시작할 수 있습니다. 단지 단일 노트북을 복사하고 수정하기만 하면 됩니다.

우리는 필요에 따라 실행 가능한 코드를 배경 자료와 함께 배치합니다. 일반적으로, 우리는 종종 일반적으로 우리는 도구를 완전히 설명하기 전에 사용할 수 있도록 만드는 측면에서 종종 오류를 겪을 것입니다 (나중에 배경을 설명하여 후속 조치를 취합니다). 예를 들어, 왜 유용한지 또는 왜 작동하는지 완전히 설명하기 전에 확률적 경사 하강법(*Stochastic Gradient Descent*)을 사용할 수 있습니다. 이것은 실무자에게 문제를 신속하게 해결하는 데 필요한 정보를 제공하는 데 도움이되며, 이를 위해 독자가 적어도 단기적으로, 몇 가지 결정에 있어서는 우리를 신뢰해야 합니다.

전반적으로 MXNet 라이브러리로 작업하게 될 것입니다. MXNet 라이브러리는 연구하기에 충분히 유연하면서도 완성품 만들기에 있어서도 충분히 빠르다는 드문 특성을 가지고 있습니다. 이 책은 딥

러닝 개념을 처음부터 가르칠 것입니다. 때때로, 우리는 Gluon의 고급 기능에 의해 사용자로부터 숨겨진 모델에 대한 세부 사항을 탐구하고자 합니다. 이것은 특히 기본 튜토리얼에서 나옵니다. 여기서는 주어진 층에서 일어나는 모든 것을 이해하기를 원합니다. 이 경우, 우리는 일반적으로 두 가지 버전의 예제를 제시합니다. 하나는 처음부터 모든 것을 구현하고 NDArray 와 자동미분(automatic differentiation)에만 의존하고 다른 하나는 Gluon을 사용하여 간결하게 수행하는 방법을 보여줍니다. 한번만 층(layer)이 어떻게 동작하는지 가르쳐 주면, 후속 자습서에서 Gluon 버전을 사용할 수 있습니다.

## 1.3 직접 하면서 배우기

많은 교과서는 일련의 주제들을 각각 철저히 상세하게 가르칩니다. 예를 들어, Chris Bishop의 훌륭한 교과서 [Pattern Recognition and Machine Learning](#)는 각 주제를 철저히 가르쳐 줍니다. 선형 회귀 분석의 장에 들어가려면 작지 않은 양의 작업이 필요합니다. 전문가들은 이 책을 철저하기 때문에 좋아하지만, 초보자에게는 이 점이 소개 텍스트로써의 유용함을 감소시킵니다.

이 책에서는 대부분의 개념을 딱 그시점(*Just in time*)에 가르쳐 드리겠습니다. 즉, 실제 목적을 달성하기 위해 필요한 바로 그 순간에 개념을 배우게 될 것입니다. 선형대수학이나 확률과 같은 기본적인 사항을 가르치기 위해 처음에는 약간의 시간이 걸리지만, 좀 더 색다른 확률 분포에 대해 걱정하기 전에, 첫 번째 모델을 훈련하는 만족감을 느끼기를 바랍니다.

## 1.4 감사의 글

우리는 영어와 중국 초안 모두에 대한 수백 명의 기여자에게 빛을 지고 있습니다. 그들은 콘텐츠를 개선하는 데 도움이 귀중한 피드백을 제공했습니다. 특히, 우리는 모두를 위해 더 나은 버전을 만들기 위해 힘써준 영어 초안의 모든 기여자에게 감사드립니다. 그들의 Github ID와 이름 (제공된 경우)은 : bowen0701, ChaiBapChya (Chaitanya Prakash Bapat), kirk86, MLWhiz (Rahul Agarwal), mstewart141, muelleme (Mike Müller), sfermigier, sundeepteki, vishaalkapoor, YaYaB. 더해서, 아마존 웹서비스에 감사 드리며, 특히 Swami Sivasubramanian, Raju Gulabani, Charlie Bell, and Andrew Jassy에게 이 책을 쓰도록 충분히 지원해 주신 대해 감사드립니다. 사용 가능한 시간, 자원, 동료와의 토론, 지속적인 격려 덕분에 책이 만들어 질 수 있었습니다.

# 2

---

## 이 책의 사용 방법

---

우리는 모델 만들기부터 모델 학습, 그리고 컴퓨터 비전 및 자연어 처리에 대한 응용까지 딥러닝의 모든 것들에 대해서 전반적인 소개를 하는 것을 목표로 합니다. 알고리즘의 원리만을 설명하는 것이 아니라, Apache MXNet을 이용해서 직접 구현하고 운영하는 것까지 보여줄 것입니다. 이 책의 각 절은 Jupyter 노트북, 텍스트, 공식, 이미지, 코드, 그리고 수행 결과로 구성되어 있습니다. 이를 통해서 이 책을 직접 읽는 것뿐만 아니라, 직접 수행하면서 상호적인 학습 경험을 할 수 있을 것 입니다. 하지만, 이 책을 여러분에게 딥러닝에 대한 소개를 제공하는 것이기 때문에, 더 많은 것을 알고 싶다면, 연구 커뮤니티들에서 제공하는 툴박스, 캠파일러, 예제들, 튜터리얼, 그리고 소스 코드를 활용하는 것을 추천합니다. 그럼 여행을 시작해봅시다.

### 2.1 대상 독자

이 책은 딥러닝을 배우고자 하는 대학교 학부생, 엔지니어, 연구원 그리고 특히 딥러닝을 실제 문제에 적용하고자 하는 사람들을 대상으로 하고 있습니다. 모든 개념을 기초부터 설명하고 있기 때문에, 딥러닝이나 머신 러닝에 대한 배경지식이 없어도 됩니다. 딥러닝 기술이나, 응용을 설명할 때 수학이나 프로그래밍을 사용하지만, 기본적인 선형대수, 미적분, 확률 그리고 기초 Python 프로그래밍과 같은 기초적인 것들만 알고 있으면 충분합니다. 부록에서는 이 책에서 다루는 대부분의 수학을 포함하고

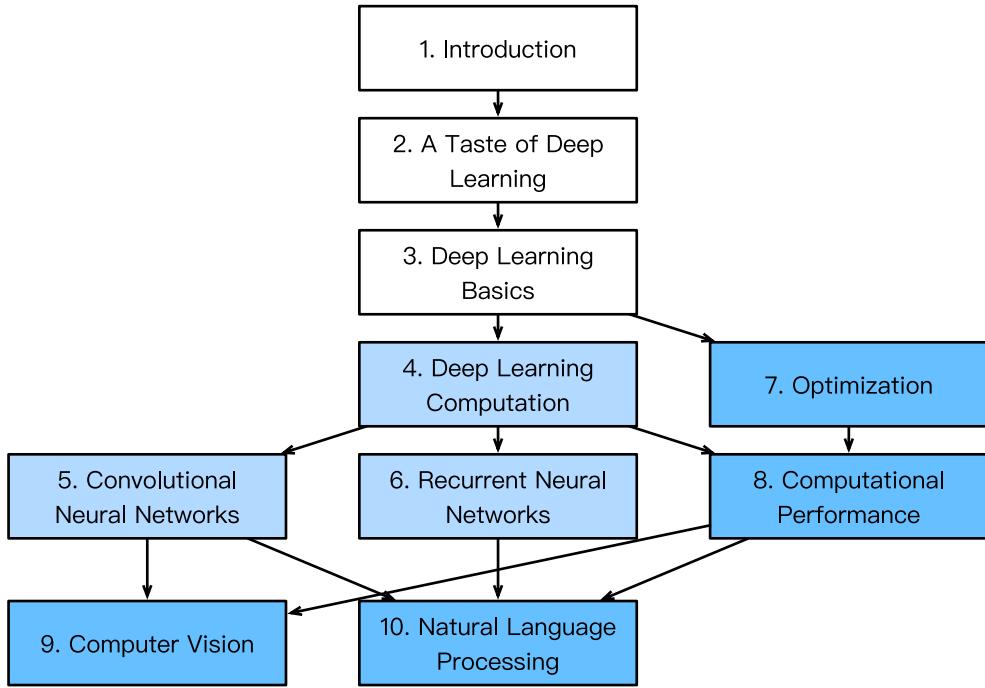
있으니 필요한 경우 참고하시기 바랍니다. 이 책은 소개서이기 때문에, 수학적으로 깊이 들어가는 것 보다는 직관과 아이디어에 더 비중을 두고 있습니다. 흥미를 갖은 독자들은 많은 훌륭한 책들을 통해서 더 자세한 것들을 배울 수 있습니다. 예를 들면, Bela Bollobas의 [Linear Analysis](#)는 선형대수와 함수 분석을 아주 자세하게 다루고 있고, [All of Statistics](#)은 통계에 대한 아주 훌륭한 가이드입니다. 만약 Python을 사용해보지 않았다면, [Python tutorial](#)을 참고하면 좋습니다. 물론, 수학적인 내용만 관심있다면, 프로그래밍 부분은 생략하면서 읽어도 됩니다. 물론 반대의 경우도 그렇습니다.

## 2.2 내용 및 구성

이 책은 크게 세 파트로 구성되어 있습니다.

- 첫번째 파트는 전제조건과 기본사항들을 다룹니다. 1장에서는 딥러닝에 대한 소개 와 이 책에 대한 활용 방법을 설명합니다. [딥러닝 맛보기](#) 장에서는 이 책의 코드를 어디서 받을 수 있고 어떻게 실행하는지 등과 같은 딥러닝 핸즈온에 필요한 것들을 이야기합니다. 만약 시간이 없거나 딥러닝의 가장 기본적인 개념과 기법들만을 배우고자 한다면, 이 파트만 읽어도 충분합니다.
- 두번째 파트인 다음 세 장은 최신 딥러닝 기법들을 다룹니다. [딥러닝 계산](#)은 딥러닝 연산의 다양한 주요 요소들을 설명하면서, 더 복잡한 모델 구현을 위한 기본을 다질 수 있도록 합니다. 다음 장은 [Convolutional Neural Networks](#)인데, 이는 최근 몇년동안 컴퓨터 비전에서 성과를 거두고 있는 딥러닝 기술입니다. 그리고 순서가 있는 데이터를 처리하는데 일반적으로 사용되는 [Recurrent Neural Networks](#)를 다룹니다. 여러분은 이 두번째 파트를 읽으면서 최근 딥러닝 기술을 이해할 수 있을 것입니다.
- 마지막 파트는 확장성, 효율성과 응용을 다룹니다. 딥러닝 모델을 학습시키는데 사용되는 다양한 [Optimization Algorithms](#)을 설명한 후, 정규화와 같이 딥러닝 연산 Performance에 영향을 주는 중요한 요소들에 대해서 살펴봅니다. 9장과 10장은 컴퓨터 비전과 자연어처리에서 사용되는 딥러닝의 응용에 대해서 알아봅니다. 이 파트는 여러분의 관심에 따라서 선택적으로 읽어도 됩니다.

이 책의 구성은 다음 그림과 같습니다. 화살표는 선행되어야하는 관계를 의미합니다. 만약 빠른 시간안에 딥러닝의 기본적인 개념과 기법들을 배워야한다면, 1장~3장만 읽으면됩니다. 더 심도있는 내용을 원하면, 그 다음 3장(4장~6장)을 읽으세요. 마지막 4장은 독자의 관심에 따라서 옵션입니다.



## 2.3 코드

이 책은 모든 절에 동작하는 코드를 포함하고 있습니다. 코드들을 수정하고 다시 수행해서 결과에 어떤 영향을 미치는지도 확인할 수 있습니다. 이렇게 한 이유는 딥러닝에서 상호적인 학습 경험이 중요하다는 것을 알아냈기 때문입니다. 어렵게도 딥러닝은 이론적으로 잘 이해되지 않고 있습니다. 그렇기 때문에, 많은 논의들은 코드를 수행해서 얻은 경험에 많이 의존하고 있습니다. 글로 설명하는 것은 최선의 노력을 해도 모든 자세한 것들을 다루기에 충분하지 않을 수 있습니다. 이론적인 진전이 더 만들어지면 그 때는 이런 상황이 좋아질 것을 기대하지만, 지금은 독자들이 코드를 바꾸고, 결과를 관찰하고, 전반적인 과정을 요약하는 것을 통해서 이해도를 높이고, 직관을 키우시길 바랍니다.

이 책의 코드는 Apache MXNet을 기반으로 합니다. MXNet은 딥러닝을 위한 오픈소스 프레임워크입니다. 이는 AWS(Amazon Cloud Services)가 선호하는 프레임워크이며, 많은 대학과 회사에서 사용되고 있습니다. 이 책의 모든 코드는 MXNet 1.2.0을 이용해서 테스트되었으나, 딥러닝의 빠른 발전으로 어떤 코드는, 이후 MXNet 버전에서는 이 책의 인쇄버전의 코드가 잘 작동하지 않을 수 있습니다. 하지만, 온라인 버전은 계속 최신을 유지할 것입니다. 만약 그런 경우를 만난다면, “[Installation and Running](#)”를 참고해서 코드와 실행환경을 업데이트하세요. 그리고, 불필요한 반복을 피하기 위해서, 이 책에서 자주 import 되거나 자주 참조되는 함수, 클래스 등은 d2l 패키지에 넣었습니다. d2l 패키지

의 버전은 1.0.0 입니다. 포함된 함수와 클래스에 대한 자세한 내용은 “*d2l package index*”에 있습니다. 이 책은 MXNet 소개서로 활용될 수도 있습니다. 코드를 사용한 주요 목적은 텍스트, 이미지, 공식과 더불어 딥러닝 알고리즘을 배우는 또 다른 방법을 제공하는데 있습니다. 이 책은 각 모델과 알고리즘의 실제 데이터에 대한 실제 영향을 이해하기 위해서 인터랙티브한 환경을 제공합니다. 딥러닝 알고리즘의 구현에 대한 자세한 내용을 설명하기 위해서 MXNet 모듈의 기본적인 기능 - ndarray, autograd, gluon 등 - 만을 사용합니다. 여러분의 연구나 업무에서 다른 딥러닝 프레임워크를 사용하는 경우에도, 이 코드는 딥러닝 알고리즘에 대한 이해를 높이는데 도움이 될 것이라고 기대합니다.

## 2.4 토론(Forum)

이 책의 내용에 대한 논의 포럼은 [discuss.mxnet.io](https://discuss.mxnet.io) 에 있습니다. 이 책의 어느 부분이던지 질문이 있을 때는, 각 절 끝에 있는 QR 코드를 스캔해서 논의에 참여할 수 있습니다. 이 책의 저자와 MXNet 개발자들이 포럼에 자주 방문하고 참여하고 있습니다.

## 2.5 문제

1. 이 책의 논의 포럼 [discuss.mxnet.io](https://discuss.mxnet.io) 에 계정을 생성하세요.
2. 여러분의 컴퓨터에 Python을 설치하세요.

## 2.6 Scan the QR Code to Discuss



# 3

---

## 딥러닝 소개

---

2016년, 네임드 데이터 과학자인 죄엘 그루스(Joel Grus)는 한 유명 인터넷 기업에서 면접을 보았습니다. 보통 그러하듯이 인터뷰 담당자는 그의 프로그래밍 기술을 평가하는 문제를 냈습니다. 간단한 어린이 게임인 FizzBuzz를 구현하는 것이 과제였습니다. 그 안에서 플레이어는 1부터 카운트하면서 3으로 나눌 수 있는 숫자는 ‘fizz’로, 5로 나눌 수 있는 숫자는 ‘buzz’로 바꿉니다. 15로 나눌 수 있는 숫자는 ‘FizzBuzz’가 됩니다. 즉, 플레이어는 시퀀스를 생성합니다.

```
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 ...
```

전혀 예상하지 못한 일이 벌어졌습니다. 거의 몇 줄의 Python 코드로 알고리즘을 구현해서 문제를 해결하는 대신, 그는 데이터를 활용한 프로그램으로 문제를 풀기로 했습니다. 그는 (3, fizz), (5, buzz), (7, 7), (2, 2), (15, fizzbuzz)의 쌍을 활용하여 분류기를 학습시켰습니다. 그가 작은 뉴럴 네트워크(neural network)를 만들고, 그것을 이 데이터를 활용하여 학습시켰고 그 결과 꽤 높은 정확도를 달성하였습니다(면접관이 좋은 점수를 주지 않아서 채용되지는 못했습니다).

이 인터뷰와 같은 상황은 프로그램 설계가 데이터에 의한 학습으로 보완 되거나 대체되는 컴퓨터 과학의 획기적인 순간입니다. 예로 든 상황은 면접이라서가 아니라 어떠한 목표를 손쉽게 달성 할 수 있게 해 준다는 점에서 중요성을 가집니다. 일반적으로는 위에서 설명한 오버스러운 방식으로 FizzBuzz를 해결하지는 않겠지만, 얼굴을 인식하거나, 사람의 목소리 또는 텍스트로 감정을 분류하거나, 음성

을 인식할 때는 완전히 다른 이야기입니다. 좋은 알고리즘, 많은 연산 장치 및 데이터, 그리고 좋은 소프트웨어 도구들로 인해 이제는 대부분의 소프트웨어 엔지니어가 불과 10년 전에는 최고의 과학자들에게도 너무 도전적이라고 여겨졌던 문제를 해결하는 정교한 모델을 만들 수 있게 되었습니다.

이 책은 머신러닝을 구현하는 여성에 들어선 엔지니어를 돋는 것을 목표로 합니다. 우리는 수학, 코드, 예제를 쉽게 사용할 수 있는 패키지로 결합하여 머신러닝을 실용적으로 만드는 것을 목표로 합니다. 온라인으로 제공되는 Jupyter 노트북 예제들은 노트북이나 클라우드 서버에서 실행할 수 있습니다. 우리는 이를 통해서 새로운 세대의 프로그래머, 기업가, 통계학자, 생물학자 및 고급 머신러닝 알고리즘을 배포하는 데 관심이 있는 모든 사람들이 문제를 해결할 수 있기를 바랍니다.

### 3.1 데이터를 활용하는 프로그래밍

코드를 이용하는 프로그래밍과 데이터를 활용하는 프로그래밍의 차이점을 좀 더 자세히 살펴 보겠습니다. 이 둘은 보이는 것보다 더 심오하기 때문입니다. 대부분의 전통적인 프로그램은 머신러닝을 필요로 하지 않습니다. 예를 들어 전자 레인지용 사용자 인터페이스를 작성하려는 경우 약간의 노력으로 몇 가지 버튼을 설계할 수 있습니다. 다양한 조건에서 전자 레인지의 동작을 정확하게 설명하는 몇 가지 논리와 규칙을 추가하면 완료됩니다. 마찬가지로 사회 보장 번호의 유효성을 확인하는 프로그램은 여러 규칙이 적용되는지 여부를 테스트하면 됩니다. 예를 들어, 이러한 숫자는 9 자리 숫자를 포함해야 하며 000으로 시작하지 않아야 한다와 같은 규칙입니다.

위의 두 가지 예에서 프로그램의 논리를 이해하기 위해 현실 세계에서 데이터를 수집할 필요가 없으며, 그 데이터의 특징을 추출할 필요가 없다는 점에 주목할 가치가 있습니다. 많은 시간이 있다면, 우리의 상식과 알고리즘 기술은 우리가 작업을 완료하기에 충분합니다.

우리가 전에 관찰 한 바와 같이, 심지어 최고의 프로그래머의 능력을 넘는 많은 예가 있지만, 많은 아이들, 심지어 많은 동물들이 쉽게 그들을 해결할 수 있습니다. 이미지에 고양이가 포함되어 있는지 여부를 감지하는 문제를 고려해보겠습니다. 어디서부터 시작해야 할까요? 이 문제를 더욱 단순화해 보겠습니다. 모든 이미지가 동일한 크기 (예, 400x400 픽셀)라고 가정하고, 각 픽셀이 빨강, 녹색 및 파랑 값으로 구성된 경우 이미지는 480,000 개의 숫자로 표시됩니다. 우리의 고양이 탐지기가 관련된 정보가 어디에 있는지 결정하는 것은 불가능합니다. 그것은 모든 값의 평균일까요? 네 모서리의 값일까요? 아니면 이미지의 특정 지점일까요? 실제로 이미지의 내용을 해석하려면 가장자리, 질감, 모양, 눈, 코와 같은 수천 개의 값을 결합 할 때만 나타나는 특징을 찾아야합니다. 그래야만 이미지에 고양이가 포함되어 있는지 여부를 판단할 수 있습니다.

다른 전략은 최종 필요성에 기반한 솔루션을 찾는 것입니다. 즉, 이미지 예제 및 원하는 응답 (cat, cat 없음) 을 출발점으로 사용하여 데이터로 프로그래밍하는 것입니다. 우리는 고양이의 실제 이미지 (인터넷에서 인기있는 주제)들과 다른 것들을 수집할 수 있습니다. 이제 우리의 목표는 이미지에

고양이가 포함되어 있는지 여부를 배울 수 있는 함수를 찾는 것입니다. 일반적으로 함수의 형태 (예, 다향식)는 엔지니어에 의해 선택되며 그 함수의 파라미터들은 데이터에서 학습됩니다.

일반적으로 머신러닝은 고양이 인식과 같은 문제를 해결하는 데 사용할 수 있는 다양한 종류의 함수를 다룹니다. 딥 러닝은 특히 신경망에서 영감을 얻은 특정 함수의 클래스를 사용해서, 이것을 특별한 방법으로 학습(함수의 파라미터를 계산하는 것)시키는 방법입니다. 최근에는 빅 데이터와 강력한 하드웨어 덕분에 이미지, 텍스트, 오디오 신호 등과 같은 복잡한 고차원 데이터를 처리하는 데 있어 딥 러닝이 사실상의 표준으로(de facto choice) 자리잡았습니다.

## 3.2 기원

딥 러닝은 최근의 발명품이지만, 인간은 데이터를 분석하고 미래의 결과를 예측하려는 욕구를 수세기 동안 가지고 있어왔습니다. 사실, 자연 과학의 대부분은 이것에 뿌리를 두고 있습니다. 예를 들어, 베르누이 분포는 야곱 베르누이 (1655-1705) 의 이름을 따서 명명되었으며, 가우시안 분포는 칼 프리드리히 가우스 (1777-1855) 에 의해 발견되었습니다. 예를 들어, 그는 최소 평균 제곱 알고리즘을 발명했는데, 이것은 보험 계산부터 의료 진단까지 다양한 분야에서 오늘날 까지도 계속 사용되고 있습니다. 이러한 기술들은 자연 과학에서 실험적인 접근법을 불러 일으켰습니다. 예를 들어 저항기의 전류 및 전압에 관한 옴의 법칙은 선형 모델로 완벽하게 설명됩니다.

중세 시대에도 수학자들은 예측에 대한 예리한 직관을 가지고 있었습니다. 예를 들어, 야곱 코벨 (1460-1533)의 기하학책에서는 발의 평균 길이를 얻기 위해 성인 남성 발 16개의 평균을 사용했습니다.



그림 1.1은 이 평균이 어떻게 얻어졌는지를 보여줍니다. 16명의 성인 남성은 교회를 떠날 때 한 줄로 정렬하도록 요구받았습니다. 그런 다음 총 길이를 16으로 나누어 현재 1피트 금액에 대한 추정치를 얻습니다. 이 '알고리즘'은 나중에 잘못된 모양의 밭을 다루기 위해 개선되었습니다 - 각각 가장 짧고

긴 발을 가진 2 명의 남성은 제외하고 나머지 발들에 대해서만 평균값을 계산합니다. 이것은 절사 평균 추정치의 초기 예 중 하나입니다.

통계는 실제로 데이터의 수집 및 가용성으로 시작되었습니다. 거장 중 한명인 로널드 피셔 (1890-1962)는 이론과 유전학의 응용에 크게 기여했습니다. 그의 알고리즘들 (예, 선형 판별 분석)과 수식들 (예, Fisher 정보 매트릭스)은 오늘날에도 여전히 자주 사용되고 있습니다 (1936년에 발표한 난초(Iris) 데이터셋도 머신러닝 알고리즘을 설명하는 데 사용되기도 합니다).

머신러닝에 대한 두 번째 영향은 정보 이론 (클로드 쇄넌, 1916-2001) 과 앤런 튜링 (1912-1954)의 계산 이론에서 나왔습니다. 튜링은 그의 유명한 논문, *기계 및 지능 컴퓨팅, Computing machinery and intelligence* (Mind, 1950년 10월)에서, 그는 “기계가 생각할 수 있습니까?”라는 질문을 제기했습니다. 그의 튜링 테스트로 설명 된 것처럼, 인간 평가자가 텍스트 상호 작용을 통해 기계와 인간의 응답을 구별하기 어려운 경우 “기계는 지능적이다”라고 간주될 수 있습니다. 오늘날까지 지능형 기계의 개발은 신속하고 지속적으로 변화하고 있습니다.

또 다른 영향은 신경 과학 및 심리학에서 발견 될 수 있습니다. 결국, 인간은 분명히 지적인 행동을 합니다. 이러한 행동 및 이에 필요한 통찰력을 설명하고, 아마도 리버스 엔지니어링 할 수 있는지 여부를 묻는 것은 합리적입니다. 이를 달성하기 위한 가장 오래된 알고리즘 중 하나는 도널드 헤브 (1904-1985)에 의해 공식화 되었습니다.

그의 획기적인 책 *행동의 조직*, *The Organization of Behavior* (John Wiley & Sons, 1949) 에서, 그는 뉴런이 긍정적인 강화를 통해 학습할 것이라고 가정했습니다. 이것은 Hebbian 학습 규칙으로 알려지게 되었습니다. 그것은 Rosenblatt의 퍼셉트론 학습 알고리즘의 원형이며 오늘날 딥 러닝을 뒷받침하는 많은 stochastic gradient descent 알고리즘의 기초를 마련했습니다: 뉴럴 네트워크의 좋은 가중치를 얻기 위해 바람직한 행동은 강화하고 바람직하지 않은 행동을 감소시킵니다.

생물학적 영감으로부터 신경망(Neural Network)이라는 명칭이 탄생하였습니다. 알렉산더 베인(1873)과 제임스 세링턴(1890)이 신경망 모델을 제안한 이래 한세기 이상 연구자들은 상호 작용하는 뉴런들의 네트워크와 유사한 계산 회로를 구현하려고 시도해 왔습니다. 시간이 지남에 따라 생물학과의 연관성은 느슨해 졌지만 신경망이라는 이름은 그대로 사용하고 있습니다. 오늘날 대부분의 신경망에서 찾을 수 있는 몇 가지 주요 핵심 원칙은 다음과 같습니다:

- ’레이어’라고 불리우는 선형 및 비선형 처리 유닛들의 교차 구조
- 제인 규칙 (일명 역 전파)을 사용하여 한 번에 전체 네트워크의 매개 변수를 조정

초기 급속한 진행 이후, 뉴럴 네트워크의 연구는 1995년경부터 2005년까지 쇠퇴했습니다. 여러 가지 이유들이 있습니다. 우선 네트워크 학습은 매우 많은 계산량을 필요로 합니다. 지난 세기 말경에 이르러 메모리는 충분해 졌지만 계산 능력이 부족했습니다. 두번째 이유는 데이터셋이 상대적으로 작았다는 것입니다. 실제로 1932년에 나온 피셔(Fisher)의 ‘난초(Iris) 데이터셋’은 각각 50장의 세 종류의 난초 사진으로 구성되어 있는데 알고리즘의 효능을 테스트하는 데 널리 사용되는 도구였습니다. 지금은

학습 예제에 흔히 쓰이는 60,000개의 손으로 쓴 숫자들로 구성된 MNIST조차 당시에는 너무 거대한 데이터로 취급되었습니다.

데이터 및 계산능력이 부족한 경우, 커널 방법, 의사 결정 트리 및 그래픽 모델과 같은 강력한 통계 도구쪽이 우수한 성능을 보여줍니다. 신경망과는 달리 이것들은 훈련하는 데 몇 주씩 걸리지 않으면서도 강력한 이론적 보장으로 예측 가능한 결과를 제공합니다.

### 3.3 딥 러닝으로의 길

시간이 흐르면서 월드 와이드 웹이 등장하고 수억명의 온라인 사용자에게 서비스를 제공하는 회사가 출현하였으며 저렴한 고품질의 센서, 데이터 저장 비용 감소([Kryder의 법칙](#)), 그리고 특히 원래 컴퓨터 게임을 위해 설계된 GPU의 가격 하락([무어의 법칙](#))이 진행됨에 따라 많은 것들이 바뀌게 되었습니다. 갑자기 계산이 불가능한 것처럼 보이는 알고리즘과 모델이 의미를 지니게 된 것입니다(반대의 경우도 마찬가지입니다). 이것은 아래 표에 가장 잘 설명되어 있습니다.

연대	데이터셋	메모리	초당 부동소수점 연산수
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

RAM이 데이터의 증가와 보조를 맞추지 않은 것은 매우 분명합니다. 동시에 계산 능력의 향상은 사용 가능한 데이터의 증가를 앞서고 있습니다. 즉, 통계 모델은 메모리 효율성이 항상 되어야하고 (일반적으로 비선형을 추가하여 달성됨) 컴퓨팅 예산 증가로 인해 이러한 매개변수를 최적화하는 데 더 많은 시간을 할애해야 했습니다. 결국 머신러닝 및 통계에 적절한 방법은 선형 모델 및 커널 방법에서 딥 네트워크로 이동했습니다. 이것은 다층 퍼셉트론(Multilayer Perceptron) (예, 맥컬록 & 피츠, 1943), 컨볼루션 뉴럴 네트워크(Convolutional Neural Network) (Le Cun, 1992), Long Short Term Memory (Hochreiter & Schmidhuber, 1997), Q-러닝 (왓킨스, 1989)과 같은 딥 러닝의 많은 주류 이론들이 상당 시간 동안 휴면기에 있다가 최근 10년간 재발견된 이유 중에 하나입니다.

통계 모델, 응용 프로그램 및 알고리즘의 최근 발전은 때때로 카브리아 폭발 (Cambrian Dexplosion)에 비유되고 있습니다: 종의 진화가 급속히 진행되는 순간입니다. 실제로, 현 시점에서 가장 좋은 성과들(state of art)은 수십 년동안 만들어져온 오래된 알고리즘이 적용된 결과가 아닙니다. 아래 목록은 연구자들이 지난 10년간 엄청난 진전을 달성하는 데 도움이 된 아이디어의 극히 일부분입니다.

- 드롭아웃(Drop out) [3] 과 같은 새로운 용량 제어 방법, 즉 학습 데이터의 큰 부분을 암기하는 위험 없이 비교적 큰 네트워크의 학습이 가능합니다. 이것은 학습 목적을 위해 무작위 변수로 가중치를 대체하여 네트워크 전체에 노이즈 주입 [4] 을 적용하여 달성되었습니다.
- 어텐션 메커니즘(Attention Mechanism)은 1 세기 이상 통계를 괴롭히던 두 번째 문제를 해결했습니다: 수를 늘리지 않고 시스템의 메모리와 복잡성을 증가시키는 방법, 학습 가능한 매개 변수 [5] 는 학습 가능한 포인터 구조로만 볼 수 있는 것을 사용하여 우아한 해결책을 찾았습니다. 즉, 전체 문장을 기억할 필요없이 (예: 고정 차원 표현의 기계 번역의 경우) 저장해야하는 모든 것은 번역 프로세스의 중간 상태에 대한 포인터였습니다. 이것은 문장을 생성하기 전에 모델이 더 이상 전체 문장을 기억할 필요가 없기 때문에 긴 문장의 정확도를 크게 높일 수 있었습니다.
- 다단계 디자인 (예: Memory Network [6] 및 Neural programmer-interpreters [7]) 를 통해 통계 모델러가 추론에 대한 반복적인 접근법을 이용해 묘사할 수 있게 하였습니다. 이러한 기술은 딥 네트워크의 내부 상태가 반복적으로 변경가능하도록 하였습니다. 그에따라 추론 체인의 후속 단계를 진행하고, 이는 프로세서가 계산을 위해 메모리를 수정할 수 있는 것과 유사합니다.
- 또 다른 중요한 발전은 적대적 생성 신경망(Generative Adversarial Networks)의 발명입니다[8]. 밀도추정 및 생성모델에 대한 전통적인 통계 방법은, 적절한 확률 분포와 그들로부터 샘플링에 대한 (종종 근사) 알고리즘을 찾는 데 초점을 맞추었습니다. 결과적으로, 이러한 알고리즘은 통계 모델에 내재 된 유연성 부족으로 인해 크게 제한되었습니다. GAN의 중요한 혁신은 샘플러를 다른 파라미터들을 가진 임의의 알고리즘으로 대체한 것입니다. 그런 다음 Discriminator(사실상 두 샘플 테스트)에 의해 가짜와 실제 데이터를 구분할 수 없도록 조정됩니다. 임의의 알고리즘을 이용하여 데이터를 생성하는 기술을 통해, 다양한 분야의 밀도추정이 가능해 졌습니다. 달리는 얼룩말[9] 과 가짜 유명인 얼굴 [10] 의 예는 이러한 발전에 대한 증명입니다.
- 대부분의 경우 단일 GPU는 학습에 필요한 많은 양의 데이터를 처리하기에는 부족합니다. 지난 10년간 병렬 분산 학습 알고리즘을 개발하는 능력은 크게 향상되었습니다. 확장 가능한 알고리즘을 설계할 때, 가장 큰 과제 중 하나는 딥 러닝 최적화, 즉 확률적 그래디언트 디센트의 핵심은 처리할 데이터에 비해 상대적으로 작은 미니배치(minibatches)에 의존한다는 점입니다. 이러한 미니배치(minibatch) 그래서, 하나의 작은 batch 때문에 GPU를 최대한 활용하지 못합니다. 따라서 1024개의 GPU로, batch당 32개의 이미지를 처리하는 미니배치 학습은, 한번의 병합된 32K 개의 이미지 처리와 같습니다. 최근에는, 처음 Li [11] 에 이어 You[12] 와 Jia[13]가 최대 64K 개의 데이터를 ResNet50으로 ImageNet을 학습 시간을 7분 미만으로 단축시켰습니다. 초기에 이 학습 시간 측정은 일 단위 이었습니다.
- 계산을 병렬화 하는 능력은, 시뮬레이션이 가능한 상황에서 강화학습(Reinforcement learning) 분야에 결정적인 기여를 하였습니다. 이것은 바둑, 아타리 게임, 스타크래프트, 그리고 물리 시뮬레이션(예를 들면 MuJoCo의 사용) 분야에서 컴퓨터가 인간능력에 다가서거나 넘어 설 수 있도록 하는데 필요한 중요한 발전을 이끌어 내었습니다. 실례로, Silver [18]는 AlphaGo가 어떻

게 이것을 달성 했는지 설명하고 있습니다. 요컨대, 많은 양의 상태, 행동, 보상의 세가지 조합 데이터들을 사용할 수 있다면, 강화학습은 각각의 데이터들을 어떻게 연관시킬 수 있을 지, 매우 많은 양의 데이터로 시도해 볼수 있게 합니다. 시뮬레이션은 그런 방법을 제공합니다.

- 딥러닝 프레임워크는 아이디어를 널리 퍼트리는데 중요한 역할을 했습니다. 손쉬운 모델링을 위한 프레임워크의 첫 번째 세대는 [Caffe](#), [Torch](#), [Theano](#) 입니다. 많은 영향력 있는 논문들이 이 도구를 이용해 작성 되었습니다. 지금에 이르러 이들은 [TensorFlow](#)[TensorFlow](#)로 대체 되었습니다. 고수준 API 인 [Keras](#), [CNTK](#), [Caffe 2](#) 및 [Apache MxNet](#)도 이를 사용합니다. 3 세대 툴, 즉 딥러닝을 위한 명령형 툴은 틀림없이, 모델을 기술하기 위해 파이썬 NumPy와 유사한 구문을 사용하는 [Chainer](#)에 의해 주도될 것입니다. 이 아이디어는 [PyTorch](#)와 [MxNet](#)의 [Gluon API](#)에도 채택 되었습니다. 이 책에서는 [MxNet](#)의 [Gluon API](#)를 사용하였습니다.

학습을 위해 더 나은 툴을 만드는 연구자와 더 나은 신경망을 만들기 위한 통계모델러 간, 작업 시스템 분리는 많은 것들을 단순화 하였습니다. 한 예로, 2014년 카네기멜론대학의 머신러닝 박사과정 학생에게 선형 회귀 분석 모델을 학습시키는 것은 매우 중요한 과제 였습니다. 지금은 이 작업은 10줄이 안되는 코드로 완료 가능하고, 프로그래머들이 확실히 이해할수 있게 만들었습니다.

## 3.4 성공 사례

인공지능은 풀기 어려웠던 여러가지 문제들을 다른 방법으로 해결해 온 오래된 역사가 있습니다. 하나의 예로, 편지는 광학 문자 인식 기술을 이용해 정렬됩니다. 이 시스템은 90년대부터 사용되었습니다.(이것이 결국, 유명한 MINIST 및 USPS 필기 숫자셋의 출처입니다.) 동시에 은행 예금의 수표책과 신청자의 신용 점수를 읽는 데에도 적용됩니다. 또 금융 거래에서는 자동으로 사기 여부를 체크 합니다. 페이팔, 스트라이프, 알리페이, 위챗, 애플, 비자, 마스터카드 등과 같은 많은 e-커머스 지불 시스템의 근간을 이룹니다. 체스 프로그램은 수십 년간 경쟁력을 유지해 왔습니다. 머신러닝은 인터넷상에서 검색, 추천, 개인화 및 랭킹을 제공해 왔습니다. 즉, 인공 지능과 머신러닝은 비록, 종종 시야에서 숨겨져 있지만, 널리 퍼져 있습니다.

최근에야 AI가 각광을 받고 있는데, 이는 대부분 이전에 다루기 어려운 문제들을 AI가 해결하고 있기 때문입니다.

- 애플의 시리 (Siri), 아마존의 알렉사 (Alexa), 구글의 조수 (assistant)와 같은 지능형 조수들은 말로 전달된 질문에 대해 합당한 정도의 정확도로 대답 할 수 있습니다. 여기에는 조명 스위치를 켜고 이발사와 약속을 잡고, 대화형 전화 지원을 제공하는 등의 일상적인 작업이 포함됩니다. 이것은 AI가 우리 삶에 영향을 미치는 가장 두드러진 사례들 일 것입니다.
- 디지털 조수의 핵심 요소는 말을 정확하게 인식 할 수있는 능력입니다. 점차적으로 이러한 시스템의 정확도는 특정 응용 분야에서 인간과 유사한 수준에 도달 할 정도로 올라갔습니다[14].

- 객체 인식기술도 마찬가지로 오랜 시간이 걸렸습니다. 2010년에 그림에서 객체를 추정하는 것은 매우 어려운 작업이었습니다. ImageNet 벤치마크에서 Lin[15]은 top-5 에러율에서 28%를 달성하였습니다. 2017년에 Hu[16]은 이 에러율을 2.25%까지 낮추었습니다. 놀라운 결과들은 새를 분석하거나, 피부암을 진단하기 위한 분야에서 이루어졌습니다.
- 게임은 인간 지능이 우월한 마지막 분야 같았습니다. TDGammon[23]에서 시작된, 시간차 강화 학습(Temporal Difference Reinforcement Learning)을 사용한 주사위 놀이 게임처럼, 알고리즘과 컴퓨터의 발전은 광범위한 분야의 애플리케이션을 위한 알고리즘들을 이끌어 냈습니다. 주사위 놀이와는 달리, 체스는 훨씬 더 복잡한 상태공간과 행동 조합을 갖고 있습니다. Campbell[17]의 DeepBlue가 게임트리를 이용한 효율적인 검색과 특수한 목적의 하드웨어를 이용한 대량 병렬계산을 통해, Gary Kasparov를 이겼습니다. 바둑은 그것의 커다란 상태 공간 때문에, 여전히 체스 보다 더 어렵습니다. 2015년에 Silver[18]의 Monte Carlo 트리샘플링을 조합한 딥러닝을 사용한, 알파고가 인간의 수준에 다다랐습니다. 포커에 있어서 어려운 점은, 상태 공간이 크고, 전체가 관찰되지 않는다는 점입니다.(우리는 상대방의 카드를 알 수 없습니다.) Brown과 Sandholm[19]의 Libratus는 효과적으로 구조화된 전략들을 이용해 인간의 능력을 넘어섰습니다. 이것은 향상된 알고리즘이 게임 분야의 인상적인 발전에 중요한 역할을 하고 있음을 나타냅니다.
- AI 발전의 또 다른 지표는 자율 주행 자동차와 트럭의 출현입니다. 아직 완전한 자율 주행에 도달한 것은 아니지만, [모멘타](#), [테슬라](#), [엔비디아](#), [모바일아이](#), [Waymo.com](#)와 같은, 적어도 부분적인 자율성을 가능하게 하는 제품을 생산하는 회사들이 이러한 방향으로 엄청난 발전을 만들어 냈습니다. 완전한 자율 주행을 만드는 것은 매우 어려운 일입니다. 운전을 잘하기 위해서는 하나의 시스템에서 신호를 인식하고, 추론하고, 조합할 수 있는 능력을 필요로 하기 때문입니다. 요즘에는, 이러한 문제를 컴퓨터비전(Computer Vision)에 딥러닝을 주로 사용합니다. 나머지 부분은 엔지니어들이 많은 부분을 조정합니다.

다시 말하지만, 위의 목록은 인공지능으로 간주되는 것과 머신러닝이 분야에서 일어난 놀라운 발견들의 극히 일부분에 불과합니다. 오늘날의 로봇 공학, 물류, 전산생물학, 입자 물리학, 천문학은 크던 작던 머신러닝의 발전의 혜택을 누리고 있습니다. 이제 머신러닝은 엔지니어와 과학자를 위한 범용적인 도구가 되어가고 있는 것입니다.

종종 AI 종말론이나 인공 지능 특이성에 대한 질문들이 비기술적인 기사에서 제기되곤 합니다. 머신러닝 시스템이 지각을 갖게 될 것이고, 그것을 만든 프로그래머와는 독립적으로 인간의 생활에 직접적인 영향을 끼칠 것들을 결정할 것이라는 것을 두려워합니다. 하지만 이미 AI는 인간의 삶에 영향을 미치고 있습니다. 신용도가 자동으로 평가되고, 오토파일럿(autopilot)은 자동차를 안전하게 운전할 수 있게 해 주며, 입력된 통계 데이터를 사용해서 보석 허용 여부를 결정하고 있습니다. 조금 더 친근한 사례로 우리는 Alexa에게 커피 머신을 켜달라고 요청할 수 있으며, Alexa가 장치에 연결되어 있다면 요청을 수행할 수 있습니다.

다행히도 우리는 인간 창조자를 노예로 만들거나 커피를 태울 준비가 된, 지각 있는 AI 시스템과는

거리가 멍니다. 첫째, AI 시스템은 특정 목표 지향적 방식으로 설계, 학습, 배포됩니다. 그들의 행동은 범용AI에 대한 환상을 줄 수 있지만, 어디까지나 현재 인공지능 디자인의 기초는 규칙, 휴리스틱, 통계 모델의 조합입니다. 둘째, 아직까지는 일반적인 일을 수행하면서 스스로 개선하고, 스스로에 대해서 사고, 스스로의 아키텍처를 개선확장하고 개선하는 일반적인 인공지능을 위한 도구는 존재하지 않습니다.

훨씬 더 현실적인 관심사는 AI가 일상생활에서 어떻게 사용되는지입니다. 트럭 운전사 및 상점 보조자가 수행하는 사소한 일들이 자동화될 수 있고 자동화될 가능성이 있습니다. 농장 로봇은 유기농업 비용을 줄일 수 있고, 또한 수확 작업을 자동화할 것입니다. 산업 혁명의 이 단계는 사회의 많은 이들의 삶에 있어서 중대한 변화를 가져올 것입니다. 트럭 운전사와 매장 점원은 많은 주에서 가장 일반적인 직업 중 하나입니다. 게다가 통계 모델이 부주의하게 적용되면 인종적, 성별 또는 연령 편견이 발생할 수 있습니다. 이러한 알고리즘이 세심한 주의를 가지고 사용되는지 확인하는 것이 중요합니다. 이것은 인류를 멸망시킬 수 있는 악의적인 초지능의 탄생에 대해 걱정하는 것보다 훨씬 더 현실적이고 중요한 문제입니다.

### 3.5 주요 요소들

머신러닝은 데이터를 사용하여 예제 간의 변환을 학습합니다. 예를 들어 숫자 이미지는 0에서 9 사이의 정수로 변환되고, 오디오는 텍스트(음성 인식)로 변환되고, 텍스트는 다른 언어의 텍스트로 변환되거나(기계 번역), 머그샷이 이름으로 변환됩니다(얼굴 인식). 그렇게 할 때, 알고리즘이 데이터를 처리하기에 적합한 방식으로 데이터를 표현해야 하는 경우가 종종 있습니다. 이러한 특징 변환(feature transformation)의 정도는 표현 학습을 위한 수단으로, 딥 러닝을 언급하는 이유로서 종종 사용됩니다. 사실, 국제 학습 표현 회의(the International Conference on Learning Representations)의 명칭은 이것으로부터 유래합니다. 동시에 머신러닝은 통계(특정 알고리즘이 아닌 매우 큰 범위의 질문까지)와 데이터 마이닝(확장성 처리)을 똑같이 사용합니다.

현기증 나는 알고리즘 및 응용 프로그램 집합으로 인해 딥 러닝을 위한 성분이 무엇인지 구체적으로 평가하기가 어렵습니다. 이것은 피자에 필요한 재료를 고정시키는 것만큼 어렵습니다. 거의 모든 구성 요소는 대체 가능합니다. 예를 들어 다층 퍼셉트론이 필수 성분이라고 가정할 수 있습니다. 그러나 convolution만 사용하는 컴퓨터 비전 모델이 있습니다. 다른 것들은 시퀀스 모델만 사용하기도 합니다.

틀림없이 이러한 방법에서 가장 중요한 공통점은 종단간(end-to-end) 학습을 사용하는 것입니다. 즉, 개별적으로 튜닝된 구성 요소를 기반으로 시스템을 조립하는 대신 시스템을 구축한 다음 성능을 공동으로 튜닝합니다. 예를 들어, 컴퓨터 비전 과학자들은 머신러닝 모델을 구축하는 과정과 특징 엔진(Neural Engine) 프로세스를 분리하곤 했습니다. Canny 에지 검출기 [20] 와 Lowe의 SIFT 특징 추출기 [21] 는 이미지를 형상 벡터에 매핑하기 위한 알고리즘으로 10여 년간 최고 였습니다. 불행히도 알고리즘에 의해 자동으로 수행 될 때 수천 또는 수백만 가지 선택에 대한, 일관된 평가와 관련하여 인간이 독창성

으로 성취할 수 있는 많은 것들이 있습니다. 딥 러닝이 적용되었을 때, 이러한 특징 추출기는 자동으로 튜닝된 필터로 대체되어 뛰어난 정확도를 달성했습니다.

마찬가지로 자연 언어 처리에서 Salton과 McGill [22] 의 bag-of-words 모델은 오랫동안 기본 선택 이었습니다. 여기서 문장의 단어는 벡터로 매핑되며 각 좌표는 특정 단어가 발생하는 횟수에 해당합니다. 이것은 단어 순서 ('개가 사람을 물었다' 대 '사람이 개를 물었다') 또는 구두점 ('먹자, 할머니' 대 '할머니를 먹자') 을 완전히 무시합니다. 불행히도, 더 나은 특징을 수동으로 엔지니어링하는 것은 다소 어렵습니다. 반대로 알고리즘은 가능한 특징(feature) 설계의 넓은 공간을 자동으로 검색 할 수 있습니다. 이것은 엄청난 진전을 이끌어 왔습니다. 예를 들어 의미상 관련성이 있는 단어 임베딩은 벡터 공간에서 '베를린 - 독일 + 이탈리아 = 로마' 형식의 추론을 허용합니다. 다시 말하지만, 이러한 결과는 전체 시스템의 end-to-end 학습을 통해 달성됩니다.

End-to-end 학습 외에도 두 번째로 중요한 것은 파라미터 기반의 통계 설명에서 완전 비 파라미터 기반의 모델로의 전환을 경험하고 있다는 것입니다. 데이터가 부족한 경우, 유용한 모델을 얻기 위해서는 현실에 대한 가정을 단순화하는 데 의존해야합니다 (예, 스펙트럼 방법을 통해). 데이터가 풍부하면 현실에 더 정확하게 맞는 비 파라미터 기반의 모형으로 대체될 수 있습니다. 어느 정도, 이것은 컴퓨터의 가용성과 함께 이전 세기 중반에 물리학이 경험한 진전과 비슷합니다. 전자가 어떻게 동작하는지에 대한 파라메트릭 근사치를 직접 해결하는 대신, 이제 연관된 부분 미분 방정식의 수치 시뮬레이션에 의존 할 수 있습니다. 이것은 설명 가능성을 희생시키면서 종종 훨씬 더 정확한 모델을 이끌어 냈습니다.

예를 들어 적대적 생성 신경망(Generative Adversarial Networks)이 있습니다. 그래픽 모델이 적절한 확률적 공식 없이도 데이터 생성 코드로 대체됩니다. 이것은 현혹적으로 현실적으로 보일 수 있는 이미지의 모델을 이끌어 냈는데, 이것은 꽤 오랜 시간 동안 너무 어려웠던 일입니다.

이전 작업의 또 다른 차이점은 볼록하지 않은 비선형 최적화 문제(nonconvex nonlinear optimization problem)를 다루면서 차선책의 솔루션을 받아들이고, 이를 증명하기 전에 시도하려는 의지입니다. 통계적 문제를 다루는 새로운 경험주의와 인재의 급속한 유입은 실질적인 알고리즘의 급속한 발전으로 이어졌습니다 (많은 경우에도 불구하고 수십 년간 존재했던 도구를 수정하고 다시 발명하는 대신).

마지막으로 딥 러닝 커뮤니티는 학술 및 기업 경계를 넘어 도구를 공유하는 것을 자랑으로 하고 있으며, 많은 우수한 라이브러리, 통계 모델 및 학습된 네트워크를 오픈 소스로 공개합니다. 이러한 공유 정신에 따라, 이 과정을 구성하는 노트북은 배포 및 사용이 자유롭습니다. 우리는 모든 사람들이 딥 러닝에 대해 배울 수 있는 진입장벽을 낮추기 위해 열심히 노력했으며 독자가 이것의 혜택을 누릴 수 있기를 바랍니다.

## 3.6 요약

- 머신러닝은 컴퓨터 시스템이 어떻게 데이터를 사용하여 성능을 향상시킬 수 있는지 연구합니다. 통계, 데이터 마이닝, 인공 지능 및 최적화의 아이디어를 결합합니다. 종종 인위적으로 지능형 솔루션을 구현하는 수단으로 사용됩니다.
- 머신러닝의 클래스로서 표현 학습은 데이터를 나타내는 적절한 방법을 자동으로 찾는 방법에 초점을 맞춥니다. 이것은 종종 학습된 변환 진행에 의해 이루어집니다.
- 최근 진전의 대부분은 값싼 센서, 인터넷 규모의 응용 프로그램에서 발생하는 풍부한 데이터, 주로 GPU를 통한 상당한 연산 발전에 의해 촉발되었습니다.
- 전체 시스템 최적화가 핵심입니다. 구성 요소를 사용하여 좋은 성능을 얻을 수 있습니다. 효율적인 딥 러닝 프레임워크의 가용성으로 인해 이 프레임워크의 설계와 구현이 훨씬 쉬워졌습니다.

## 3.7 문제

1. 현재 작성중인 코드의 어느 부분이 '학습'될 수 있습니까? 즉, 학습에 의해서, 당신이 작성한 코드에 의해 자동으로 의사결정을 함으로써, 향상시킬 수 있습니까?
2. 당신의 코드는 경험적 설계 선택을 포함합니까? 어떠한 문제들이, 이 문제를 해결하는 방법에 대한 많은 예가 있지만, 이를 자동화 할 구체적인 방법은 없습니까? 이들은 딥 러닝을 사용할 주요 후보자 일 수 있습니다.
3. 인공지능을 하나의 새로운 산업 혁명으로 볼 때, 알고리즘과 데이터 사이의 관계는 무엇입니까? 증기 엔진과 석탄의 관계와 유사합니까? (근본적인 차이점은 무엇입니까?)
4. 다른 어떠한 분야에 end-to-end 학습법을 적용할 수 있습니까? 물리학? 공학? 계량 경제학?

## 3.8 참고문헌

- [1] Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433.
- [2] Hebb, D. O. (1949). The organization of behavior; a neuropsychological theory. A Wiley Book in Clinical Psychology. 62-78.
- [3] 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.

- [4] Bishop, C. M. (1995). Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1), 108-116.
- [5] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
- [6] Sukhbaatar, S., Weston, J., & Fergus, R. (2015). End-to-end memory networks. In *Advances in neural information processing systems* (pp. 2440-2448).
- [7] Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. arXiv preprint arXiv:1511.06279.
- [8] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).
- [9] Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. arXiv preprint.
- [10] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196.
- [11] Li, M. (2017). Scaling Distributed Machine Learning with System and Algorithm Co-design (Doctoral dissertation, PhD thesis, Intel).
- [12] You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. ArXiv e-prints.
- [13] Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ... & Chen, T. (2018). Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. arXiv preprint arXiv:1807.11205.
- [14] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., ... & Zweig, G. (2017, March). The Microsoft 2016 conversational speech recognition system. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on* (pp. 5255-5259). IEEE.
- [15] Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ... & Huang, T. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. Large scale visual recognition challenge.
- [16] Hu, J., Shen, L., & Sun, G. (2017). Squeeze-and-excitation networks. arXiv preprint arXiv:1709.01507, 7.
- [17] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. *Artificial intelligence*, 134 (1-2), 57-83.

- [18] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529 (7587), 484.
- [19] Brown, N., & Sandholm, T. (2017, August). Libratus: The superhuman ai for no-limit poker. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence.
- [20] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6), 679-698.
- [21] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 91-110.
- [22] Salton, G., & McGill, M. J. (1986). Introduction to modern information retrieval.
- [23] Tesauro, G. (1995), *Transactions of the ACM*, (38) 3, 58-68

### 3.9 Scan the QR Code to Discuss



# 4

---

## 딥러닝 맛보기

---

서두르고 싶다면, 이 장은 딥러닝을 맛보기 위해 필요한 모든 세부 사항들을 담고 있습니다. 만약 제대로 살펴보고 싶은 경우에도 여전히 이장을 읽어 보길 권장합니다. 이 장에서는 어떻게 Apache MXNet을 설치하는지, 자동미분(automatic differentiation) 사용 방법, 데이터와 메모리 조작 방법을 설명하고 있습니다. 우리는 또한 선형대수와 통계학에 대해, 독자들이 기본기 위에서 속도를 낼 수 있도록 돋기 위해 만들어진 속성 가이드/기본서를 제공합니다.

### 4.1 소개

저자들은 이 책을 쓰기 전에, 많은 노동력이 필요한 일처럼, 많은 카페인이 필요했습니다.. 상상해 봅시다. 우리는 차에 올라타서 운전을 하기 시작했습니다. 아이폰을 사용자인 Alex는 핸드폰의 음성 인식 시스템을 부르기 위해서 'Hey Siri'라고 외쳤습니다. 그러자 Mu는 '블루 보틀 커피샵으로 가는길을 알려줘'라고 명령 했습니다. 핸드폰은 그의 명령을 글로 바꿔서 화면에 빠르게 보여줍니다. 우리가 길을 묻는 것을 알아채고는 우리의 요청에 응하기 위해서 지도 앱을 띠웁니다. 지도 앱이 실행되자 마자 여러 경로를 찾아냅니다. 각 경로 옆에는 예상 소요 시간이 함께 표시됩니다. 설명을 위해서 지어 낸 이야기이긴 하지만, 이 짧은 시나리오는 스마트폰을 통해 다양한 머신 러닝 모델이 사용되는 것을 보여주고 있습니다.

여러분이 지금까지 머신 러닝을 다뤄본적이 없다면, 무엇에 대해서 이야기를 하고 있는지 모를 수도 있습니다. 어쩌면 ‘그냥 프로그래밍으로 작동하는거 아닌가요?’라고 묻거나 ‘머신 러닝이 무엇을 의미하나요?’라는 질문을 던질 수도 있습니다. 우선 확실하게 해두기 위해서, 모든 머신 러닝 알고리즘은 컴퓨터 프로그램을 작성해서 구현됩니다. 사실 우리는 다른 컴퓨터 과학의 분야와 동일한 언어와 하드웨어를 사용합니다. 하지만, 모든 컴퓨터 프로그램이 머신 러닝을 포함하는 것은 아닙니다. 두번째 질문에 대한 답은, 머신 러닝은 방대한 분야이기 때문에 정의하기 어렵습니다. 이 질문은 마치 ’수학이 무엇인가요?’라는 질문과 비슷합니다. 하지만, 여러분이 공부를 시작할 수 있도록 직관적인 설명을 충분히 해보겠습니다.

#### 4.1.1 동기 부여를 위한 예시

우리가 매일 사용하는 컴퓨터 프로그램의 대부분은 제일 원칙(first principles)을 활용해서 코드화될 수 있습니다. 여러분이 쇼핑 카트에 물건을 담으면, 이커머스 어플리케이션은 어떤 항목(여러분의 user ID 와 제품 ID를 연관시키는)을 쇼핑 카트 데이터베이스 태이블에 저장합니다. 우리는 이런 프로그램을 제일 원칙에 따라서 작성하고, 실제 고객을 본 적이 없어도 런치할 수 있습니다. 간단한 어플리케이션을 만드는데는 굳이 머신 러닝을 사용하지 말아야합니다.

(머신 러닝 개발자 커뮤니티에게 있어서는) 다행히도, 많은 문제들에 대한 해결책이 그리 쉽지만은 않습니다. 커피를 사려가는 이야기로 돌아가서, ‘Alexa’, ‘Okay, Google’,이나 ‘Siri’ 같은 *wake word*에 응답하는 프로그램을 작성한다고 생각해보세요. 컴퓨터와 코드 편집기만 사용해서 코드를 만들어 나간다고 했을 때 제일 원칙을 이용해서 어떻게 그런 프로그램을 작성할 것인가요? 조금만 생각해 봐도 이 문제가 쉽지 않다는 것을 알 수 있습니다. 매 초마다 마이크는 대략 44,000개의 샘플을 수집합니다. 소리 조각으로부터 그 소리 조각이 *wake word*를 포함하는지 신뢰있게 {yes, no}로 예측하는 률을 만들 수 있나요? 어떻게 할지를 모른다고 해도 걱정하지 마세요. 우리도 그런 프로그램을 처음부터 어떻게 작성해야하는지 모릅니다. 이것이 바로 우리가 머신 러닝을 사용하는 이유입니다.



트릭을 알려드리겠습니다. 우리는 컴퓨터에게 명시적으로 입력과 출력을 어떻게 매핑해야하는지를 알려주는 것은 모르지만, 우리 자신은 인지적인 활동을 할 수 있는 능력이 있습니다. 즉, 우리는 ‘Alexa’라는 단어를 인지하도록 컴퓨터를 프로그램하는 방법은 모르지만, 여러분은 그 단어를 인지할 수 있습니다. 이 능력을 사용해서, 우리는 오디오 샘플과 그 오디오 샘플이 *wake word*를 포함하는지 여부를 알려주는 레이블을 아주 많이 수집할 수 있습니다. 머신 러닝 접근 방법은 우리가 *wake word*를 바로 인식할 수 있도록 명시적인 시스템 디자인을 할 수 없지만, 대신 우리는 많은 수의 파라미터들을 갖는 유연한 프로그램을 정의할 수 있습니다. 이것들은 프로그램의 행동을 바꾸기 위해서 조작하는데 사

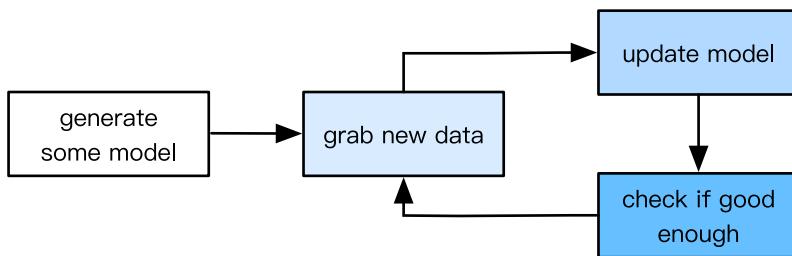
용하는 손잡이들입니다. 이 프로그램을 모델이라고 부릅니다. 일반적으로 우리의 모델은 입력을 어떤 결과로 변환하는 머신일 뿐입니다. 이 경우에는, 모델은 오디오 조각을 입력으로 받아서, {yes, no} 답을 출력으로 생성하는데, 우리는 이 결과가 wake word의 포함 여부를 담기를 원합니다.

만약 여러분이 좋은 모델을 선택했다면, ‘Alexa’ 단어를 들을 때마다 yes를 출력하는 모델을 만드는 파라미터 세트 하나가 존재할 것입니다. 마찬가지로 ‘Apricot’ 단어에 대해서 yes를 출력하는 것이 다른 조합이 있을 수 있습니다. 우리는 이 두 가지가 비슷하기 때문에, 동일한 모델이 ‘Alexa’ 인식과 ‘Apricot’ 인식에 적용되기를 기대합니다. 하지만 근본적으로 다른 입력 또는 출력을 다루기 위해서는 다른 모델이 필요할 수도 있습니다. 예를 들어, 이미지와 캡션을 매핑하는 머신과 영어 문장을 중국어 문장으로 매핑하는 모델은 서로 다른 것을 사용할 것입니다.

이미 예상했겠지만, 이 손잡이를 아무렇게나 설명할 경우, 아마도 그 모델은 ‘Alexa’, ‘Apricot’ 또는 어떤 영어 단어도 인식하지 못할 것입니다. 일반적으로 딥러닝에서는 학습(learning)은 여러 학습 기간에 걸쳐서 모델의 행동(손잡이를 돌리면서)을 업데이트하는 것을 말합니다.

학습 과정은 보통 다음과 같습니다.

1. 임의로 초기화되서 아무것도 유용한 것을 못하는 모델로 시작합니다.
2. 레이블을 갖은 데이터를 수집합니다. (예, 오디오 조각과 그에 해당하는 {yes, no} 레이블들)
3. 주어진 예제들에 모델이 조금 덜 이상하게 작동하도록 손잡이를 조정합니다.
4. 모델이 좋아질 때까지 반복합니다.



요약하면, wake word를 인식하는 코드를 작성하는 것이 아니라, 아주 많은 레이블이 있는 데이터셋이 있을 경우에 wake word를 인식하는 것을 배우는 프로그램을 작성하는 것입니다. 데이터셋을 제공해서 프로그램의 행동을 결정하는 것을 *programming with data*라고 생각할 수 있습니다.

우리는 아래 이미지들과 같은 아주 많은 고양이와 개 샘플들을 머신 러닝 시스템에 제공해서 고양이 탐지기 프로그램을 만들 수 있습니다.



이런 방법으로 탐지기는 결국에 고양이를 입력으로 받으면 아주 큰 양수를 결과로 나오게, 그리고 개를 입력으로 받으면 아주 큰 음수를 결과로 나오게 학습될 것입니다. 이 모델은 잘 모르겠으면 0과 가까운 수를 결과로 출력할 것입니다. 이 예는 머신 러닝으로 할 수 있는 일의 일부에 불과합니다.

### 4.1.2 머신 러닝의 현기증이 날 듯한 다재다능함

이것이 머신 러닝의 핵심 아이디어입니다. 정해진 행동에 대한 코드를 만드는 것이 아닌, 더 많은 경험을 하면 능력이 향상되는 프로그램을 디자인하는 것입니다. 이 기본 아이디어는 여러 형태들이 될 수 있습니다. 머신 러닝은 여러 종류의 어플리케이션 도메인 문제를 풀 수 있고, 모델의 다른 형태를 포함하고, 여러 학습 알고리즘에 따라 업데이트를 합니다. 앞에서 든 예의 경우 자동 음성 인식 문제에 적용된 지도학습(*supervised learning*)의 한 예입니다.

간단한 규칙 기반의 시스템이 실패하거나 만들기 어려운 경우에 데이터를 활용할 수 있도록 하는 다양한 도구들의 집합이 머신 러닝입니다. 이 다양함 때문에 머신 러닝을 처음 접하는 경우 혼란스러울 수 있습니다. 예를 들어, 머신 러닝 기술은 검색엔진, 자율주행차, 기계번역, 의료진단, 스팸 필터링, 게임, 얼굴 인식, 데이터 매칭, 보험 프리미엄 계산, 사진에 필터 적용 등 다양한 응용에서 이미 널리 사용되고 있습니다.

이 문제들은 겉보기에는 달라보이지만, 많은 것들은 공통적인 구조를 가지고 있고, 딥러닝 도구를 이용해서 풀 수 있습니다. 이 문제들은 코드로 직접 프로그램이 어떻게 행동해야하는지를 작성하는 것이 불가능하지만, 데이터로 프로그램을 만들 수 있다는 것이 때문에 이를 대부분이 비슷합니다. 대부분의 경우 이런 종류의 프로그램을 설명하는 가장 직접적인 언어가 수학입니다. 이 책은 다른 머신 러닝이나 뉴럴 네트워크 책과는 다르게 수학 표현은 최소화하고, 실제 예제와 실제 코드를 중심으로 설명하겠습니다.

### 4.1.3 머신 러닝 기초

wake word를 인식하는 문제를 이야기할 때, 음성 조각과 레이블로 구성된 데이터셋을 언급했습니다. (추상적이긴 하지만) 음성 조각이 주어졌을 때 레이블을 예측하는 머신 러닝 모델을 어떻게 학습시킬 수 있는지 설명했습니다. 예제로부터 레이블을 예측하는 설정은 ML의 한 종류로 지도학습(*supervised learning*)이라고 부릅니다. 딥러닝에서도 많은 접근법들이 있는데, 다른 절들에서 다루겠습니다. 머신 러닝을 진행하기 위해서는 다음 4가지가 필요합니다.

1. 데이터
2. 데이터를 어떻게 변환할지에 대한 모델
3. 우리가 얼마나 잘하고 있는지를 측정하는 loss 함수

## 4. loss 함수를 최소화하도록 모델 파라미터를 바꾸는 알고리즘

### 데이터

일반적으로는 데이터가 많아질수록 일이 더 쉬워집니다. 더 많은 데이터가 있다면, 더 강력한 모델을 학습시킬 수 있습니다. 데이터는 딥러닝 부활의 중심이고 딥러닝에서 아주 흥미로운 많은 모델들은 많은 데이터가 없으면 만들어지지 못했습니다. 머신 러닝을 수행하는 여러분들이 자주 접하게 될 몇 가지 종류의 데이터는 다음과 같습니다.

- **이미지**: 스마트폰으로 찍거나 웹에서 수집한 사진들, 인공위성 이미지, 의료 사진, 초음파, CT 또는 MRI 같은 방사선 이미지 등
- **텍스트**: 이메일, 고등학교 에세이, 트윗, 뉴스 기사, 의사의 기록, 책, 번역된 문장 등
- **오디오**: Amazon Echo, 아이폰, 또는 안드로이드 폰과 같은 스마트 디바이스에 전달될 음성 명령, 오디오 책, 전화 통화, 음악 녹음 등
- **비디오**: 텔레비전 프로그램, 영화, 유튜브 비디오, 휴대전화 수신 범위(cell phone footage), 가정 감시 카메라, 다중 카메라를 이용한 추적 등

### 모델

보통의 경우에 데이터는 이를 통해서 이루고자 하는 것과는 아주 다릅니다. 예를 들면 사람들의 사진을 가지고 있고, 사람이 행복한지 아닌지를 알아내고자 합니다. 모델이 고해상도 이미지를 받아서 행복 점수를 결과로 내도록 할 수 있습니다. 단순한 문제는 단순한 모델로 해결될 수 있지만, 이 경우에는 많은 것을 묻고 있습니다. 이를 하기 위해서는 우리의 행복 탐지기가 수십만개의 저수준(low-level) 피쳐들(픽셀값들)을 행복 점수와 같은 상당히 추상적인 것으로 변환해야합니다. 정확한 모델을 선택하는 것은 어려운 일이고, 다른 모델들은 다른 데이터셋에 더 적합합니다. 이 책에서 우리는 대부분 딥 뉴럴 네트워크에 집중할 예정입니다. 이 모델들은 데이터 변환이 많이 연속적으로 구성되어있고, 따라서 이를 딥 러닝(deep learning)이라고 합니다. 딥 네트를 논의하는 과정으로 우선 보다 간단한 또는 얕은 모델을 먼저 살펴보겠습니다.

### 손실 함수(loss function)

우리의 모델이 얼마나 잘하고 있는지 평가하기 위해서 모델의 결과와 정답(truth)을 비교할 필요가 있습니다. Loss 함수는 우리 결과가 얼마나 나쁜지를 측정하는 방법을 제공합니다. 예를 들어, 이미지로 부터 환자의 심장 박동을 추론하는 모델을 학습시켰다고 하겠습니다. 환자의 실제 심장 박동은

60bpm인데, 모델이 환자의 심장 박동을 100bpm이하고 예측했다면, 모델이 틀린 일을 하고 있다는 것을 이야기할 방법이 필요합니다.

다른 예로, 모델이 이메일이 스팸일 가능성을 점수로 알려준다면, 예측이 틀렸을 때 모델에게 알려줄 방법이 필요합니다. 일반적으로 머신 러닝의 학습 부분은 이 loss 함수를 최소화하는 것으로 구성됩니다. 보통은 모델이 많은 파라미터를 갖습니다. 이 파라미터들의 가장 좋은 값은 관찰된 데이터의 학습 데이터(*training data*)에 대한 loss를 최소화하는 것을 통해서 '배우'기를 원하는 것입니다. 불행하게도, 학습데이터에 대해서 잘 하는 것이 (본 적이 없는) 테스트 데이터에도 잘 작동한다는 것이 보장되지 않습니다. 그렇기 때문에, 우리는 두 값을 추적해야합니다.

- **학습 오류(training error):** 학습 데이터에 대해 loss를 최소화하면서 모델을 학습시킨 데이터에 대한 오류입니다. 비유하자면, 실제 시험을 준비하는 학생이 연습 시험에 대해서 모두 잘하는 것과 동일합니다. 이 결과가 좋긴하지만, 실제 시험에서도 잘본다는 보장은 없습니다.
- **테스트 오류(test error):** 보지 않은 테스트 셋에 대한 오류입니다. 학습 오류와는 상당히 다를 수 있습니다. 이런 경우 즉 보지 않은 데이터에 대한 일반화를 실패한 경우를 우리는 오버피팅(*overfitting*)이라고 합니다. 실제 생활과 비유하면, 연습 시험은 모두 잘했는데 실제 시험은 망친 것입니다.

## 최적화 알고리즘

마지막으로 loss를 최소화하기 위해서 우리는 모델과 loss 함수를 사용해서 loss를 최소화하는 파라미터 집합을 찾는 방법이 필요합니다. 뉴럴 네트워크에서 가장 유명한 최적화 알고리즘은 gradient descent 라고 불리는 방법을 따르고 있습니다. 간략하게 말하면, 각 파라미터에 대해서 파라미터를 조금 바꿨을 때 학습 셋에 대한 loss가 어느 방향으로 움직이는지를 봄으로써, loss 가 감소하는 방향으로 파라미터를 업데이트 합니다.

지금부터는 우리는 머신 러닝의 몇 가지 종류에 대해서 조금 자세히 살펴보겠습니다. 머신 러닝이 할 수 있는 것들을 나열하는 것으로 시작합니다. 목적은 이를 달성하는 방법에 대한 기술 집합(즉, 학습, 데이터 종류 등)과 보완되는 것을 기억해두세요. 아래 목록은 여러분들이 공부를 시작하고, 우리가 문제를 이야기할 때 공동 언어를 쓸 수 있을 정도입니다. 더 많은 문제는 앞으로 계속 다룰 예정입니다.

### 4.1.4 지도 학습(supervised learning)

지도학습(supervised learning)은 주어진 입력 데이터에 대한 타겟(*target*)을 예측하는 문제를 푸는 것입니다. 타겟은 종종 레이블(*label*)이라고 불리고, 기호로는  $y$ 로 표기합니다. 입력 데이터 포인트는 샘플(*sample*) 또는 인스턴스(*instance*)라고 불리기도 하고,  $x$ 로 표기됩니다. 입력  $x$ 를 예측 on  $f_{\theta}(x)$ 로 매핑하는 모델  $f_{\theta}$ 을 생성하는 것이 목표입니다.

이해를 돋기 위해서 예를 들어보겠습니다. 여러분이 의료분야에서 일을 한다면, 어떤 환자에게 심장 마비가 일어날지 여부를 예측하기를 원할 것입니다. 이 관찰, 심장 마비 또는 정상, 은 우리의 레이블  $y$  가 됩니다. 입력  $x$  는 심박동, 이완기 및 수축 혈압 등 바이탈 사인들이 될 것입니다.

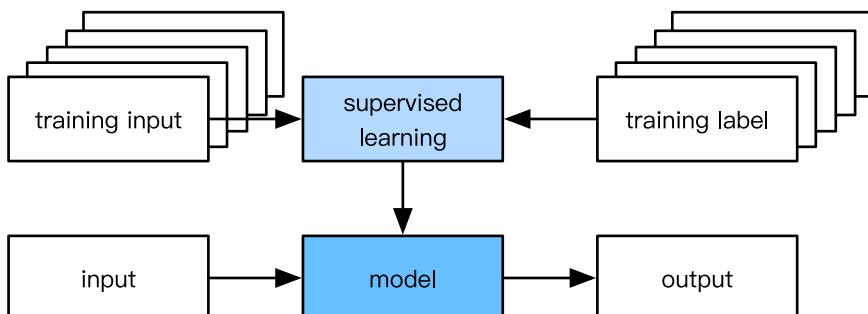
파라미터  $\theta$  를 선택하는데, 우리는(감독자) 레이블이 있는 예들,  $(x_i, y_i)$ 을 모델에게 제공하기 때문에 감독이 작동합니다. 이 때,  $x_i$  는 정확한 레이블과 매치되어 있습니다.

확률 용어로는 조건부 확률  $P(y|x)$  을 추정하는데 관심이 있습니다. 이것은 머신 러닝의 여러 접근 방법 중에 하나이지만, 지도학습은 실제로 사용되는 머신 러닝의 대부분을 설명합니다. 부분적으로, 많은 중요한 작업들이 몇 가지 이용 가능한 증거가 주어졌을 때 알 수 없는 것의 확률을 추정하는 것으로 설명될 수 있기 때문입니다 :

- CT 이미지를 보고 암 여부를 예측하기
- 영어 문장에 대한 정확한 프랑스어 번역을 예측하기
- 이번달의 제정 보고 데이터를 기반으로 다음달 주식 가격을 예측하기

‘입력으로부터 타겟을 예측한다’라고 간단하게 설명했지만, 지도학습은 입력과 출력의 타입, 크기 및 개수에 따라서 아주 다양한 형식이 있고 다양한 모델링 결정을 요구합니다. 예를 들면, 텍스트의 문자열 또는 시계열 데이터와 같은 시퀀스를 처리하는 것과 고정된 백처 표현을 처리하는데 다른 모델을 사용합니다. 이 책의 처음 9 파트에서 이런 문제들에 대해서 상세하게 다룹니다.

명백히 말하면, 학습 과정은 다음과 같습니다. 예제 입력을 많이 수집해서, 임의로 고릅니다. 각각에 대해서 ground truth를 얻습니다. 입력과 해당하는 레이블(원하는 결과)을 합쳐서 학습 데이터를 구성합니다. 학습 데이터를 지도학습 알고리즘에 입력합니다. 여기서 지도학습 알고리즘(supervised learning algorithm)은 데이터셋을 입력으로 받아서 어떤 함수(학습된 모델)를 결과로 내는 함수입니다. 그렇게 얻어진 학습된 모델을 이용해서 이전에 보지 않은 새로운 입력에 대해서 해당하는 레이블을 예측합니다.



## 회귀(regression)

아마도 여러분의 머리에 떠오르는 가장 간단한 지도학습은 회귀(regression)일 것입니다. 주택 판매 데이터베이스에서 추출된 데이터를 예로 들어보겠습니다. 각 행은 하나의 집을, 각 열은 관련된 속성(집의 면적, 침실 개수, 화장실 개수, 도심으로부터의 도보 거리 등)을 갖는 테이블을 만들습니다. 우리는 이 데이터셋의 하나의 행을 속성벡터(*feature vector*)라고 부르고, 이와 연관된 객체는 예제(*example*)이라고 부릅니다.

만약 여러분이 뉴욕이나 샌프란시스코에서 살고, 아마존, 구글, 마이크로소프트, 페이스북의 CEO가 아니라면, 여러분 집의 속성 백터(집 면적, 침실수, 화장실수, 도심까지 도보 거리)는 아마도 [100, 0, .5, 60] 가 될 것입니다. 하지만, 피츠버그에 산다면 [3000, 4, 3, 10] 와 가까울 것입니다. 이런 속성 백터는 모든 전통적인 머신 러닝 문제에 필수적인 것입니다. 우리는 일반적으로 어떤 예제에 대한 속성 백터를  $x_i$  로 표기하고, 모든 예제에 대한 속성 백터의 집합은  $X$  로 표기합니다.

결과에 따라서 어떤 문제가 회귀(*regression*)인지를 결정됩니다. 새 집을 사기 위해서 부동산을 돌아다니고 있다고 하면, 여러분은 주어진 속성에 대해서 합당한 집 가격을 추정하기를 원합니다. 타겟 값, 판매 가격은 실제 숫자(*real number*)가 됩니다. 샘플  $x_i$ 에 대응하는 각 타겟은  $y_i$ 로 표시하고, 모든 예제  $X$ 에 대한 모든 타겟들은  $y$ 로 적습니다. 타겟이 어떤 범위에 속하는 임의의 실수값을 갖는다면, 우리는 이를 회귀 문제라고 부릅니다. 우리의 모델의 목표는 실제 타겟 값을 근접하게 추정하는 예측(이 경우에는 집가격 추측)을 생성하는 것입니다. 이 예측을  $\hat{y}_i$ 로 표기합니다. 만약 표기법이 익숙하지 않다면, 다음 장들에서 더 자세히 설명할 것이기 때문에 지금은 그냥 무시해도 됩니다.

많은 실질적인 문제들이 잘 정의된 회귀 문제들입니다. 관객이 영화에 줄 평점을 예측하는 것은 회귀의 문제인데, 여러분이 2009년에 이를 잘 예측하는 대단한 알고리즘을 디자인했다면 \$1 million Netflix prize 를 받았을 것입니다. 환자가 입원일 수를 예측하는 것 또한 회귀 문제입니다. 문제가 회귀의 문제 인지를 판단하는 좋은 경험의 법칙은 얼마나 만큼 또는 얼마나 많이로 대답이 되는지 보는 것입니다.

- 이 수술은 몇 시간이 걸릴까요? - 회귀
- 이 사진에 개가 몇 마리 있나요? - 회귀

그런데 만약 주어진 문제에 대한 질문을 ‘이것은... 인가요?’ 라고 쉽게 바꿀 수 있다면, 분류의 문제입니다. 이는 다른 기본적인 문제 유형입니다. 머신 러닝을 이전에 다뤄보지 않은 경우에도 비공식적으로는 회귀의 문제들을 다뤄왔습니다. 예를 들어, 여러분의 집의 배수구를 수리하고, 수리공이  $x_1 = 3$  시간이 걸려서 하수관에서 땅어리를 제거했습니다. 이에 대해서 수리공은  $y_1 = \$350$  청구를 합니다. 여러분의 친구가 같은 수리공을 공용해서 or  $x_2 = 2$  시간 걸려서 일하고,  $y_2 = \$250$  를 청구했습니다. 어떤 사람이 하수관에서 땅어리를 제거하는 데 비용이 얼마나 될지를 물어보면, 여러분은 논리적인 추정 - 시간이 더 소요되면 더 비싸다 - 을 할 것입니다. 기본 비용이 있고, 시간당 비용이 있을 것이라고까지 추정할 것입니다. 이 가정이 맞다면, 위 두 데이터 포인트를 활용해서 수리공의 가격 구조를 알아낼 수 있습니다: 시간당 100달러 및 기본 비용 50달러. 여러분이 여기까지 잘 따라왔다며 선형

회귀에 대한 고차원의 아이디어를 이미 이해한 것입니다. (선형모델을 bias를 사용해서 디자인했습니다.)

위 예에서는 수리공의 가격을 정확하게 계산하는 파라미터를 찾아낼 수 있었습니다. 때로는 불가능한데, 예를 들면 만약 어떤 차이가 이 두 피쳐 외에 작용하는 경우가 그렇습니다. 그런 경우에는 우리는 우리의 예측과 관찰된 값의 차이를 최소화하는 모델을 학습시키고자 노력합니다. 대부분 장들에서 우리는 아주 일반적인 loss 둘 중에 하나에 집중할 것입니다. 하나는 L1 loss 로, 다음과 같고,

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

L2 loss where

다른 하나는 최소 평균 제곱 손실(least mean square loss), 즉 L2 loss 입니다. 이는 다음과 같이 표기 됩니다.

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

나중에 보겠지만,  $L_2$  loss는 우리의 데이터가 가우시안 노이즈에 영향을 받았다고 가정에 관련이 되고,  $L_1$  loss는 라플라스 분포(Laplace distribution)의 노이즈를 가정합니다.

## 분류(classification)

회귀 모델은 얼마나 많이 라는 질문에 답을 주는데는 훌륭하지만, 많은 문제들이 이 템플렛에 잘 들어맞지 않습니다. 예를 들면, 은행이 모바일앱에 수표 스캐닝 기능을 추가하고자 합니다. 이를 위해서 고객은 스마트폰의 카메라로 수표를 찍으면, 이미지에 있는 텍스트를 자동으로 이해하는 기능을 하는 머신 러닝 모델이 필요합니다. 손으로 쓴 글씨에 더 잘 동작을 해야할 필요가 있습니다. 이런 시스템은 문자인식(OCR, optical character recognition)이라고 하고, 이것이 풀려는 문제의 종류를 분류라고 합니다. 회귀 문제에 사용되는 알고리즘이는 아주 다른 알고리즘이 이용됩니다.

분류는 이미지의 픽셀값과 같은 속성 백터를 보고, 그 예제가 주어진 종류들 중에서 어떤 카테고리에 속하는지를 예측합니다. 손으로 쓴 숫자의 경우에는 숫자 0부터 9까지 10개의 클래스가 있습니다. 가장 간단한 분류의 형태는 단 두개의 클래스가 있는 경우로, 이를 이진 분류(binary classification)이라고 부릅니다. 예를 들어, 데이터셋  $X$  가 동물들의 사진이고, 이에 대한 레이블  $Y$  이 {고양이, 강아지}인 경우를 들 수 있습니다. 회귀에서는 결과가 실수 값  $\hat{y}$  가 되지만, 분류에서는 결과가 예측된 클래스인 분류기를 만들고자 합니다.

이 책에서 더 기술적인 내용을 다룰 때, 고정된 카테고리 - 예를 들면 고양이 또는 개 -에 대한 결과만을 예측하는 모델을 최적화하는 것은 어려워질 것입니다. 대신 확률에 기반한 모델로 표현하는 것이 훨씬

더 쉽습니다. 즉, 예제  $x$  가 주어졌을 때, 모델은 각 레이블  $k$  에 확률  $\hat{y}_k$  를 할당하는 것입니다. 결과가 확률값이기 때문에 모두 양수이고, 합은 1이됩니다. 이는  $K$  개의 카테고리에 대한 확률을 구하기 위해서는  $K - 1$  개의 숫자만 필요하다는 것을 의미합니다. 이진 분류를 예로 들어보겠습니다. 공정하지 않은 동전을 던져서 앞면이 나올 확률이 0.6 (60%)라면, 뒷면이 나올 확률은 0.4 (40%)다 됩니다. 동물 분류의 예로 돌아가보면, 분류기는 이미지를 보고 이미지가 고양이일 확률  $\text{Pr}(y = \text{cat}|x) = 0.9$  을 출력합니다. 우리는 이 숫자를 이미지가 고양이를 포함 것이라고 90% 정도 확신한다고 해석할 수 있습니다. 예측된 클래스에 대한 확률의 정도는 신뢰에 대한 개념을 나타냅니다. 신뢰의 개념일 뿐만 아니라, 고급 내용을 다루는 장에서는 여러 비신뢰의 개념도 논의하겠습니다.

두 개보다 많은 클래스가 있을 경우에 우리는 이 문제를 다중클래스 분류(*multiclass classification*) 이라고 합니다. 흔한 예로는 손으로 쓴 글씨 - [0, 1, 2, 3 ... 9, a, b, c, ...] - 를 인식하는 예제가 있습니다. 우리는 회귀 문제를 풀 때 L1 또는 L2 loss 함수를 최소화하는 시도를 했는데, 분류 문제에서 cross-entropy 함수가 흔히 사용되는 loss 함수는 입니다. MXNet Gluon에서는 관련된 loss 함수에 대한 내용을 [여기](#)에서 볼 수 있습니다.

가장 그럴듯한 클래스가 결정을 위해서 사용하는 것이 꼭 아닐 수도 있습니다. 여러분의 뒷풀에서 이 아름다운 벼섯을 찾는다고 가정해보겠습니다.



34

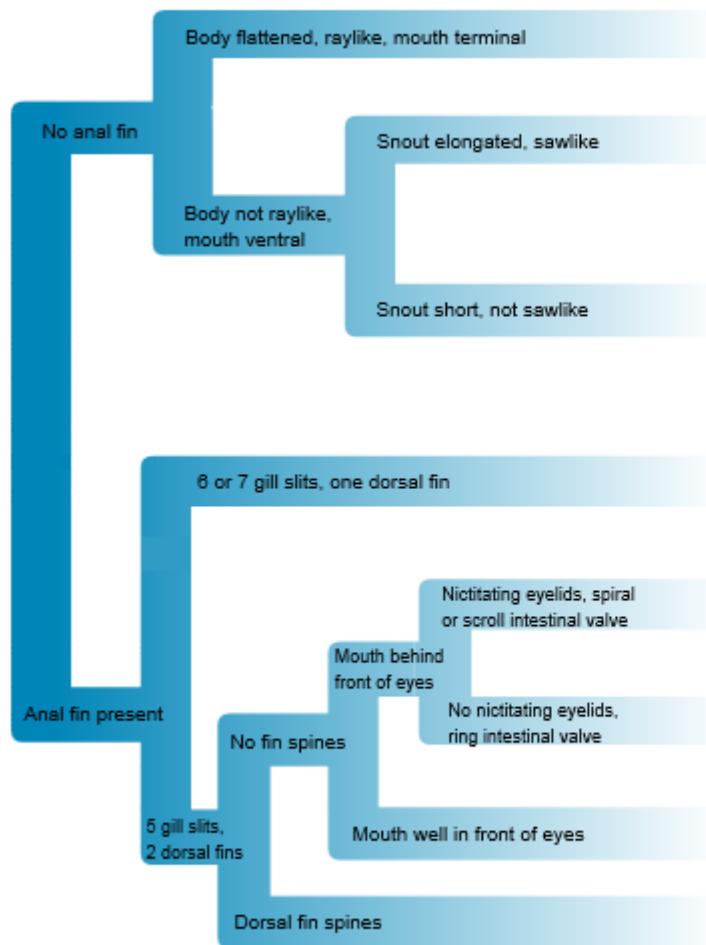
4. 딥러닝 맛보기

자, 사진이 주어졌을 때 베섯이 독이 있는 것인지를 예측하는 분류기를 만들어서 학습했다고 가정합니다. 우리의 독버섯 탐기 분류기의 결과가  $\Pr(y = \text{deathcap}|\text{image}) = 0.2$ 로 나왔습니다. 다르게 말하면, 이 분류기는 80% 확신을 갖고 이 베섯이 알광대버섯(death cap)이 아니다라고 말하고 있습니다. 하지만, 이것을 먹지는 않을 것입니다. 이 베섯으로 만들어질 멋진 저녁식사의 가치가 독버섯을 먹고 죽을 20%의 위험보다 가치가 없기 때문입니다. 이것을 수학적으로 살펴보겠습니다. 기본적으로 우리는 예상된 위험을 계산해야합니다. 즉, 결과에 대한 확률에 그 결과에 대한 이익(또는 손해)를 곱합니다.

$$L(\text{action}|x) = \mathbf{E}_{y \sim p(y|x)} [\text{loss}(\text{action}, y)]$$

따라서 베섯을 먹을 경우 우리가 얻는 loss  $L$ 은  $L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$ 인 반면에, 먹지 않을 경우 cost 또는 loss는  $L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$ 이 됩니다.

우리의 주의 깊음이 옳았습니다. 균학자들은 위 베섯이 실제로 독버섯인 알광대버섯이라고 알려줄 것이기 때문입니다. 분류 문제는 이진 분류보다 복잡해질 수 있습니다. 즉, 다중클래스 분류 문제이거나 더 나아가서는 다중 레이블 분류의 문제일 수 있습니다. 예를 들면, 계층을 푸는 분류의 종류들이 있습니다. 계층은 많은 클래스들 사이에 관계가 있는 것을 가정합니다. 따라서, 모든 오류가 동일하지 않습니다. 즉, 너무 다른 클래스로 예약하는 것보다는 관련된 클래스로 예측하는 것을 더 선호합니다. 이런 문제를 계층적 분류(*hierarchical classification*)이라고 합니다. 계층적 분류의 오랜 예는 Linnaeus가 동물을 계층으로 분류한 것을 들수 있습니다.



### SQUATINIFORMES

Triassic onwards



### PRISTIOPHORIFORMES

Jurassic onwards



### SQUALIFORMES

Jurassic onwards



### HEXANCHIFORMES

Permian onwards



### CARCHARHINIFORMES

Jurassic onwards



### LAMNIFORMES

Jurassic onwards



### ORECTOLOBIFORMES

Jurassic onwards



### HETERODONTIFORMES

Triassic onwards



## Extant Shark Orders

Recognisable external characteristics

동물 분류의 경우 푸들을 슈나이저라고 실수로 분류하는 것이 그렇게 나쁘지 않을 수 있지만, 푸들을 공룡이라고 분류한다면 그 영향이 클 수도 있습니다. 어떤 계층이 적절할지는 여러분이 모델을 어떻게 사용할 것인지에 달려있습니다. 예를 들면, 땀랑이 뱀(rattle snake)과 가터스 뱀(garter snake)은 계통 트리에서는 가까울 수 있지만, 땀랑이 뱀을 가터스 뱀으로 잘못 분류한 결과는 치명적일 수 있기 때문입니다.

## 태깅(tagging)

어떤 분류의 문제는 이진 또는 다중 클래스 분류 형태로 딱 떨어지지 않습니다. 예를 들자면, 고양이와 강아지를 구분하는 정상적인 이진 분류기를 학습시킬 수 있습니다. 현재의 컴퓨터 비전의 상태를 고려하면, 이는 상용도구을 이용해서도 아주 쉽게 할 수 있습니다. 그럼에도 불구하고, 우리의 모델이 얼마나 정확하든지 상관없이 브레맨 음악대의 사진지 주어진다면 문제가 발생할 수도 있습니다.



사진에는 고양이, 수닭, 강아지, 당나귀 그리고 배경에는 나무들이 있습니다. 우리의 모델을 이용해서 주로 무엇을 할 것인지에 따라서, 이 문제를 이진 분류의 문제로 다룰 경우 소용이 없어질 수 있습니다. 대신, 우리는 모델이 이미지에 고양이, 강아지, 당나귀 그리고 수닭이 있는 것을 알려주도록 하고 싶을 것입니다.

서로 배타적이 아닌(*not mutually exclusive*) 아닌 클래스들을 예측하는 문제를 멀티-레이블 분류라고 합니다. 자동 태깅 문제가 전형적인 멀티 레이블 분류 문제입니다. 태그의 예는 기술 문서에 붙이는 태그 - 즉, ‘머신 러닝’, ‘기술’, ‘가젯’, ‘프로그램언어’, ‘리눅스’, 클라우드 컴퓨팅’, ‘AWS’ - 를 생각해 봅시다. 일반적으로 기사는 5-10개 태그를 갖는데, 그 이유는 태그들이 서로 관련이 있기 때문입니다. ‘클라우드 컴퓨팅’에 대한 글은 ‘AWS’를 언급할 가능성이 높고, ‘머신 러닝’ 관련 글은 ‘프로그램 언어’와 관련된 것일 수 있습니다.

우리는 연구자들이 리뷰를 많이 할 수 있도록 하기 위해서 올바른 태그를 다는 것이 중요한 생물 의학 문헌을 다룰 때 이런 문제를 다뤄야합니다. 의학 국립 도서관에는 많은 전문 주석자들이 PubMed에 색인된 아티클들을 하나씩 보면서 MeSH (약 28,000개 태그의 집합) 중에 관련 된 태그를 연관시키는 일을 하고 있습니다. 이것은 시간이 많이 소모되는 일로서, 주석자들이 태그를 다는데는 보통 1년이 걸립니다. 머신 러닝을 사용해서 임시 태그를 달고, 이후에 매뉴얼 리뷰를 하는 것이 가능합니다. 실제로 몇 년 동안 BioASQ 에서는 이에 대한 대회를 열었었습니다.

## 검색(search)과 랭킹(ranking)

때로는 각 예제들에 대해서 어떤 클래스 또는 실제 값을 할당하는 것만을 원하지 않습니다. 정보 검색 분야의 경우에는 아이템 집합에 순위를 매기고 싶어합니다. 웹 검색을 예로 들어보면, 목표는 특정 페이지가 쿼리에 관련이 있는지 여부를 판단하는 것보다는 검색 결과들 중에 어떤 것이 사용자에게 먼저 보여줘야하는 것에 있습니다. 관련 검색 결과의 순서에 대해서 관심이 많고, 우리의 러닝 알고리즘은 큰 집합의 일부에 대한 순서를 매길 수 있어야합니다. 즉, 알파벳에서 처음 5개 글자가 무엇인지를 물어봤을 경우, A B C D E 를 결과로 주는 것과 C A B E D 를 결과로 주는 것에는 차이가 있습니다. 결과 집합은 같은 경우라도, 집합안에서 순서도 중요합니다.

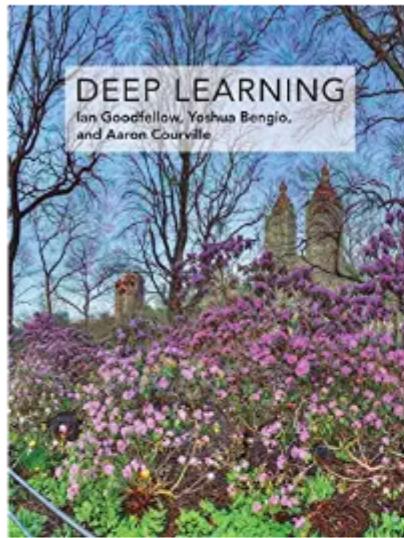
이 문제에 대한 가능한 해결방법은 가능한 집합의 원소들에 관련성 점수를 부여하고, 점수가 높은 항목들을 검색하는 것입니다. PageRank 가 관련성 점수를 적용한 예로, 특성 중 하나는 이것은 실제 쿼리에 의존하지 않는다는 것입니다. 대신, 쿼리 단어들을 포함한 결과들을 순서를 부여하는 것을 합니다. 요즘의 검색 엔진은 머신 러닝과 행동 모델을 이용해서 쿼리와 관련된 관련성 점수를 얻습니다. 이 주제만 다루는 컨퍼런스가 있습니다.

## 추천 시스템

추천 시스템은 검색과 랭킹과 관련된 또 다른 문제 세팅입니다. 사용자에게 관련된 상품을 보여주는 것이 목표이기에 문제는 비스합니다. 주요 차이점은 추천 시스템에서는 특정 사용자에 대한 개인화 (*personalization*)를 중점으로 한다는 것입니다. 예를 들어, 영화 추천의 경우에는 SciFi 에 대한 결과 페이지와 우디 앤슨 코미디에 대한 결과 페이지가 아주 다르게 나옵니다.

이런 문제는 영화, 제품 또는 음악 추천에서 발생합니다. 어떤 경우에는 고객은 얼마나 그 제품을 좋아하는지를 직접 알려주기도 합니다 (예를 들면 아마존의 제품 리뷰). 어떤 경우에는 결과에 만족하지 못한 경우 피드백을 간단하게 주기도 합니다 (재생 목록의 타이틀을 건너뛰는 형식으로). 일반적으로는 이런 시스템은 어떤 점수  $y_{ij}$  를 예측하고자 하는데, 이 예측은 사용자  $u_i$  와 제품  $p_j$  가 주어졌을 때 예상된 평점 또는 구매 확률이 될 수 있습니다.

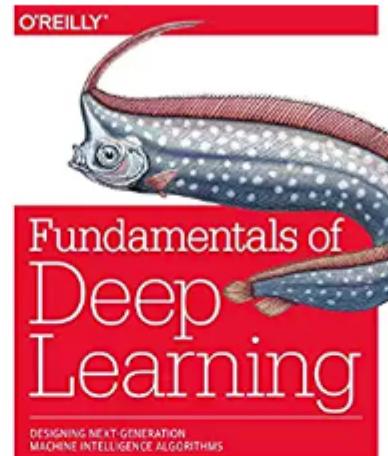
이런 모델은 어떤 사용자에 대해서 가장 큰 점수  $y_{ij}$  를 갖는 객체들의 집합을 찾아주는데, 이것이 추천으로 사용됩니다. 운영 시스템은 매우 복잡하고, 점수를 계산할 때 자세한 사용자의 활동과 상품의 특징까지 고려합니다. 아래 이미지는 아마존이 저자들의 관심을 반영한 개인화 알고리즘을 기반으로 아마존이 추천한 딥러닝 책들의 예입니다.



**Deep Learning (Adaptive Computation and Machine Learning series)** Nov 18, 2016

\$63.99 - \$72.00 ✓prime  
Hardcover, Kindle Edition

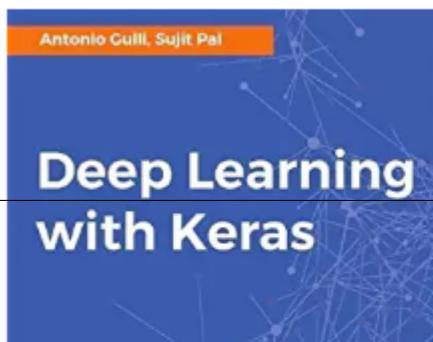
★★★★★ 85



**Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms** Jun 2017

\$22.49 - \$34.60  
✓prime | FREE One-Day  
Paperback, Kindle Edition

★★★★★ 5



4.1. 소개



## 시퀀스 러닝(sequence learning)

지금까지는 고정된 개수의 입력을 받아서 고정된 개수의 결과를 출력하는 문제를 봤습니다. 면적, 침실 개수, 옥실 개수, 다운타운까지 도보 거리와 같은 고정된 특성들로부터 주택 가격을 예측하는 것을 고려하기 앞서서, (고정된 차원의) 이미지를 고정된 수의 클래스들에 속할 예측된 확률로 매핑하는 것, 사용자 ID와 제품 ID를 받아서 별점수를 예측하는 문제들도 논의했습니다. 이 경우들은, 우리가 고정된 길이의 입력을 모델에 넣어서 결과를 얻으면, 모델은 무엇을 봤는지 바로 읽어버립니다.

만약 입력이 정말로 모두 같은 차원을 갖거나, 연속된 입력들이 서로 관련이 아무런 관련이 없을 경우에는 문제가 없습니다. 하지만, 비디오 영상의 단편을 다뤄야 한다면 어떨까요? 이 경우에는 각 단편은 서로 다른 여러 프레임들로 구성되어 있을 거입니다. 각 프레임에서 무엇이 일어나고 있는지에 대한 추측은 이전 또는 이후 프레임을 고려할 경우에 더 확실할 것입니다. 언어도 마찬가지입니다. 기계번역은 유명한 딥러닝 문제들 중에 하나입니다: 이는, 어떤 언어의 문장을 받아서, 다른 언어로 번역을 추측하는 것입니다.

이와 같은 문제는 의학에서도 찾을 수 있습니다. 우리는 중환자실의 환자를 모니터링하면서, 24시간 안에 생명에 대한 위험은 어느 정도를 넘을 경우 경고를 발생하는 모델을 원할 수 있습니다. 당연히 이 모델이 사용했던 지난 몇 시간 동안의 기록을 버리고, 오직 가장 최근의 기록만을 사용해서 예측을 하는 것을 원하지는 않을 것입니다.

머신 러닝의 아주 흥미로운 응용들이 이런 문제들에 속합니다. 이런 문제들은 시퀀스 러닝(*sequence learning*)의 예들입니다. 입력 시퀀스들을 받거나, 출력 시퀀스를 생성하는 (또는 모두) 모델이 필요합니다. 종종 우리는 이런 문제들을 seq2seq 문제라고 합니다. 언어 번역은 seq2seq 문제입니다. 음성으로 부터 텍스트를 추출하는 것 또한 seq2seq 문제입니다. 시퀀스 변환(*sequence transformation*)의 모든 종류를 고려하기는 어렵지만, 특별한 몇 가지 사례는 여기서 언급할 가치가 있습니다.

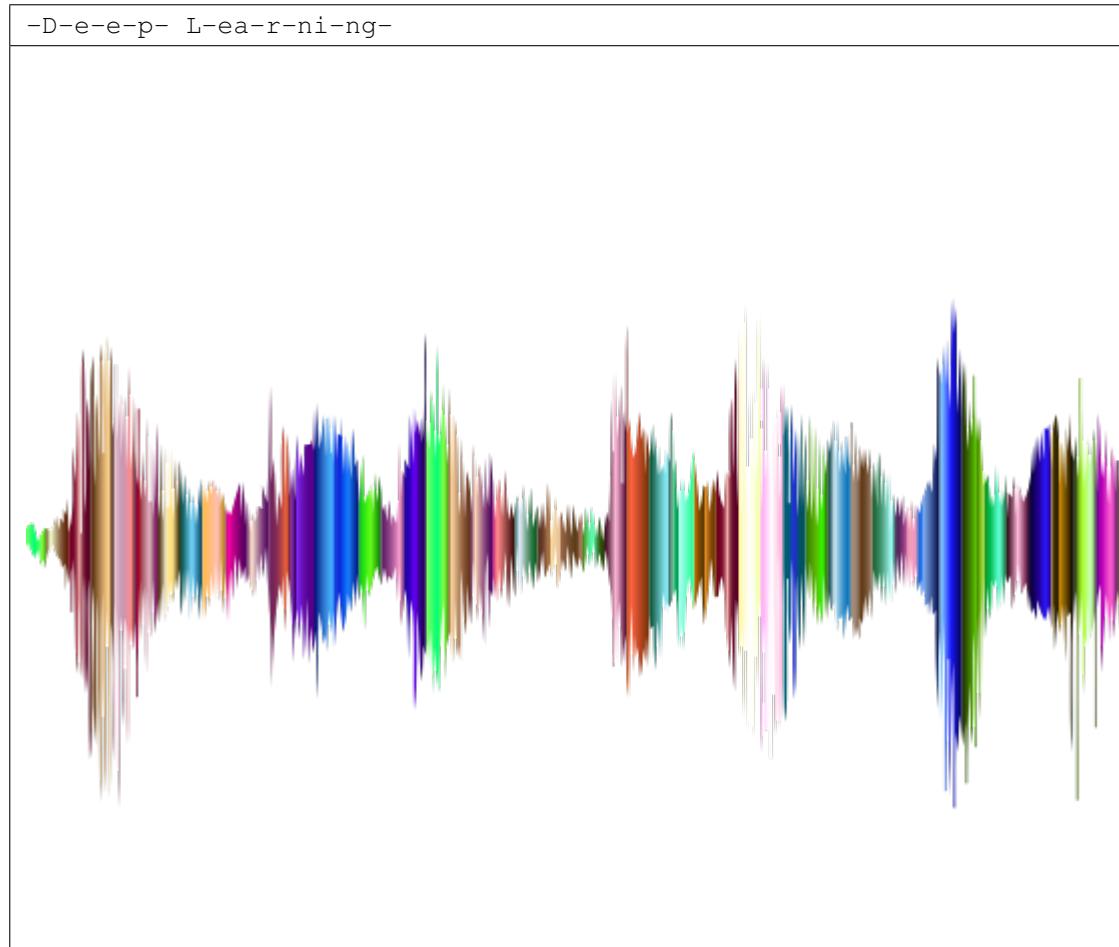
## 태깅(Tagging)과 파싱(Parsing)

이것은 텍스트 시퀀스에 속성들로 주석을다는 것입니다. 이 때, 입력과 출력의 개수가 정확하게 같습니다. 예를 들면, 동사와 주어가 어디에 있는지를 알기를 원합니다. 또는, 어떤 단어가 이름을 갖는 개체인지를 알고자 할 수 있습니다. 일반적으로, 이런 예들에서는 구조나 문법적인 추정에 근거해서 분해(decompose)를 하고 주석을다는 것이 목표입니다. 다음은 문장이 주어졌을 때 어떤 단어가 이름을 갖는 개체를 가르키는지를 태그로 주석을다는 아주 간단한 예제입니다.

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

## 자동 음성 인식(automatic speech recognition)

음성 인식에서는 입력 시퀀스  $x$ 는 화자의 음성이고, 출력  $y$ 는 화자가 말한 원문의 기록입니다. 텍스트보다 오디오 프레임의 수가 훨씬 더 많다는 점이 챌린지입니다. 즉, 수천개의 오디오 샘플이 하나의 발화된 단어에 해당되기 때문에, 오디오와 텍스트의 1:1 대응이 없습니다.



## 텍스트를 음성으로 변환하기(Text to Speech)

Text-to-Speech(TTS)는 음성 인식과 정반대입니다. 즉, 입력  $x$ 는 텍스트이고, 출력  $y$ 는 오디오 파일입니다. 이 경우, 출력은 입력보다 훨씬 깁니다. 사람이 잘못된 음성 파일을 알아내는 것은 쉽지만,

컴퓨터에게는 쉬운일이 아닙니다.

### 기계 번역(Machine Translation)

대응하는 입력과 출력이 같은 순서인 음성 인식과는 다르게, 기계 번역의 경우 순서가 뒤바뀌는 것이 중요할 수 있습니다. 즉, 하나의 시퀀스를 다른 시퀀스로 바꾸는 일을 하지만, 입력과 출력의 개수나 대응하는 데이터 포인트들의 순서가 같다고 가정하지 않습니다. 동사를 문장의 맨 끝에 놓는 독일인의 경향에 대한 예제를 생각해보겠습니다.

독일어	Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
영어	Did you already check out this excellent tutorial?
잘못된 배치	Did you yourself already this excellent tutorial looked-at?

비슷한 문제는 아주 많이 존재합니다. 예를 들면, 사용자가 웹 페이지를 어떤 순서로 읽는지를 결정하는 것은 2차원적인 레이아웃 분석 문제입니다. 비슷하게 대화 문제에서는 세상에 대한 지식과 이전 상태를 고려해야 합니다. 이것들은 활발한 연구 영역입니다.

#### 4.1.5 비지도 학습(Unsupervised learning)

지금까지 모든 예제들은 지도 학습(*supervised learning*)과 관련된 것들 였습니다. 즉, 모델에 예제들과 관련된 타겟 값들을 입력하는 학습였습니다. 지도 학습을 굉장히 특화된 일과 매우 분석적인 상사를 갖는 것으로 생각할 수 있습니다. 상사는 여러분의 어깨 넘어에서 모든 상황마다 정확히 무엇을 해야하는지를 알려주며, 이는 여러분이 상황을 행동으로 연결하는 것을 배울 때까지 지속됩니다. 그런 상사와 일하는 것은 아주 귀찮은 것입니다. 다른 한편으로는 이런 상사를 기쁘게 만드는 것은 쉽습니다. 패턴을 가능한 빨리 인지해서 그들의 행동을 모방하면 됩니다.

이와는 완전히 반대인 경우, 즉 여러분이 무엇을 해야할지를 전혀 모르는 상사와 일하는 것이 실망스러울 수 있습니다. 하지만, 여러분이 데이터 과학자가 되기를 계획하고 있다면, 이에 익숙해져야 합니다. 여러분의 상사는 엄청나게 많은 데이터를 주면서, 이것으로 데이터 과학을 좀 해봐!라고 할 수도 있습니다. 이것은 모호하기 때문에, 모호하게 들립니다. 이런 문제 종류를 우리는 비지도 학습(*unsupervised learning*)이라고 하며, 우리가 물을 수 있는 질문의 종류와 수는 우리들의 창의성에 달려있습니다. 다음 장들에서 다양한 비지도 학습 기법에 대해서 알아볼 예정이나, 여러분의 궁금증을 달래주기 위해서, 여러분이 물을 몇가지 질문들에 대해서 설명하겠습니다.

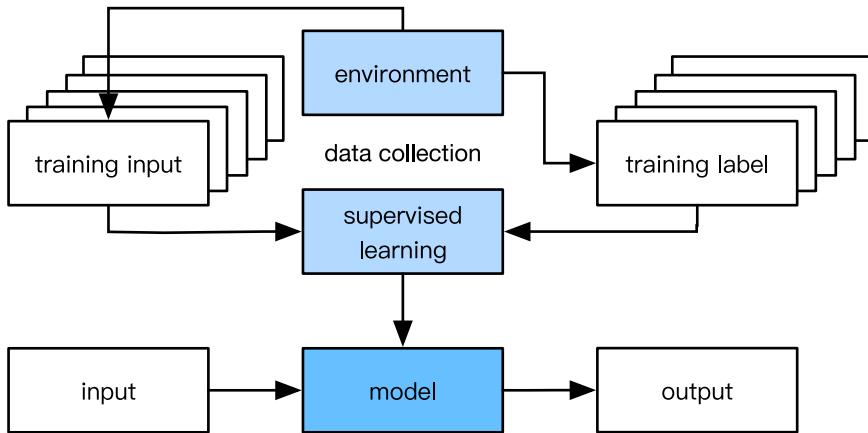
- 데이터를 정확하게 요약하는 작은 개수의 프로토타입을 찾을 수 있을까요? 사진들이 주어졌을 때, 사진들을 풍경 사진, 강아지 사진, 아기들, 고양이들, 산정상 등으로 그룹을 나눌 수 있을

까요? 비슷하게, 사용자의 브라우징 행동들에 대한 데이터가 주어졌을 때, 비슷한 행동을 하는 사용자들로 그룹을 나눌 수 있나요? 이런 문제는 일반적으로 **클러스터링(clustering)**이라고 합니다.

- 데이터에 대해서 관련있는 특성을 정확하게 포착하는 몇 개의 파라미터들 찾을 수 있을까요? 공의 궤적은 공의 속도, 직경, 질량으로 아주 잘 설명됩니다. 재봉사는 옷을 맞출 목적에 따라서 사람 몸 모양을 꽤 정확하게 설명하는 몇 개의 파라미터를 만들었습니다. 이 문제들은 **부분 공간 추정(subspace estimation)** 문제로 알려져 있습니다. 의존이 선형인 경우에는 우리는 이를 **주성분 분석(principal component analysis)**이라고 합니다.
- 유클리디안 공간에서 (임의로 조직화된) 객체에 대해서 심볼릭 성질이 잘 일치하는 표현이 있나요? 이는 **표현 학습(representation learning)**이라고 불리며, 엔티티들과 그것들의 관계를 설명하는데 사용됩니다. 예를 들면, Rome - Italy + France = Paris.
- 우리가 관찰한 많은 양의 데이터에 대한 주원인(root cause)에 대한 설명이 있나요? 예를 들면, 주택 가격, 공해, 범죄, 위치, 교육, 급여 등에 대한 인구 통계학 데이터가 있을 때, 단순히 경험적인 데이터를 기반으로 이것들이 어떻게 연관되어 있는지를 찾을 수 있나요? **방향성 그래프 모델(directed graphical model)**과 **인과 관계(causality)**가 이것을 다룹니다.
- Generative adversarial network**는 최근에 개발된 중요하고 흥미로운 네트워크입니다. 기본적으로는 이것은 데이터를 합성하는 단계적인 방법입니다. 근본적인 통계적 메카니즘은 실제 데이터와 가짜 데이터가 같은지를 점검하는 테스트들입니다. 나중에 몇 개의 노트북에 걸쳐서 살펴보도록 하겠습니다.

#### 4.1.6 환경과 상호 작용하기

지금까지는 데이터가 실제로 어디서 왔는지 또는 머신 러닝 모델이 결과를 만들 때 실제로 어떤 일이 일어나는지를 논의하지 않았습니다. 그 이유는지도 학습과 비지도 학습은 이런 이슈를 아주 복잡한 방법으로 해결하지 않기 때문입니다. 둘 중 어떤 경우에도, 우리는 많은 양의 데이터를 받아서, 환경과 상호 작용을 하지 않고 패턴 인식을 수행합니다. 이런 모든 학습은 알고리즘이 환경과 분리된 상태에서 일어나기 때문에, 이를 **오프라인 학습(offline learning)**이라고 합니다. 지도 학습의 경우, 프로세스는 다음과 같습니다.



오프라인 학습의 간결함은 매력적입니다. 좋은 점은 다른 문제들을 고려할 필요 없는 격리된 상태에서 패턴 인식을 하면 된다는 것이지만, 단점은 문제를 공식화(problem formulation)하는 것이 아주 제한적이라는 것입니다. 여러분이 더 의욕적이거나, Asimov's Robot Series를 읽으면서 자랐다면, 예측을 할 뿐만 아니라 세상에서 행동을 취할 수 있는 인공지능 봇을 떠올릴 수도 있습니다. 이는 단지 예측을 하는 것이 아니라 행동을 선택하는 것을 생각해봐야한다는 것을 뜻합니다. 나아가 예측과는 달리 행동은 실제로 환경에 영향을 미칩니다. 우리가 지능적인 에이전트를 학습시키기를 원한다면, 행동이 에이전트의 미래 관찰에 미치는 영향에 대해서도 설명해야합니다.

환경과의 상호 작용을 고려하는 것은 완전히 새로운 모델링에 대한 질문을 가져옵니다. 환경이,

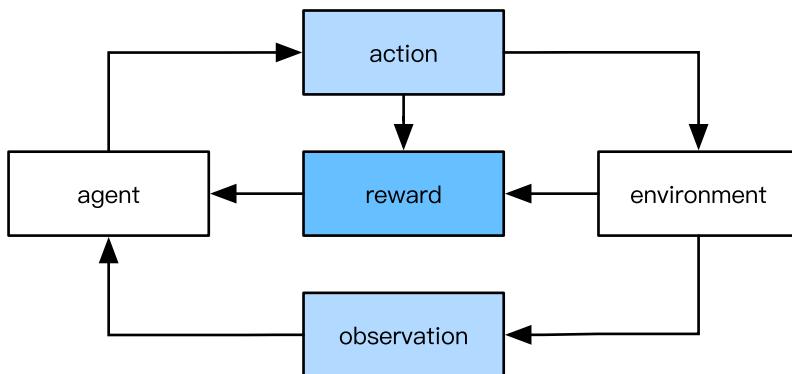
- 우리가 이전에 행한 것을 기억하나요?
- 우리를 돋기를 원하나요? 즉, 사용자가 음성 인식기에 텍스트를 읽는 경우.
- 우리를 이야기를 원하나요? 즉, 스팸 필터(스팸 발송자에 대항하는) 같은 적대적 설정 또는 (상대와) 게임을 플레이하는 것.
- not care (as in most cases)?
- 변하는 역동성을 가지고 있나요(시간에 따라 그대로인지 변하는지)?

마지막 질문은 공변량 변화(covariate shift) 문제를 일으킵니다.(학습 데이터와 테스트 데이터가 다를 때) 숙제는 TA가 제출하는 반면에, 시험은 강의를 하는 사람이 낸 문제를 풀 때 이런 것을 경험했을 것입니다. 환경과 상호 작용을 명시적으로 고려하는 강화 학습(reinforcement learning)과 적대적 학습(adversarial learning)에 대해서 간단히 살펴보겠습니다.

## 강화 학습(reinforcement learning)

환경과 작용을 하면서 행동을 위하는 에이전트를 만드는 머신 러닝에 관심이 있다면, 여러분은 아맞도 강화 학습(*reinforcement learning, RL*)에 집중할 것입니다. 로보틱스, 분석 시스템 심지어는 비디오 게임에 대한 AI 개발에 적용될 수 있습니다. 딥 뉴럴 네트워크를 RL 문제에 적용한 딥 강화 학습(*deep reinforcement learning, DRL*)이 인기가 높습니다. deep Q-network that beat humans at Atari games using only the visual input와 AlphaGo program that dethroned the world champion at the board game Go이 유명한 두 예입니다.

강화 학습은 에이전트가 일련의 시간 단계(*time steps*)에 걸쳐서 환경과 작용을 문제에 대한 일반적인 기술을 정의합니다. 각 시간 단계  $t$ 마다, 에이전트는 환경으로 부터 어떤 관찰  $o_t$ 를 받아서, 행동  $a_t$ 을 선택해야하고, 이 행동은 다시 환경으로 전달이 됩니다. 그리고 나면 에이전트는 환경으로 부터 보상  $r_t$ 를 받습니다. 그 후, 에이전트는 다음 관찰을 받아서, 다음 행동을 취하는 것을 반복합니다. RL에 이전트의 행동은 정책(*policy*)에 의해서 정의됩니다. 간단하게 말하면, 정책(*policy*)은 (환경으로 부터 얻은) 관찰을 행동으로 매핑시키는 함수입니다. 강화 학습의 목표는 좋은 정책을 만드는 것입니다.



RL 프레임워크의 일반성을 과장하는 것은 어렵습니다. 예를 들어, 임의의 강화 학습 문제를 RL 문제로 바꿀 수 있습니다. 분류의 문제를 예로 들어보겠습니다. 우리는 한 행동(*action*)이 각 클래스 대응하는 에이전트를 만들 수 있습니다. 그리고는 원래의 지도 학습에서 사용하는 손실 함수(*loss function*)와 완전히 같은 보상을 주는 환경을 생성합니다.

즉, RL은 지도 학습이 해결하지 못하는 많은 문제를 해결할 수 있습니다. 예를 들면, 지도 학습에서는 학습 입력에 대한 정확한 레이블이 있어야합니다. 하지만 RL의 경우에는 관찰에 대해서 환경은 최적의 행동을 알려준다고 가정하지 않습니다. 일반적으로 우리는 어떤 보상을 받을 뿐입니다. 즉, 환경은 어떤 행동이 보상을 받을 수 있는지 조차 알려주지 않을 수도 있습니다.

체스 게임을 생각해 보세요. 오직 실제 보상은 게임이 끝났을 때 주어집니다. 즉, 우리는 게임을 이겨서 보상 1을 받거나, 졌을 때 보상 -1을 받게됩니다. 강화 학습은 신뢰 할당 문제(*credit assignment problem*)

을 다뤄야 합니다. 10월 11일에 승진을 하는 직원의 경우도 같은 것이 적용됩니다. 승진은 지난 해에 잘 선택된 많은 행동들을 반영할 가능성이 높습니다. 미래에 더 많은 승진을 하기 위해서는 어떠한 행동이 승진으로 이어졌었는지를 알아내야 합니다.

강화 학습은 일부 관찰의 문제(problem of partial observability)도 다뤄야 할 수도 있습니다. 즉, 현재 관찰은 현재 상태에 대한 모든 것을 알려주지 않을 수도 있습니다. 청소 로봇이 집에서 비슷하게 생긴 여러 옷장들 중 하나에 갇혔다고 하겠습니다. 로봇의 정확한 위치 (즉 상태)를 추론하는 것은 옷장에 들어가기 전에 이전 상태를 고려해야 할 것입니다.

마지막으로, 어느 시점에 강화 학습은 좋은 정책 하나를 알게 될 수도 있지만, 에이전트가 전혀 시도하지 않은 더 좋은 정책들이 많이 있을 수도 있습니다. 강화 학습은 정책이 주는 현재 알려진 최고의 전략을 개척(exploit)하거나 지식을 얻기 위해서 단기적인 보상을 포기하면서 전략들의 공간을 탐사(explore)할지를 지속적으로 선택해야 합니다.

### MDP, 반딧(bandit), 친구들

일반적인 강화 학습 문제는 아주 일반적인 설정을 갖습니다. 행동은 이후의 관찰에 영향을 줍니다. 보상은 선택된 행동에 대해서만 관찰됩니다. 환경은 전체 또는 일부만 관찰될 수 있습니다. 이 모든 복잡성을 한번에 설명하기에는 연구자들에게 너무 많은 것을 묻는 것입니다. 그 결과로 연구자들은 강화 학습의 특별한 경우들을 연구해왔습니다.

환경이 완전히 관찰되는 경우에 우리는 RL 문제를 마코브 결정 프로세스(*Markov Decision Process, MDP*)라고 부릅니다. 상태가 이전의 행동에 의존하지 않을 경우, 우리는 이런 문제를 문맥상 반딧 문제(*contextual bandit problem*)이라고 합니다. 상태가 없고, 초기에 알려지지 않은 보상을 갖는 가능한 행동의 집합만 있는 경우, 이런 문제를 우리는 고전적인 멀티-암드 반딧 문제(*multi-armed bandit problem*)이라고 합니다.

#### 4.1.7 요약

머신 러닝은 광대합니다. 우리는 아마도 전체를 다루지 못할 것입니다. 하지만, 뉴럴 네트워크는 간단하고, 기초적인 수학만 요구합니다. 그럼 시작해봅시다. (하지만 먼저 MXNet을 설치하세요)

#### 4.1.8 Scan the QR Code to Discuss



## 4.2 Gluon 시작하기

시작하기에 앞서 노트북들을 수행하는데 필요한 코드를 다운로드하고 설치해야 합니다. 이 절을 그냥 넘어가도 다음 절들에서 설명하는 이론적인 내용을 이해하는데는 문제가 없지만, 직접 코드를 실행해 보는 것을 꼭 권장합니다. 코드를 작성하고, 수정하고 결과를 보는 것이 이 책으로부터 더 많은 것을 얻을 수 있는 방법이기 때문입니다. 요약하자면, 아래 단계들을 수행하면 됩니다.

1. conda 설치하기
2. 이 책에 필요한 코드를 다운로드하기
3. GPU를 가지고 있는데 아직 설치하지 않았다면 GPU 드라이버 설치하기
4. MXNet와 이 책의 예제들을 수행할 conda 환경 빌드하기

### 4.2.1 Conda

라이브러리를 설치의 간편함을 위해서, 유명한 Python 패키지 관리자인 `conda`를 권장합니다.

1. [conda.io/en/latest/miniconda.html](https://conda.io/en/latest/miniconda.html)를 방문해서 여러분의 OS에 맞는 `Miniconda`를 다운로드한 후 설치하세요.
2. 리눅스인 경우 `source ~/.bashrc`, MacOS인 경우 `source ~/.bash_profile`를 수행해서 셸을 업데이트 합니다. PATH 환경 변수에 Anaconda를 꼭 추가하세요.
3. 이 책의 노트북들이 담긴 tarball를 다운로드하세요. 이 파일은 [www.d2l.ai/d2l-en-1.0.zip](http://www.d2l.ai/d2l-en-1.0.zip)에 있습니다. 또는 GitHub 리포지토리에서 최신 버전을 복사해도 됩니다.
4. ZIP 파일의 압축을 풀어서 원하는 디렉토리에 옮겨 놓으세요.

리눅스의 경우 아래 명령들을 명령행에서 수행하면 됩니다. MacOS를 사용하는 경우, 처음 두 줄에 있는 Linux를 MacOS로 바꾸면 됩니다. Windows 사용자는 위 가이드에 있는 링크들을 참고하세요.

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh  
sh Miniconda3-latest-Linux-x86_64.sh  
mkdir d2l-en  
cd d2l-en  
curl https://www.d2l.ai/d2l-en-1.0.zip -o d2l-en.zip  
unzip d2l-en-1.0.zip  
rm d2l-en-1.0.zip
```

## 4.2.2 GPU 지원

기본 설정인 경우 MXNet은 모든 컴퓨터에서 수행되는 것을 보장하기 위해서 GPU 지원을 하지 않도록 설치됩니다. 만약 여러분의 컴퓨터에 GPU가 있다면, conda 환경을 수정해서 CUDA가 활성화된 빌드를 다운로드해야합니다. 당연하지만, 필요한 드라이버들이 설치되어 있어야 합니다. 아래 단계들을 수행하세요.

1. 여러분이 가지고 있는 GPU에 맞는 NVIDIA Drivers 가 설치되어 있는지 확인하세요.
2. GPU를 위한 프로그래밍 언어인 CUDA 를 설치하세요.
3. 딥러닝을 위한 다양한 최적화 라이브러리를 제공하는 CUDNN 을 설치하세요.
4. 추가적인 가속을 위해서 필요한 경우, TensorRT 를 설치합니다.

설치는 다소 오래 걸리고, 라이센스 동의를 해야하기도 하고 여러 설치 스크립트를 사용해야합니다. 여러분의 OS나 하드웨어에 따라 설치 방법은 달라질 수 있습니다.

위 과정을 마친 후, environment.yml 의 환경 설정을 업데이트 합니다. mxnet 을 mxnet-cu92 또는 설치된 CUDA 버전에 맞도록 바꿉니다. 만약 CUDA 8.0 버전이 설치된 경우, mxnet-cu92 를 mxnet-cu80 으로 바꿔야 합니다. 이것은 꼭 conda 환경을 만들기 전에 해야합니다. 그렇지 않은 경우, 빌드를 다시 수행해야하기 때문입니다. 리눅스 사용자는 아래 명령으로 수정할 수 있습니다. (Windows 사용자는 Notepad 등을 사용해서 environment.yml를 직접 수정합니다.)

```
cd d2l  
emacs environment.yml
```

## 4.2.3 Conda 환경

요약하면, conda는 Python 라이브러리들을 반복가능하고 안정적인 방법으로 설정하는 방법을 제공합니다. 이를 통해서 모든 소프트웨어의 의존성을 만족시킬 수 있습니다. 시작하기에 필요한 것을 다음과

같습니다.

1. conda를 이용해서 환경을 생성하고 활성화합니다. 편리하게 하기 위해서 environment.yml 파일에 모든 설정을 넣어 놨습니다.
2. 환경을 활성화합니다.
3. Jupyter 노트북을 열어서 실험을 시작합니다.

## Windows

명령창을 엽니다.

```
conda env create -f environment.yml  
cd d2l-en  
activate gluon  
jupyter notebook
```

이후에 라이브러리들을 다시 활성화할 필요가 있다면, 첫번째 줄은 넘어가세요. 이는 여러분의 설정이 활성화되어 있음을 확인시켜줍니다. Jupyter Notebook(jupyter notebook)을 대신에, JupyterLab(jupyter lab)을 사용할 수 있음을 알아두세요. 활성화된 conda gluon 환경에서 conda install jupyterlab 을 실행해서 직접 설치할 수 있습니다.

웹 브라우저 통합이 잘 작동한다면, Jupyter 를 실행하면 웹 브라우저의 새 창이 생성됩니다. 만약 그렇지 않다면, 직접 <http://localhost:8888>를 열어보세요. 어떤 노트북은 필요한 데이터와 사전-학습(pre-trained) 모델을 자동으로 다운로드하기도 합니다. MXNET\_GLUON\_REPO 변수를 수정해서 리포지토리 위치를 바꿀 수도 있습니다.

## Linux와 MacOS

리눅스에서도 비슷하게 합니다. 단지, anaconda 명령 옵션이 약간 다릅니다.

```
conda env create -f environment.yml  
cd d2l-en  
source activate gluon  
jupyter notebook
```

Windows와 다른 설치들의 주요 차이는 Windows에서는 activate gluon 을 사용하지만, Linux나 MacOS는 source activate gluon 을 사용한다는 것입니다. 그 외에는 Windows에서 했던 것과 동일합니다. 더 강력한 환경을 원한다면 JupyterLab을 설치하세요.

#### 4.2.4 Gluon 업데이트하기

새로운 CUDA 버전이나 MXNet을 설치해서 리포지토리를 업데이트하고 싶다면, conda 명령으로 간단하게 할 수 있습니다. 아래 명령으로 패키지를 업데이트 합니다.

```
cd d2l-en  
conda env update -f environment.yml
```

#### 4.2.5 요약

- conda는 모든 소프트웨어의 의존성을 만족하도록 해주는 Python 패키지 매니저입니다.
- environment.yml은 이 책에 필요한 모든 설정을 가지고 있습니다. 모든 노트북에 대한 다운로드 링크를 제공하고, GitHub를 통해서도 제공합니다.
- GPU를 가지고 있다면, GPU 드라이버와 관련 설정을 업데이트하세요. 학습 시간을 아주 많이 줄일 수 있습니다.

#### 4.2.6 문제

1. 이 책의 코드를 다운로드하고, 실행 환경을 설치하세요.
2. 질문이나 도움이 필요하면 이 절의 아래에 있는 링크를 따라 포럼을 이용하세요.
3. 포럼의 계정을 만들고 여러분을 소개하세요.

#### 4.2.7 Scan the QR Code to Discuss



## 4.3 데이터 조작(data manipulation)

데이터를 변경할 수 없다면 아무것도 할 수 없습니다. 일반적으로 우리는 데이터를 사용해서 두가지 중요한 일을 합니다. (i) 데이터를 얻고, (ii) 컴퓨터에서 들어오면 처리하기. 데이터를 저장하는 방법을 모른다면 데이터를 얻는 것의 의미가 없으니, 합성된 데이터를 다루는 것부터 시작하겠습니다.

MXNet에서 데이터를 저장하고 변경하는 주요 도구인 NDArray를 소개하겠습니다. NumPy를 사용해봤다면, NDArray가 NumPy의 다차원 배열과 디자인 측면에서 비슷하다는 것을 눈치챌 것입니다. 하지만, 주요 장점들이 있습니다. 첫번째로는 NDArray는 CPU, GPU 그리고 분산 클라우드 아키텍처에서 비동기 연산을 지원합니다. 두번째는, 자동 미분을 지원합니다. 이 특징들 때문에 NDArray는 머신 러닝에 이상적인 요소라고 할 수 있습니다.

### 4.3.1 시작하기

이 절에서는 여러분은 기본적인 것을 다룰 것입니다. 요소별 연산이나 표준 분포와 같은 기본적인 수학 내용을 이해하지 못해도 걱정하지 마세요. 다음 두 절에서 필요한 수학과 어떻게 코드로 구현하는지를 다룰 예정입니다. 수학에 대해서 더 알고 싶다면, 부록에 “[Math](#)”를 참고하세요.

MXNet과 MXNet의 ndarray 모듈을 import 합니다. 여기서는 ndarray를 nd라고 별칭을 주겠습니다.

```
[1]: import mxnet as mx  
from mxnet import nd
```

우리가 만들 수 있는 가장 단순한 객체는 벡터입니다. arange 는 12개의 연속된 정수를 갖는 행 벡터를 생성합니다.

```
[2]: x = nd.arange(12)  
x  
[2]:  
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]  
<NDArray 12 @cpu(0)>
```

x를 출력할 때 나온 <NDArray 12 @cpu(0)>로 부터, 우리는 이것이 길이가 12인 일차원 배열이고, CPU의 메인메모리에 저장되어 있다는 것을 알 수 있습니다. @cpu(0)에서 0은 아무런 의미가 없고, 특정 코어를 의미하지도 않습니다.

NDArray 인스턴스의 shape은 shape 속성으로 얻습니다.

```
[3]: x.shape  
[3]: (12,)
```

size 속성은 NDArray 인스턴스의 원소 총 개수를 알려줍니다. 우리는 벡터를 다루고 있기 때문에 두 결과는 같습니다.

```
[4]: x.size  
[4]: 12
```

행 벡터를 3행, 4열의 행렬로 바꾸기 위해서, 즉 shape을 바꾸기 위해서 reshape 함수를 사용합니다. 모양(shape)이 바뀌는 것을 제외하고는 x의 원소와 크기는 변하지 않습니다.

```
[5]: x = x.reshape((3, 4))  
x  
[5]:  
[[ 0.  1.  2.  3.]  
 [ 4.  5.  6.  7.]  
 [ 8.  9. 10. 11.]]  
<NDArray 3x4 @cpu(0)>
```

위와 같이 행렬의 모양을 바꾸는 것은 좀 이상할 수 있습니다. 결국, 3개 행을 갖는 행렬을 원한다면 총 원소의 개수가 12개가 되기 위해서 열이 4가 되어야한다는 것을 알아야합니다. 또는, NDArray에게 행의 개수가 몇 개이든지 모든 원소를 포함하는 열이 4개인 행렬을 자동으로 찾아내도록 요청하는 것도 가능합니다. 즉, 위의 경우에는 `x.reshape((3, 4))` 는 `x.reshape((-1, 4))` 와 `x.reshape((3, -1))` 와 같습니다.

```
[6]: nd.empty((3, 4))  
[6]:  
[[1.805371e+13 4.574118e-41 1.805371e+13 4.574118e-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]]  
<NDArray 3x4 @cpu(0)>
```

`empty` 메소드는 모양(shape)에 따른 메모리를 잡아서 원소들의 값을 설정하지 않고 행렬을 반환합니다. 이는 아주 유용하지만, 원소들이 어떤 형태의 값이라도 가질 수 있는 것을 의미합니다. 이는 매우 큰 값들일 수도 있습니다. 하지만, 일반적으로는 행렬을 초기화하는 것을 원합니다.

보통은 모두 0으로 초기화하기를 원합니다. 수학자들은 이차원 보다 큰 객체들에 대해서는 특별한 이름을 쓰지 않지만, 우리는 이것들은 텐서(tensor)라고 부르겠습니다. 모든 원소가 0이고 모양(shape)이

(2,3,4)인 텐서를 하나 만들기 위해서 다음과 같이 합니다.

```
[7]: nd.zeros((2, 3, 4))  
  
[7]:  
[[[0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]]  
  
[[0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]]]  
<NDArray 2x3x4 @cpu(0)>
```

NumPy 처럼, 모든 원소가 1인 텐서를 만드는 방법은 다음과 같습니다.

```
[8]: nd.ones((2, 3, 4))  
  
[8]:  
[[[1. 1. 1. 1.]  
 [1. 1. 1. 1.]  
 [1. 1. 1. 1.]]  
  
[[1. 1. 1. 1.]  
 [1. 1. 1. 1.]  
 [1. 1. 1. 1.]]]  
<NDArray 2x3x4 @cpu(0)>
```

Python 리스트를 이용해서 NDArray의 각 원소 값을 지정하는 것도 가능합니다.

```
[9]: y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])  
y  
  
[9]:  
[[2. 1. 4. 3.]  
 [1. 2. 3. 4.]  
 [4. 3. 2. 1.]]  
<NDArray 3x4 @cpu(0)>
```

어떤 경우에는, NDArray의 값을 임의로 채우기를 원할 때가 있습니다. 이는 특히 뉴럴 네트워크의 파라미터로 배열을 사용할 때 일반적입니다. 아래 코드는 shape가 (3,4) NDArray를 생성하고, 각 원소는 평균이 0이고 분산이 1인 표준 분포로부터 임의로 추출한 값을 갖습니다.

```
[10]: nd.random.normal(0, 1, shape=(3, 4))

[10]:
[[ 2.2122064  0.7740038  1.0434405  1.1839255 ]
 [ 1.8917114 -1.2347414 -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]]
<NDArray 3x4 @cpu(0)>
```

### 4.3.2 연산들

우리는 종종 배열에 함수를 적용할 필요가 있습니다. 아주 간단하고, 굉장히 유용한 함수 중에 하나로 요소별(element-wise) 함수가 있습니다. 이 연산은 두 배열의 동일한 위치에 있는 원소들에 대해 스칼라 연산을 수행하는 것입니다. 스칼라를 스칼라로 매핑하는 함수를 사용하면 언제나 요소별(element-wise) 함수를 만들 수 있습니다. 수학 기호로는 이런 함수를  $f : \mathbb{R} \rightarrow \mathbb{R}$ 로 표현합니다. 같은 모양(shape)의 두 벡터  $\mathbf{u}$  와  $\mathbf{v}$  와 함수  $f$ 가 주어졌을 때, 모든  $i$ 에 대해서  $c_i \leftarrow f(u_i, v_i)$ 을 갖는 벡터  $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ 를 만들 수 있습니다. 즉, 우리는 스칼라 함수를 벡터의 요소별로 적용해서 벡터 함수  $F : \mathbb{R}^d \rightarrow \mathbb{R}^d$ 를 만들었습니다. MXNet에서는 일반적인 표준 산술 연산자들(+,-,/,\*,\*\*\*)은 모양(shape)이 무엇이든지 상관없이 두 텐서의 모양(shape)이 같을 경우 모두 요소별 연산으로 간주되어 계산됩니다. 즉, 행렬을 포함한 같은 모양(shape)을 갖는 임의의 두 텐서에 대해서 요소별 연산을 수행할 수 있습니다.

```
[11]: x = nd.array([1, 2, 4, 8])
y = nd.ones_like(x) * 2
print('x =', x)
print('x + y', x + y)
print('x - y', x - y)
print('x * y', x * y)
print('x / y', x / y)

x =
[1. 2. 4. 8.]
<NDArray 4 @cpu(0)>
x + y
[ 3. 4. 6. 10.]
<NDArray 4 @cpu(0)>
x - y
[-1. 0. 2. 6.]
<NDArray 4 @cpu(0)>
x * y
```

(continues on next page)

```
[ 2.  4.  8. 16.]
<NDArray 4 @cpu(0)>
x / y
[0.5 1.  2.  4. ]
<NDArray 4 @cpu(0)>
```

제곱과 같은 더 많은 연산들이 요소별 연산으로 적용될 수 있습니다.

```
[12]: x.exp()
[12]:
[2.7182817e+00 7.3890562e+00 5.4598148e+01 2.9809580e+03]
<NDArray 4 @cpu(0)>
```

요소별 연산과 더불어서, dot 함수를 이용한 행렬의 곱처럼 행렬의 연산들도 수행할 수 있습니다. 행렬  $x$  와  $y$  의 전치행렬에 대해서 행렬의 곱을 수행해보겠습니다.  $x$  는 행이 3, 열이 4인 행렬이고,  $y$  는 행이 4개 열이 3개를 갖도록 전치시킵니다. 두 행렬을 곱하면 행이 3, 열이 3인 행렬이 됩니다. (이것이 어떤 의미인지 헷갈려도 걱정하지 마세요. *linear algebra* 절에서 행렬의 연산에 대한 것들을 설명할 예정입니다.)

```
[13]: x = nd.arange(12).reshape((3,4))
y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
nd.dot(x, y.T)
[13]:
[[ 18.  20.  10.]
 [ 58.  60.  50.]
 [ 98. 100.  90.]]
<NDArray 3x3 @cpu(0)>
```

여러 NDArray들을 합치는 것도 가능합니다. 이를 위해서는 어떤 차원(dimension)을 따라서 합쳐야 하는지를 알려줘야 합니다. 아래 예제는 각각의 차원 0 (즉, 행들)과 차원 1 (열들)을 따라서 두 행렬을 합칩니다.

```
[14]: nd.concat(x, y, dim=0)
nd.concat(x, y, dim=1)
[14]:
[[ 0.  1.  2.  3.  2.  1.  4.  3.]
 [ 4.  5.  6.  7.  1.  2.  3.  4.]
 [ 8.  9. 10. 11.  4.  3.  2.  1.]]
```

(continues on next page)

&lt;NDArray 3x8 @cpu(0)&gt;

NumPy에서와 같이 논리 문장을 사용해서 이진 NDArray를 만들 수 있습니다.  $x == y$  를 예로 들어보겠습니다. 만약  $x$  와  $y$  가 같은 원소가 있다면, 새로운 NDArray는 그 위치에 1을 갖고, 다른 값이면 0을 갖습니다.

[15]: `x == y`

[15]:

```
[[0. 1. 0. 1.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
<NDArray 3x4 @cpu(0)>
```

NDArray의 모든 요소를 더하면 하나의 원소를 갖는 NDArray가 됩니다.

[16]: `x.sum()`

[16]:

```
[66.]
<NDArray 1 @cpu(0)>
```

`asscalar` 함수를 이용해서 결과를 Python의 스칼라로 바꿀 수 있습니다. 아래 예제는  $x$  의  $\ell_2$  놈을 계산합니다. 이 결과는 하나의 원소를 갖는 NDArray이고, 이를 Python의 스칼라 값으로 바꿉니다.

[17]: `x.norm().asscalar()`

[17]: 22.494442

표기를 편하게 하기 위해서 `y.exp()`, `x.sum()`, `x.norm()`, 등을 각각 `nd.exp(y)`, `nd.sum(x)`, `nd.norm(x)` 처럼 쓸 수도 있습니다.

### 4.3.3 브로드캐스트 메카니즘

위 절에서 우리는 같은 모양(shape)의 두 NDArray 객체에 대한 연산을 어떻게 수행하는지를 살펴봤습니다. 만약 모양(shape)이 다른 경우에는 NumPy와 같이 브로드캐스팅 메카니즘이 적용됩니다: 즉, 두 NDArray가 같은 모양(shape)을 갖도록 원소들이 복사된 후, 요소별로 연산을 수행하게 됩니다.

```
[18]: a = nd.arange(3).reshape((3, 1))
b = nd.arange(2).reshape((1, 2))
a, b
```

```
[18]: (
    [[0.]
     [1.]
     [2.]]
<NDArray 3x1 @cpu(0)>,
[[0. 1.]]
<NDArray 1x2 @cpu(0)>)
```

a와 b는 각각 (3x1), (1x2) 행렬이기 때문에, 두 행렬을 더하기에는 모양(shape)이 일치하지 않습니다. NDArray는 이런 상황을 두 행렬의 원소들을 더 큰 행렬 (3x2)로 ‘브로드캐스팅’ 해서 해결합니다. 즉, 행렬 a는 컬럼을 복제하고, 행렬 b는 열을 복제한 후, 요소별 덧셈을 수행합니다.

```
[19]: a + b
[19]:
[[0. 1.]
 [1. 2.]
 [2. 3.]]
<NDArray 3x2 @cpu(0)>
```

#### 4.3.4 인덱싱과 슬라이싱(slicing)

다른 Python 배열처럼 NDArray의 원소들도 인덱스를 통해서 지정할 수 있습니다. Python에서처럼 첫 번째 원소의 인덱스는 0이고, 범위는 첫번째 원소는 포함하고 마지막은 포함하지 않습니다. 즉, 1:3은 두번째와 세번째 원소를 선택하는 범위입니다. 행렬에서 행들을 선택하는 예는 다음과 같습니다.

```
[20]: x[1:3]
[20]:
[[ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 2x4 @cpu(0)>
```

값을 읽는 것 말고도, 행렬의 원소 값을 바꾸는 것도 가능합니다.

```
[21]: x[1, 2] = 9
x
[21]:
[[ 0.  1.  2.  3.]
 [ 4.  5.  9.  7.]]
```

(continues on next page)

```
[ 8.  9. 10. 11.]
<NDArray 3x4 @cpu(0)>
```

여러 원소에 같은 값을 할당하고 싶을 경우에는, 그 원소들에 대한 인덱스를 모두 지정해서 값을 할당하는 것으로 간단히 할 수 있습니다. 예를 들어 `[0:2, :]` 는 첫번째와 두번째 행을 의미합니다. 행렬에 대한 인덱싱을 이야기해왔지만, 벡터나 2개 보다 많은 차원을 갖는 텐서에도 동일하게 적용됩니다.

```
[22]: x[0:2, :] = 12
x
[22]:
[[12. 12. 12. 12.]
 [12. 12. 12. 12.]
 [ 8.  9. 10. 11.]]
<NDArray 3x4 @cpu(0)>
```

### 4.3.5 메모리 절약하기

앞의 예제들 모두 연산을 수행할 때마다 새로운 메모리를 할당해서 결과를 저장합니다. 예를 들어, `y = x + y` 를 수행하면, 원래의 행렬 `y` 에 대한 참조는 제거되고, 새로 할당된 메모리를 참조하도록 동작합니다. 다음 예제에서는, 객체의 메모리 주소를 반환하는 Python의 `id()` 함수를 이용해서 이를 확인해보겠습니다. `y = x + y` 수행 후, `id(y)` 는 다른 위치를 가리키고 있습니다. 이렇게 되는 이유는 Python은 `y + x` 연산 결과를 새로운 메모리에 저장하고, `y` 가 새로운 메모리를 참조하도록 작동하기 때문입니다.

```
[23]: before = id(y)
y = y + x
id(y) == before
[23]: False
```

이는 두가지 이유로 바람직하지 않을 수 있습니다. 첫번째로는 매번 불필요한 메모리를 할당하는 것을 원하지 않습니다. 머신 러닝에서는 수백 메가 바이트의 파라미터들을 매 초마다 여러 번 업데이트를 수행합니다. 대부분의 경우 우리는 이 업데이트를 같은 메모리(*in-place*)를 사용해서 수행하기를 원합니다. 두번째는 여러 변수들이 같은 파라미터를 가리키고 있을 수 있습니다. 같은 메모리에 업데이트를 하지 않을 경우, 메모리 누수가 발생하고, 래퍼런스가 유효하지 않은 파라미터를 만드는 문제가 발생할 수 있습니다.

다행히도 MXNet에서 같은 메모리 연산은 간단합니다. 슬라이스 표기법  $y[:] = <\text{expression}>$ 을 이용하면 이전에 할당된 배열에 연산의 결과를 저장할 수 있습니다. `zeros_like` 함수를 사용해서 동일한 모양(shape)을 갖고 원소가 모두 0인 행렬을 하나 복사해서 이것이 어떻게 동작하는지 보겠습니다.

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

id(z): 140197672472192
id(z): 140197672472192
```

멋져 보이지만,  $x+y$ 는 결과값을 계산하고 이를  $y[:]$ 에 복사하기 전에 이 값을 저장하는 임시 버퍼를 여전히 할당합니다. 메모리를 더 잘 사용하기 위해서, `ndarray` 연산(이 경우는 `elemwise_add`)을 직접 호출해서 임시 버퍼의 사용을 피할 수 있습니다. 모든 `ndarray` 연산자가 제공하는 `out` 키워드를 이용하면 됩니다.

```
[25]: before = id(z)
nd.elemwise_add(x, y, out=z)
id(z) == before

[25]: True
```

$x$  값이 프로그램에서 더 이상 사용되지 않을 경우,  $x[:] = x + y$ 이나  $x += y$ 로 연산으로 인한 메모리 추가 사용을 줄일 수 있습니다.

```
[26]: before = id(x)
x += y
id(x) == before

[26]: True
```

### 4.3.6 NDArray와 NumPy간 상호 변환

MXNet NDArray를 NumPy로 변환하는 것은 간단합니다. 변환된 배열은 메모리를 공유하지 않습니다. 이것은 사소하지만 아주 중요합니다. CPU 또는 GPU 하나를 사용해서 연산을 수행할 때, NumPy가 동일한 메모리에서 다른 일을 수행하는 것을 MXNet이 기다리는 것을 원하지 않기 때문입니다. `array` 와 `asnumpy` 함수를 이용하면 변환을 할 수 있습니다.

```
[27]: import numpy as np

a = x.asnumpy()
print(type(a))
b = nd.array(a)
print(type(b))

<class 'numpy.ndarray'>
<class 'mxnet.ndarray.NDArray'>
```

### 4.3.7 문제

1. 이 절의 코드를 실행하세요. 조건문  $x == y$  를  $x < y$  이나  $x > y$  로 바꿔서 결과가 어떻게 되는지 확인하세요.
2. 다른 모양(shape)의 행렬들에 브로드캐스팅이 적용되는 연산을 수행하는 두 NDArray를 바꿔보세요. 예를 들면 3 차원 텐서로 바꿔보세요. 예상한 결과도 같나요?
3. 행렬 3개  $a, b, c$  가 있을 경우,  $c = \text{nd.dot}(a, b.T) + c$  를 가장 메모리가 효율적인 코드로 바꿔보세요.

### 4.3.8 Scan the QR Code to Discuss



## 4.4 선형 대수

자, 이제 데이터를 저장하고 조작하는 방법을 배웠으니, 모델에 대한 대부분을 이해하는데 필요한 기초적인 선형 대수 일부를 간단하게 살펴보겠습니다. 기초적인 개념과 관련된 수학 표기법, 그리고 코드로의 구현까지 모두 소개할 것입니다. 기본 선형 대수에 익숙하다면, 이 절은 빨리 읽거나 다음 절로 넘어가도 됩니다.

```
[1]: from mxnet import nd
```

#### 4.4.1 스칼라(scalar)

선형대수나 머신 러닝을 배워본 적이 없다면, 아마도 한번에 하나의 숫자를 다루는데 익숙할 것입니다. 예를 들어, 팔로 알토의 기온이 화씨 52도입니다. 공식 용어를 사용하면 이 값은 스칼라(*scalar*)입니다. 이 값을 섭씨로 바꾸기를 원한다면,  $c = (f - 32) * 5/9$  공식에  $f$  값으로 52 대입하면 됩니다. 이 공식에서 각 항들 32, 5, 9은 스칼라 값입니다. 플레이스 홀더  $c$  와  $f$  를 변수라고 부르고, 아직 정해지지 않은 스칼라 값들을 위해 있습니다.

수학적인 표기법으로는 우리는 스칼라를 소문자( $x_{\text{ff}}, y_{\text{ff}}, z_{\text{ff}}$ )로 표기 합니다. 또한 모든 스칼라에 대한 공간은  $\mathcal{R}_{\text{ff}}$ 로 적습니다. 이후에 공간이 정확히 무엇인지를 알아보겠지만, 편의상 지금은  $x_{\text{ff}}$  가 스칼라라고 이야기하는 것은  $x \in \mathcal{R}_{\text{ff}}$ 로 표현하기로 하겠습니다.

MXNet에서 스칼라는 하나의 원소를 갖는 NDArray로 표현됩니다. 아래 코드에서는 두 개의 스칼라를 생성하고, 친숙한 수치 연산 - 더하기, 빼기, 나누기, 그리고 제곱을 수행합니다.

```
[2]: x = nd.array([3.0])
y = nd.array([2.0])

print('x + y = ', x + y)
print('x * y = ', x * y)
print('x / y = ', x / y)
print('x ** y = ', nd.power(x,y))

x + y =
[5.]
<NDArray 1 @cpu(0)>
x * y =
[6.]
<NDArray 1 @cpu(0)>
x / y =
[1.5]
<NDArray 1 @cpu(0)>
x ** y =
[9.]
<NDArray 1 @cpu(0)>
```

`asscalar` 메소드를 이용하면 NDArray를 Python의 float 형으로 변환할 수 있습니다. 하지만 이렇

게 하는 것은 좋은 아이디어가 아님을 알아두세요. 그 이유는 이를 수행하는 동안, 프로세스 제어를 Python에게 줘야하기 때문에 NDArray는 결과를 내기 위한 다른 것들을 모두 멈춰야 합니다. 어렵게도, Python은 병렬로 일을 처리하는데 좋지 못합니다. 이런 연산을 코드나 네트워크에서 수행한다면 학습하는데 오랜 시간이 걸릴 것입니다.

```
[3]: x.asscalar()  
[3]: 3.0
```

#### 4.4.2 벡터(vector)

벡터를 `[1.0, 3.0, 4.0, 2.0]`처럼 숫자들의 리스트로 생각할 수 있습니다. 벡터의 각 숫자는 하나의 스칼라 변수로 이루어져 있습니다. 이 숫자들을 우리는 벡터의 원소 또는 구성 요소라고 부릅니다. 종종 우리는 실제 세상에서 중요한 값을 담은 벡터들에 관심을 갖습니다. 예를 들어 채무 불이행 위험을 연구하고 있다면, 모든 지원자를 원소가 수입, 재직 기간, 이전의 불이행 횟수 등을 포함한 벡터와 연관을 지을지도 모릅니다. 병원 내 환자들의 심장 마비 위험을 연구하는 사람은, 환자들을 최근 바이탈 사인, 콜레스테롤 지수, 하루 운동 시간 등을 원소로 갖는 벡터로 표현 할 것입니다. 수학적인 표기법을 이용할 때 벡터는 굵은 글씨체로 소문자 (`u, v, w`) 를 사용해서 표현합니다. MXNet에서는 임의의 숫자를 원소로 갖는 1D NDArray를 이용해서 벡터를 다루게 됩니다.

```
[4]: x = nd.arange(4)  
print('x = ', x)  
  
x =  
[0. 1. 2. 3.]  
<NDArray 4 @cpu(0)>
```

첨자를 사용해서 벡터의 요소를 가리킬 수 있습니다. 즉, `uff` 의 4번째 요소는 `u4ff`로 표현합니다. `u4ff`는 스칼라이기 때문에 굵은 글씨가 아닌 폰트로 표기하는 것을 유의하세요. 코드에는 NDArray의 `iff` 번째 인덱스로 이를 지정합니다.

```
[5]: x[3]  
[5]: [3.]  
<NDArray 1 @cpu(0)>
```

#### 4.4.3 길이, 차원(dimensionality), 모양(shape)

앞 절에서 소개한 개념 몇 개를 다시 살펴보겠습니다. 벡터는 숫자들의 배열입니다. 모든 배열은 길이를 갖듯이, 벡터도 길이를 갖습니다. 벡터  $\mathbf{x}$  가  $n$  개의 실수 값을 갖는 스칼라들로 구성되어 있다면, 이는 수학적인 표현으로  $\mathbf{x} \in \mathcal{R}^n$  로 적습니다. 벡터의 길이는 일반적으로 차원(dimension) 이라고 합니다. Python 배열처럼, NDArray의 길이도 Python의 내장 함수 `len()` 를 통해서 얻을 수 있습니다.

벡터의 길이는 `.shape` 속성으로도 얻을 수 있습니다. `shape`은 NDArray 객체의 각 축에 대한 차원의 목록으로 표현됩니다. 벡터는 축이 하나이기 때문에, 벡터의 모양(shape)은 하나의 숫자로 표현됩니다.

```
[6]: x.shape  
[6]: (4,)
```

차원(dimension)이라는 단어가 여러가지 의미로 사용되기 때문에, 헷갈릴 수도 있습니다. 어떤 경우에는 벡터의 차원(dimensionality) 을 벡터의 길이 (원소들의 수)로 사용하기도 하고, 어떤 경우에는 배열의 축의 개수로 사용되기도 합니다. 후자의 경우에는 스칼라는 0 차원을 갖고, 벡터는 1차원을 갖습니다.

흔동을 줄이기 위해서, 우리는 2D 배열 또는 3D 배열이라고 말할 때, 축이 각각 2개 3개인 배열을 의미하도록 합니다. 하지만 만약  $n$ -차원 벡터라고 하는 경우에는, 길이가  $n$  인 벡터를 의미합니다.

```
[7]: a = 2  
x = nd.array([1,2,3])  
y = nd.array([10,20,30])  
print(a * x)  
print(a * x + y)  
  
[2. 4. 6.]  
<NDArray 3 @cpu(0)>  
  
[12. 24. 36.]  
<NDArray 3 @cpu(0)>
```

#### 4.4.4 행렬(matrix)

벡터가 오더 0인 스칼라를 오더 1로 일반화하는 것처럼, 행렬은 1D에서 2D로 벡터를 일반화합니다. 일반적으로 대문자 ( $A, B, C$ )로 표현하는 행렬은 코드에서는 축이 2개인 배열로 표현합니다. 시각화

한다면, 행렬은 원소  $a_{ij}$  가  $i$ -열,  $j$ -행에 속하는 표로 그릴 수 있습니다.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \text{ff}$$

MXNet에서는  $n$  행,  $m$  열을 갖는 행렬을 만드는 방법은 두 요소를 갖는  $(n, m)$  모양(shape)을 이용해서 ones 또는 zeros 함수를 호출을 통해 ndarray를 얻는 것입니다.

```
[8]: A = nd.arange(20).reshape((5, 4))
print(A)
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>
```

행렬은 유용한 자료 구조입니다. 행렬을 이용해서 서로 다른 양식의 변형을 갖는 데이터를 구성할 수 있습니다. 예를 들어보면, 행렬의 행들은 서로 다른 환자에 대한 정보를, 열은 서로 다른 속성에 대한 정보를 의미할 수 있습니다.

행렬  $A$  의 스칼라 원소  $a_{ij}$  을 지정하는 방법은 행( $i$ )과 열( $j$ )에 대한 인덱스를 지정하면 됩니다. : 를 사용해서 공백으로 두면, 해당 차원의 모든 원소를 의미합니다. (앞 절에서 봤던 방법입니다.)

행렬을 전치하는 방법은 T 를 이용합니다. 전치 행렬은 만약  $B = A^T$  이면, 모든  $i$ ff 과  $j$ ff 에 대해서  $b_{ij} = a_{ji}$  인 행렬을 의미합니다.

```
[9]: print(A.T)
```

```
[[ 0.  4.  8. 12. 16.]
 [ 1.  5.  9. 13. 17.]
 [ 2.  6. 10. 14. 18.]
 [ 3.  7. 11. 15. 19.]]
<NDArray 4x5 @cpu(0)>
```

#### 4.4.5 텐서(tensor)

벡터가 스칼라를 일반화하고, 행렬이 벡터를 일반화하는 것처럼 더 많은 축을 갖는 자료 구조를 만들 수 있습니다. 텐서(tensor)는 임의의 개수의 축을 갖는 행렬을 표현하는 일반적인 방법을 제공합니다. 예를 들어 벡터는 1차 오더(order) 텐서이고, 행렬은 2차 오더(order) 텐서입니다.

3D 자료 구조를 갖는 이미지를 다룰 때 텐서를 사용하는 것은 아주 중요하게 됩니다. 즉, 각 축이 높이, 넓이, 그리고 세가지 색(RGB) 채널을 의미합니다. 지금은 이것에 대한 내용은 생략하고, 기본적인 것을 확실하게 아는 것을 목표로 하겠습니다.

```
[10]: X = nd.arange(24).reshape((2, 3, 4))
print('X.shape =', X.shape)
print('X =', X)

X.shape = (2, 3, 4)
X =
[[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
 [[12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]]
<NDArray 2x3x4 @cpu(0)>
```

#### 4.4.6 텐서 연산의 기본 성질

스칼라, 벡터, 행렬, 그리고 어떤 오더를 갖는 텐서들은 우리가 자주 사용할 유용한 특성들을 가지고 있습니다. 요소별 연산(element-wise operation)의 정의에서 알 수 있듯이, 같은 모양(shape)들에 대해서 연산을 수행하면, 요소별 연산의 결과는 같은 모양(shape)을 갖는 텐서입니다. 또 다른 유용한 특성은 모든 텐서에 대해서 스칼라를 곱하면 결과는 같은 모양(shape)의 텐서입니다. 수학적으로 표현하면, 같은 모양(shape)의 두 텐서  $X$ 와  $Y$ 가 있다면  $\alpha X + Y$  는 같은 모양(shape)을 갖습니다.

```
[11]: a = 2
x = nd.ones(3)
y = nd.zeros(3)
print(x.shape)
print(y.shape)
```

(continues on next page)

```
print((a * x).shape)
print((a * x + y).shape)

(3,)
(3,)
(3,)
(3,)
```

더하기와 스칼라 곱으로 보존되는 특성이 모양(shape) 뿐만은 아닙니다. 이 연산들은 벡터 공간의 맴버쉽을 보존해줍니다. 하지만, 여러분의 첫번째 모델을 만들어서 수행하는데 중요하지 않기 때문에, 이 장의 뒤에서 설명하겠습니다.

#### 4.4.7 합과 평균

임의의 텐서들로 수행할 수 있는 조금 더 복잡한 것은 각 요소의 합을 구하는 것입니다. 수학기호로는 합을  $\sum f_i$ 로 표시합니다. 길이가  $d$ 인 벡터  $u$ 의 요소들의 합은  $\sum_{i=1}^d u_i$ 로 표현하고, 코드에서는 `nd.sum()` 만 호출하면 됩니다.

```
[12]: print(x)
print(nd.sum(x))

[1. 1. 1.]
<NDArray 3 @cpu(0)>

[3.]
<NDArray 1 @cpu(0)>
```

임의의 모양(shape)을 갖는 텐서의 원소들의 합을 비슷하게 표현할 수 있습니다. 예를 들어  $m \times n$  행렬  $A$ 의 원소들의 합은  $\sum_{i=1}^m \sum_{j=1}^n a_{ij}$ 이고, 코드로는 다음과 같습니다.

```
[13]: print(A)
print(nd.sum(A))

[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]]
```

(continues on next page)

```
[16. 17. 18. 19.]
<NDArray 5x4 @cpu(0)>
```

```
[190.]
<NDArray 1 @cpu(0)>
```

합과 관련된 것으로 평균(*mean*)이 있습니다. (*average*라고도 합니다.) 평균은 원소들의 합을 원소들의 개수로 나눠서 구합니다. 어떤 벡터  $\mathbf{u}$ 의 평균을 수학 기호로 표현하면  $\frac{1}{d} \sum_{i=1}^d u_i$ 이고, 행렬  $A$ 에 대한 평균은  $s \frac{1}{n \cdot m} \sum_{i=1}^m \sum_{j=1}^n a_{ij}$ 이 됩니다. 코드로 구현하면, 임의의 `shape`을 갖는 텐서의 평균은 `nd.mean()`을 호출해서 구합니다.

```
[14]: print(nd.mean(A))
print(nd.sum(A) / A.size)
```

```
[9.5]
<NDArray 1 @cpu(0)>
```

```
[9.5]
<NDArray 1 @cpu(0)>
```

#### 4.4.8 점곱(dot product)

지금까지는 원소들 사이에 연산인 더하기와 평균에 대해서 살펴봤습니다. 이 연산들이 우리가 할 수 있는 전부라면, 선형대수를 별도의 절로 만들어서 설명할 필요가 없을 것입니다. 즉, 가장 기본적인 연산들 중에 하나로 점곱(dot product)이 있습니다. 두 벡터,  $\mathbf{u}$  와  $\mathbf{v}$ , 가 주어졌을 때, 점곱,  $\mathbf{u}^T \mathbf{v}$ , 은 요소들끼리 곱을 한 결과에 대한 합이 됩니다. 즉,  $\mathbf{u}^T \mathbf{v} = \sum_{i=1}^d u_i \cdot v_i$ .

```
[15]: x = nd.arange(4)
y = nd.ones(4)
print(x, y, nd.dot(x, y))
```

```
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>
```

(continues on next page)

[ 6 . ]

&lt;NDArray 1 @cpu(0)&gt;

두 벡터의 점곱 `nd.dot(x, y)`, 은 원소들끼리의 곱을 수행한 후, 합을 구하는 것과 동일합니다.

[ 16 ] : `nd.sum(x * y)`

[ 16 ] :

[ 6 . ]  
<NDArray 1 @cpu(0)>

점곱은 다양한 경우에 유용하게 사용됩니다. 예를 들어, 가중치들의 집합  $\mathbf{w}$ 에 대해서, 어떤 값  $u$ 의 가중치를 적용한 합은 점곱인  $\mathbf{u}^T \mathbf{w}$ 으로 계산될 수 있습니다. 가중치들이 0 또는 양수이고, 합이 1 ( $\sum_{i=1}^d w_i = 1$ )인 경우, 행렬의 곱은 가중치 평균(*weighted average*)을 나타냅니다. 길이가 1인 두 벡터 (길이가 무엇인지는 아래에서 `norm`을 설명할 때 다릅니다)가 있을 때, 점곱을 통해서 두 벡터 사이의 코사인 각을 구할 수 있습니다.

#### 4.4.9 행렬-벡터 곱

점곱을 어떻게 계산하는지 알아봤으니, 행렬-벡터 곱을 알아볼 준비가 되었습니다. 우선 행렬  $A$ 와 열벡터  $\mathbf{x}$ 를 시각적으로 표현하는 것으로 시작합니다.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

행렬을 다시 행벡터 형태로 표현이 가능합니다.

$$A = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix}, \quad \mathbf{x}$$

여기서 각  $\mathbf{a}_i^T \in \mathbb{R}^m$ 는 행렬의  $i$  번째 행을 표시하는 행벡터입니다.

그러면 행렬-벡터 곱  $\mathbf{y} = A\mathbf{x}$  은 컬럼 벡터  $\mathbf{y} \in \mathbb{R}^{n \times 1}$ 이며, 각 원소  $y_i$  는 점곱  $\mathbf{a}_i^T \mathbf{x}$ 입니다.

$$A\mathbf{x} = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T \mathbf{x} \\ \mathbf{a}_2^T \mathbf{x} \\ \vdots \\ \mathbf{a}_n^T \mathbf{x} \end{pmatrix}$$

즉, 행렬  $A \in \mathbb{R}^{n \times m}$  로 곱하는 것을 벡터를  $\mathbb{R}^{m \times 1}$ 에서  $\mathbb{R}^{n \times 1}$ 로 사영시키는 변환으로도 생각할 수 있습니다.

이런 변환은 아주 유용하게 쓰입니다. 예를 들면, 회전을 정사각 행렬의 곱으로 표현할 수 있습니다. 다음 절에서 보겠지만, 행렬-벡터 곱을 뉴럴 네트워크의 각 층의 연산을 표현하는데도 사용합니다.

`ndarray` 를 이용해서 행렬-벡터의 곱을 계산할 때는 점곱에서 사용했던 `nd.dot()` 함수를 동일하게 사용합니다. 행렬  $A$  와 벡터  $x$  를 이용해서 `nd.dot(A, x)` 를 호출하면, MXNet은 행렬-벡터 곱을 수행해야한다는 것을 압니다.  $A$  의 열의 개수와  $x$  의 차원이 같아야 한다는 점을 유의하세요.

```
[17]: nd.dot(A, x)
```

```
[17]: [ 14.  38.  62.  86. 110.]
<NDArray 5 @cpu(0)>
```

#### 4.4.10 행렬-행렬 곱

점곱과 행렬-벡터 곱을 잘 이해했다면, 행렬-행렬 곱은 아주 간단할 것입니다.

행렬  $A \in \mathbb{R}^{n \times k}$ ,  $B \in \mathbb{R}^{k \times m}$  가 있다고 하겠습니다.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{pmatrix} \text{ff}$$

행렬의 곱  $C = AB$  를 계산하기 위해서,  $A$  를 행벡터들로,  $B$  를 열벡터들로 생각하면 쉽습니다.

$$A = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix}, \quad B = \begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{pmatrix}.$$

각 행벡터  $\mathbf{a}_i^T$  는  $\mathbb{R}^k$  에 속하고, 각 열벡터  $\mathbf{b}_j$  는  $\mathbb{R}^k$  에 속한다는 것을 주의하세요.

그러면, 행렬  $C \in \mathbb{R}^{n \times m}$  의 각 원소  $c_{ij}$  는  $\mathbf{a}_i^T \mathbf{b}_j$  로 구해집니다.

$$C = AB = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix} \begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T \mathbf{b}_1 & \mathbf{a}_1^T \mathbf{b}_2 & \cdots & \mathbf{a}_1^T \mathbf{b}_m \\ \mathbf{a}_2^T \mathbf{b}_1 & \mathbf{a}_2^T \mathbf{b}_2 & \cdots & \mathbf{a}_2^T \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^T \mathbf{b}_1 & \mathbf{a}_n^T \mathbf{b}_2 & \cdots & \mathbf{a}_n^T \mathbf{b}_m \end{pmatrix}$$

행렬-행렬 곱  $AB$  을 단순히  $m$  개의 행렬-벡터의 곱을 수행한 후, 결과를 붙여서  $n \times m$  행렬로 만드는 것으로 생각할 수도 있습니다. 일반적인 점곱과 행렬-벡터 곱을 계산하는 것처럼 MXNet에서 행렬-행렬의 곱은 `nd.dot()` 으로 계산됩니다.

```
[18]: B = nd.ones(shape=(4, 3))
nd.dot(A, B)
```

```
[18]: [[ 6.  6.  6.]
 [22. 22. 22.]
 [38. 38. 38.]
 [54. 54. 54.]
 [70. 70. 70.]]
<NDArray 5x3 @cpu(0)>
```

#### 4.4.11 놈(norm)

모델을 구현하기 전에 배워야 할 개념이 하나 더 있습니다. 선형대수에서 가장 유용한 연산 중에 놈(norm) 이 있습니다. 엄밀하지 않게 설명하면, 놈은 벡터나 행렬이 얼마나 큰지를 알려주는 개념입니다.  $\|\cdot\|$  으로 놈을 표현하는데,  $\cdot$  은 행렬이나 벡터가 들어갈 자리입니다. 예를 들면, 벡터  $\mathbf{x}$  나 행렬  $A$  를 각각  $\|\mathbf{x}\|$  or  $\|A\|$  로 적습니다.

모든 놈은 다음 특성을 만족시켜야 합니다.

1.  $\|\alpha A\| = |\alpha| \|A\|$
2.  $\|A + B\| \leq \|A\| + \|B\|$
3.  $\|A\| \geq 0$
4. If  $\forall i, j, a_{ij} = 0$ , then  $\|A\| = 0$

위 규칙을 말로 설명하면, 첫번째 규칙은 행렬이나 벡터의 모든 원소에 상수  $\alpha$  만큼 스캐일을 바꾸면, 놈도 그 상수의 절대값 만큼 스캐일이 바뀐다는 것입니다. 두번째 규칙은 친숙한 삼각부등식입니다. 세번째는 놈은 음수가 될 수 없다는 것입니다. 거의 모든 경우에 가장 작은 크기가 0이기에 이 규칙은 당연합니다. 마지막 규칙은 가장 작은 놈은 행렬 또는 벡터가 0으로 구성되었을 경우라는 기본적인 것에 대한 것입니다. 0이 아닌 행렬에 놈이 0이 되도록 놈을 정의하는 것이 가능합니다. 하지만, 0인 행렬에 0이 아닌 놈이 되게 하는 놈을 정의하는 것은 불가능합니다. 길게 설명했지만, 이것을 이해했다면 중요한 개념을 얻었을 것입니다.

수학시간에 배운 유클리디안 거리(Euclidean distance)를 기억한다면, 0이 아닌 것과 삼각부등식이 떠오를 것입니다. 놈이 거리를 측정하는 것과 비슷하다는 것을 인지했을 것입니다.

사실 유클리디안 거리  $\sqrt{x_1^2 + \dots + x_n^2}$  는 놈입니다. 특히, 이를  $\ell_2$ -놈이라고 합니다. 행렬의 각 원소에 대해서 유사하게 계산한 것  $\sqrt{\sum_{i,j} a_{ij}^2}$  을 푸로베니우스 놈(Frobenius norm)이라고 합니다. 머신러닝에서는 자주 제곱  $\ell_2$  놈을 사용합니다. ( $\ell_2^2$ 로 표현합니다.)  $\ell_1$  놈도 흔히 사용합니다.  $\ell_1$  놈은 절대값들의 합으로, 이상치(outlier)에 덜 중점을 주는 편리한 특성이 있습니다.

$\ell_2$  놈의 계산은 `nd.norm()` 으로 합니다.

```
[19]: nd.norm(x)
[19]:
[3.7416573]
<NDArray 1 @cpu(0)>
```

$\ell_1$  놈을 계산하는 방법은 각 원소의 절대값을 구한 후, 모두 합하는 것입니다.

```
[20]: nd.sum(nd.abs(x))
[20]:
[6.]
<NDArray 1 @cpu(0)>
```

#### 4.4.12 놈(norm)와 목적(objective)

더 깊이 나가지는 않겠지만, 이 개념들이 왜 중요한지 궁금할 것입니다. 머신 러닝에서 우리는 종종 최적화 문제를 풀기를 시도합니다 - 즉, 관찰된 데이터에 할당된 확률을 최대화하기, 예측된 값과 실제 값의 차이를 최소화하기, 단어, 제품, 새로운 기사와 같은 아이템들에 가까운 아이템들의 거리가 최소화되는 벡터를 할당하기 등을 시도합니다. 아마도 머신 러닝 알고리즘의 (데이터를 제외한) 가장 중요한 요소인 이 목적(objective)들은 자주 놈(norm)으로 표현됩니다.

#### 4.4.13 중급 선형 대수

여러분이 여기까지 잘 따라오면서 모든 내용을 이해했다면, 솔직하게 여러분은 모델을 시작할 준비가 되었습니다. 만약 조급함을 느낀다면, 이 절의 나머지는 넘어가도 됩니다. 실제로 적용할 수 있는 유용한 모델들을 구현하는데 필요한 모든 선형대수에 대해서 알아봤고, 더 알고 싶으면 다시 돌아올 수 있습니다.

하지만, 머신 러닝만 고려해봐도 선형대수에 대한 더 많은 내용이 있습니다. 이후 어느 시점에 여러분이 머신 러닝 경력을 만들기를 원한다면, 여기서 다룬 것보다 더 많은 것을 알아야할 것입니다. 유용하고 더 어려운 개념을 소개하면서 이 절을 마치겠습니다.

#### 4.4.14 벡터의 기본 성질들

벡터는 숫자를 담는 자료 구조보다 더 유용합니다. 벡터의 원소에 숫자를 읽고 적는 것, 유용한 수학 연산을 수행하는 것과 더불어, 벡터를 재미있는 방법으로 분석할 수 있습니다.

벡터 공간의 개념은 중요한 개념입니다. 벡터 공간이 되기에 필요한 조건은 다음과 같습니다.

- **더하기 공리(Additive axioms)** ( $x, y, z$ 가 모두 벡터라고 가정합니다.):  $x + y = y + x$ ,  $(x + y) + z = x + (y + z)$ ,  $0 + x = x + 0 = x$ . 그리고  $(-x) + x = x + (-x) = 0$ .
- **곱하기 공리(Multiplicative axioms)** ( $x$ 는 벡터이고  $a, b$ 는 스칼라입니다.):  $0 \cdot x = 0$ ,  $1 \cdot x = x$ ,  $(ab)x = a(bx)$ .
- **분배 공리(Distributive axioms)** ( $x$ 와  $y$ 는 벡터,  $a, b$ 는 스칼라로 가정합니다.):  $a(x + y) = ax + ay$  and  $(a + b)x = ax + bx$ .

#### 4.4.15 특별한 행렬들

이 책에서 사용할 특별한 행렬들이 있습니다. 그 행렬들에 대해서 조금 자세히 보겠습니다.

- **대칭 행렬(Symmetric Matrix)** 이 행렬들은 대각선 아래, 위의 원소들이 같은 값을 갖습니다. 즉,  $M^\top = M$  입니다. 이런 예로는 짹들의(pairwise) 거리를 표현하는 행렬  $M_{ij} = \|x_i - x_j\|$ 이 있습니다. 페이스북 친구 관계를 대칭 행렬로 표현할 수 있습니다.  $i$  와  $j$  가 친구라면  $M_{ij} = 1$ 이 되고, 친구가 아니라면  $M_{ij} = 0$ 로 표현하면 됩니다. 하지만, 트위터 그래프는 대칭이 아님을 주목해세요.  $M_{ij} = 1$ , 즉  $i$  가  $j$  를 팔로우하는 것이 꼭  $j$  가  $i$  를 팔로우하는 것,  $M_{ji} = 1$ , 은 아니기 때문입니다.
- **비대칭 행렬(Antisymmetric Matrix)**  $M^\top = -M$  를 만족하는 행렬입니다. 임의의 행렬은 대칭 행렬과 비대칭 행렬로 분해될 수 있습니다. 즉,  $M = \frac{1}{2}(M + M^\top) + \frac{1}{2}(M - M^\top)$  로 표현될 수 있습니다.
- **대각 지배 행렬(Diagonally Dominant Matrix)** 대각 원소들 보다 대각이 아닌 원소들이 작은 행렬입니다. 즉,  $M_{ii} \geq \sum_{j \neq i} M_{ij}$  이고  $M_{ii} \geq \sum_{j \neq i} M_{ji}$  입니다. 어떤 행렬이 이 특성을 갖는다면, 대각원소를 사용해서  $M$  을 추정할 수 있고, 이를  $\text{diag}(M)$  로 표기합니다.
- **양의 정부호 행렬(Positive Definite Matrix)** 이 행렬은  $x \neq 0$  이면,  $x^\top M x > 0$  인 좋은 특성을 갖습니다. 직관적으로 설명하면, 벡터의 제곱 놈,  $\|x\|^2 = x^\top x$ , 의 일반화입니다.  $M = A^\top A$  이면 이 조건이 만족시킨다는 것을 쉽게 확인할 수 있습니다. 이유는  $x^\top M x = x^\top A^\top A x = \|Ax\|^2$  이기 때문입니다. 모든 양의 정부호 행렬은 이런 형태로 표현될 수 있다는 더 심오한 이론이 있습니다.

#### 4.4.16 요약

몇 페이지들(또는 Jupyter 노트북 한개)을 통해서 뉴럴 네트워크의 중요한 부분들을 이해하는데 필요한 모든 선형대수에 대해서 알아봤습니다. 물론 선형대수에는 더 많은 내용이 있고, 이것들은 머신러닝에 유용하게 쓰입니다. 예를 들어, 행렬을 분해할 수 있는데, 이 분해는 실세계의 데이터셋의 아래 차원의 구조를 알려주기도 합니다. 행렬 분해를 이용하는데 집중하는 머신러닝의 별도의 분야가 있습니다. 이를 이용해서 데이터의 구조를 밝히고 예측 문제를 풀기 위해서 고차원의 텐서를 일반화하기도 합니다. 하지만 이 책에서는 딥러닝에 집중합니다. 여러분이 실제 데이터를 사용해서 유용한 머신러닝 모델을 만들기 시작한다면, 수학에 대해서 더 관심을 갖게될 것이라고 믿습니다. 하지만 수학적인 내용은 나중에 더 설명하기로 하고, 이 절은 여기서 마무리하겠습니다.

선형 대수에 대해서 더 배우기를 원한다면, 유용한 교재들이 있습니다.

- 탄탄한 기초를 쌓고 싶으면, Gilbert Strang의 책 [Introduction to Linear Algebra](#)를 참고하세요.
- Zico Kolter's [Linear Algebra Review and Reference](#)

#### 4.4.17 Scan the QR Code to Discuss



## 4.5 자동 미분(automatic differentiation)

머신 러닝에서 우리는 경험의 함수로써 모델을 더욱 좋게 만들려는 목적으로 학습을 합니다. 보통은 더 좋게 만드는 것은 손실 함수(*loss function*) , 즉, “모델이 얼마나 못하냐”에 대한 점수를 최소화하는 것을 의미합니다. 뉴럴 네트워크에서는 파라미터에 대해서 미분이 가능한 손실 함수(*loss function*)를 선택합니다. 간단하게 설명하면, 모델의 각 파라미터들이 손실(loss)을 얼마만큼 증가 또는 감소 시키는데 영향을 주는지를 결정할 수 있다는 것입니다. 미적분이 직관적이지만, 복잡한 모델들에 대한 미분을 직접 계산하는 것은 너무 힘들고 오류를 범할 수 있는 일입니다.

`autograd` 패키지는 미분을 자동으로 계산해주는 기능 제공합니다. 다른 대부분의 라이브러리들은 자동 미분을 하기 위해서는 심볼 그래프 컴파일을 우선 수행해야하지만, `autograd`는 일반적인 명령형 코드(imperative code)를 작성하면서 미분을 수행할 수 있게 해줍니다. 모델에 값을 대입할 때마다, `autograd`는 그래프를 즉석으로 만들고, 이를 사용해서 그래디언트(gradient)를 역으로 계산합니다. 이 절에서 나오는 수학에 익숙하지 않은 독자들은 부록의 “[Mathematical Basics](#)” 절을 참고하세요.

```
[1]: from mxnet import autograd, nd
```

### 4.5.1 간단한 예제

장난감 예제로 시작해보겠습니다. 함수  $y = 2\mathbf{x}^\top \mathbf{x}$  를 컬럼 벡터  $\mathbf{x}$ 에 대해서 미분을 하고 싶습니다. 우선은, 변수  $\mathbf{x}$ 를 생성하고, 초기 값을 할당합니다.

```
[2]: x = nd.arange(4).reshape((4, 1))
print(x)
```

```
[[0.]
 [1.]
 [2.]]
```

(continues on next page)

```
[3.]  
<NDArray 4x1 @cpu(0)>
```

$x$ 에 대한  $y$ 의 미분을 계산한 결과를 저장할 공간이 필요합니다. `attach_grad` 함수를 호출하면 `NDArray`는 그래디언트(gradient)를 저장할 공간을 마련합니다.

```
[3]: x.attach_grad()
```

자 이제  $y$ 를 계산합니다. MXNet은 즉석에서 연산 그래프를 생성해줍니다. 이는 마치 MXNet이 기록장치를 켜고, 각 변수들이 생성되는 정확한 경로를 모두 집어내는 것과 같습니다.

연산 그래프를 생성하는 데에는 적지 않은 연산량이 필요함을 기억해두세요. 따라서, MXNet은 요청을 받았을 때만 그래프를 생성하도록 되어 있습니다. 즉, 이는 `with autograd.record():` 블록 안에 넣었을 때 작동합니다.

```
[4]: with autograd.record():
    y = 2 * nd.dot(x.T, x)
print(y)
```

```
[[28.]]  
<NDArray 1x1 @cpu(0)>
```

$x$ 의 모양(shape)이 (4,1)이니,  $y$ 는 스칼라 값이 됩니다. 이후에 우리는 `backward` 함수를 호출해서 그래디언트(gradient)를 자동으로 구합니다. 만약  $y$ 가 스칼라 타입이 아니면, 기본 설정으로 MXNet은 우선  $y$ 의 모든 항목을 더해서 새로운 값을 얻은 결과의  $x$ 에 대한 그래디언트(gradient)를 찾습니다.

```
[5]: y.backward()
```

함수  $y = 2\mathbf{x}^\top \mathbf{x}$ 의  $\mathbf{x}$ 에 대한 미분은  $4\mathbf{x}$ 입니다. 미분이 올바르게 계산되는지 확인해보겠습니다.

```
[6]: print((x.grad - 4 * x).norm().asscalar() == 0)
print(x.grad)
```

True

```
[[ 0.]
 [ 4.]
 [ 8.]]
```

(continues on next page)

```
[12.]>
<NDArray 4x1 @cpu(0)>
```

## 4.5.2 학습 모드와 예측 모드

위에서 보았듯이 record 함수를 호출하면 MXNet은 그래디언트(gradients)를 계산하고 기록합니다. 더불어 기본 설정으로는 autograd 는 실행 모드를 예측 모드에서 학습 모드로 바꿉니다. 이는 is\_training 함수를 호출해서 확인할 수 있습니다.

```
[7]: print(autograd.is_training())
with autograd.record():
    print(autograd.is_training())

False
True
```

어떤 경우에는 같은 모델이 학습 모드와 예측 모드에서 다르게 동작합니다. 배치 정규(normalization)가 그렇습니다. 다른 어떤 모델은 그래디언트(gradients)를 쉽게 계산하기 위해서 더 많은 보조 변수를 저장하기도 합니다. 다음 장들에서 차이에 대해서 자세하게 다룰 예정이나, 지금은 자세한 내용에 대해서 걱정할 필요는 없습니다.

## 4.5.3 Python 제어 흐름의 그래디언트(gradients) 계산하기

자동 미분을 사용하는 장점 중에 하나는 함수의 연산 그래프가 Python 제어 흐름(if문과 loop 제어)을 가지고 있어도 변수에 대한 그래디언트(gradients)를 찾아낼 수 있다는 것입니다. 아래 프로그램을 예로 들어보겠습니다. while loop의 반복 횟수와 if 조건문의 수행 횟수는 입력 b 의 값에 영향을 받습니다.

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

그래디언트(gradient)를 계산하기 위해서 record 함수를 호출해야 하고, 그래디언트(gradient)를 찾기 위해서 backward 함수를 호출합니다.

```
[9]: a = nd.random.normal(shape=1)
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

위에서 정의한 함수  $f$  를 분석해보겠습니다. 실제로, 임의의 입력  $a$ 가 주어지면, 그 출력은  $f(a) = x * a$ 의 형태여야 하며, 여기서 스칼라 계수  $x$ 의 값은 입력  $a$ 에 의존합니다.  $c = f(a)$  는  $a$ 에 대한  $x$ 의 기울기와  $c / a$ 의 값을 갖기 때문에 다음과 같이 이 예에서 제어 흐름의 그래디언트(gradient) 결과의 정확성을 확인할 수 있습니다.

```
[10]: print(a.grad == (d / a))
```

```
[1.]
<NDArray 1 @cpu(0)>
```

#### 4.5.4 헤드 그래디언트(head gradient)와 체인룰

\*주의: 이 파트는 까다롭지만, 이 책의 나머지 부분을 이해하는데 꼭 필요한 내용은 아닙니다. 만약 새로운 네트워크를 직접 만들고 싶은 경우에 필요한 내용이기 때문에, 이 책을 처음 읽을 때는 넘어가도 됩니다.

$y$  가  $x$  의 함수이고, 이 함수의 backward 함수,  $y.backward()$  를 호출할 때, 우리는  $y$  의  $x$  에 대한 미분을 구하는 것에 관심이 있습니다. 수학자들은 이를  $\frac{dy(x)}{dx}$  로 기술합니다. 어떤 경우에는  $z$  가  $y$  의 함수일 때,  $z$  의  $x$  에 대한 미분을 구해야 합니다. 즉,  $\frac{d}{dx} z(y(x))$  를 구해야 합니다. 체인룰을 떠올려보면, 다음과 같이 계산됩니다.

$$\frac{d}{dx} z(y(x)) = \frac{dz(y)}{dy} \frac{dy(x)}{dx}.$$

$y$  가 더 큰 함수  $z$  의 일부이고,  $x.grad$  가  $\frac{dz}{dx}$  를 저장하도록 하려면, 헤드 그래디언트(head gradient)  $\frac{dz}{dy}$  를 backward() 의 입력으로 넣으면 됩니다. 기본 인자는 `nd.ones_like(y)` 입니다. 자세한 내용은 Wikipedia 를 참조하세요.

```
[11]: with autograd.record():
    y = x * 2
    z = y * x

head_gradient = nd.array([10, 1., .1, .01])
z.backward(head_gradient)
print(x.grad)
```

```
[[0.]
 [4.]
 [0.8]
 [0.12]]
<NDArray 4x1 @cpu(0)>
```

## 4.5.5 요약

- MXNet은 미분 계산을 자동화해주는 autograd 패키지를 제공합니다.
- MXNet의 autograd 패키지는 일반적인 명령형(imperative) 프로그램 형식으로 사용할 수 있습니다.
- MXNet의 실행 모드로는 학습 모드와 예측 모드가 있습니다. 현재의 실행 모드는 autograd.is\_training()을 통해서 구할 수 있습니다.

## 4.5.6 문제

1. 이 절에서 예로 들었던 제어 흐름(control flow)의 그래디언트(gradients)를 찾고자 할 때, 변수 a가 랜덤 벡터 또는 행렬에 따라서 바뀌고 있습니다. 이 때, c 계산 결과는 더 이상 스칼라 형태가 아닙니다. 결과에 무슨 일이 일어났나요? 이것을 어떻게 분석할 수 있나요?
2. 제어 흐름(control flow)의 그래디언트(gradients)를 구하는 예를 다시 설계해보세요. 실행하고 결과를 분석해보세요.
3. 세컨드 가격 경매 (예, eBay)에서는 두번째로 높은 가격을 부른 사람이 물건을 사게 됩니다. autograd를 이용해서 물건의 가격에 대해서 경매 참여에 대한 그래디언트(gradients)를 계산하세요. 왜 병리학적인 결과가 나올까요? 이 결과는 메카니즘에 대해서 무엇을 말해주나요? 더 자세한 내용은 [Edelman, Ostrovski and Schwartz, 2005](#) 페이지를 읽어보세요.
4. 왜 일차미분보다 이차미분이 훨씬 더 많은 연산량이 필요하나요?

5. 체인룰의 헤드 그래디언트(head gradient) 관계를 유도해보세요. 막히는 경우 [Wikipedia Chain Rule](#) 를 참고하세요.
6. 함수  $f(x) = \sin(x)$  를 가정합니다. 심볼 계산을 하지 않고 (즉,  $f'(x) = \cos(x)$  을 이용하지 말고)  $\frac{df(x)}{dx}$  를 구해서  $f(x)$  와  $\frac{df(x)}{dx}$  의 그래프를 그려보세요.

#### 4.5.7 Scan the QR Code to Discuss



## 4.6 확률과 통계

머신 러닝은 어떤 방식이든지 결국 예측을 수행하는 것입니다. 어떤 환자의 의료 기록을 바탕으로 내년에 심장 마비를 겪을 확률 예측하기를 예로 들어볼 수 있습니다. 비정상 탐지를 위해서, 비행기 제트 엔진의 센서 데이터가 정상적으로 동작할 때 어떤 값을 갖게 될지 예측을 할 수도 있습니다. 강화학습에서는 에이전트가 주어진 환경에서 똑똑하게 동작하게 만드는 것이 목표입니다. 이 경우에는 주어진 행동들 중에 가장 높은 보상을 받는 확률을 고려해야합니다. 추천 시스템을 만드는 경우에도 확률을 고려해야합니다. 예를 들어 여러분이 대형 온라인 서점에서 일을 한다면, 어떤 책을 홍보했을 때 특정 사용자가 그 책을 구매할지에 대한 확률을 추정하고 싶어할 것입니다. 이를 위해서 우리는 확률과 통계의 언어를 사용할 필요가 있습니다. 확률을 다루는 별도의 과정, 전공, 논문, 직업 심지어는 부서 까지도 있습니다. 이 책의 목표는 이 모든 주제들에 대해서 배워보는 것은 아니고, 여러분이 스스로 머신 러닝 모델을 만들 수 있을 정도의 내용을 알려주고, 이후에 스스로 공부해 볼 수 있는 주제들을 선택할 수 있도록 하는 것입니다.

지금까지 확률에 대해서 많이 이야기를 해왔지만, 확률에 정확하게 무엇인지를 설명하지 않았고 구체적인 예제를 들지는 않았습니다. 동물의 사진이 주어졌을 때, 고양이인지 개인지를 구분하는 문제를 조금 자세하게 살펴 보겠습니다. 이 문제는 간단해 보이지만, 사실 쉽지 않은 문제가 있습니다. 우선은 문제의 난이도가 이미지의 해상도에 따라 차이가 있을 수 있습니다.

10px	20px	40px	80px	160px
				
				

사람이 320 픽셀 해상도의 이미지에서 개와 고양이를 구분하는 것은 쉽습니다. 하지만, 40 픽셀이 되면 그 분류가 어렵고, 10픽셀로 줄어들면 거의 불가능합니다. 즉, 개와 고양이를 먼 거리에서 판별하는 것은 (또는 낮은 해상도의 이미지에서) 동전 던지기를 해서 추측하는 것과 동일해집니다. 확률은 확실성에 대한 추론을 하는 공식적인 방법을 제공합니다. 만약, 이미지에 고양이가 있다는 것을 완벽하게 확신한다면, 해당 레이블  $l$  이 고양이일 확률,  $P(l = \text{cat})$  는 1.0이라고 말합니다. 만약  $l = \text{cat}$  인지  $l = \text{dog}$  에 대한 아무런 판단을 못한다면, 두 확률은 동일하다고 하다고 말하며,  $P(l = \text{cat}) = 0.5$  이 됩니다. 만약 이미지에 고양이가 있다는 것을 확실하지는 않지만 어느 정도 확신한다면, 확률은  $.5 < P(l = \text{cat}) < 1.0$  로 주어질 것입니다.

이제 두번째 예를 들어보겠습니다. 대만 날씨에 대한 데이터를 관찰한 데이터가 있을 때, 내일 비가 내릴 확률을 예측하고자 합니다. 예를 들어 비가 내릴 확률이 0.5 정도가 될 것입니다. 위 두 가지 예제 모두 살펴볼 가치가 있습니다. 두 경우 모두 결과에 대한 불확실성이 있지만, 주요 차이점이 있습니다. 첫번째 예제는 이미지가 고양이인지 개이지만, 우리가 어떤 것인지 모르는 경우이고, 두번째 예제는 결과가 실제로 임의로 일어나는 이벤트일 수도 있습니다. 즉, 확률이란 우리의 확실성에 대한 사고를 하기 위한 유연한 언어이며, 다양한 경우에 효과적으로 적용될 수 있습니다.

## 4.6.1 기초 확률 이론

주사위를 던져서 다른 숫자가 아닌 1일 나오는 확률이 얼마나 되는지 찾는 경우를 생각해보겠습니다. 주사위가 공정하다면, 모든 6개 숫자들,  $\mathcal{X} = \{1, \dots, 6\}$ , 은 일어날 가능성이 동일합니다. 학술용어로는 “1은 확률  $\frac{1}{6}$ 로 일어난다”라고 말합니다.

공장에서 막 만들어진 주사위에 대해서 우리는 이 비율을 알지 못할 수 있고, 주사위가 공정한지 확인해야 할 필요가 있습니다. 주사위를 조사하는 유일한 방법은 여러 번 던져보면서 결과를 기록하는 것입니다. 주사위를 던질 때마다, 우리는  $\{1, 2, \dots, 6\}$ 에 하나의 숫자를 얻게 되고, 이 결과들이 주어지면, 각 숫자들이 일어날 수 있는 확률을 조사할 수 있습니다.

가장 자연스러운 방법은 각 숫자들이 나온 횟수를 전체 던진 횟수로 나누는 것입니다. 이를 통해서 우리는 특정 이벤트에 대한 확률을 추정 합니다. 큰 수의 법칙(the law of large numbers)에 따라, 던지는 횟수가 늘어날 수록 이 추정은 실제 확률과 계속 가까워집니다. 더 자세한 논의를 하기 전에, 실제로 실험을 해보겠습니다.

우선 필요한 패키지들을 import 합니다.

```
[1]: import mxnet as mx  
from mxnet import nd
```

다음으로는 주사위를 던지는 것을 해야합니다. 통계에서는 확률 분포에서 샘플을 뽑는 것을 샘플링이라고 합니다. 연속되지 않은 선택들에 확률이 부여된 분포를 우리는 다항(multinomial) 분포라고 합니다. 분포(distribution)에 대한 공식적인 정의는 다음에 다루겠고, 지금은 분포를 이벤트들에 확률을 할당하는 것 정도로 생각하겠습니다. MXNet에서 `nd.random.multinomial` 함수를 이용하면 다항 분포에서 샘플을 추출할 수 있습니다.

```
[2]: probabilities = nd.ones(6) / 6  
nd.random.multinomial(probabilities)
```

```
[2]:  
[3]  
<NDArray 1 @cpu(0)>
```

여러 샘플을 뽑아보면, 매번 임의의 숫자를 얻는 것을 확인할 수 있습니다. 주사위의 공정성을 추정하는 예제에서 우리는 같은 분포에서 많은 샘플을 추출하기를 원합니다. Python의 `for` loop을 이용하면 너무 느리기 때문에, `random.multinomial`이 여러 샘플을 한번째 뽑아주는 기능을 이용해서 우리가 원하는 모양(shape)의 서로 연관이 없는 샘플들의 배열을 얻겠습니다.

```
[3]: print(nd.random.multinomial(probabilities, shape=(10)))  
print(nd.random.multinomial(probabilities, shape=(5,10)))
```

```
[3 4 5 3 5 3 5 2 3 3]
<NDArray 10 @cpu(0)>

[[2 2 1 5 0 5 1 2 2 4]
 [4 3 2 3 2 5 5 0 2 0]
 [3 0 2 4 5 4 0 5 5 5]
 [2 4 4 2 3 4 4 0 4 3]
 [3 0 3 5 4 3 0 2 2 1]]
<NDArray 5x10 @cpu(0)>
```

이제 주사위를 던지는 샘플을 구하는 방법을 알았으니, 100번 주사위를 던지는 시뮬레이션을 해서, 각 숫자들이 나온 횟수를 카운팅합니다.

```
[4]: rolls = nd.random.multinomial(probabilities, shape=(1000))
counts = nd.zeros((6,1000))
totals = nd.zeros(6)
for i, roll in enumerate(rolls):
    totals[int(roll.asscalar())] += 1
    counts[:, i] = totals
```

1000번을 던져본 후에 최종 합계를 확인합니다.

```
[5]: totals / 1000
[5]:
[0.167 0.168 0.175 0.159 0.158 0.173]
<NDArray 6 @cpu(0)>
```

결과에 따르면, 모든 숫자 중에 가장 낮게 추정된 확률은 약 0.15이고, 가장 높은 추정 확률은 0.188입니다. 공정한 주사위를 사용해서 데이터를 생성했기 때문에, 각 숫자들은  $1/6$  즉 0.167의 확률을 갖는다는 것을 알고 있고, 예측도 매우 좋게 나왔습니다. 시간이 지나면서 이 확률이 의미 있는 추정치로 어떻게 수렴하는지를 시각해 볼 수도 있습니다.

이를 위해서 우선은  $(6, 1000)$ 의 모양(shape)을 갖는 counts 배열을 살펴봅시다. 1000번을 수행하는 각 단계마다, counts 는 각 숫자가 몇 번 나왔는지를 알려줍니다. 그렇다면, counts 배열의  $j$  번째 열의 그때까지 던진 총 횟수로 표준화해서, 그 시점에서의 추정 확률 current 를 계산합니다. counts 객체는 다음과 같습니다.

```
[6]: counts
[6]:
[[ 0.  0.  0. ... 165. 166. 167.]
 [ 1.  1.  1. ... 168. 168. 168.]
 [ 0.  0.  0. ... 175. 175. 175.]
 [ 0.  0.  0. ... 159. 159. 159.]
 [ 0.  1.  2. ... 158. 158. 158.]
 [ 0.  0.  0. ... 173. 173. 173.]]
<NDArray 6x1000 @cpu(0)>
```

던진 총 횟수로 표준화 하면,

```
[7]: x = nd.arange(1000).reshape((1,1000)) + 1
estimates = counts / x
print(estimates[:,0])
print(estimates[:,1])
print(estimates[:,100])

[0. 1. 0. 0. 0. 0.]
<NDArray 6 @cpu(0)>

[0. 0.5 0. 0. 0.5 0. ]
<NDArray 6 @cpu(0)>

[0.1980198 0.15841584 0.17821783 0.18811882 0.12871288 0.14851485]
<NDArray 6 @cpu(0)>
```

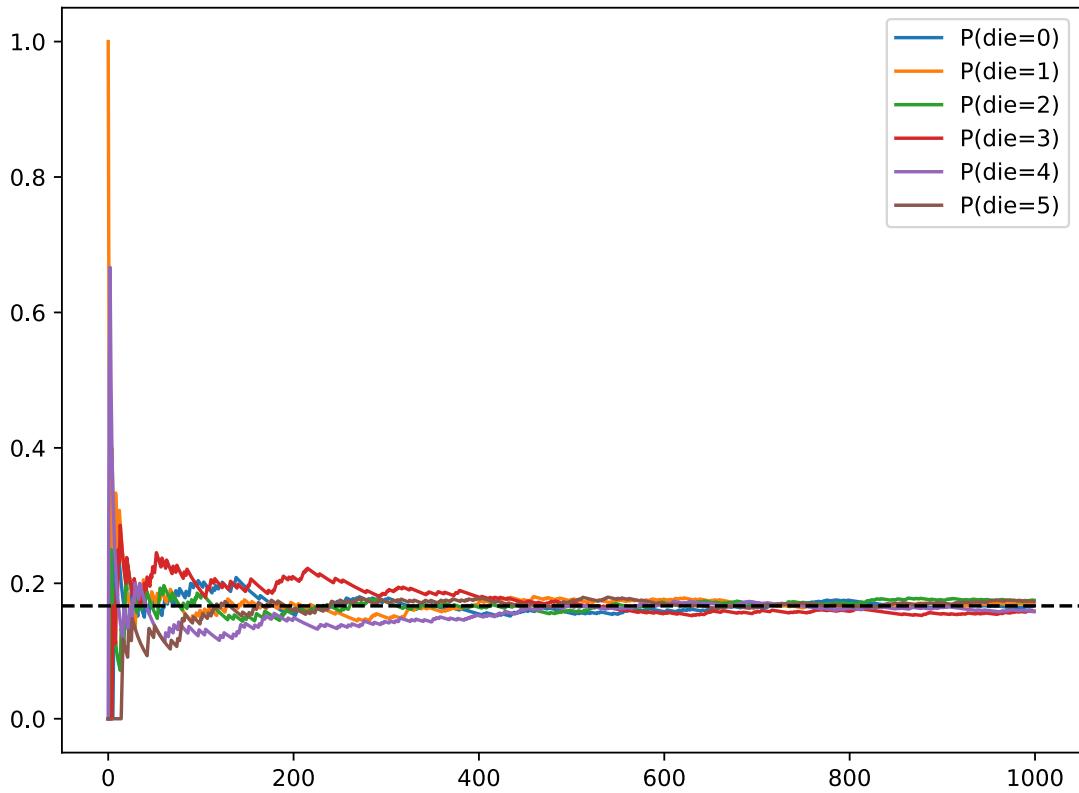
결과에서 보이듯이, 주사위를 처음 던진 경우 하나의 숫자에 대한 확률이 1.0이고 나머지 숫자들에 대한 확률이 0인 극단적인 예측을 하지만, 100번을 넘어서면 결과가 상당히 맞아 보입니다. 플롯을 그리는 패키지 matplotlib을 이용해서 이 수렴 과정을 시각화 해봅니다. 이 패키지를 아직 설치하지 않았다면, `install it`를 참고해서 지금 하세요.

```
[8]: %matplotlib inline
from matplotlib import pyplot as plt
from IPython import display
display.set_matplotlib_formats('svg')

plt.figure(figsize=(8, 6))
for i in range(6):
    plt.plot(estimates[i, :].asnumpy(), label="P(die=" + str(i) + ")")
```

(continues on next page)

```
plt.axhline(y=0.16666, color='black', linestyle='dashed')
plt.legend()
plt.show()
```



각 선은 주사위의 숫자 중에 하나를 의미하고, 1000번 주사위 던지기를 수행하면서 각 횟수마다 각 숫자가 나올 확률의 추정값을 나타내는 그래프입니다. 검은 점선은 진짜 확률(true probability, 1/6)을 표시합니다. 횟수가 늘어가면 선들이 진짜 확률에 수렴하고 있습니다.

주사위 던지기 예를 통해서 **확률 변수(random variable)**라는 개념을 소개했습니다. 여기서  $X$ 로 표현할 확률 변수는 어떤 양이 될 수 있고, 결정적이지 않을 수 있습니다. 확률 변수는 여러 가능성들의 집합에서 하나의 값을 나타낼 수도 있습니다. 집합은 팔호를 이용해서 표현합니다. 예를 들면, {cat, dog, rabbit} 입니다. 집합에 속한 아이템들은 원소(element)라고 하고, 어떤 원소  $x$  가 집합  $S$ 에 속한다 라고 하면 표기는  $x \in S$ 로 합니다. 기호  $\in$ 는 “속한다”라고 읽고, 포함 관계를 표현합니다. 예를 들어,  $\text{dog} \in \{\text{cat}, \text{dog}, \text{rabbit}\}$  입니다. 주사위 던지는 것의 경우, 확률 변수  $X \in \{1, 2, 3, 4, 5, 6\}$

입니다

연속적이지 않은 확률변수(예를 들어 주사위의 6면)와 연속적인 확률변수(예를 들어 사람의 몸무게나 키) 사이에는 미묘한 차이점이 있다는 것을 기억하세요. 두 사람의 키가 정확하게 같은지를 묻는 경우는 드물 것입니다. 아주 정확한 측정 방법이 있어서 이를 적용한다면, 이 세상에 키가 완전하게 같은 사람 두 사람이 없습니다. 사실, 적당히 정교한 측정을 하는 경우에도 아침에 일어났을 때의 키와 밤에 잠자기 전에 챴 키는 다르게 나옵니다. 즉, 어떤 사람의 키가  $2.00139278291028719210196740527486202$  미터일 확률을 물어보는 것은 의미가 없습니다. 전체 인구에 대해서도 이 확률은 거의 0입니다. 따라서, 어떤 사람의 키가 어느 구간(예를 들면 1.99 와 2.01 미터 사이)에 속하는지를 묻는 것이 더 의미가 있습니다. 이런 경우들에는 우리는 어떤 값을 밀도(density)로 볼 가능성을 정량화 합니다. 정확하게 2.0 미터인 키에 대한 확률은 없지만, 밀도는 0이 아닙니다. 서로 다른 두 키의 구간에 대해서는 확률값이 0이 아닌 수가 됩니다.

기억해 두어야 할 몇가지 중요한 확률에 대한 공리(axiom)들이 있습니다.

- 어떤 이벤트  $z$ 에 대해서, 확률은 절대로 음수가 아닙니다. 즉,  $\Pr(Z = z) \geq 0$
- 두 이벤트  $Z = z$  과  $X = x$ 에 대해서, 두 이벤트의 합집합(union)에 대한 확률은 각 이벤트의 확률의 합보다 클 수 없습니다. 즉,  $\Pr(Z = z \cup X = x) \leq \Pr(Z = z) + \Pr(X = x)$ ff.
- 어떤 확률 변수에 대해서, 모든 값들에 대한 확률의 합은 항상 1입니다. 즉,  $\sum_{i=1}^n \Pr(Z = z_i) = 1$ .
- 서로 겹치지 않는 두 사건,  $Z = z$  과  $X = x, t$ ,에 대해서, 둘 중에 한 사건이 일어날 확률은 각 사건의 확률의 합과 같습니다. 즉,  $\Pr(Z = z \cup X = x) = \Pr(Z = z) + \Pr(X = x)$ .

#### 4.6.2 여러 확률 변수 다루기

종종 하나 이상의 확률 변수를 동시에 다룰 필요가 생깁니다. 질병과 증상의 관계를 모델링하는 경우를 들 수 있습니다. 질병과 증상이 주어졌을 때, 예를 들면 ‘독감’과 ‘기침’, 두개는 어떤 확률로 환자에게 일어날 수도 일어나지 않을 수 있습니다. 이 둘에 대한 확률이 작기를 기대하지만, 더 좋은 의료 처방을 할 수 있도록 확률과 둘 사이의 관계를 예측하고자 합니다.

더 복잡한 예로, 수백만 픽셀로 이루어진 이미지를 들어보겠습니다. 즉, 수백만 확률 변수가 존재합니다. 많은 경우에 이미지들은 이미지에 있는 객체를 지칭하는 레이블을 갖습니다. 이 레이블도 확률 변수로 생각할 수 있습니다. 더 나아가서는, 위치, 시간, 구경(aperture), 초점 거리, ISO, 초점, 카메라 종류 등과 같은 모든 메타 데이터를 확률 변수로 생각할 수도 있습니다. 이 모든 것은 연관되어 발생하는 확률 변수들입니다. 여러 확률 변수를 다룰 때 몇가지 중요한 것들이 있습니다. 첫번째는 교차 확률 분포  $\Pr(A, B)$ 입니다. 두 원소  $a$  와  $b$  가 주어졌을 때, 교차 확률 분포는 동시에  $A = a$  이고  $B = b$

일 확률이 얼마인지에 대한 답을 줍니다. 임의의 값  $a$  와  $b$  에 대해서,  $\Pr(A, B) \leq \Pr(A = a)$  이라는 사실은 쉽게 알 수 있습니다.

$A$  와  $B$  가 일어났기 때문에,  $A$  가 발생하고,  $B$  또한 발생해야 합니다. (또는 반대로). 즉,  $A$  와  $B$  가 동시에 일어나는 것은  $A$  와  $B$  가 별도로 일어나는 것보다는 가능성이 낮습니다. 이 사실로 흥미로운 비율을 정의할 수 있습니다. 즉,  $0 \leq \frac{\Pr(A, B)}{\Pr(A)} \leq 1$ . 우리는 이것을 조건부 확률(conditional probability)이라고 부르며,  $\Pr(B|A)$  로 표현합니다. 다시 말하면,  $A$  가 일어났을 때  $B$  가 일어날 확률입니다.

조건부 확률의 정의를 이용하면, 확률에서 가장 유용하고 유명한 방정식을 도출할 수 있는데, 이것이 바로 베이즈 이론(Bayes' theorem)입니다. 이를 도출하는 방법으로  $\Pr(A, B) = \Pr(B|A) \Pr(A)$  로부터 출발합니다. 대칭성을 적용하면,  $\Pr(A, B) = \Pr(A|B) \Pr(B)$  이 됩니다. 조건 변수들 중 하나에 대해서 풀어보면 다음 공식을 얻게 됩니다.

$$\Pr(A|B) = \frac{\Pr(B|A) \Pr(A)}{\Pr(B)}$$

어떤 것으로부터 다른 어떤 것을 추론(즉 원인과 효과)하고자 하는데, 반대 방향에 대한 것만 알고 있을 경우에 아주 유용합니다. 주변화(marginalization)는 이것이 작동하게 만드는데 아주 중요한 연산입니다. 이 연산은  $\Pr(A, B)$  로부터  $\Pr(A)$  와  $\Pr(B)$  를 알아내는 연산입니다.  $A$  가 일어날 확률은 모든  $B$  에 대한 교차 확률(joint probability)의 값으로 계산됩니다. 즉,

$$\Pr(A) = \sum_{B'} \Pr(A, B') \text{ and } \Pr(B) = \sum_{A'} \Pr(A', B)$$

점검해야 할 아주 유용한 특성은 종속과 독립입니다. 독립은 하나의 사건의 발생이 다른 사건의 발생에 영향을 주지 않는 것을 의미합니다. 위 경우에는  $\Pr(B|A) = \Pr(B)$  를 의미합니다. 그 외의 경우들은  $A$  와  $B$ 가 종속적이라고 합니다. 주사위를 두 번 연속으로 던지는 것은 독립적이나, 방의 전등 스위치의 위치와 방의 밝기는 그렇지 않습니다.(이 둘이 완전히 결정적이지는 않습니다. 왜냐하면, 전구가 망가질 수도 있고, 전원이 나갈 수도 있고, 스위치가 망가질 경우 등이 있기 때문입니다.)

그럼 배운 것을 테스트해보겠습니다. 의사가 환자에게 AIDS 테스트를 하는 것을 가정하겠습니다. 이 테스트는 상당히 정확해서, 환자가 음성일 경우 이를 틀리게 예측하는 확률이 1%이고, 환자가 양성일 경우 HIV 검출을 실패하지 않습니다.  $D$  는 진단 결과를  $H$  는 HIV 상태를 표기합니다.  $\Pr(D|H)$  결과를 표로 만들어보면 다음과 같습니다.

결과	HIV 양성	HIV 음성
테스트 결과 - 양성	1	0.01
테스트 결과 - 음성	0	0.99

같은 열의 값을 더하면 1이나, 행으로 더하면 그렇지 않습니다. 그 이유는 조건부 확률도 합이 확률처럼 1이여야하기 때문입니다. 테스트 결과가 양성일 경우 환자가 AIDS에 걸렸을 확률을 계산해보겠습니다. 당연하게 도 이는 질병이 얼마나 일반적인가에 따라 달라집니다. 인구의 대부분이 건강하다고 가정하겠습니다. 즉  $\Pr(\text{HIV positive}) = 0.0015$ . 베이즈 이론(Bayes' Theorem)을 적용하기 위해서 우리는 다음을 결정해야합니다.

$$\begin{aligned}\Pr(\text{Test positive}) &= \Pr(D = 1|H = 0)\Pr(H = 0) + \Pr(D = 1|H = 1)\Pr(H = 1) \\ &= 0.01 \cdot 0.9985 + 1 \cdot 0.0015 \\ &= 0.011485\end{aligned}$$

따라서, 우리가 얻는 것은 다음과 같습니다.

$$\begin{aligned}\Pr(H = 1|D = 1) &= \frac{\Pr(D = 1|H = 1)\Pr(H = 1)}{\Pr(D = 1)} \\ &= \frac{1 \cdot 0.0015}{0.011485} \\ &= 0.131\end{aligned}$$

이 결과는 99% 정확도로 테스트 결과가 양성으로 나올지라도 환자가 실제로 AIDS에 걸렸을 확률은 13.1% 밖에 되지 않는다는 것을 의미입니다. 이 결과에서 보듯이, 통계는 매우 직관적이지 않을 수 있습니다.

### 4.6.3 조건부 독립성

그렇다면, 환자가 이렇게 무서운 결과를 받았을 때 어떻게 해야할까요? 아마도 환자는 의사에게 테스트를 다시 해봐달라고 요청할 것입니다. 두번째 테스트는 다르게 나왔다고 하겠습니다. (즉, 첫번째 만큼 좋지 않습니다.)

결과	HIV 양성	HIV 음성
테스트 결과 - 양성	0.98	0.03
테스트 결과 - 음성	0.02	0.97

안타깝게도 두번째 테스트 역시 양성으로 나오고 있습니다. 베이즈 이론(Bayes' Theorem)을 적용하기 위한 필요한 확률값들을 계산해봅니다.

- $\Pr(D_1 = 1 \text{ and } D_2 = 1) = 0.0003 \cdot 0.9985 + 0.98 \cdot 0.0015 = 0.00176955$
- $\Pr(H = 1|D_1 = 1 \text{ and } D_2 = 1) = \frac{0.98 \cdot 0.0015}{0.00176955} = 0.831$

즉, 두번째 테스트 결과는 좋지 않다는 것에 더 확신하게 만듭니다. 두번째 결과는 첫번째 보다 덜 정확함에도 불구하고, 예측 결과를 더 향상시켰습니다. 그렇다면, 첫번째 테스트를 두번하지 않을까요? 결국, 첫번째 테스트가 더 정확했습니다. 두번째 테스트가 필요한 이유는 첫번째 테스트를 독립적으로 확인하기 위함입니다. 즉,  $\Pr(D_1, D_2|H) = \Pr(D_1|H) \Pr(D_2|H)$  이라는 암묵적인 가정을 했습니다. 통계학에서는 이런 확률 변수를 조건에 독립적이라고 하며,  $D_1 \perp\!\!\!\perp D_2|H$  라고 표현합니다.

#### 4.6.4 요약

이 절에서 우리는 확률, 독립, 조건 독립, 그리고 기본적인 결론을 도출하는데 이것들을 어떻게 사용하는지를 알아봤습니다. 이 개념들은 아주 유용합니다. 다음 절에서는 나이브 베이즈 분류기(Naive Bayes)를 사용한 기본적인 예측을 하는데 이 개념들이 어떻게 사용되는지 살펴보겠습니다.

#### 4.6.5 문제

1.  $\Pr(A)$  과  $\Pr(B)$  확률로 두 사건이 주어졌을 때,  $\Pr(A \cup B)$  와  $\Pr(A \cap B)$  의 상한과 하한을 구하세요. 힌트 - [Venn Diagram](#)을 사용하는 상황을 그려보세요.
2. 연속적인 사건, 즉  $A, B, C$ , 들이 있는데,  $B$  는  $A$ 에만 의존하고,  $C$  는  $B$ 에만 의존한다고 가정합니다. 이 경우 교차 확률(joint probability)를 간단하게 할 수 있을까요? 힌트 - 이는 [Markov Chain](#)입니다.

#### 4.6.6 Scan the QR Code to Discuss



### 4.7 나이브 베이즈 분류(Naive Bayes Classification)

조건에 독립적인 것은 데이터를 다루는데 있어서 많은 공식을 간단하게 해주기 때문에 유용합니다. 간단하고 유명한 알고리즘으로 나이브 베이즈 분류(Naive Bayes Classifier)가 있습니다. 이 알고리즘의 주요 가정은 주어진 레이블들에 대해서 모든 속성들이 서로 영향을 주지 않는다는 것입니다. 즉,

다음과 같은 수식을 만족시킵니다.

$$p(\mathbf{x}|y) = \prod_i p(x_i|y)$$

베이즈 이론을 적용해보면,  $p(y|\mathbf{x}) = \frac{\prod_i p(x_i|y)p(y)}{p(\mathbf{x})}$  분류를 얻을 수 있습니다. 하지만 어렵게도,  $p(x)$ 를 모르기 때문에 다루기 어렵습니다. 다행인 것은,  $\sum_y p(y|\mathbf{x}) = 1$  인 것을 알고 있기 때문에,  $p(x)$ 가 필요가 없습니다. 따라서, 우리는 항상 표준화(normalization)를 구할 수 있습니다.

$$p(y|\mathbf{x}) \propto \prod_i p(x_i|y)p(y).ff$$

스팸 메일과 일반 메일을 분류하는 것을 예로 들어서 설명해보겠습니다. Nigeria, prince, money, rich 와 같은 단어들이 이메일에 있다는 것은 그 이메일이 스팸일 가능성이 있다는 것을 암시한다고 할 수 있습니다. 반면에, theorem, network, Bayes, statistics 같은 단어는 메시지가 실질적인 내용을 담고 있는 것을 암시한다고 할 수 있습니다. 따라서, 이런 단어들이 속한 클래스가 주어졌을 때 각각에 대한 확률에 대한 모델을 만들고, 문장이 스팸일 가능성에 대한 점수를 매기는데 사용하는 것이 가능합니다. 실제로 이 방법은 많은 오랫동안 Bayesian spam filters가 사용하는 방법입니다.

#### 4.7.1 광학 문자 인지 (optical character recognition)

이미지가 다루기 더 쉽기 때문에, MNIST 데이터셋의 숫자를 구분하는 문제에 나이브 베이즈 분류를 적용해보겠습니다. 여기서 문제는  $p(y)$  와  $p(x_i|y)$  를 모른다는 것입니다. 그렇게 때문에 우리가 해야 할 일은 주어진 학습 데이터를 사용해서 이 확률들을 추정해야합니다. 즉, 모델을 학습시키는 것이 필요합니다.  $p(y)$  를 추정하는 것은 어려운 일이 아닙니다. 우리가 다루는 클래스의 개수가 10개이기 때문에, 아주 쉽게 추정할 수 있습니다 - 즉, 각 숫자가 나오는 것을 카운팅 한 수  $n_y$  전체 데이터 개수  $n$  으로 나누면 됩니다. 예를 들어 숫자 8이 나온 횟수가  $n_8 = 5,800$  이고 전체 데이터 개수가 of  $n = 60,000$  개이면, 추정 확률은  $p(y = 8) = 0.0967$  입니다.

이제 조금 더 어려운  $p(x_i|y)$ 에 대해서 이야기해보겠습니다. 이미지가 흑백으로 되어 있기 때문에,  $p(x_i|y)$  는 픽셀  $i$  가 클래스  $y$  에 속할 확률을 나타냅니다. 앞에서 적용한 방법을 이용해서 어떤 이벤트가 발생한 횟수  $n_{iy}$  를 카운트하고,  $y$ 가 일어난 전체 횟수  $n_y$  로 나눌 수 있습니다. 그러나 약간 까다로운 문제가 있습니다 - 어떤 픽셀은 절대로 검정색이지 않을 수 있습니다. (만약, 이미지를 잘 잘라내면, 코너 픽셀들은 항상 흰색일 것이기 때문입니다.) 통계학자들이 이런 문제를 다루는 편리한 방법으로 의사(pseudo) 카운트를 모든 것에 더하는 것입니다. 즉,  $n_{iy}$  를 사용하는 대신  $n_{iy} + 1$  를,  $n_y$  대신  $n_y + 1$  를 사용하는 것입니다. 이 방법은 Laplace Smoothing이라고 불리는 것입니다.

```
[1]: %matplotlib inline
from matplotlib import pyplot as plt
from IPython import display
display.set_matplotlib_formats('svg')
import mxnet as mx
from mxnet import nd
import numpy as np

# We go over one observation at a time (speed doesn't matter here)
def transform(data, label):
    return (nd.floor(data/128)).astype(np.float32), label.astype(np.float32)
mnist_train = mx.gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = mx.gluon.data.vision.MNIST(train=False, transform=transform)

# Initialize the counters
xcount = nd.ones((784,10))
ycount = nd.ones((10))

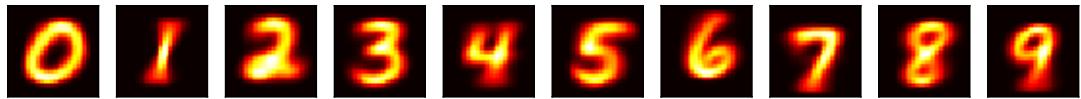
for data, label in mnist_train:
    y = int(label)
    ycount[y] += 1
    xcount[:,y] += data.reshape((784))

# using broadcast again for division
py = ycount / ycount.sum()
px = (xcount / ycount.reshape(1,10))
```

모든 픽셀들에 대해서 픽셀 별로 등장하는 횟수를 계산했으니, 우리의 모델이 어떻게 동작하는지 그림을 그려서 보겠습니다. 이미지를 이용하면 아주 편한 점은 시각화가 가능하다는 것입니다. 28x28x10 확률을 시각화하는 것은 무의미한 일이니, 이미지 형태로 그려서 빠르게 살펴보겠습니다. 눈치가 빠른 분은 숫자를 닮은 어떤 평균임을 알아차렸을지도 모르겠습니다.

```
[2]: import matplotlib.pyplot as plt
fig, figarr = plt.subplots(1, 10, figsize=(10, 10))
for i in range(10):
    figarr[i].imshow(xcount[:, i].reshape((28, 28)).asnumpy(), cmap='hot')
    figarr[i].axes.get_xaxis().set_visible(False)
    figarr[i].axes.get_yaxis().set_visible(False)

plt.show()
print('Class probabilities', py)
```



Class probabilities

```
[0.09871688 0.11236461 0.09930012 0.10218297 0.09736711 0.09035161
```

```
0.09863356 0.10441593 0.09751708 0.09915014]
```

```
<NDArray 10 @cpu(0)>
```

우리의 모델에 근거해서 이미지의 가능성들에 대한 계산을 할 수 있게 되었습니다. 통계 용어로 말하면,  $p(x|y)$  를 계산할 수 있습니다. 즉, 이는 주어진 이미지가 특정 레이블에 속할 확률이 됩니다. “모든 픽셀이 독립적이다”라고 가정하는 나이브 베이즈 모델에 따르면, 다음 공식을 얻습니다.

$$p(\mathbf{x}|y) = \prod_i p(x_i|y)$$

따라서, 베이즈 법칙을 사용하면,  $p(y|\mathbf{x})$  의 값은 다음 식으로 구할 수 있습니다.

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{\sum_{y'} p(\mathbf{x}|y')}$$

그럼 코드를 실행해 보겠습니다.

```
[3]: # Get the first test item
data, label = mnist_test[0]
data = data.reshape((784,1))

# Compute the per pixel conditional probabilities
xprob = (px * data + (1-px) * (1-data))
# Take the product
xprob = xprob.prod(0) * py
print('Unnormalized Probabilities', xprob)
# Normalize
xprob = xprob / xprob.sum()
print('Normalized Probabilities', xprob)

Unnormalized Probabilities
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 10 @cpu(0)>
Normalized Probabilities
[nan nan nan nan nan nan nan nan nan]
```

(continues on next page)

&lt;NDArray 10 @cpu(0)&gt;

완전히 틀린 결과를 얻었습니다. 이유를 찾기 위해 각 픽셀의 확률을 살펴보겠습니다. 일반적으로 확률값은 0.001 와 1 사이의 값입니다. 우리는 784 개의 값을 곱했습니다. 바로 이 부분에서 문제가 발생합니다. 즉, 고정소수점 연산을 수행하면서 수치적인 언더플로우(numerical underflow)가 발생합니다. 작은 수들을 계속 곱하면 결국 0이 되는 되고, 결국 0을 나누는 일이 발생하기 때문에 결과가 nan 이 되는 것입니다.

이 문제를 해결하기 위해서  $\log ab = \log a + \log b$  이라는 것을 사용하겠습니다. 즉, 로그(logarithm) 합으로 바꾸기 입니다. 이를 적용하면 log-공간에서 비표준화 된 확률을 얻게 되니, 표준화하기 위해서 다음과 같은 공식을 활용합니다.

$$\frac{\exp(a)}{\exp(a) + \exp(b)} = \frac{\exp(a + c)}{\exp(a + c) + \exp(b + c)}$$

분모 항의 하나는 1이 되도록  $c = -\max(a, b)$  로 선택합니다.

```
[4]: logpx = nd.log(px)
logpxneg = nd.log(1-px)
logpy = nd.log(py)

def bayespost(data):
    # We need to incorporate the prior probability p(y) since p(y/x) is
    # proportional to p(x/y) p(y)
    logpost = logpy.copy()
    logpost += (logpx * data + logpxneg * (1-data)).sum(0)
    # Normalize to prevent overflow or underflow by subtracting the largest
    # value
    logpost -= nd.max(logpost)
    # Compute the softmax using logpx
    post = nd.exp(logpost).asnumpy()
    post /= np.sum(post)
    return post

fig, figarr = plt.subplots(2, 10, figsize=(10, 3))

# Show 10 images
ctr = 0
for data, label in mnist_test:
    x = data.reshape((784,1))
    y = int(label)
```

(continues on next page)

```

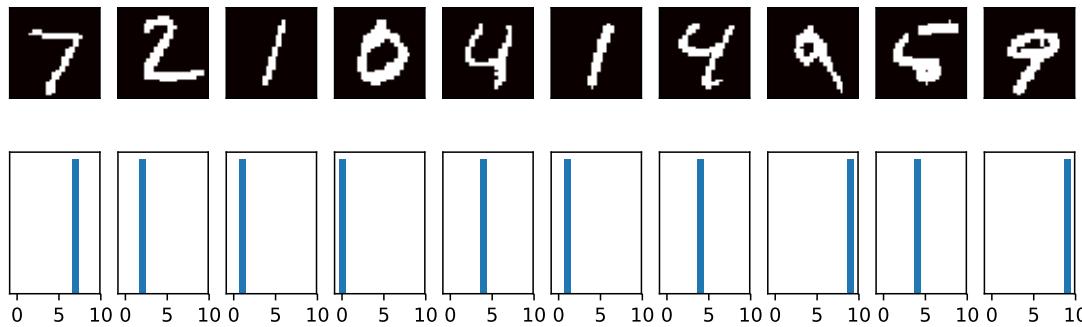
post = bayespost(x)

# Bar chart and image of digit
figarr[1, ctr].bar(range(10), post)
figarr[1, ctr].axes.get_yaxis().set_visible(False)
figarr[0, ctr].imshow(x.reshape((28, 28)).asnumpy(), cmap='hot')
figarr[0, ctr].axes.get_xaxis().set_visible(False)
figarr[0, ctr].axes.get_yaxis().set_visible(False)
ctr += 1

if ctr == 10:
    break

plt.show()

```



보이는 것처럼, 이 분류기가 많은 경우 잘 동작하고 있습니다. 하지만, 뒤에서 두번째 숫자는 예측이 틀리기도 하고 그 잘못된 예측에 대해서 너무 높은 확신값을 주고 있습니다. 즉, 완전히 틀린 추측일 경우에도 확률을 0 또는 1에 아주 가깝게 출력하고 있습니다. 이런 모델은 사용할 수 있는 수준이 아닙니다. 이 분류기의 전반적인 정확도를 계산해서 이 모델이 얼마나 좋은지 확인합니다.

```

[5]: # Initialize counter
ctr = 0
err = 0

for data, label in mnist_test:
    ctr += 1
    x = data.reshape(784,1)
    y = int(label)

```

(continues on next page)

```

post = bayespost(x)
if (post[y] < post.max() ) :
    err += 1

print('Naive Bayes has an error rate of', err/ctr)

Naive Bayes has an error rate of 0.1574

```

현대 딥 네트워크는 0.01 보다 낮은 에러율을 달성합니다. 나이브 베이즈 분류기는 80년대나 90년대에 스팸 필터 등을 만드는데 많이 사용되었지만, 이제는 더이상 사용되지 않습니다. 성능이 나쁜 이유는 우리가 모델을 만들때 했던 통계적인 가정이 틀렸기 때문입니다 - 즉, 모든 피셀은 서로 연관이 없고, 오직 레이블에만 관련이 있다고 가정한 것이 틀렸기 때문입니다. 사람들이 숫자를 적는 방법이 다양하다는 것을 반영하지 못하는 틀린 가정이 잘 작동하지 않는 분류기를 만들어낸 것입니다. 자 이제부터 딥 네트워크를 만드는 것을 시작해보겠습니다.

## 4.7.2 요약

- 나이브 베이즈는  $p(\mathbf{x}|y) = \prod_i p(x_i|y)$  를 가정하는 분류기를 만들기 쉽습니다.
- 분류기는 학습시키기 쉽지만, 예측이 많이 틀리기 쉽습니다.
- 전반적인 신뢰수준과 틀린 예측을 해결하기 위해서,  $p(x_i|y)$  확률을 Laplace smoothing 과 같은 방법을 적용할 수 있습니다. 즉, 모든 카운트에 상수를 더하는 방법을 적용할 수 있습니다.
- 나이브 베이즈 분류기는 관찰(observation)들 사이의 관계를 고려하지 않습니다.

## 4.7.3 문제

- $p(x_i|y)$  가 표준 분포를 따를 때, 나이브 베이즈 회귀 모델(Naive Bayes regression estimator)을 만들어보세요.
- 어떤 경우 나이브 베이즈가 잘 동작할까요?
- 어떤 목격자가加害자를 다시 봤을 경우, 90% 정확도로 그 사람을 인식할 수 있다고 확신합니다.
  - 5명의 용의자가 있을 경우, 이 사실이 유용할까요?
  - 50명일 경우에도 유용할까요?

#### 4.7.4 Scan the QR Code to Discuss



## 4.8 샘플링

난수는 확률 변수의 한 형태인데, 컴퓨터는 숫자를 다루는 것을 아주 잘하기 때문에 코드를 제외한 거의 모든 것은 결국 숫자로 바뀝니다. 난수를 만드는데 필요한 기본 도구 중에 하나는 어떤 분포에서 샘플을 추출하는 것입니다. 그럼 난수 생성기를 사용했을 때 어떤 일이 벌어지는지 보겠습니다. 우선 필요한 모듈을 import 하겠습니다.

```
[1]: %matplotlib inline
from matplotlib import pyplot as plt
import mxnet as mx
from mxnet import nd
import numpy as np
```

```
[2]: import random
for i in range(10):
    print(random.random())
0.5275699109732184
0.8450735952230787
0.09797154444181
0.5038644410496992
0.9144282547562258
0.882482826471394
0.7540969877613545
0.5544011600051396
0.3497308621920453
0.4328291654808599
```

## 4.8.1 균등 분포(uniform distribution)

위 결과의 숫자들은 굉장히 임의로 선택된 것들입니다. 이 숫자들은 0과 1사이의 범위에서 잘 분포되어 있습니다. 즉, 어떤 특정 구간에 숫자들이 몰려있지 않습니다. (사실 진짜 난수 발생기가 아니기 때문에 그럴수도 있습니다.) 어떤 숫자가 [0.2, 0.3] 구간에서 선택될 가능성과 [.593264, .693264) 구간에서 선택될 가능성이 비슷하다는 의미입니다. 이 난수 발생기는 내부적으로 임의의 정수를 먼저 만들고, 그 다음에 이 숫자를 최대 구간의 숫자로 나누는 방식으로 동작합니다. 정수를 얻고 싶은 경우에는, 다음과 같이하면 됩니다. 아래 코드는 0과 100사이의 임의의 정수를 생성합니다.

```
[3]: for i in range(10):
    print(random.randint(1, 100))

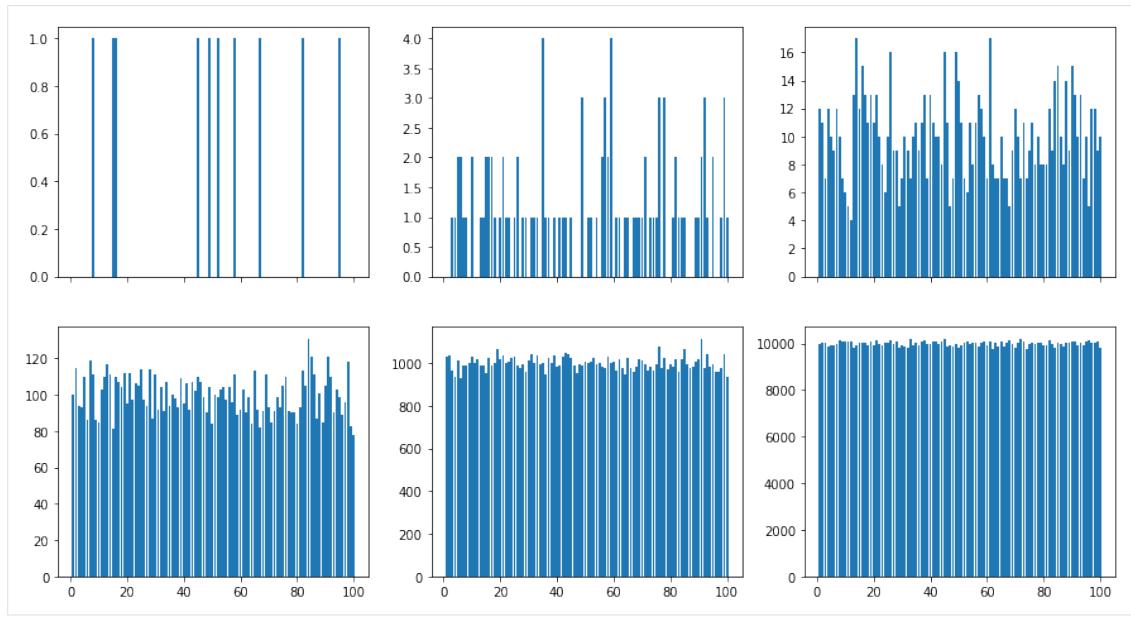
17
50
44
27
74
34
65
37
37
11
```

`randint` 가 정말로 균일하다는 것을 확인을 어떻게 할 수 있을까요? 직관적으로 가장 좋은 방법은 100만개의 난수를 생성해서 각 숫자가 몇 번 나오는지 계산하고, 이 분포가 균일한지를 확인하는 것입니다.

```
[4]: import math

counts = np.zeros(100)
fig, axes = plt.subplots(2, 3, figsize=(15, 8), sharex=True)
axes = axes.reshape(6)
# Mangle subplots such that we can index them in a linear fashion rather than
# a 2d grid

for i in range(1, 1000001):
    counts[random.randint(0, 99)] += 1
    if i in [10, 100, 1000, 10000, 100000, 1000000]:
        axes[int(math.log10(i))-1].bar(np.arange(1, 101), counts)
plt.show()
```



위 결과로 나온 그림으로 확인하면, 초기의 숫자는 균등해 보이지 않습니다. 100개의 정수에 대한 분포에서 100개보다 적은 개수를 뽑는 경우는 당연한 결과입니다. 하지만, 1000 샘플을 뽑아도 여전히 상당한 변동이 있습니다. 우리가 원하는 결과는 숫자를 뽑았을 때의 확률이  $p(x)$  가 되는 것입니다.

## 4.8.2 카테고리 분포(categorical distribution)

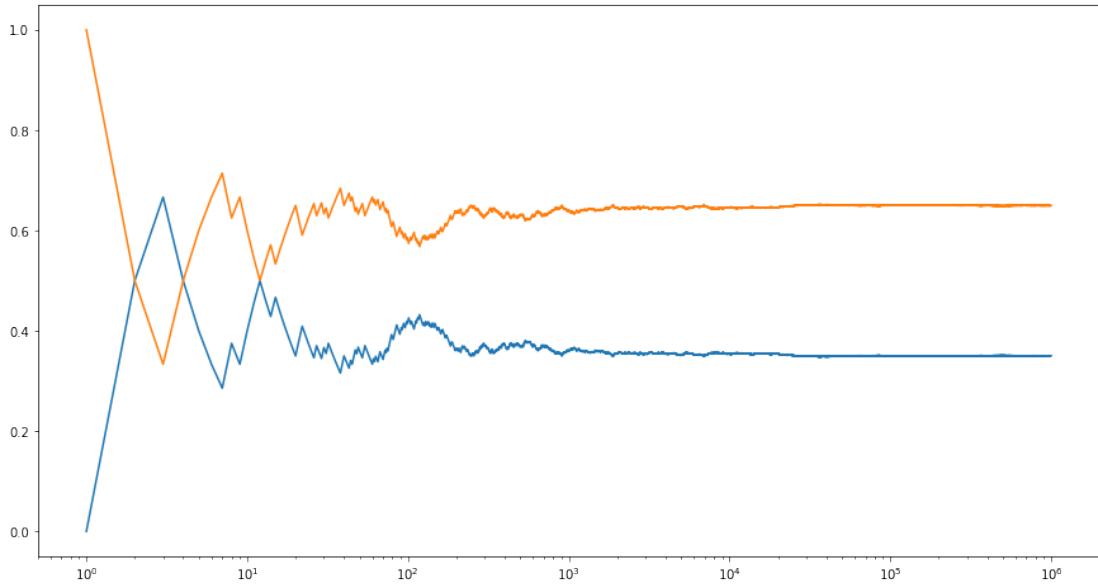
사실, 100개 중에서 균일한 분포로 뽑기를 하는 것은 아주 간단합니다. 만약에 불균일한 확률을 사용해야 한다면 어떻게 해야 할까요? 동전을 던졌을 때 앞면이 나올 확률이 0.35이고 뒷면이 나올 확률이 0.65가 나오는 편향된 (biased) 동전을 간단한 예로 들어보겠습니다. 이를 구현하는 간단한 방법은  $[0, 1]$  구간에서 균일하게 숫자를 선택하고 그 수가 0.35 보다 작으면 앞면으로, 크면 뒷면으로 하는 것입니다. 그럼 코드를 보겠습니다.

```
[5]: # Number of samples
n = 1000000
y = np.random.uniform(0, 1, n)
x = np.arange(1, n+1)

# Count number of occurrences and divide by the number of total draws
p0 = np.cumsum(y < 0.35) / x
p1 = np.cumsum(y >= 0.35) / x
```

(continues on next page)

```
plt.figure(figsize=(15, 8))
plt.semilogx(x, p0)
plt.semilogx(x, p1)
plt.show()
```



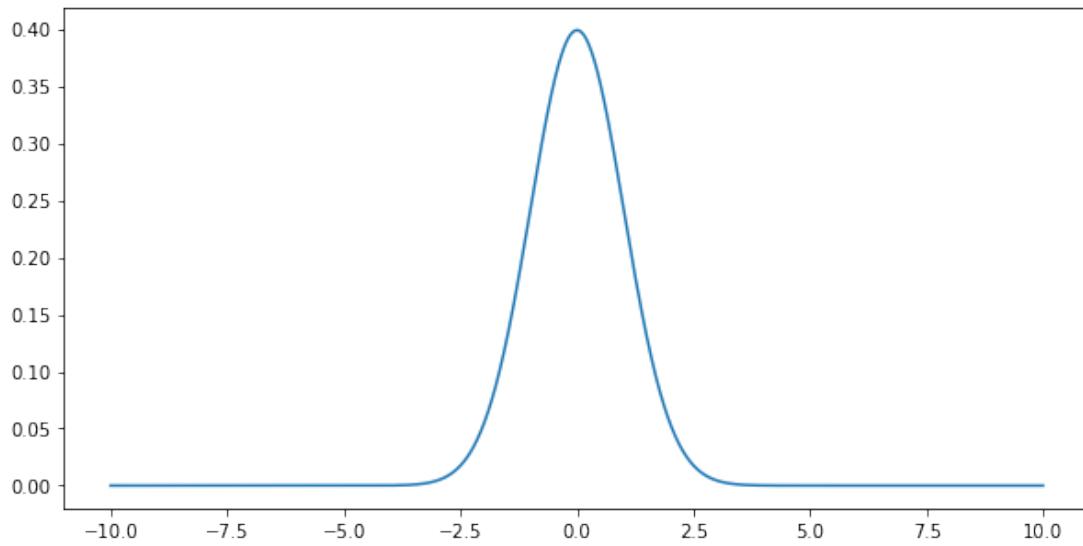
결과에서 보이듯이, 평균적으로 보면 35%가 0이고, 65%가 1입니다. 두 개 이상의 결과가 있다면 어떻게 할까요? 위 아이디어를 다음과 같이 일반화하면 됩니다. 예를 들면  $p = [0.1, 0.2, 0.05, 0.3, 0.25, 0.1]$  와 같은 분포가 있다고 하면, 누적된 분포를 계산해서  $F = [0.1, 0.3, 0.35, 0.65, 0.9, 1]$  를 얻습니다. (이는 Python의 `cumsum` 함수를 이용하면 간단히 할 수 있습니다.) 이전과 동일하게  $U[0, 1]$  범위의 균일 분포에서 난수  $x$  를 뽑고,  $F[i - 1] \leq x < F[i]$  를 만족시키는 구간을 찾아서  $i$  를 샘플 결과로 리턴합니다. 이렇게 하면 난수가  $[F[i - 1], F[i]]$  구간에 속할 확률이  $p(i)$ 가 됩니다.

위 방법보다 훨씬 더 효율적인 알고리즘이 많이 있습니다. 예를 들면,  $n$  개의 랜덤 변수에 대해서  $F$  에 대한 이진 검색을 수행하면  $O(\log n)$  시간이 걸립니다.  $O(n)$  만큼의 전처리를 하면 샘플링에 상수의 시간이 걸리는 [Alias Method](#) 와 같은 더 좋은 알고리즘들이 있습니다.

### 4.8.3 표준 분포(normal distribution)

표준 분포 (또는 가우시안 분포)는  $p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right)$  로 정의됩니다. 그림으로 확인해보겠습니다.

```
[6]: x = np.arange(-10, 10, 0.01)
p = (1/math.sqrt(2 * math.pi)) * np.exp(-0.5 * x**2)
plt.figure(figsize=(10, 5))
plt.plot(x, p)
plt.show()
```



이 분포에서 샘플링을 하는 것은 그리 간단하지 않습니다. 우선 서포트가 무한합니다. 즉, 어떤  $x$  값이 주어지든지 확률 밀도  $p(x)$  값이 양수입니다. 두번째 특징은 확률 밀도는 균일하지 않습니다. 이 분포에서 샘플링을 수행하는 많은 기법이 있는데, 모든 알고리즘에 사용되는 주요 아이디어는  $p(x)$ 를 계층화해서 균일한 분포  $U[0, 1]$ 로 매핑시키는 것입니다. 확률 적분 변환이 그 방법 중에 하나입니다.

$p$ 의 누적 분포 함수(cumulative distribution function, CDF)를  $F(x) = \int_{-\infty}^x p(z)dz$ 로 표기합니다. 이 방법은 앞에서 누적합의 연속된 버전이라고 할 수 있습니다. 이전과 같은 방법으로 균일하게 뽑은  $\xi$ 에 대해서 역으로 매핑하는  $F^{-1}(\xi)$ 를 정의할 수 있습니다. 벡터  $F$ 에 대한 정확한 구간을 찾는 이전의 문제와 다르게, 우리는  $F(x)$ 의 역을 구해야 합니다.

실제로 가우시안의 경우 CDF의 역을 구하는 것은 다소 까다롭습니다. 두 개의 균등한 분포들을 다뤄야 하지만 이차원 적분이 더 다루기 쉽기 때문에 두 개의 표준 확률 변수로 만듭니다. 지금은 이 문제를 해결해주는 알고리즘이 있다고 해두겠습니다.

표준 분포는 또 다른 중요한 특징이 있습니다. 만약 어떤 분포에서 충분히 많은 뽑기를 해서 평균을 구한다면, 모든 분포는 표준 분포에 수렴합니다. 이를 더 자세히 이해하기 위해서, 세가지 중요한 개념들인 기대값(expected value), 평균, 그리고 분산을 소개하겠습니다.

- 분포  $p$ 를 따르는 함수  $f$ 에 대한 기대값  $\mathbf{E}_{x \sim p(x)}[f(x)]$ 은 적분  $\int_x p(x)f(x)dx$ 으로 계산됩니다. 즉, 이는  $p$ 에 따라 주어지는 모든 결과에 대한 평균값입니다.
- 함수  $f(x) = x$ 에 대한 기대값은 굉장히 중요합니다. 이 함수의 기대값은  $\mu := \mathbf{E}_{x \sim p(x)}[x]$ 입니다. 이는 전형적인  $x$ 에 대한 아이디어를 제공해주기 때문입니다.
- 중요한 다른 개념으로는 분산이 있습니다. 이는  $\sigma^2 := \mathbf{E}_{x \sim p(x)}[(x - \mu)^2]$ 으로 표현되며, 평균으로부터 얼마나 떨어져 있는지를 알려줍니다. 간단한 계산을 하면 분산은  $\sigma^2 = \mathbf{E}_{x \sim p(x)}[x^2] - \mathbf{E}_{x \sim p(x)}[x]^2$ 로 표현되기도 합니다.

위 개념은 확률 변수의 평균과 분산을 바꿀 수 있게 해줍니다. 예를 들면 확률 변수  $x$ 의 평균이  $\mu$ 일 경우 확률 변수  $x + c$ 의 평균은  $\mu + c$ 이 됩니다. 또한, 확률 변수가  $\gamma x$  일 경우에는 분산은  $\gamma^2 \sigma^2$ 이 됩니다. 평균  $\mu$ 이고 분산이  $\sigma^2$ 인 확률 변수에 표준 분포(normal distribution)를 적용하면  $p(x) = \frac{1}{\sqrt{2\sigma^2\pi}} \exp(-\frac{1}{2\sigma^2}(x - \mu)^2)$ 의 형태가 됩니다. 스캐일 팩터  $\frac{1}{\sigma}$ 가 적용된 것에 주의하세요. 이렇게 한 이유는 이 분포를  $\sigma$  만큼 늘릴 경우, 같은 확률 값을 갖게 하기 위해서  $\frac{1}{\sigma}$  만큼 줄여야 할 필요가 있기 때문입니다. (왜냐하면 분포의 가중치들의 합은 항상 1이어야하기 때문입니다.)

자 이제 통계학에서 가장 기본적인 이론 중에 하나에 대해서 알아볼 준비가 되었습니다. 이는 Central Limit Theorem입니다. 이 이론은 충분히 잘 행동하는 확률 변수에 대해서, 특히 잘 정의된 평균과 분산을 가지고 있는 확률 변수, 전체 합은 표준 분포로 근접합니다. 이해를 돋기 위해서 시작할 때 사용했던 실험을 정수값  $\{0, 1, 2\}$ 을 갖는 확률 변수를 사용해봅니다.

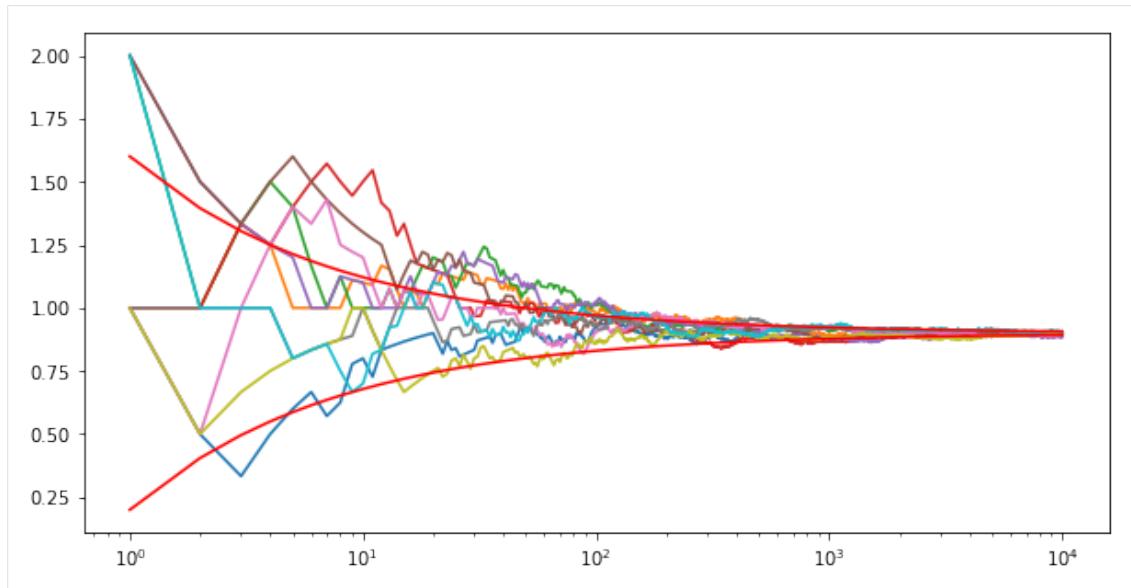
```
[7]: # Generate 10 random sequences of 10,000 uniformly distributed random variables
tmp = np.random.uniform(size=(10000,10))
x = 1.0 * (tmp > 0.3) + 1.0 * (tmp > 0.8)
mean = 1 * 0.5 + 2 * 0.2
variance = 1 * 0.5 + 4 * 0.2 - mean**2
print('mean {}, variance {}'.format(mean, variance))

# Cumulative sum and normalization
y = np.arange(1,10001).reshape(10000,1)
z = np.cumsum(x, axis=0) / y

plt.figure(figsize=(10,5))
for i in range(10):
    plt.semilogx(y,z[:,i])

plt.semilogx(y,(variance**0.5) * np.power(y,-0.5) + mean, 'r')
plt.semilogx(y,-(variance**0.5) * np.power(y,-0.5) + mean, 'r')
plt.show()

mean 0.9, variance 0.49
```



위 결과를 보면 많은 변수들의 평균만을 보면 처음 예제와 아주 비슷하게 보입니다. 즉, 이론이 맞다는 것을 보여줍니다. 확률 변수의 평균과 분산은 다음과 같이 표현됩니다.

$$\mu[p] := \mathbf{E}_{x \sim p(x)}[x] \text{ 와 } \sigma^2[p] := \mathbf{E}_{x \sim p(x)}[(x - \mu[p])^2]$$

그러면,  $\lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} \sum_{i=1}^n \frac{x_i - \mu}{\sigma} \rightarrow \mathcal{N}(0, 1)$  이 됩니다. 즉, 어떤 값으로부터 시작했는지 상관없이, 가우시안 분포에 항상 수렴하게 됩니다. 이것이 통계에서 가우시안 분포가 유명한 이유들 중에 하나입니다.

#### 4.8.4 여러 분포들

그 외에도 유용한 분산들이 많이 있습니다. 더 자세한 내용은 통계책이나 위키피디아를 참조하세요.

- **이항 분포(Binomial Distribution)** 같은 분포에서 여러번 뽑을 때의 분포를 설명하는데 사용됩니다. 즉, 편향된 동전(동전 앞면이 나올 확률이  $\pi \in [0, 1]$  인 동전을 사용할 때)을 10번 던져서 앞면이 나오는 횟수. 이산 분포는  $p(x) = \binom{n}{x} \pi^x (1 - \pi)^{n-x}$  입니다.
- **다항 분포(Multinomial Distribution)** 두개보다 많은 결과가 있을 경우에 해당합니다. 즉, 주사위를 여러번 던지는 경우를 예로 들 수 있습니다. 이 경우 분포는  $p(x) = \frac{n!}{\prod_{i=1}^k x_i!} \prod_{i=1}^k \pi_i^{x_i}$  로 주어집니다.

- **포아송 분포(Poisson Distribution)** 주어진 속도(rate)에 따라서 일어나는 이벤트를 모델링할 때 사용됩니다. 예를 들면, 어느 공간에 일정 시간 동안 떨어지는 빗방울의 수가 됩니다. (특이한 사실은, 프리시안 군인들이 말의 발길에 치여서 죽은 수가 이 분포를 따르고 있습니다.) 속도  $\lambda$ 에 대해서, 일이 일어날 확률은  $p(x) = \frac{1}{x!} \lambda^x e^{-\lambda}$ 로 표현됩니다.
- **베타, 디리치(Dirichlet), 감마, 위ഷ트(Wishart)** 분포 통계학자들은 이것들을 각각 이산, 다항, 포아송, 그리고 가우시안 분포의 변종이라고 설명하고 있습니다. 이 분포들은 분포들의 집합에 대한 계수를 위한 사전 순위로 사용되는데, 자세한 설명은 생략하겠습니다. 이산 결과들의 확률을 모델링하는데 사전 순위로의 베타 분포 같은 것입니다.

#### 4.8.5 Scan the QR Code to Discuss



## 4.9 문서(documentation)

이 책에서 MXNet 함수와 클래스를 모두 설명하기는 불가능하니, API 문서나 추가적인 튜토리얼과 예제를 참고하면 이 책에서 다루지 못한 많은 내용을 찾아볼 수 있습니다.

### 4.9.1 모듈의 모든 함수와 클래스 찾아보기

모듈에서 어떤 함수와 클래스가 제공되는지 알기 위해서 `dir` 함수를 이용합니다. 예를 들어, `nd.random` 모듈의 모든 멤버와 속성을 다음과 같이 조회할 수 있습니다.

```
[1]: from mxnet import nd
print(dir(nd.random))

['NDArray', '_Null', '__all__', '__builtins__', '__cached__', '__doc__',
 ← '__file__', '__loader__', '__name__', '__package__', '__spec__',
 ← '__internal', '__random_helper', 'current_context', 'exponential',
 ← 'exponential_like', 'gamma', 'gamma_like', 'generalized_negative_binomial',
 ← 'generalized_negative_binomial_like', 'multinomial', 'negative_binomial',
 ← 'negative_binomial_like', 'normal', 'normal_like', 'numeric_types',
 ← 'poisson', 'poisson_like', 'randint', 'randn', 'shuffle', 'uniform',
 ← 'uniform_like']
```

일반적으로 이름이 \_\_ 로 시작하는 함수(Python에서 특별한 객체를 나타냄)나 \_ 로 시작하는 함수(보통은 내부 함수들)는 무시해도 됩니다. 나머지 멤버들에 대해서는 이름을 통해 추측해보면, 다양한 난수를 생성하는 메소드들로 추측할 수 있습니다. 즉, 균일한 분포에서 난수를 생성하는 uniform, 표준 분산에서 난수를 생성하는 normal 그리고 Poisson 샘플링인 poisson 등의 기능을 제공함을 알 수 있습니다.

## 4.9.2 특정 함수들과 클래스들의 사용법 찾아보기

help 함수를 이용하면 특정 함수나 클래스의 사용법 확인할 수 있습니다. NDArray의 ones\_like 함수를 예로 살펴봅니다.

```
[2]: help(nd.ones_like)

Help on function ones_like:

ones_like(data=None, out=None, name=None, **kwargs)
    Return an array of ones with the same shape and type
    as the input array.

Examples::

x = [[ 0.,  0.,  0.],
      [ 0.,  0.,  0.]]

ones_like(x) = [[ 1.,  1.,  1.],
                  [ 1.,  1.,  1.]]


Parameters
-----
data : NDArray
    The input

out : NDArray, optional
```

(continues on next page)

The output NDArray to hold the result.

#### Returns

-----

`out : NDArray or list of NDArrays`  
The output of this function.

문서를 보면, `ones_like` 함수는 NDArray 객체와 모두 1로 설정된 같은 모양(shape)의 새로운 객체를 만들어 줍니다. 확인해보겠습니다.

```
[3]: x = nd.array([[0, 0, 0], [2, 2, 2]])
y = x.ones_like()
y
```

```
[3]: [[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @cpu(0)>
```

Jupyter 노트북에서는 ? 를 이용해서 다른 윈도우에 문서를 표시할 수 있습니다. 예를 들어 `nd.random.uniform?` 를 수행하면 `help(nd.random.uniform)` 과 거의 동일한 내용이 다른 윈도우에 나옵니다. 그리고, `nd.random.uniform??` 와 같이 ? 를 두개 사용하면, 함수를 구현하는 코드도 함께 출력됩니다.

### 4.9.3 API 문서

API에 대한 더 자세한 내용은 MXNet 웹사이트 <http://mxnet.apache.org/> 를 확인하세요. Python 및 이외의 다른 프로그램 언어에 대한 내용들을 웹 사이트에서 찾을 수 있습니다.

### 4.9.4 문제

1. API 문서에서 `ones_like` 와 `autograd`를 찾아보세요.

#### 4.9.5 Scan the QR Code to Discuss





# 5

---

## 딥러닝 기초

---

이 장에서는 딥러닝의 기본적인 내용들을 소개합니다. 네트워크 아키텍처, 데이터, 손실 함수(loss function), 최적화, 그리고 용량 제어를 포함합니다. 이해를 돋기 위해서, 선형 함수, 선형 회귀, 그리고 확률적 경사 하강법(stochastic gradient descent)과 같은 간단한 개념부터 시작합니다. 이것들은 softmax나 다층 퍼셉트론(multilayer perceptron)와 같은 보다 복잡한 개념의 기초가 됩니다. 우리는 이미 상당히 강력한 네트워크를 디자인할 수 있지만, 필수적인 제어나 기교는 배우지 않았습니다. 이를 위해서, 용량 제어, 오버피팅(overfitting)과 언더피팅(underfitting)에 대한 개념을 이해할 필요가 있습니다. 드롭아웃(dropout), 수치 안정화(numerical stability), 그리고 초기화에 대한 설명으로 이 장을 마무리할 예정입니다. 우리는 실제 데이터에 모델을 적용하는 방법에 집중하겠습니다. 이를 통해서 여러분은 기본 개념 뿐만 아니라 딥 네트워크를 실제 문제에 적용할 수 있도록 할 예정입니다. 성능, 확장성 그리고 효율성은 다음 장들에서 다룹니다.

### 5.1 선형 회귀(Linear Regression)

우선 간단한 문제인 회귀 문제(regression)를 살펴보겠습니다. 회귀 문제는 주어진 데이터포인트  $x$ 에 해당하는 실제 값으로 주어지는 타겟  $y$ 를 예측하는 과제입니다. 회귀 문제는 현실에서 많이 보이는 문제입니다. 예를 들면, 주택 가격, 기온, 판매량 등과 같은 연속된 값을 예측하는 문제들을 들 수 있

습니다. 이는 결과 값이 이미지 분류와 같이 파일의 종류를 예측하는 이산적인(discrete) 구분 문제(classification)와는 다릅니다.

### 5.1.1 선형 회귀의 기본 요소들

가장 간단하지만 가장 유용한 접근 방법인 선형 회귀를 정의하는 방법으로 예측 함수가 입력 피처들의 선형 조합으로 표현된다고 가정합니다. 이 때문에, 이를 선형 회귀(linear regression)라는 이름으로 부릅니다.

### 5.1.2 선형 모델

간단한 예로, 집의 면적(제곱 미터)과 지어진 후 몇 년이 되었는지를 입력으로 사용해서 주택 가격을 예측하는 문제를 들어보겠습니다. 이 경우 모델을 다음과 같은 수식으로 표현할 수 있습니다.

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b$$

이 공식은 명확해 보이는데, 두 개 이상의 입력 변수가 사용되는 경우는 굉장히 긴 공식이 됩니다. (변수 이름을 지정하는 것조차 지루한 일입니다.) 하지만, 수학자들이 발명한 벡터를 사용하면 간단하게 표현이 가능합니다.  $d$  개의 변수가 있다고 하면, 모델은 아래와 같이 표현됩니다.

$$\hat{y} = w_1 \cdot x_1 + \dots + w_d \cdot x_d + b$$

데이터 포인트들을  $X$ 로, 타겟 변수는  $y$ 로 표현합니다. 각 데이터 포인트  $x_i$  와 이에 대한 label 값인  $y_i$ 를 추정해서 연관시켜주는  $\Gamma_j X$ (weight) 벡터  $w$  와 bias  $b$ 를 찾아보는 것을 시도해봅니다. 이를 조금 전문적인 수학 기호로 표현하면, 위 긴 합은  $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$  이 됩니다. 마지막으로, 데이터 포인트들의 집합  $X$  와 예측 값  $\hat{y}$  은 아래와 같은 행렬-벡터 곱의 공식이 됩니다.

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$$

$x$  와  $y$  의 관계가 대략 선형적이라고 가정하는 것은 상당히 합리적입니다. 측정하는데 다소 오류가 발생할 수도 있습니다. 예를 들면, 주택 가격은 일반적으로 하락하지만, 오래될 수록 가치가 더 해지는, 오래된 역사적인 주택의 경우는 해당되지 않을 수 있습니다. 파라메터  $w$  를 찾기 위해서는 두 가지가 더 필요합니다. 하나는, 현재 모델의 품질(quality)을 측정하는 방법과 두 번째는 품질을 향상시킬 수 있는 방법입니다.

### 5.1.3 학습 데이터

우선 필요한 것은 데이터입니다. 예를 들면, 여러 집들의 실제 판매가격과 그 집들의 크기와 지어진 후 몇 년이 지났는지에 대한 데이터가 필요합니다. 우리가 하고자 하는 것은 모델이 예측한 집 가격과 실제 가격의 차이(오류)를 최소화하는 모델 파라미터를 찾는 것입니다. 머신러닝의 용어로는, 데이터 셋은 ‘학습 데이터’ 또는 ‘학습 셋’이라고 하고, 하나의 집 (집과 판매가격)을 ‘샘플’, 그리고 그 집의 판매가격을 ‘레이블(label)’이라고 합니다. 레이블을 예측하기 위해서 사용된 두 값은 ‘피쳐(feature)’ 또는 ‘공변량(covariate)’이라고 합니다. 피쳐는 샘플의 특징을 표현하는데 사용됩니다.

일반적으로 수집한 샘플의 개수를  $n$  으로 표기하고, 각 샘플은 인덱스  $i$  를 사용해서  $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$  와 레이블은  $y^{(i)}$  로 표현합니다.

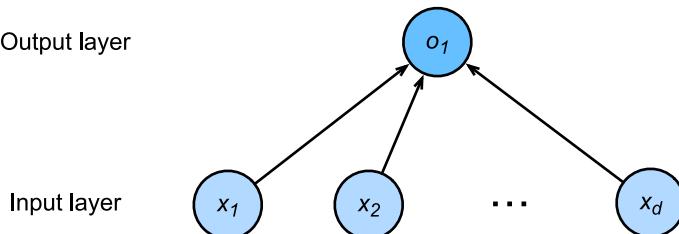
### 5.1.4 Loss 함수

모델 학습을 위해서는 모델이 예측된 가격과 실제 가격의 오차를 측정해야 합니다. 보통 오차는 0 또는 양수값을 선택하고, 값이 작을 수록, 오차가 적음을 의미합니다. 일반적으로 제곱 함수를 사용하며, index  $i$  의 샘플에 대한 오차 계산은 다음과 같이 합니다.

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2,$$

수식에 곱해진  $1/2$  상수값은 2차원 항목을 미분했을 때 값이 1이되게 만들어서 조금 더 간단하게 수식을 만들기 위해서 사용된 값입니다. 이를 사용하면, 오류값이 작을 수록 예측된 값이 실제 가격과 더 비슷해지고, 두 값이 같으면 오류는 0이 됩니다. 학습 데이터셋이 주어졌을 때, 이 오류는 모델 파라미터들에만 의존합니다. 즉, 이 함수를 모델 파라미터를 파라미터로 갖는 함수로 생각할 수 있습니다. 머신러닝에서는 이와같이 오류를 측정하는 함수를 ‘loss function’ 이라고 부릅니다. 위에서 정의한 제곱 오류 함수는 ‘square loss’라고 부릅니다.

조금 더 구체적으로 살펴보기 위해서, 집값이 집 크기에만 의존한다는 모델을 가정해서 일차원 문제로 회귀 문제를 도식화한 것을 예로 들어보겠습니다.



위 그래프에서 보이는 것처럼, 이차 의존성 (quadratic dependence)으로 인해서 예측값  $\hat{y}^{(i)}$  과 실제값

$y^{(i)}$  의 큰 차이는 loss 에서는 더 크게 반영됩니다. 전체 데이터셋에 대해서 모델의 품질을 측정하기 위해서 학습셋에 대한 loss의 평균값을 사용할 수 있습니다.

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

학습 샘플들의 평균 loss를 최소화하는 모델 파라미터  $\mathbf{w}^*$  와  $b^*$  를 찾는 것이 모델을 학습시키는 것입니다.

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

### 5.1.5 최적화 알고리즘

모델과 loss 함수가 상대적으로 간단하게 표현되는 경우 앞에서 정의한 loss를 최소화하는 방법은 역행렬을 사용해서 명확한 수식으로 표현할 수 있습니다. 이 수식은 다양하고 좋은 수학적 분석을 적용할 수 있어서 좋지만, 적은 경우에만 적용할 수 있는 제약이 있습니다.(즉, multilayer perceptron이나 비선형 레이어가 있으면 적용할 수 없습니다.) 대부분 딥러닝 모델은 위 분석적 방법을 적용할 수 없습니다. loss 함수의 값은 점진적인 최적화 알고리즘을 사용해서 모델 파라미터를 유한한 횟수로 업데이트하면서 줄이는 방법을 적용해야만 합니다.

딥러닝에서는 산술적인 솔루션으로 미니 배치를 적용한 stochastic gradient descent 방법이 널리 사용되고 있습니다. 사용되는 알고리즘은 간단합니다: 우선, 일반적으로는 난수를 이용해서 모델 파라미터를 초기화합니다. 그 후, 데이터를 반복적으로 사용해서 loss 함수의 값을 줄이는 것을 반복합니다. 각 반복에서는 학습 데이터에서 미리 정한 개수만큼의 샘플들을 임의로 또 균일하게 뽑아서 미니 배치  $\mathcal{B}$  를 구성하고, 미니 배치의 값들에 대한 평균 loss 값의 모델 파라미터에 대한 미분을 구합니다. 마지막으로 이 결과와 미리 정의된 스텝 크기  $\eta > 0$  를 곱해서 loss 값이 최소화되는 방향으로 파라미터를 변경합니다. 수식으로 표현하면 다음과 같습니다.

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

이차원 loss 및 선형 함수에 대해서는 아래와 같이 명시적으로 이를 계산할 수 있습니다. 여기서  $\mathbf{w}$  와  $\mathbf{x}$  는 벡터입니다. 벡터를 잘 사용하면  $w_1, w_2, \dots, w_d$  와 같은 계수를 읽기 쉬운 수식으로 표현할 수

있습니다.

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

위 수식에서  $|\mathcal{B}|$  는 각 미니 배치의 샘플 개수를 의미하고,  $\eta$  는 ‘학습 속도(learning rate)’를 의미 합니다. 학습 속도는 양수 값을 사용합니다. 여기서 중요한 점은 미니 배치 크기와 학습 속도는 모델 학습을 통해서 찾아지는 값이 아니라 여러분이 직접 선택을 해야하는 값들입니다. 따라서, 우리는 이 값들을 *hyper-parameters* 라고 부릅니다. 우리가 흔히 *hyper-parameters* 튜닝이라고 하는 일은 이 값들을 조정하는 것을 의미합니다. 아주 나쁜 경우에는 적당한 *hyper-parameters* 를 찾기까지 반복된 실험을 수행해야 할 수도 있습니다. 더 좋은 방법으로는 모델 학습의 일부로 이 값들을 찾는 것이 있습니다만, 심화 주제이기 때문에 여기서는 다루지 않겠습니다.

### 5.1.6 모델을 이용한 예측

모델 학습이 끝나면 모델 파라미터  $\mathbf{w}, b$ 에 해당하는 값  $\hat{\mathbf{w}} @ \hat{b}$  을 저장합니다. 학습을 통해서 loss 함수를 최소화 시키는 최적의 값  $\mathbf{w}^*, b^*$  를 구할 필요는 없습니다. 다만, 이 최적의 값에 근접하는 값을 학습을 통해서 찾는 것입니다. 이후, 학습된 선형 회귀 모델  $\hat{\mathbf{w}}^\top x + \hat{b}$  을 이용해서 학습 데이터셋에 없는 집 정보에 대한 집 가격을 추정합니다. “추정”을 “모델 예측(prediction)” 또는 “모델 추론 (inference)” 라고 합니다.

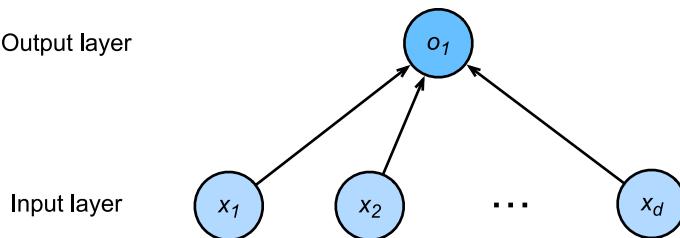
“추론(inference)”이라는 용어는 실제로는 잘못 선택된 용어지만, 딥러닝에서는 많이 사용하는 용어로 자리잡았습니다. 통계에서 추론은 다른 데이터를 기반으로 파라미터들과 결과를 추정하는 것을 의미하기 때문에, 통계학자들과 이야기할 때 이 용어로 인해서 혼동을 가져오기도 합니다. 하지만, 이미 보편적으로 사용되고 있기 때문에, 학습된 모델에 새로운 데이터를 적용하는 것을 추론이라는 용어로 사용하겠습니다. (수백년을 걸친 통계학자들에게 미안함을 표합니다.)

### 5.1.7 선형 회귀에서 딥 네트워크로

지금까지 선형 함수만을 이야기했는데, 뉴럴 네트워크는 이 보다 많은 것을 다룹니다. 물론 선형 함수는 중요한 구성 요소입니다. 이제 모든 것을 ‘층(layer)’ 표기법으로 다시 기술해 보겠습니다.

### 5.1.8 뉴럴 네트워크 다이어그램

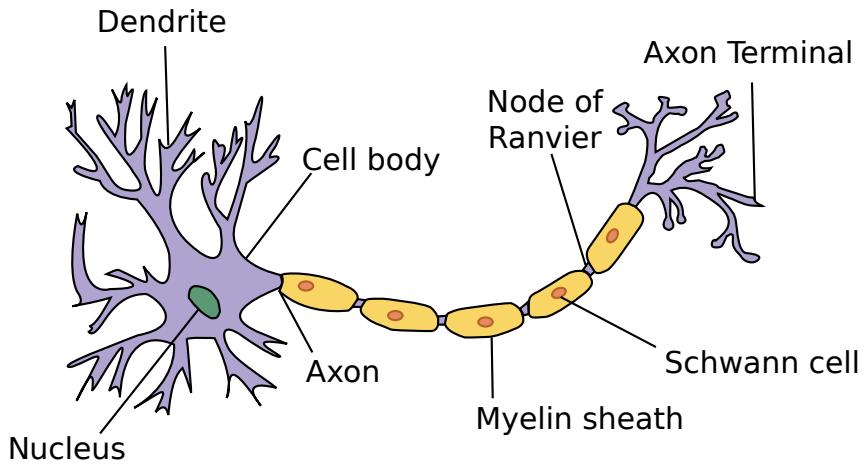
딥러닝에서는 모델의 구조를 뉴럴 네트워크 다이어그램으로 시각화할 수 있습니다. 뉴럴 네트워크 구조로 선형 회귀를 좀 더 명확하게 표현해보면, 그림 3.1에서와 같이 뉴럴 네트워크 다이어그램을 이용해서 이 절에서 사용하고 있는 선형 회귀 모델을 도식화 할 수 있습니다. 이 뉴럴 네트워크 다이어그램에서는 모델 파라미터인 가중치와 bias를 직접 표현하지 않습니다.



위 뉴럴 네트워크에서 입력값은  $x_1, x_2, \dots, x_d$  입니다. 때로는 입력값의 개수를 피처 차원(feature dimension)이라고 부르기도 합니다. 이 경우에는 입력값의 개수는  $d$  이고, 출력값의 개수는 1입니다. 출력값을 선형 회귀의 결과를 직접 결과로 사용한다는 것을 기억해두세요. 입력 레이어에는 어떤 비선형이나 어떤 계산이 적용되지 않기 때문에, 이 네트워크의 총 레이어의 개수는 1개입니다. 종종 이런 네트워크를 단일 뉴론이라고 부르기도 합니다. 모든 입력들이 모든 출력(이 경우는 한개의 출력)과 연결되어 있기 때문에, 이 레이어는 fully connected layer 또는 dense layer라고 불립니다.

### 5.1.9 생물학으로 우회

뉴럴 네트워크라는 이름은 신경과학으로부터 나왔습니다. 얼마나 많은 네트워크 구조가 만들어졌는지 잘 이해하기 위해서, 우선 뉴론(neuron)의 기본적인 구조를 살펴볼 필요가 있습니다. 비유하자면, 입력 단자인 수상돌기(dendrites), CPU인 핵(nucleus), 출력 연결인 축삭(axon), 그리고, 시냅스를 통해서 다른 뉴런과 연결하는 축삭 단자(axon terminal)라고 하면 충분합니다.



수상돌기는 다른 뉴론들로부터 온 정보  $x_i$ 를 받습니다. 구체적으로는 그 정보는 시텝틱 가중치  $w_i$ 가 적용된 정보값입니다. 이 가중치는 입력에 얼마나 반응을 해야하는지 정의합니다. (즉,  $x_i w_i$ 를 통해서 활성화 됨) 이 모든 값들은 핵에서  $y = \sum_i x_i w_i + b$ ,로 통합되고, 이 정보는 축삭(axon)으로 보내져서 다른 프로세스를 거치는데, 일반적으로는  $\sigma(y)$ 를 통해서 비선형 처리가 됩니다. 이 후, 최종 목적지 (예를 들면 근육) 또는 수상돌기를 거쳐서 다른 뉴론으로 보내집니다.

뇌의 구조는 아주 다양합니다. 어떤 것들은 다소 임의적으로 보이지만, 어떤 것들은 아주 규칙적인 구조를 가지고 있습니다. 예를 들면, 여러 곤충들의 시각 시스템은 아주 구조적입니다. 이 구조들에 대한 분석을 통해서 신경과학자들은 새로운 아키텍처를 제안하는데 영감을 받기도 하고, 어떤 경우에는 아주 성공적이었습니다. 하지만, 비행기가 새로부터 영감을 받아서 만들어 졌지만 차이가 많은 것과 같이, 이 둘의 직접적인 관계를 찾아보는 것은 오류가 되기도 합니다. 수학과 컴퓨터과학이 영감의 같은 근원이라고 볼 수 있습니다

### 5.1.10 벡터화로 빠르게 만들기

모델 학습 및 예측을 수행할 때, 벡터 연산을 사용하고 이를 통해서 여러 값들은 한번에 처리합니다. 이것이 왜 중요한지 설명하기 위해서 벡터들을 더하는 두 가지 방법을 생각해봅시다. 우선 1000 차원의 벡터 두 개를 생성합니다.

```
[1]: from mxnet import nd
from time import time

a = nd.ones(shape=10000)
b = nd.ones(shape=10000)
```

두 벡터를 더하는 방법 중에 하나는 for loop을 이용해서 벡터의 각 값들을 하나씩 더하는 것입니다.

```
[2]: start = time()
c = nd.zeros(shape=10000)
for i in range(10000):
    c[i] = a[i] + b[i]
time() - start
[2]: 1.8443152904510498
```

다른 방법으로는 두 벡터를 직접 더할 수 있습니다.

```
[3]: start = time()
d = a + b
time() - start
[3]: 0.00046372413635253906
```

당연하게도 벡터를 직접 더하는 방법이 훨씬 더 빠릅니다. 코드를 벡터화하는 것은 연산 속도를 빠르게 하는 좋은 방법입니다. 마찬가지로 연산식을 간단하게 하고, 표기에 있어서 잠재적인 오류를 줄여주는 효과도 있습니다.

### 5.1.11 표준 분포와 제곱 Loss

아래 내용은 옵션이니, 다음으로 넘어가도 됩니다. 하지만, 딥러닝 모델을 구성에 있어서 디자인 선택에 대한 이해를 높이는데 도움이 됩니다. 위에서 봤듯이, squared loss  $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$  는 간단한 편미분  $\partial_y l(y, \hat{y}) = (\hat{y} - y)$  과 같은 좋은 특징들을 가지고 있습니다. 즉, gradient가 예측값과 실제 값의 차이로 계산됩니다. 선형 회귀는 전통적인 통계 모델입니다. Legendre가 처음으로 least squares regression을 1805년에 개발했고, 1809년에 Gauss에 의해서 재발견되었습니다. 이를 조금 더 잘 이해하기 위해서 평균이  $\mu$ 이고 분산이  $\sigma^2$ 인 정규 분포(normal distribution)를 떠올려봅시다.

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

이는 다음과 같이 시각화될 수 있습니다.

```
[4]: %matplotlib inline
from matplotlib import pyplot as plt
from IPython import display
from mxnet import nd
import math
```

(continues on next page)

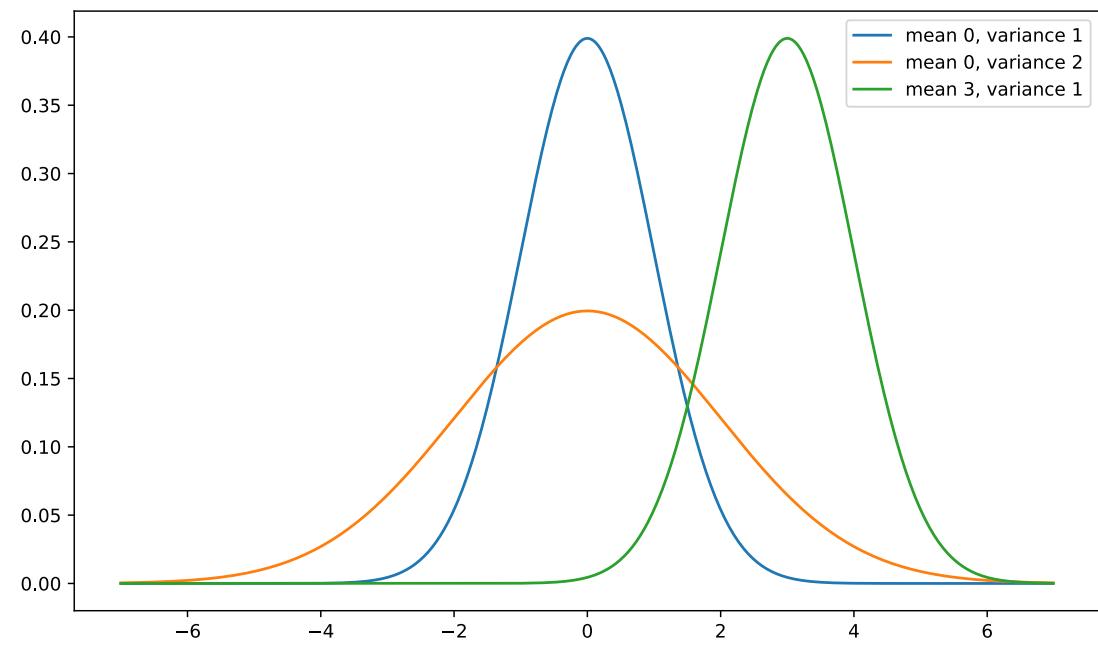
```

x = np.arange(-7, 7, 0.01)
# Mean and variance pairs
parameters = [(0,1), (0,2), (3,1)]

# Display SVG rather than JPG
display.set_matplotlib_formats('svg')
plt.figure(figsize=(10, 6))
for (mu, sigma) in parameters:
    p = (1/math.sqrt(2 * math.pi * sigma**2)) * np.exp(-(0.5/sigma**2) * (x-mu)**2)
    plt.plot(x, p, label='mean ' + str(mu) + ', variance ' + str(sigma))

plt.legend()
plt.show()

```



위 그림에서 보이듯이, 평균을 변경하면 함수를 이동시키고, 편차를 증가시키면 피크는 낮추고 더 페쳐지게 만듭니다. least mean square loss를 적용한 선형 회귀에서 중요한 가정은 관찰들은 노이즈가 있는 관찰에서 얻어지고, 이 노이즈들은 데이터에 더해진다는 것입니다.

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

이는 주어진  $x$ 에 대해서 특정  $y$ 를 얻을 가능성을(likelihood)는 다음과 같이 표현됩니다.

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

가장 근접한  $b$ 와  $\mathbf{w}$  값을 찾는 좋은 방법은 전체 데이터셋에 대한 likelihood를 최대화하는 것입니다.

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

파라미터들에 대해서 데이터의 likelihood를 최대화하는 것은 *Maximum Likelihood Principle*로 잘 알려져 있고, 이런 estimator들은 *Maximum Likelihood Estimators* (MLE)라고 불립니다. 어렵게도, 많은 지수 함수들의 곱을 최적화하는 것은 구현하는 것이나, 종이에 적어보는 것이나 아주 어렵습니다. 대신, 더 좋은 방법은  $Negative Log-Likelihood - \log P(Y|X)$ 를 최소화하는 것입니다. 위 예는 다음 수식으로 표현됩니다.

$$-\log P(Y|X) = \sum_{i=1}^n \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b)^2$$

이 공식을 잘 살펴보면  $-\log P(Y|X)$ 를 최소화할 때는 첫번째 항목을 무시할 수 있습니다. 왜냐하면, 첫번째 항목은  $w$ ,  $b$  그리고 데이터와도 연관이 없기 때문입니다. 두번째 항목은 우리가 이전에 봤던 objective와 상수  $\frac{1}{\sigma^2}$  가 곱해진 것을 빼면 동일합니다. 이 값은 가장 근접한 솔루션을 찾는 것만 원한다면 무시할 수 있고, 이렇게 하면 additive Gaussian noise를 갖는 선형 모델의 likelihood를 최대화하는 것은 squared loss를 적용한 선형 회귀와 동일한 문제로 정의됩니다.

### 5.1.12 요약

- 머신러닝에서 중요한 요소는 학습 데이터, loss 함수, 최적화 알고리즘, 그리고 당연하지만 모델 자체입니다.
- 벡터화는 모든 것(수학)을 좋게 만들고, (코드를) 빠르게 만들어 줍니다.
- objective 함수를 최소화하는 것과 maximum likelihood를 구하는 것은 같은 것입니다.
- 선형 모델도 뉴럴 모델입니다.

### 5.1.13 문제

1. 데이터  $x_1, \dots, x_n \in \mathbb{R}$  가 있다고 가정합니다. 우리의 목표는  $\sum_i (x_i - b)^2$  를 최소화시키는 상수  $b$  를 찾는 것입니다.
  - 최적의 닫힌 형식의 해(closed form solution)을 찾아보세요.
  - 표준 분포의 용어로 이것은 무엇을 의미하나요?
2. Quadratic loss를 사용한 선형 회귀의 최적화 문제를 닫힌 형식으로 풀기를 원한다고 가정합니다. 간단하게 하기 위해서, 문제에서 편향(bias)  $b$  는 생략해도 됩니다.
  - 문제를 행렬과 벡터 표기를 사용해서 다시 기술해보세요. (힌트 - 모든 데이터를 하나의 행렬로 취급합니다.)
  - 최적화 문제를  $w$  에 대한 경사(gradients)를 계산하세요.
  - 행렬 방식을 풀어서 닫힌 형식의 해(closed form solution)를 구하세요.
  - 언제 이 방법이 확률적 경사 하강법(stochastic gradient descent) (즉, 위에서 우리가 논의한 점진적 최적화 접근법)보다 좋을까요? 언제 이 방법이 틀릴까요? (힌트 -  $x$ 가 고차원이면 어떻게 될까요? 만약 많은 관찰들이 아주 작으면 어떻게 될까요?)
3. 부가 노이즈  $\epsilon$  을 결정하는 노이즈 모델이 지수 분포라고 가정합니다. 즉,  $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$ .
  - 모델  $-\log p(Y|X)$  의 데이터에 대한 네가티브 로그-가능도(log-likelihood)를 적어보세요.
  - 닫힌 형식의 해를 찾을 수 있나요?
  - 이 문제를 푸는 확률적 경사 하강법(stochastic gradient descent) 알고리즘을 제안하세요. 무엇이 잘못될 가능성이 있나요? (힌트 - 파라미터를 업데이트할 때, 정류점(stationary point) 근처에서 어떤 일이 일어날까요?)
4. NumPy와 같은 다른 패키지들이나 MATLAB과 같은 다른 프로그램 언어를 사용해서 두 벡터를 더할 때의 실행시간들을 비교해보세요.

### 5.1.14 Scan the QR Code to Discuss



## 5.2 선형 회귀를 처음부터 구현하기

선형 회귀에 대한 어느 정도의 배경 지식을 습득했으니 이제 실제 구현을 해보도록 하겠습니다. 좋은 딥러닝 프레임워크를 이용하면 반복적인 일을 줄일 수 있지만, 일을 쉽게 하기 위해서 프레임워크에 너무 의존하면 딥러닝이 어떻게 동작하는지 이해하기 어렵게 될 수 있습니다. 예를 들면, 이후에 레이어, loss 함수 등을 직접 정의해야하는 경우에 특히 그렇습니다. 따라서, NDArray와 autograd만을 이용해서 선형 회귀 학습을 직접 구현해보겠습니다.

시작하기 앞서, 이 절에서 필요한 패키지와 모듈을 import 합니다. matplotlib는 도표를 그리고 화면에 표시하는데 사용됩니다.

```
[1]: %matplotlib inline
from IPython import display
from matplotlib import pyplot as plt
from mxnet import autograd, nd
import random
```

### 5.2.1 데이터셋 생성하기

간단한 학습 데이터셋을 직접 만들어서 학습된 파라미터와 실제 모델의 파라미터의 차이를 시각적으로 비교해볼 수 있습니다. 학습 데이터셋의 샘플 개수는 1000개로 하고, 특성(feature) 개수는 2개로 합니다. 임의로 생성한 배치 샘플 특성(feature)  $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$  와 실제 가중치 값  $\mathbf{w} = [2, -3.4]^\top$  와 편향(bias)  $b = 4.2$ 를 사용하겠습니다. 그리고, 임의의 노이즈 값  $\epsilon$ 도 사용합니다.

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon$$

노이즈 항목  $\epsilon$ 은 평균이 0이고 표준편차가 0.01인 정규 분포를 따르도록 정의합니다. 아래 코드로 실제 데이터셋을 생성합니다.

```
[2]: num_inputs = 2
num_examples = 1000
true_w = nd.array([2, -3.4])
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = nd.dot(features, true_w) + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

features의 각 행은 2차원 데이터 포인트로 구성되고, labels의 각 행은 1차원 타겟 값으로 구성됩니다.

```
[3]: features[0], labels[0]
```

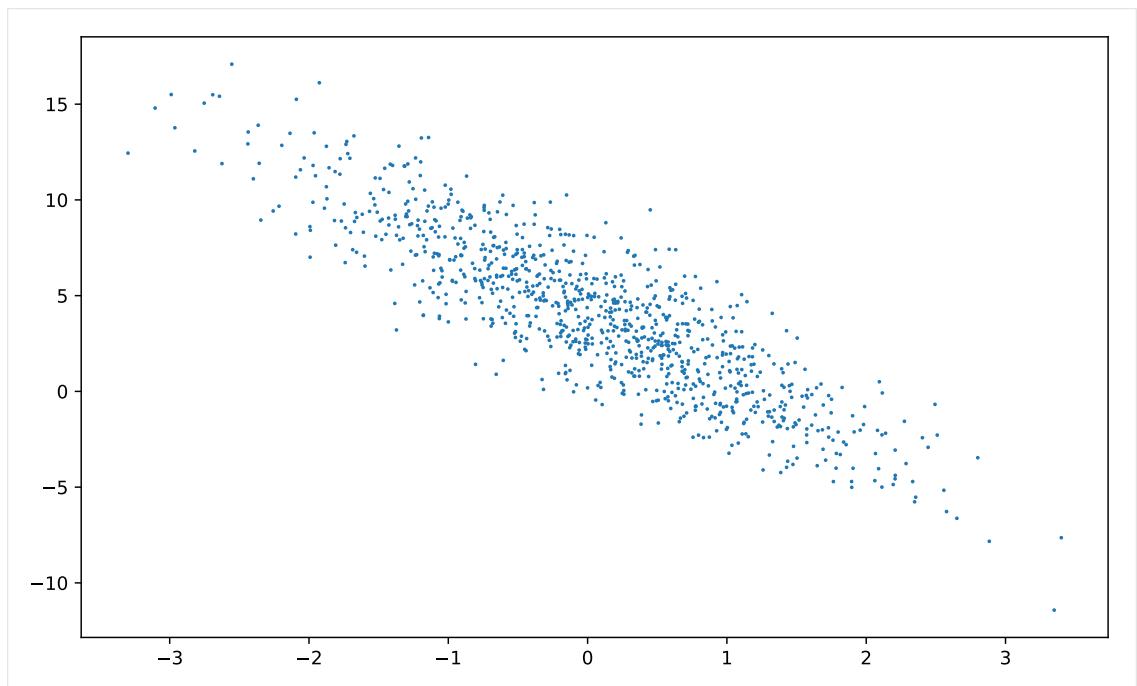
```
[3]: (
 [2.2122064 0.7740038]
<NDArray 2 @cpu(0)>,
[6.000587]
<NDArray 1 @cpu(0)>)
```

features[:, 1] 과 labels 를 이용해서 scatter plot을 생성해보면, 둘 사이의 선형 관계를 명확하게 관찰할 수 있습니다.

```
[4]: def use_svg_display():
    # Display in vector graphics
    display.set_matplotlib_formats('svg')

def set_figsize(figsize=(3.5, 2.5)):
    use_svg_display()
    # Set the size of the graph to be plotted
    plt.rcParams['figure.figsize'] = figsize

set_figsize()
plt.figure(figsize=(10, 6))
plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```



plot을 그려주는 함수 plt, use\_svg\_display 함수와 set\_figsize 함수는 g21 패키지에 정의되어 있습니다. 앞으로는 plot을 그리기 위해서 g21=plt 를 직접 호출하겠습니다. plt 은 g21 패키지의 전역 변수로 정의되어 있기 때문에, 벡터 다이어그램과 크기를 정하기 위해서는 plot을 그리기 전에 g21.set\_figsize() 를 호출하면 됩니다.

### 5.2.2 데이터 읽기

모델을 학습시킬 때, 전체 데이터셋을 반복적으로 사용하면서 각 데이터의 미니 배치를 얻어야 합니다. 이를 위해서 함수를 하나 정의하겠습니다. 이 함수는 임의로 선택된 특성(feature)들과 태그(tag)들을 배치 크기(batch size)의 개수만큼 리턴해주는 역할을 합니다. 왜 한번에 하나의 샘플을 사용하지 않고 여러 샘플을 리턴하는 iterator를 작성할까요? 이유는 최적화를 효율적으로 하기 위함입니다. 한번에 하나의 1-차원 값을 처리했을 때 성능이 아주 느렸던 것을 기억해볼까요. 하나의 샘플을 처리하는 것처럼, 하나의 벡터가 아닌 행렬로 표현된 샘플들의 전체 배치를 한번에 처리하는 것도 동일하게 할 수 있습니다. 특히, GPU는 행렬을 다룰 때 아주 빠른 속도로 연산을 수행합니다. 이것이 딥러닝에서 보통 하나의 샘플 보다는 미니 배치 단위로 연산을 하는 이유 중에 하나입니다.

```
[5]: # This function has been saved in the d2l package for future use
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):
        j = nd.array(indices[i: min(i + batch_size, num_examples)])
        yield features.take(j), labels.take(j)
        # The "take" function will then return the corresponding element based
        # on the indices
```

첫번째 작은 배치를 읽어서 출력해보겠습니다. 각 배치의 특성(feature)들의 모양(shape)은 배치 크기와 입력 차원의 수와 연관됩니다. 마찬가지로, 배치 크기와 동일한 레이블(label)들을 얻습니다.

```
[6]: batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, y)
    break

[[[-0.32821104  2.5574841]
 [-1.5036768   1.4741625]
 [-0.41308445  0.6788356]
 [ 0.29109526 -0.95134956]
 [ 0.4241809   1.691645]
 [ 1.0132014   0.9695031]
 [-0.62550926  0.00357655]
 [ 0.32212737 -1.5355504]
 [-0.6614866   0.09907386]
 [ 0.35622832  2.113086]]
 <NDArray 10x2 @cpu(0)>
[-5.162363  -3.8148558  1.0485888  8.011182  -0.7100331  2.954539
 2.9411407  10.066744   2.5347762 -2.2713506]
<NDArray 10 @cpu(0)>
```

당연하겠지만, iterator를 다시 수행하면, 전체 데이터를 모두 소진할 때까지 다른 미니 배치를 반환합니다. 위에서 구현한 iterator는 다소 비효율적입니다 (모든 데이터를 메모리에 로딩한 후, 메모리를 접근하는 것을 반복하기 때문입니다.) 패키지에서 제공하는 iterator는 더 효율적으로 구현되어 있고, 파일에 저장된 데이터를 접근하거나 데이터 스트림을 통한 접근도 가능합니다.

### 5.2.3 모델 파라미터들 초기화하기

가중치는 평균값이 0이고 표준편차가 0.01인 정규분포를 따르는 난수값들로 초기화합니다. 편향(bias)  $b$ 는 0으로 설정합니다.

```
[7]: w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
b = nd.zeros(shape=(1,))
```

이후, 모델이 데이터를 잘 예측할 수 있도록 이 파라미터들을 업데이트할 것입니다. 이를 위해서 손실 함수(loss function)의 파라미터에 대한 그래디언트(gradient), 즉 다변수 미분을 구해야 합니다. 손실(loss) 값을 줄이는 방향으로 각 파라미터를 업데이트 할 것입니다. autograd 가 적당한 데이터 구조를 준비하고, 변경을 추적할 수 있도록, 그래디언트(gradient)들을 명시적으로 붙여줘야 합니다.

```
[8]: w.attach_grad()
b.attach_grad()
```

### 5.2.4 모델 정의하기

다음으로는 모델을 정의합니다. 아주 간단하고 유용한 뉴럴 네트워크인 선형 모델을 정의하겠습니다. 선형 모델의 결과를 계산하기 위해서, 입력 값과 모델의 가중치  $w$  를 곱하고 오프셋(offset)  $b$  를 더합니다.

```
[9]: # This function has been saved in the d2l package for future use
def linreg(X, w, b):
    return nd.dot(X, w) + b
```

### 5.2.5 손실 함수(loss function) 정의하기

이전 절에서 선형 회귀 손실(loss)를 정의하는데 사용한 제곱 손실 함수(squared loss function)를 사용하겠습니다. 이를 구현하기 위해서 우선 실제 값  $y$  의 모양(shape)을 예측 값  $y_{\text{hat}}$ 의 모양(shape)과 동일하게 변형합니다. 다음 함수의 리턴 값은  $y_{\text{hat}}$  의 모양(shape)과 동일하게 바꿉니다.

```
[10]: # This function has been saved in the d2l package for future use
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

## 5.2.6 선형 회귀 알고리즘 정의하기

선형 회귀 문제는 잘 정의된 솔루션이 있습니다. 하지만, 우리가 다루게 될 많은 재미있는 모델은 분석적인 방법으로 풀릴 수 없습니다. 그렇기 때문에, 이 문제를 확률적 경사 하강법(stochastic gradient descent) sgd 를 이용해서 풀어볼 것입니다. 각 단계에서 데이터셋 중 임의로 선택한 배치를 이용해서, 가중치(weight)들에 대한 손실(loss)의 그래디언트(gradients)를 추정합니다. 그리고, 손실(loss)을 줄이는 방향으로 파라미터들을 조금씩 업데이트합니다. 여기서, 자동 미분 모듈로 계산된 그래디언트(gradients)는 샘플들의 배치의 그래디언트(gradients) 합입니다. 평균을 구하기 위해서, 이 값을 배치 크기로 나누고, 학습 속도(learning rate) lr 로 정의된 값에 비례해서 업데이트를 하게 됩니다.

```
[11]: # This function has been saved in the d2l package for future use
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

## 5.2.7 학습

학습은 데이터를 반복해서 사용하면서 모델의 파라미터를 최적화시키는 것입니다. 각 반복에서는 현재 얻어진 미니 배치의 데이터 샘플들(피처  $X$  와 레이블  $y$ )을 이용해서 역함수인 backward 함수를 호출해서 미니 배치에 대한 확률적 경사(stochastic gradient)를 계산합니다. 이후, 최적화 알고리즘 sgd 을 호출해서 모델 파라미터들을 업데이트하게 됩니다. 앞의 소스 코드에서 배치 크기 batch\_size 를 10으로 설정했으나, 각 미니 배치별로 손실(loss) 1 의 모양(shape)은 (10,1)이 됩니다.

- 파라미터  $(\mathbf{w}, b)$  를 초기화합니다.
- 끝날 때까지 다음을 반복합니다.
  - 그래디언트(gradients) 계산하기  $\mathbf{g} \leftarrow \partial_{(\mathbf{w}, b)} \frac{1}{B} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$
  - 파라미터 업데이트하기  $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

그래디언트(gradients)를 계산하는 코드를 직접 작성해야 한다면, 지루하기도 하고 코드에 오류가 있을 수 있기 때문에,  $g$  를 계산해주는 자동 미분(automatic differentiation)을 이용합니다. 이에 대한 자세한 내용은 “Automatic Gradient” 절을 참조하세요. loss 1 은 스칼라 변수가 아니기 때문에, 1. backward() 를 수행하면 1 의 모든 항목들을 합해서 새로운 변수를 만들고, 이를 이용해서 다양한 모델 파라미터의 gradient를 계산합니다.

매 epoch 마다, data\_iter 함수를 이용해서 학습 데이터셋의 전체 샘플을 한 번씩 학습에 사용합니다. 이 때, 전체 샘플의 개수는 배치 크기의 배수라고 가정합니다. 하이퍼파라미터(hyper-parameter) 인 총 epoch 수 num\_epochs 와 학습 속도(learning rate) lr 는 각각 3, 0.03으로 설정합니다. 실제

상황에서는 하이퍼파라미터(hyper-parameter)들은 반복된 실험을 통해서 가장 좋은 값을 직접 찾아야합니다. 예를 들면, 어떤 모델은 학습을 오래하면 정확도가 높아지기도 합니다. 물론 이 때, 연산 비용이 증가하게 됩니다. 마찬가지로, 학습을 수행하는 중간에 학습 속도(learning rate)를 변경하고 싶은 경우도 있습니다. 이에 대한 내용은 “최적화 알고리즘, Optimization Algorithms”에서 자세히 살펴보겠습니다.

```
[12]: 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 data set are used once in one epoch
    # iteration. The features and tags of mini-batch 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 0.040728
epoch 2, loss 0.000153
epoch 3, loss 0.000050
```

학습된 모델을 평가하는 방법으로 실제 파라미터와 학습을 통해서 찾아낸 파라미터를 비교해보면, 이것들이 매우 비슷해졌습니다.

```
[13]: print('Error in estimating w', true_w - w.reshape(true_w.shape))
print('Error in estimating b', true_b - b)

Error in estimating w
[ 0.00045753 -0.00028944]
<NDArray 2 @cpu(0)>
Error in estimating b
[0.00035954]
<NDArray 1 @cpu(0)>
```

모델의 파라미터를 아주 정확하게 찾아내는 것이 간단한 일은 아닙니다. 아주 특별한 분류의 문제에서만 간단히 찾는 것이 가능합니다. 예를 들면, 노이즈가 있는 샘플들을 이용해도 숨겨진 상관관

계를 정확하게 찾아낼 수 있을 정도로 많은 양의 데이터가 주어진 강한 볼록 최적화(strongly convex optimization) 문제가 그런 경우입니다. 대부분의 경우, 문제는 이런 분류가 아닙니다. 실제로는 학습 데이터가 사용되는 순서를 포함해서 모든 경우가 동일하지 않은 경우가 아니면 딥 네트워크의 파라미터들이 비슷하거나 같은 값을 갖습니다. 그럼에도 불구하고, 뉴럴 네트워크는 좋은 모델을 만들어내는데, 이는 예측을 잘하는 여러 파라미터 집합들이 존재하기 때문입니다.

### 5.2.8 요약

이 절에서 NDArray와 autograd만을 사용해서 레이어 정의나 멋진 옵티마이저(optimizer)를 위한 별도의 도구 없이도 딥 네트워크 구현 및 최적화를 어떻게 할 수 있는지 살펴봤습니다. 이 예제는 어떤 것들이 가능한지에 대한 기본만 다뤘고, 다음 절에서는 지금까지 배운 것들을 기반으로 다양한 딥러닝 모델들을 살펴보고, 보다 간결하게 구현하는 방법에 대해서 살펴보겠습니다.

### 5.2.9 문제

1. 가중치(weight)들을 0으로 ( $w = 0$ ) 초기화를 하면 어떤 일이 발생할까요? 알고리즘이 여전히 동작할까요?
2. 여러분이 전압과 전류간의 모델을 만들고자 하는 Georg Simon Ohm이라고 가정해 보세요. 여러분의 모델 파라미터를 학습시키기 위해서 autograd를 사용할 수 있을까요?
3. 스펙트럼 에너지 밀도를 사용해서 물체의 온도를 결정하는 데 Planck's Law를 사용할 수 있나요?
4. autograd를 이차 미분으로 확장한다면 어떤 문제를 만날 수 있을까요?
5. squared\_loss 함수에서 reshape 함수가 왜 필요한가요?
6. 다양한 학습 속도(learning rate)들을 사용해서 실험하고, 그 결과 손실 함수(loss function) 값이 얼마나 빠르게 감소하는지 알아보세요.
7. 예제들의 개수가 배치 크기로 나누어 떨어지지 않을 경우에, data\_iter 함수는 어떻게 동작할까요?

## 5.2.10 Scan the QR Code to Discuss



## 5.3 선형 회귀의 간결한 구현

딥러닝 프레임워크들의 개발로 인해서, 딥러닝 어플리케이션을 개발하는 것이 나날이 쉬워지고 있습니다. 이번에는 앞 절에서 구현한 모델을 보다 간결하게 구현하고 학습시키는 방법을 MXNet에서 제공하는 Gluon 인터페이스를 이용해서 구현해보도록 하겠습니다.

### 5.3.1 데이터셋 만들기

이전 절에서 사용한 것처럼 동일한 데이터셋을 생성합니다.

```
[1]: from mxnet import autograd, nd

num_inputs = 2
num_examples = 1000
true_w = nd.array([2, -3.4])
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = nd.dot(features, true_w) + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

### 5.3.2 데이터 읽기

Gluon은 데이터를 읽는데 사용할 수 있는 `data` 모듈을 제공합니다. `data`라는 이름은 변수 이름으로 많이 사용하기 때문에, `gdata`라고 별명을 붙여서 사용하겠습니다. 매 반복(iteration)마다, 10개 데이터 인스턴스를 갖는 미니 배치를 읽어보겠습니다.

```
[2]: from mxnet.gluon import data as gdata

batch_size = 10
```

(continues on next page)

```
# Combine the features and labels of the training data
dataset = gdata.ArrayDataset(features, labels)
# Randomly reading mini-batches
data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
```

`data_iter` 는 이전 절에서 사용한 방식과 같습니다. 이를 이용해서 첫번째 미니 배치를 읽어서 내용을 출력합니다.

```
[3]: for X, y in data_iter:
    print(X, y)
    break
```

```
[[ 0.8007321 -0.46828032]
 [-0.33806685  0.9888303 ]
 [-1.5036768   1.4741625 ]
 [-1.0249715   1.47838   ]
 [ 1.0380163   0.8255258 ]
 [ 2.3101764   0.24071898]
 [ 0.83400416 -2.3646805 ]
 [-1.3119828   1.1535704 ]
 [-0.10079116  0.48418596]
 [-1.436299   -1.2416507 ]]
<NDArray 10x2 @cpu(0)>
[ 7.403495   0.16907343 -3.8148558  -2.8686962   3.4770956   8.0141945
 13.898395  -2.3660097   2.3647711   5.559958   ]
<NDArray 10 @cpu(0)>
```

### 5.3.3 모델 정의하기

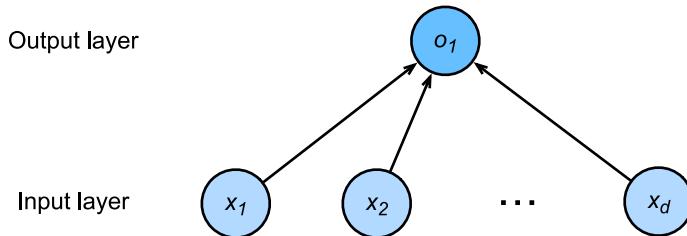
선형 회귀 모델을 직접 구현했을 때는 모델 파라미터를 정의하고, 모델을 수행하는 매 단계를 직접 작성했는데, 더 복잡한 모델을 이렇게 만들어야 한다면 구현이 더 복잡해질 것입니다. Gluon은 이미 정의된 다양한 레이어를 제공하고 있어서, 레이어를 어떻게 구현하는지가 아니라, 레이어를 사용한 모델을 설계하는데 집중할 수 있도록 해줍니다.

선형 모델을 구현하는 방법은, 우선 `nn` 모듈을 `import`하는 것으로 시작합니다. `nn`은 `neural network`의 약자입니다. 이름이 의미하듯이 이 모듈은 많은 종류의 뉴럴 네트워크층들을 바로 사용할 수 있도록 제공합니다. `Sequential` 인스턴스인 `net` 모델 변수를 정의합니다. Gluon에서 `Sequential` 인스

텐스는 다양한 레이어를 순차적으로 담는 컨테이너로 사용됩니다. 모델을 정의할 때, 이 컨테이너에 층을 원하는 순서대로 추가합니다. 입력값이 주어지면, 컨테이너의 각 층 순서대로 계산이 이뤄지고, 각 층의 결과값은 다음 층의 입력으로 사용됩니다.

```
[4]: from mxnet.gluon import nn  
net = nn.Sequential()
```

단일층 네트워크를 다시 생각해봅시다. 층은 모든 입력들과 모든 출력들을 연결하는 완전 연결(fully connected)로 구성되어 있고, 이는 행렬-벡터 곱으로 표현했습니다. Gluon의 Dense 인스턴스는 완전 연결층(fully connected layer)을 의미합니다. 한 개의 스칼라 출력을 생성해야하기 때문에 갯수를 1로 정의합니다.



```
[5]: net.add(nn.Dense(1))
```

Gluon의 특별한 점은, 각 레이어의 입력 shape를 별도로 지정할 필요가 없다는 것입니다. 예를 들면, 선형 회귀의 입력 개수 같은 것. 예를 들면 net(X) 가 수행되면, 모델이 데이터를 보면서 각 레이어의 입력 개수를 자동으로 계산해줍니다. 이 원리에 대한 내용은 “딥러닝 계산” 장에서 자세히 다루겠습니다. Gluon의 이런 디자인은 모델 개발을 더 쉽게 만들어줍니다.

### 5.3.4 모델 파라미터들 초기화하기

net 을 사용하기전에 선형 회귀 모델(linear regression model)의 가중치와 편향(bias) 같은 모델 파라미터들을 초기화를 해야합니다. 이를 위해서 MXNet의 initializer 모듈을 import 합니다. 이 모듈을 이용하면 다양한 방법으로 모델 파라미터를 초기화할 수 있습니다. import 한 init 은 initializer 의 간략한 이름입니다. init.Normal(sigma=0.01) 을 통해서 평균이 0이고 표준편차가 0.01인 정규 분포를 따르는 난수값들로 각 가중치 파라미터들을 초기화합니다. 편향(bias)은 0으로 기본 설정 되게 됩니다.

```
[6]: from mxnet import init  
net.initialize(init.Normal(sigma=0.01))
```

위 코드는 매우 직관적으로 보이지만, 실제로는 상당히 이상한 일이 일어납니다. 이 단계에서 우리는 아직 Gluon에게 입력이 어떤 차원을 갖는지를 알려주지 않은 상태에서 파라미터를 초기화하고 있습니다. 네트워크를 어떻게 정의하는지에 따라서 따라서 2가 될 수도, 2,000이 될 수도 있기 때문에, 이 시점에서 메모리를 미리 할당할 수도 없습니다. 파라미터 할당 및 초기화는 최초의 데이터가 네트워크에 입력되는 시점으로 미뤄지는 방식으로 동작합니다. 이런 방식으로 모든 설정을 미리 준비한 후, 사용자는 설정들에 대해서 걱정하지 않아도 됩니다. 단지 유의해야 할 점은 파라미터들이 아직 초기화가 되지 않았기 때문에 값을 바꾸는 일을 못한다는 것입니다.

### 5.3.5 손실 함수(loss function) 정의하기

Gluon의 loss 모듈은 다양한 손실 함수(loss function)를 정의하고 있습니다. loss 모듈을 import 할 때 이름을 gloss 바꾸고, 제곱 손실(squared loss)을 모델의 손실 함수(loss function)로 사용하도록 합니다.

```
[7]: from mxnet.gluon import loss as gloss
loss = gloss.L2Loss() # The squared loss is also known as the L2 norm loss
```

### 5.3.6 최적화 알고리즘 정의하기

마찬가지로 미니 배치에 대한 확률적 경사 하강법(stochastic gradient descent)을 직접 구현할 필요가 없습니다. Gluon을 import 한 후, Trainer 인스턴스를 만들면서, 학습 속도(learning rate)가 0.03을 갖는 미니 배치 확률적 경사 하강법(stochastic gradient descent)을 최적화(optimization) 알고리즘으로 설정하면 됩니다. 이 최적화 알고리즘은 add 함수를 이용해서 net 인스턴스에 추가된 레이어들의 모든 파라미터들에 적용할 것인데, 이 파라미터들은 collect\_params 함수를 통해서 얻습니다.

```
[8]: from mxnet import gluon
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

### 5.3.7 학습

Gluon을 이용해서 모델을 표현하는 것이 보다 간결하다는 것을 눈치챘을 것입니다. 예를 들면, 파라미터들을 일일이 선언하지 않았고, 손실 함수(loss function)를 직접 정의하지 않았고, 확률적 경사 하강법(stochastic gradient descent)을 직접 구현할 필요가 없습니다. 더욱 복잡한 형태의 모델을 다루기 시작하면, Gluon의 추상화를 사용하는 것이 얼마나 많은 이득이 되는지 알게 될 것입니다. 기본적인 것들이 모두 설정된 후에는 학습 loop은 앞에서 직접 구현한 것과 비슷합니다.

앞서 구현했던 모델 학습 단계를 다시 짚어보면, 전체 학습 데이터(train\_data)를 정해진 에포크(epoch) 만큼 반복해서 학습을 수행합니다. 하나의 에포크(epoch)는 다시 미니 배치로 나눠지는데 미니 배치는 입력과 출력에 해당하는 진실 값(ground-truth label)들로 구성됩니다. 각 미니 배치에서는 다음 단계들이 수행됩니다.

- 네트워크를 따라 순전파(forward pass)을 수행해서 예측 값net (x) 을 생성하고, 손실(loss) l 을 계산합니다.
- l.backward() 수행으로 역전파(backward pass)를 실행해서 그래디언트(gradients)들을 계산합니다.
- SGD 옵티마이저(optimizer)를 수행해서 모델 파라미터들을 업데이트합니다. (trainer.step() 을 보면, 어떤 파라미터를 업데이트 해야하는지를 명시하고 있지 않고 학습할 데이터만 전달하고 있습니다. 이유는, trainer를 초기화할 때 이미 학습할 파라미터를 명시했기 때문입니다.)

학습 상태를 관찰하기 위해서, 매 에포크(epoch) 마다 전체 특성(feature)들에 대해서 손실(loss()) 값을 계산해서 출력합니다.

```
[9]: 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: 0.040390  
epoch 2, loss: 0.000157  
epoch 3, loss: 0.000051

학습된 모델 파라미터들과 실제 모델 파라미터를 비교해봅니다. 모델의 파라미터들의 값을 확인하는 방법은, 우선 net 인스턴스로부터 총을 얻어내고, 그 총의 가중치(weight) 변수와 편향(bias) 변수를 접근하는 것입니다. 학습된 파라미터들과 실제 파라미터들이 매우 비슷한 것을 볼 수 있습니다.

```
[10]: 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
[[ 0.00035167 -0.00055599]]
<NDArray 1x2 @cpu(0)>
Error in estimating b
[0.00045252]
<NDArray 1 @cpu(0)>
```

### 5.3.8 요약

- Gluon을 이용해서 모델을 매우 간단하게 구현할 수 있습니다.
- Gluon에서 data 모듈은 데이터 프로세싱하는 도구를, nn 모듈은 많은 종류의 뉴럴 네트워크 레이어들의 정의를 그리고 loss 모듈은 다양한 손실 함수(loss function)를 제공합니다.
- MXNet의 initializer 모듈은 모델 파라미터를 초기화하는 다양한 방법을 제공합니다.
- 모델 파라미터의 차원(dimensionality)과 저장 공간은 할당은 실제 사용될 때까지 미뤄집니다.  
(따라서, 초기화되기 전에 파라미터를 접근하는 경우에 주의해야 합니다.)

### 5.3.9 문제

- 1 = loss(output, y) 를 1 = loss(output, y).mean() 로 바꿀 경우, trainer.step(batch\_size) 를 trainer.step(1) 바꿔야합니다. 왜 일까요?
- gluon.loss 와 init 모듈에 어떤 손실 함수(loss function)와 초기화 방법들이 포함되어 있는지 MXNet 문서를 읽어보세요. 손실 함수(loss function)를 Huber's loss로도 바꿔보세요.
- dense.weight 의 그래디언트(gradient)를 어떻게 접근할 수 있을까요?

### 5.3.10 Scan the QR Code to Discuss



## 5.4 Softmax 회귀(regression)

앞 두 절, *from scratch* 와 *using Gluon*을 통해서 선형 회귀 모델을 직접 구현해보기도 했고, Gluon을 이용해서 구현해보았습니다. Gluon을 이용하면 파라미터 정의나 초기화, 손실 함수(loss function) 정의, optimizer 구현과 같은 반복된 일을 자동화할 수 있었습니다.

회귀(regression)는 몇 개인지, 얼마인지 등에 대한 답을 구할 때 사용하는 도구로, 예를 들면 집 가격이 얼마인지, 어떤 야구팀이 몇 번 승리를 할 것인지 등을 예측하는데 사용할 수 있는 방법입니다. 다른 예로는, 환자가 몇 일 만에 퇴원할 것인지 예측하는 것도 회귀(regression) 문제입니다.

하지만, 현실에서는 어떤 카테고리에 해당하는지를 예측하는 문제를 더 많이 접하게 됩니다.

- 메일이 스팸인지 아닌지
- 고객이 구독 서비스에 가입할지 아닐지
- 이미지에 있는 객체가 무엇인지 (원숭이, 강아지, 고양이, 닭 등)
- 고객이 어떤 물건을 구매할 것인지

카테고리별로 값을 할당하거나, 어떤 카테고리에 속할 확률이 얼마나 되는지를 예측하는 것은 분류(classification)라고 부릅니다. 앞 절들에서 살펴본 모델은 확률을 예측하는 문제에 적용하기 어렵습니다.

### 5.4.1 분류 문제들

입력 이미지의 높이와 넓이가 2 픽셀이고, 색은 회색인 이미지를 입력으로 다루는 간단한 문제부터 시작해보겠습니다. 이미지의 4개 픽셀의 값은  $x_1, x_2, x_3, x_4$  으로 표현하고, 각 이미지의 실제 레이블(label)는 “고양이”, “닭”, “강아지” 중에 하나로 정의되어 있다고 하겠습니다. (4 픽셀로 구성된 이미지가 3개 동물 중에 어떤 것인지를 구별할 수 있다고 가정합니다.)

이 레이블(label)들을 표현하는데 두 가지 방법이 있습니다. 첫번째 방법은 {강아지, 고양이, 닭}을 각  $y \in \{1, 2, 3\}$  으로 정의합니다. 이 방법은 컴퓨터에 정보를 저장하는 좋은 방법이지만, 이 방법은 회귀 문제에 적합합니다. 더구나 이 숫자들의 순서가 분류의 문제에서는 의미가 없습니다. 우리의 간단한 예제에서는 적어도 수학적으로는 고양이가 강아지보다는 닭과 더 비슷하다는 것을 의미할 수도 있게 됩니다. 하지만, 실제 문제들에서 이런 비교가 잘되지 않습니다. 그렇기 때문에, 통계학자들은 원-핫-인코딩(one hot encoding) 을 통해서 표현하는 방법을 만들었습니다.

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$$

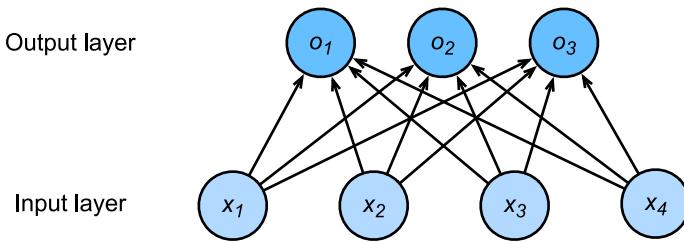
즉,  $y$  는 3차원 벡터로  $(1,0,0)$ 은 고양이를,  $(0,1,0)$ 은 닭은,  $(0,0,1)$ 은 강아지를 의미합니다.

## 5.4.2 네트워크 아키텍처

여러 클래스들에 대한 분류를 예측할 때는 카테고리 개수와 같은 수의 출력들이 필요합니다. 이점이 회귀 문제와 가장 다른 점입니다. 4개 특성(feature)들과 3개의 동물 카테고리 출력(output)들이 있으니, 가중치( $w$ )는 12개의 스칼라들로 구성되고 편향(bias) ( $b$ )는 3개의 스칼라로 정의됩니다. 각 입력에 대해서 3개의 출력 ( $o_1, o_2, o_3$ )는 다음과 같이 계산됩니다.

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1, \\ o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2, \\ o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3. \end{aligned}$$

아래 뉴럴 네트워크 다이어그램은 위 연산을 표현하고 있습니다. 선형 회귀처럼, softmax 회귀는 단 일층의 뉴럴 네트워크로 구성됩니다. 출력 ( $o_1, o_2, o_3$ ) 는 모든 입력 ( $x_1, x_2, x_3, x_4$ ) 값들과 연관되서 계산되기 때문에, softmax 회귀의 출력층은 완전 연결층입니다.



## 5.4.3 Softmax 연산

위 표기법은 다소 장황해 보입니다. 이를 벡터 표현으로 하면  $\mathbf{o} = \mathbf{Wx} + \mathbf{b}$  와 같이 쓰기도 간단하고 코딩하기도 간단해집니다. 하지만, 분류 문제는 이산(discrete) 예측 결과가 필요하기 때문에,  $i$  번째 카테고리에 대한 확신 수준(confidence level)을 표현하기 위해서 출력을  $o_i$ 로 표현하는 간단한 방법을 사용합니다. 이렇게 구성하면, 어떤 카테고리에 속하는지를 결과 값들 중에 가장 큰 값의 클래스로 선택하면 되고,  $\text{argmax}_i o_i$ 로 간단히 계산할 수 있습니다. 예를 들면, 결과  $o_1, o_2, o_3$  가 각각 0.1, 10, 0.1 이라면, 예측된 카테고리는 2, 즉 “닭”이 됩니다.

하지만, 출력층의 값을 직접 사용하기에는 두 가지 문제가 있습니다. 첫번째는 출력값의 범위가 불확실해서, 시각적으로 이 값들의 의미를 판단하기 어렵다는 것입니다. 예를 들어, 이전 예에서 결과 10 은 주어진 이미지가 “닭” 카테고리에 속할 것이라고 “매우 확신”한다는 것을 의미합니다. 왜냐하면,

다른 두 카테고리들의 값보다 100배 크기 때문입니다. 만약에  $o_1 = o_3 = 10^3$  이라면, 10이라는 출력값은 이미지가 “닭” 카테고리에 속할 가능성이 매우 낮다는 것의 의미하게 됩니다. 두번째 문제는 실제 레이블(label)은 이산(discrete) 값을 갖기 때문에, 불특정 범위를 갖는 출력값과 레이블 값의 오류를 측정하는 것이 매우 어렵다는 것입니다.

출력값들이 확률값으로 나오도록 해볼 수 있겠지만, 새로운 데이터가 주어졌을 때 확률값이 0 또는 양수이고, 전체 합이 1이 된다는 것을 보장할 수는 없습니다. 이런 이산 값(discrete value) 예측 문제를 다루기 위해서 통계학자들은 softmax 로지스틱 회귀(logistic regression)이라는 분류 모델을 만들었습니다. 선형 회귀(linear regression)과는 다르게, softmax 회귀(regression)의 결과는 모든 결과값들의 합이 1이 되도록 하는 비선형성에 영향을 받고, 각 결과 값은 0 또는 양수값을 갖습니다. 비선형 변환은 다음 공식으로 이뤄집니다.

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \text{ where } \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

모든  $i$ 에 대해서  $0 \leq \hat{y}_i \leq 1$ 이고  $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ 를 만족하는 것을 쉽게 확인할 수 있습니다. 따라서,  $\hat{y}$ 은 적절한 확률 분포이고,  $o$  값은 쉽게 측정할 수 있는 값으로 간주할 수 있습니다. 아래 공식은 가장 가능성 있는 클래스를 찾아줍니다.

$$\hat{i}(\mathbf{o}) = \underset{i}{\operatorname{argmax}} o_i = \underset{i}{\operatorname{argmax}} \hat{y}_i$$

즉, softmax 연산은 예측하는 카테고리의 결과를 바꾸지 않으면서, 결과  $o$ 에 대한 적절한 의미를 부여해줍니다. 이것을 벡터 표현법으로 요약해보면, get  $\mathbf{o}^{(i)} = \mathbf{Wx}^{(i)} + \mathbf{b}$ ,  $\hat{\mathbf{y}}^{(i)} = \text{softmax}(\mathbf{o}^{(i)})$  이 됩니다.

#### 5.4.4 미니 배치를 위한 벡터화

연산 효율을 더 높이기 위해서, 데이터의 미니 배치에 대한 연산을 벡터화합니다. 차원이  $d$ 이고 배치 크기가  $n$ 인 데이터들의 미니 배치  $\mathbf{X}$ 가 있고, 결과로  $q$ 개의 카테고리가 있다고 가정하겠습니다. 그러면, 미니 배치 feature  $\mathbf{X}$ 는  $\mathbb{R}^{n \times d}$ 에 속하고, 가중치들  $\mathbf{W}$ 는  $\mathbb{R}^{d \times q}$ 에, 편향(bias)  $\mathbf{b}$ 는  $\mathbb{R}^q$ 에 속합니다.

$$\begin{aligned}\mathbf{O} &= \mathbf{WX} + \mathbf{b} \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O})\end{aligned}$$

이렇게 정의하면 가장 많이 차지하는 연산을 가속화할 수 있습니다. 즉,  $\mathbf{WX}$ 이 행렬-벡터의 곱에서 행렬-행렬의 곱으로 변환됩니다. softmax는 결과  $\mathbf{O}$ 의 모든 항목에 지수 함수를 적용하고, 지수

함수들의 값의 합으로 정규화(normalize) 하는 것으로 계산됩니다.

### 5.4.5 손실 함수(loss function)

확률 결과를 출력하는 방법을 정의했으니, 이 값이 얼마나 정확한지를 측정하는 값으로 변환하는 것이 필요합니다. 즉, 손실 함수(loss function)가 필요합니다. 선형 회귀에서 사용했던 것과 동일한 개념을 사용하는데, 이는 가능도 최대화(likelihood maximization)라고 합니다.

#### 로그 가능도(Log-Likelihood)

softmax 함수는 결과  $\mathbf{o}$  를 여러 결과들에 대한 확률,  $p(y = \text{cat}|\mathbf{x})$ , 들의 벡터로 변환합니다. 이는, 예측된 값이 얼마나 잘 예측하고 있는지를 확인하는 것으로 실제 값과 예측 결과에 대한 비교를 할 수 있습니다.

$$p(Y|X) = \prod_{i=1}^n p(y^{(i)}|x^{(i)}) \text{이고, 그러므로 } -\log p(Y|X) = \sum_{i=1}^n -\log p(y^{(i)}|x^{(i)})$$

잘 예측하는 것은  $-\log p(Y|X)$  를 최소화하는 것을 의미합니다. 이를 통해서 손실 함수(loss function)를 다음과 같이 정의할 수 있습니다. (표기를 간단하게 하기 위해서  $i$  는 제외했습니다.)

$$l = -\log p(y|x) = -\sum_j y_j \log \hat{y}_j$$

여기서  $\hat{y} = \text{softmax}(\mathbf{o})$  이고, 벡터  $\mathbf{y}$  는 해당하는 레이블이 아닌 위치에는 모두 0을 갖습니다. (예를 들면 (1,0,0)). 따라서, 모든  $j$  에 대한 합을 하면, 하나의 항목만 남게 됩니다. 모든  $\hat{y}_j$  는 확률값이기 때문에, 이에 대한 로그를 적용한(logarithm) 값은 0보다 커질 수 없습니다. 그 결과, 주어진  $x$ 에 대해서  $y$ 를 잘 예측하는 경우라면 (즉,  $p(y|x) = 1$ ), 손실 함수(loss function)는 최소화될 것입니다.

### 5.4.6 Softmax와 미분(derivative)

Softmax와 이에 대한 손실(loss)는 많이 사용되기 때문에, 어떻게 계산되는지 자세히 살펴볼 필요가 있습니다.  $o$  를 손실  $l$  의 정의에 대입하고, softmax의 정의를 이용하면, 다음과 같이 표현을 얻습니다.

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j$$

어떤 일이 일어나는지 더 살펴보기 위해서, 손실 함수(loss function)를  $o$ 에 대해서 미분을 해보면 아래 공식을 유도할 수 있습니다.

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = \Pr(y = j|x) - y_j$$

다르게 설명해보면, 그래디언트(gradients)는 모델이  $p(y|x)$  확률 표현식으로 예측한 것과 실제 값  $y_j$ 의 차이입니다. 이는 회귀 문제에서 보았던 것과 아주 비슷합니다. 회귀 문제에서 그래디언트(gradients)가 관찰된 실제 값  $y$ 와 예측된 값  $\hat{y}$ 의 차이로 계산되었습니다. 이는 너무 우연으로 보이는데, 사실은 그렇지 않습니다. [exponential](#) 계열의 모델의 경우에는, 로그 가능성(log-likelihood)의 그래디언트(gradients)는 정확하게 이 항목으로 주어집니다. 이로 인해서 그래디언트(gradients)를 구하는 것이 실제 적용할 때 매우 간단해집니다.

### 크로스-엔트로피 손실(cross-entropy loss)

자 이제는 하나의 결과에 대한 관찰을 하는 경우가 아니라, 결과들에 대한 전체 분포를 다루는 경우를 생각해봅시다.  $y$ 에 대한 표기를 이전과 동일하게 사용할 수 있습니다. 오직 다른 점은  $(0,0,1)$ 과 같이 이진(binary) 값을 갖는 것이 아니라  $(0.1, 0.2, 0.7)$ 과 같이 일반적인 확률 벡터를 사용한다는 것입니다. 손실  $l$ 의 정의도 동일한 수학을 사용하지만, 이에 대한 해석은 조금 더 일반적입니다. 레이블들의 분포에 대한 손실의 기대값을 의미합니다.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_j y_j \log \hat{y}_j$$

이렇게 정의된 손실은 크로스-엔트로피 손실(cross-entropy loss)이라고 부릅니다. 이것은 다중 클래스 분류에 가장 흔히 사용되는 손실입니다. 이 이름에 대해서 알아보기 위해서는 정보 이론(information theory)에 대한 설명이 필요하며, 지금부터 설명하겠습니다. 다음 내용은 넘어가도 됩니다.

#### 5.4.7 정보 이론(Information theory) 기초

정보 이론(information theory)은 정보 (또는 데이터)를 가능한 한 간결한 형식으로 인코딩, 디코딩, 전송, 및 변조하는 문제를 다룹니다.

## 엔트로피(Entropy)

데이터(또는 난수)에 몇개의 정보 비트들이 담겨있는지가 중요한 개념입니다. 이는 분포  $p$  의 entropy로 다음과 같이 수치화할 수 있습니다.

$$H[p] = \sum_j -p(j) \log p(j)$$

정보 이론의 근본적인 이론 중에 하나로 분포  $p$ 로부터 임의로 추출된 데이터를 인코드하기 위해서는 최소  $H[p]$  개의 'nat'이 필요하다는 것이 있습니다. 여기서 'nat'은 비트와 동일하나, 베이스(base) 2가 아니라 베이스(base)  $e$ 를 이용합니다. 즉, 1 nat은  $\frac{1}{\log(2)} \approx 1.44$  비트이고,  $H[p]/2$  는 종종 이진 앤트로피(binary entropy)라고 불립니다.

조금 더 이론적으로 들어가보겠습니다.  $p(1) = \frac{1}{2}$  이고,  $p(2) = p(3) = \frac{1}{4}$ 인 분포를 가정하겠습니다. 이 경우, 이 분포에서 추출한 데이터에 대한 최적의 코드를 굉장히 쉽게 설계할 수 있습니다. 즉, 1의 인코딩은 0, 2와 3에 대한 인코딩은 각각 10, 11로 정의하면 됩니다. 예상되는 비트 개수는  $1.5 = 0.5 * 1 + 0.25 * 2 + 0.25 * 2$  이고, 이 숫자는 이진 앤트로피(binary entropy)  $H[p]/\log 2ff$  와 같다는 것을 쉽게 확인할 수 있습니다.

## Kullback Leibler Divergence

두 분포간에 차이를 측정하는 방법 중에 하나로 앤트로피(entropy)를 이용하는 방법이 있습니다.  $H[p]$ 는 분포  $p$ 를 따르는 데이터를 인코드하는데 필요한 최소 비트 수를 의미하기 때문에, 틀린 분포  $q$ 에서 뽑았을 때 얼마나 잘 인코딩이 되었는지를 물어볼 수 있습니다.  $q$ 를 인코딩하는데 추가로 필요한 비트 수는 두 분포가 얼마나 다른지에 대한 아이디어를 제공합니다. 직접 계산해보겠습니다. 분포  $q$ 에 대해 최적인 코드를 이용해서  $j$ 를 인코딩하기 위해서는  $-\log q(j)$  nat이 필요하고,  $p(j)$ 인 모든 경우에서 이를 사용하면, 다음 식을 얻습니다.

$$D(p\|q) = - \sum_j p(j) \log q(j) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)}$$

$q$ 에 대해서  $D(p\|q)$ 를 최소화하는 것은 크로스-엔트로피 손실(cross-entropy loss)을 최소화하는 것과 같습니다. 이는  $q$ 에 의존하지 않는  $H[p]$ 를 빼버리면 바로 얻을 수 있습니다. 이를 통해서 우리는 softmax 회귀(regression)는 예측된 값  $\hat{y}$ 이 아니라 실제 레이블  $y$ 를 봤을 때 얻는 놀라움(비트 수)을 최소화하려는 것임을 증명했습니다.

### 5.4.8 모델 예측 및 평가

학습된 softmax 회귀(regression) 모델을 사용하면, 새로운 특성(feature)가 주어졌을 때, 각 결과 카테고리에 속할 확률값을 예측할 수 있습니다. 일반적으로는 가장 크게 예측된 확률값을 갖는 카테고리를 결과 카테고리라고 정의합니다. 실제 카테고리(label)와 일치하는 경우에 예측이 정확하다고 합니다. 다음에는 모델의 성능을 평가하는 방법으로 accuracy 정확도를 사용할 예정입니다. 이는 정확하게 예측한 개수와 전체 예측의 개수의 비율과 같습니다.

### 5.4.9 요약

- 벡터를 확률로 변환하는 softmax 연산을 알아봤습니다.
- softmax 회귀(regression)은 분류의 문제에 적용할 수 있습니다. softmax 연산을 이용해서 얻은 결과 카테고리의 확률 분포를 이용합니다.
- 크로스 엔트로피(cross entropy)는 두 확률 분포의 차이를 측정하는 좋은 방법입니다. 이는 주어진 모델이 데이터를 인코드하는데 필요한 비트 수를 나타냅니다.

### 5.4.10 문제

1. Kullback-Leibler divergence  $D(p\|q)$  가 모든 음수가 분포  $p, q$  에 대해서 음수가 아님을 증명하세요. 힌트 - Jensen의 부등식을 이용하세요. 예를 들여,  $-\log x$  가 볼록 함수(convex function)이라는 사실을 사용하세요.
2.  $\log \sum_j \exp(o_j)$  가  $o$ 에서 볼록 함수(convex function)임을 증명하세요.
3. 지수승 집단과 softmax를 관계를 심도있게 알아볼 수 있습니다.
  - softmax에 대한 크로스 엔트로피 손실  $l(y, \hat{y})$  의 이차 미분을 계산하세요.
  - softmax( $o$ )로 주어지는 분포의 분산을 계산하고, 위에서 계산한 이차 이분과 같음을 증명하세요.
4. 동일한 확률로 일어나는 3개 클래스가 있다고 가정합니다. 즉, 확률 벡터가  $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ 입니다.
  - 이 문제를 위한 이진 코드(binary code)를 설계하고자 하면 어떤 문제가 있을까요? 앤트로피의 하한(lower bound)를 필요한 비트수로 같게 할 수 있나요?
  - 더 좋은 코드를 설계할 수 있나요? 힌트 - 두 독립적인 관찰을 인코딩하려면 어떤 일이 생기나요?  $n$ 개의 관찰을 연관해서 인코드하면 어떨까요?

5. Softmax는 위에서 소개된 매핑에 대한 잘못된 이름입니다 (하지만 딥러닝에서 많은 사람들이 쓰고 있습니다.) 실제 softmax는  $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$ 로 정의됩니다.

- $\text{RealSoftMax}(a, b) > \max(a, b)$  임을 증명하세요.
- $\lambda > 0$  일 경우, 모든  $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$ 에 대해서 이것이 성립함을 증명하세요
- $\lambda \rightarrow \infty$  이면,  $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \rightarrow \max(a, b)$  임을 증명하세요.
- soft-min은 어떻게 생겼을까요?
- 이를 두개 이상의 숫자들로 확장해보세요.

#### 5.4.11 Scan the QR Code to Discuss



## 5.5 이미지 분류 데이터 (Fashion-MNIST)

softmax 회귀(regression) 구현에 앞서 적절한 데이터셋이 필요합니다. 시각적으로 돋보이게 만들기 위해서, 분류 문제 중에서 선택해보겠습니다.

다음 장들에서 모델 정확도의 차이를 관찰하거나, 비교 알고리즘의 연산 효율성에 대한 이야기를 할 때에도 반복해서 사용할 예제입니다. 가장 흔한 이미지 분류 데이터셋은 MNIST 손글씨 숫자 인식 데이터셋이 있습니다. 이 데이터셋은 1990년 대에 Lecun, Cortes와 Burges에 의해서 제안되었습니다. 하지만, 거의 모든 모델이 MNIST 데이터셋에 대해서 95% 이상의 정확도를 보여주기 때문에, 모델들 사이의 차이를 설명하기에 적합하지 않습니다. 알고리즘들의 차이를 보다 직관적으로 보여주기 위해서, 더 복잡한 데이터셋을 사용하겠습니다. 이 데이터셋은 Fashion-MNIST라는 것으로 2017년에 Xio, Rasul 그리고 Vollgraf가 제안했습니다.

### 5.5.1 데이터 구하기

우선, 이 절에서 필요한 패키지와 모듈을 import 합니다.

```
[1]: import sys  
sys.path.insert(0, '..')  
  
%matplotlib inline  
import d2l  
from mxnet.gluon import data as gdata  
import sys  
import time
```

다음으로, Gluon의 data 패키지를 이용해서 이 데이터셋을 다운로드합니다. 데이터셋은 처음 불렸을 때, 인터넷으로부터 자동으로 다운로드됩니다. train 파라미터를 통해서 학습 데이터셋을 받을 것인지 테스트 데이터셋을 받을 것인지를 정할 수 있습니다. 테스트 데이터셋 또는 테스팅 데이터셋은 모델의 성능을 평가할 때만 쓰이고, 학습에는 사용되지 않는 데이터입니다.

```
[2]: mnist_train = gdata.vision.FashionMNIST(train=True)  
mnist_test = gdata.vision.FashionMNIST(train=False)
```

학습 데이터셋과 테스트 데이터셋은 각 카테고리별로 각각 6,000개와 1,000개의 이미지들로 구성되어 있습니다. 카테고리 개수는 10개이기에, 학습 데이터는 총 60,000개의 이미지들로 테스팅 셋은 10,000개 이미지들을 가지고 있습니다.

```
[3]: len(mnist_train), len(mnist_test)  
[3]: (60000, 10000)
```

[] 을 이용하면 각 샘플을 접근할 수 있습니다. 첫번째 데이터의 이미지와 레이블을 얻어보겠습니다.

```
[4]: feature, label = mnist_train[0]
```

feature 변수는 높이와 넓이가 모두 28 픽셀인 이미지 데이터를 가지고 있습니다. 각 픽셀은 8-bit 부호없는 정수(uint8)이고, 0부터 255 사이의 값을 갖습니다. 이는 3차원 NDArray에 저장됩니다. 마지막 차원은 채널의 개수를 의미합니다. 데이터셋이 회색 이미지이기 때문에, 채널의 수는 1이 됩니다. 간단하게 하기 위해서, 이미지의 모양이 높이 h, 넓이는 w 픽셀인 경우 이미지의 모양(shape)을  $h \times w$  또는  $(h, w)$ 로 표기하도록 하겠습니다.

```
[5]: feature.shape, feature.dtype  
[5]: ((28, 28, 1), numpy.uint8)
```

각 이미지에 대한 레이블은 NumPy의 스칼라로 저장되어 있고, 이는 32-bit 정수 형태입니다.

```
[6]: label, type(label), label.dtype  
[6]: (2, numpy.int32, dtype('int32'))
```

Fashion-MNIST에는 10개의 카테고리가 있는데, 이들은 티셔츠, 바지, 풀오버, 드레스, 코트, 샌들, 셔츠, 스니커, 가방, 발목 부츠입니다. 숫자 형태의 레이블을 텍스트 레이블로 바꿔주는 함수를 아래와 같이 정의합니다.

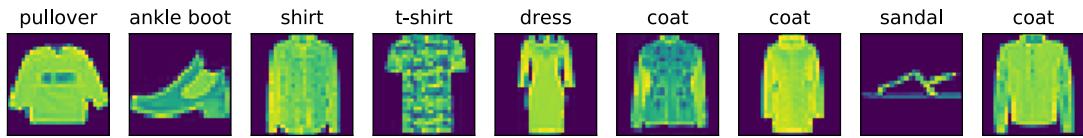
```
[7]: # This function has been saved in the d2l package for future use  
def get_fashion_mnist_labels(labels):  
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',  
                  'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']  
    return [text_labels[int(i)] for i in labels]
```

아래 함수는 한 줄에 여러 이미지와 그 이미지의 레이블을 그리는 것을 정의합니다.

```
[8]: # This function has been saved in the d2l package for future use  
def show_fashion_mnist(images, labels):  
    d2l.use_svg_display()  
    # Here _ means that we ignore (not use) variables  
    _, figs = d2l.plt.subplots(1, len(images), figsize=(12, 12))  
    for f, img, lbl in zip(figs, images, labels):  
        f.imshow(img.reshape((28, 28)).asnumpy())  
        f.set_title(lbl)  
        f.axes.get_xaxis().set_visible(False)  
        f.axes.get_yaxis().set_visible(False)
```

학습 데이터셋의 처음 9개의 샘플들에 대한 이미지와 텍스트 레이블을 살펴보겠습니다.

```
[9]: X, y = mnist_train[0:9]  
show_fashion_mnist(X, get_fashion_mnist_labels(y))
```



## 5.5.2 미니 배치 읽기

학습 데이터나 테스트 데이터를 읽는 코드를 “선형 회귀를 처음부터 구현하기”에서처럼 직접 작성하지 않고 DataLoader를 사용하도록 하겠습니다. 데이터 로더는 매 번 batch\_size 개수의 샘플을

갖는 미니 배치를 읽습니다.

실제 수행을 할 때, 데이터를 읽는 것이 성능의 병목이 되는 것을 볼 수 있습니다. 특히, 모델이 간단하거나 컴퓨터가 빠를 경우에 더욱 그렇습니다. `DataLoader`의 유용한 특징은 데이터 읽기 속도를 빠르게 하기 위해서 멀티 프로세스들 사용할 수 있다는 것입니다. (단, 현재 Windows에서는 지원되지 않습니다) 예를 들면, `num_workers` 설정을 통해서 4개의 프로세스가 데이터를 읽도록 만들 수 있습니다.

추가적으로 `ToTensor` 클래스를 이용해서 이미지 데이터를 `uint8`에서 `32bit` 부동 소수점 숫자로 변환합니다. 이후, 모든 숫자를 255로 나눠서 모든 픽셀의 값이 0과 1사이가 되도록 합니다. `ToTensor` 클래스는 이미지 채널을 마지막 차원에서 첫번째 차원으로 바꿔주는 기능이 있는데, 이는 다음에 소개할 컨볼루션 뉴럴 네트워크(convolutional neural network) 계산과 관련이 있습니다. 데이터셋의 `transform_first` 함수를 이용하면, `ToTensor`의 변환을 각 데이터 샘플(이미지와 레이블)의 첫번째 원소인 이미지에 적용할 수 있습니다.

```
[10]: batch_size = 256
transformer = gdata.vision.transforms.ToTensor()
if sys.platform.startswith('win'):
    # 0 means no additional processes are needed to speed up the reading of
    # data
    num_workers = 0
else:
    num_workers = 4

train_iter = gdata.DataLoader(mnist_train.transform_first(transformer),
                             batch_size, shuffle=True,
                             num_workers=num_workers)
test_iter = gdata.DataLoader(mnist_test.transform_first(transformer),
                            batch_size, shuffle=False,
                            num_workers=num_workers)
```

Fashion-MNIST 데이터 셋을 가지고 와서 읽는 로직은 `g2l.load_data_fashion_mnist` 함수 내부에 구현되어 있습니다. 이 함수는 다음 장들에서 사용될 예정입니다. 이 함수는 `train_iter` 와 `test_iter` 두 변수를 리턴합니다. 이 책에서는 내용이 깊어짐에 따라 이 함수를 향상시켜보겠습니다. 전체 구현에 대한 자세한 내용은 “Deep Convolutional Neural Networks (AlexNet)” 절에서 설명하겠습니다.

학습 데이터를 읽는데 걸리는 시간을 측정해 보겠습니다.

```
[11]: start = time.time()
for X, y in train_iter:
```

(continues on next page)

```

continue
'%.2f sec' % (time.time() - start)
[11]: '1.27 sec'

```

### 5.5.3 요약

- Fashion-MNIST는 의류 분류 데이터 셋으로 10개의 카테고리로 분류되어 있습니다. 다음 장들에서 다양한 알고리즘의 성능을 테스트하는데 사용할 예정입니다.
- 이미지의 모양(shape)은 높이 h 픽셀, 넓이 w 픽셀을 이용해서  $h \times w$  나 ( $h, w$ )로 저장됩니다.
- 데이터 이터레이터(iterator)는 효율적인 성능을 위한 중요한 컴포넌트입니다. 가능하면 제공되는 것들을 사용하세요.

### 5.5.4 문제

1. batch\_size를 줄이면 (예를 들면 1) 읽기 성능에 영향을 미칠까요?
2. Windows 사용자가 아니라면, num\_workers을 바꾸면서 읽기 성능이 어떻게 영향을 받는지 실험해보세요.
3. mxnet.gluon.data.vision에서 어떤 데이터셋들이 제공되는지 MXNet 문서를 통해서 확인해보세요.
4. mxnet.gluon.data.vision.transforms에서 어떤 변환들이 제공되는지 MXNet 문서를 통해서 확인해보세요.

### 5.5.5 Scan the QR Code to Discuss



## 5.6 Softmax 회귀(regression)를 처음부터 구현하기

선형 회귀를 직접 구현해본 것처럼, softmax 회귀(regression)도 직접 구현해보는 것이 도움이 될 것입니다. 이후에, 같은 내용을 Gluon을 사용해서 구현하면서 비교를 해보겠습니다. 필요한 패키지와 모듈을 import 하는 것으로 시작합니다.

```
[1]: import sys  
sys.path.insert(0, '..')  
  
%matplotlib inline  
import d2l  
from mxnet import autograd, nd
```

Fashion-MNIST 데이터셋을 사용하고, 배치 크기는 256으로 하겠습니다.

```
[2]: batch_size = 256  
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

### 5.6.1 모델 파라미터 초기화하기

선형 회귀처럼 샘플들을 벡터로 표현합니다. 각 예제가  $28 \times 28$  픽셀의 이미지이기 때문에 784 차원의 벡터에 저장합니다. 그리고 10개의 카테고리가 있으니 단일 레이어를 갖는 네트워크의 output 차원은 10으로 정의합니다. 이렇게 하면, softmax 회귀(regression)의 가중치와 편향(bias) 파라미터들은 각각 크기가  $784 \times 10$ ,  $1 \times 10$  인 행렬이 됩니다.  $W$  를 가우시안 노이즈를 이용해서 초기화합니다.

```
[3]: num_inputs = 784  
num_outputs = 10  
  
W = nd.random.normal(scale=0.01, shape=(num_inputs, num_outputs))  
b = nd.zeros(num_outputs)
```

이전처럼 모델 파라미터에 그래디언트(gradient)를 붙이겠습니다.

```
[4]: W.attach_grad()  
b.attach_grad()
```

## 5.6.2 Softmax

softmax 회귀(regression)을 정의하기에 앞서, sum 과 같은 연산이 NDArray의 특정 차원에서 어떻게 동작하는지를 보도록 하겠습니다. 행렬  $x$  의 같은 열 (axis=0) 또는 같은 행 (axis=1)의 값들을 모두 더할 수 있습니다. 합을 수행한 후 결과의 차원수를 줄이지 않고 그대로 유지하는 것도 가능합니다. 이를 위해서 keepdims=True 파라미터 값을 설정하면 됩니다.

```
[5]: X = nd.array([[1, 2, 3], [4, 5, 6]])
X.sum(axis=0, keepdims=True), X.sum(axis=1, keepdims=True)

[5]: (
[[5. 7. 9.]]
<NDArray 1x3 @cpu(0)>,
[[ 6.]
[15.]]
<NDArray 2x1 @cpu(0)>)
```

자 이제 우리는 softmax 함수를 정의할 준비가 되었습니다. 우선 각 항에  $\exp$  를 적용해서 지수값을 구하고, 정규화 상수(normalization constant)를 구하기 위해서 각 행의 값들을 모두 더합니다. 각 행을 정규화 상수(normalization contatnt)로 나누고 그 결과를 리턴합니다. 코드를 보기 전에 수식을 먼저 보겠습니다.

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}$$

분모는 파티션(partition) 함수라고 불리기도 합니다. 이 이름은 파티클의 양상률에 대한 분포를 모델링하는 통계 물리에서 기원합니다. *Naive Bayes*에서 그랬던 것처럼 행렬의 항목들이 너무 크거나 작아서 생기는, 숫자가 너무 커지는 오버플로우(overflow)나 너무 작아지는 언더플로우(underflow)를 고려하지 않고 함수를 구현하겠습니다.

```
[6]: def softmax(X):
    X_exp = X.exp()
    partition = X_exp.sum(axis=1, keepdims=True)
    return X_exp / partition # The broadcast mechanism is applied here
```

보는 것처럼, 임의의 난수 입력에 대해서, 각 항목을 0 또는 양의 숫자로 변환합니다. 또한, 확률에서 요구하는 것처럼 각 행의 합은 1이 됩니다.

```
[7]: X = nd.random.normal(shape=(2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(axis=1)
```

```
[7]: (
  [[0.21324193 0.33961776 0.1239742 0.27106097 0.05210521]
   [0.11462264 0.3461234 0.19401033 0.29583326 0.04941036]]
  <NDArray 2x5 @cpu(0)>,
  [1.0000001 1.          ]
  <NDArray 2 @cpu(0)>)
```

### 5.6.3 모델

Softmax 연산을 이용해서 softmax 회귀(regression) 모델을 정의하겠습니다. `reshape` 함수를 이용해서 원본 이미지를 길이가 `num_inputs`인 벡터로 변환합니다.

```
[8]: def net(X):
    return softmax(nd.dot(X.reshape((-1, num_inputs)), W) + b)
```

### 5.6.4 손실 함수(loss function)

앞 절에서 softmax 회귀(regression)에서 사용하는 크로스-엔트로피 손실 함수(cross-entropy loss function)를 소개했습니다. 이는 모든 딥러닝에서 등장하는 손실 함수(loss function)들 중에 가장 일반적인 손실 함수(loss function)입니다. 이유는 회귀(regression) 문제보다는 분류 문제가 더 많기 때문입니다.

크로스-엔트로피(cross-entropy)의 계산은 레이블(label)의 예측된 확률값을 얻고, 이 값에 로그(logarithm) –  $\log p(y|x)$  을 적용하는 것임을 기억해두세요. Python의 `for` loop을 사용하지 않고 (비효율적임), softmax를 적용한 행렬에서 적당한 항목을 뽑아주는 `pick` 함수를 이용하겠습니다. 3개의 카테고리와 2개의 샘플의 경우 아래와 같이 구할 수 있습니다.

```
[9]: y_hat = nd.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y = nd.array([0, 2], dtype='int32')
nd.pick(y_hat, y)

[9]:
[0.1 0.5]
<NDArray 2 @cpu(0)>
```

이를 이용해서 크로스-엔트로피 손실 함수(cross-entropy loss function)를 다음과 같이 정의합니다.

```
[10]: def cross_entropy(y_hat, y):
    return -nd.pick(y_hat, y).log()
```

## 5.6.5 분류 정확도

예측된 확률 분포들  $y_{\text{hat}}$  주어졌을 때, 가장 높은 예측 확률을 갖는 것을 결과 카테고리로 사용합니다. 실제 카테고리  $y$  와 일치하는 경우, 이 예측은 '정확하다'라고 합니다. 분류 정확도는 정확한 예측들과 전체 예측 수의 비율로 정의됩니다.

정확도를 계산하기 위해서 accuracy 함수를 다음과 같이 정의합니다.  $y_{\text{hat}}.\text{argmax}(\text{axis}=1)$  는 행렬  $y_{\text{hat}}$  에서 가장 큰 원소의 인덱스를 리턴하고, 그 결과의 shape은  $y$  변수의 shape과 동일합니다. 이제 해야 할 일은 두 개가 일치하는지 확인하는 것입니다. 동등 연산자 == 는 데이터 타입에 민감하기 때문에, 두 개를 동일한 타입으로 바꿔야합니다. float32 로 하겠습니다. 결과는 각 항목이 거짓일 경우 0, 참일 경우 1의 값을 갖는 NDArray가 됩니다. 이에 대한 평균을 계산하면 원하는 결과를 얻을 수 있습니다.

```
[11]: def accuracy(y_hat, y):
        return (y_hat.argmax(axis=1) == y.astype('float32')).mean().asscalar()
```

예측된 확률 분포와 레이블(label)에 대한 변수로 pick 함수에서 정의했던  $y_{\text{hat}}$  과  $y$  를 계속 사용하겠습니다. 첫번째 샘플의 예측 카테고리는 2 (첫번째 행에서 가장 큰 값은 0.6이고 이 값의 인덱스는 2)임을 확인할 수 있고, 이는 실제 레이블 0과 일치하지 않습니다. 두번째 샘플의 예측 카테고리는 2 (두번째 행에서 가장 큰 값이 0.5이고 이값의 인덱스는 2)이고, 이는 실제 레이블 2와 일치합니다. 따라서, 이 두 예들에 대한 분류 정확도는 0.5 입니다.

```
[12]: accuracy(y_hat, y)
[12]: 0.5
```

마찬가지로, `data_iter` 로 주어지는 데이터셋에 대한 모델 `net` 결과에 대한 정확도를 평가해볼 수 있습니다.

```
[13]: # The function will be gradually improved: the complete implementation will be
# discussed in the "Image Augmentation" section
def evaluate_accuracy(data_iter, net):
    acc_sum, n = 0.0, 0
    for X, y in data_iter:
        y = y.astype('float32')
        acc_sum += (net(X).argmax(axis=1) == y).sum().asscalar()
        n += y.size
    return acc_sum / n
```

이 모델 `net` 은 난수 값으로 가중치 값들이 초기화되어 있기 때문에, 정확도는 임의로 추측하는 것과 유사한 0.1 (10개의 클래스)로 나올 것입니다.

```
[14]: evaluate_accuracy(test_iter, net)
```

```
[14]: 0.0925
```

## 5.6.6 모델 학습

softmax 회귀(regression) 학습은 선형 회귀 학습과 아주 유사합니다. 모델의 손실 함수(loss function)를 최적화하기 위해서 미니 배치 확률적 경사 하강법(stochastic gradient descent)를 이용합니다. 모델 학습에서 num\_epochs 에포크(epoch) 횟수와 lr 학습 속도(learning rate)는 모두 바꿀 수 있는 하이퍼파라미터(hyper-parameter)입니다. 이 값을 바꾸면서, 모델의 분류 정확도를 높일 수 있습니다.

```
[15]: num_epochs, lr = 5, 0.1
```

```
# This function has been saved in the d2l package for future use
def train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
    params=None, lr=None, trainer=None):
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
        for X, y in train_iter:
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y).sum()
                l.backward()
                if trainer is None:
                    d2l.sgd(params, lr, batch_size)
                else:
                    # This will be illustrated in the next section
                    trainer.step(batch_size)
                y = y.astype('float32')
                train_l_sum += l.asscalar()
                train_acc_sum += (y_hat.argmax(axis=1) == y).sum().asscalar()
                n += y.size
        test_acc = evaluate_accuracy(test_iter, net)
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
            % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))

train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs,
    batch_size, [W, b], lr)
```

```
epoch 1, loss 0.7881, train acc 0.747, test acc 0.794
```

```
epoch 2, loss 0.5742, train acc 0.810, test acc 0.822
```

```
epoch 3, loss 0.5295, train acc 0.822, test acc 0.832
```

(continues on next page)

```
epoch 4, loss 0.5056, train acc 0.830, test acc 0.837
epoch 5, loss 0.4890, train acc 0.835, test acc 0.843
```

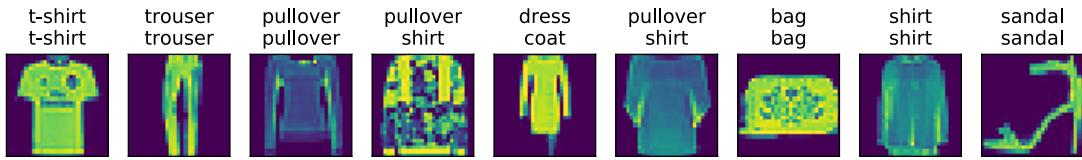
## 5.6.7 예측

학습이 완료되었으면, 모델을 이용해서 이미지를 분류 해보겠습니다. 이미지들이 주어졌을 때, 실제 레이블들(텍스트 결과의 첫번째 줄)과 모델 예측(텍스트 결과의 두번째 줄)을 비교 해보세요.

```
[16]: for X, y in test_iter:
    break

true_labels = d2l.get_fashion_mnist_labels(y.asnumpy())
pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1).asnumpy())
titles = [truelabel + '\n' + predlabel
          for truelabel, predlabel in zip(true_labels, pred_labels)]

d2l.show_fashion_mnist(X[0:9], titles[0:9])
```



## 5.6.8 요약

softmax 회귀(regression)을 이용해서 다중 카테고리 분류를 할 수 있습니다. 학습은 선형 회귀와 비슷하게 수행됩니다: 데이터를 획득하고, 읽고, 모델과 손실 함수(loss function)를 정의한 후, 최적화 알고리즘을 이용해서 모델을 학습시킵니다. 사실은 거의 모든 딥러닝 모델의 학습 절차는 이와 비슷합니다.

## 5.6.9 문제

1. 이 절에서 softmax 연산의 수학적인 정의에 따라 softmax 함수를 직접 정의해봤습니다. 이 경우 어떤 문제가 발생할 수 있을까요? (힌트 -  $\exp(50)$ 의 크기를 계산해보세요)

2. 이 절의 cross\_entropy 함수 크로스-엔트로피 손실 함수(cross-entropy loss function)의 정의를 따라서 구현되었습니다. 이 구현에 어떤 문제가 있을까요? (힌트 - logarithm의 도메인을 고려해보세요)
3. 위 두가지 문제를 어떻게 해결할 수 있는지 생각해보세요
4. 가장 유사한 레이블을 리턴하는 것이 항상 좋은 아이디어일까요? 예를 들면, 의료 진단에서 그 렇게 하겠나요?
5. 어떤 특성(feature)들을 기반으로 다음 단어를 예측하기 위해서 softmax 회귀(regression)을 사용하기를 원한다고 가정하겠습니다. 단어 수가 많은 경우 어떤 문제가 있을까요?

### 5.6.10 Scan the QR Code to Discuss



## 5.7 Softmax 회귀(regression)의 간결한 구현

우리는 이미 [선형 회귀 구현](#)에서 Gluon을 이용하는 것이 아주 편리하다는 것을 확인했습니다. 이제 Gluon이 분류에 어떻게 적용되는지 보도록 하겠습니다. 역시 몇 가지 패키지와 모듈을 import하는 것으로 시작합니다.

```
[1]: import sys
sys.path.insert(0, '..')

%matplotlib inline
import d2l
from mxnet import gluon, init
from mxnet.gluon import loss as gloss, nn
```

앞 절과 동일하게 Fashion-MNIST 데이터셋과 같은 배치 크기를 사용합니다.

```
[2]: batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

## 5.7.1 모델 파라미터 초기화하기

앞 절에서 언급했듯이 softmax regression의 output 레이어는 fully connected 레이어입니다. 따라서, 10개의 output을 갖는 fully connected 레이어를 추가하고, weight를 평균이 0이고 표준 편차가 0.01인 분포에서 난수를 뽑아서 초기화를 합니다.

```
[3]: net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

## 5.7.2 The Softmax

이전 예제에서는 모델의 결과를 계산하고, 이 결과에 크로스-엔트로피 손실(cross-entropy loss)을 적용했습니다. 이를 위해서 `-nd.pick(y_hat, y).log()` 를 이용했습니다. 수학적으로는 이렇게 하는 것이 매우 논리적입니다. 하지만, 이미 수차례 언급하였듯이 연산의 관점에서 보면 어려운 문제 가 될 수 있습니다. (예를 들면, *Naive Bayes* 의 예 또는 이전 장의 문제들처럼).  $y_{\text{hat}}$  의  $j$  번째 원소를  $\hat{y}_j$  라고 하고, 입력인  $y_{\text{linear}}$  변수의  $j$  번째 원소를  $z_j$  라고 할 때, softmax 함수는  $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}$  를 계산합니다.

만약 몇 개의  $z_i$  가 매우 큰 값을 갖는다면,  $e^{z_i}$  값이 float 변수가 표현할 수 있는 값보다 훨씬 커질 수 있습니다(overflow). 따라서, 분모(또는 분자)가 inf 가 돼서 결과  $\hat{y}_j$  가 0, inf 또는 nan 가 될 수 있습니다. 어떤 경우에든지 cross\_entropy 는 잘 정의된 값을 리턴하지 못할 것입니다. 이런 문제 때문에, softmax 함수에서는 모든  $z_i$  에서  $\max(z_i)$  를 뺍니다. 이렇게  $z_i$  를 이동시키는 것이 softmax 의 리턴값을 변화시키지 않는다는 사실을 확인해볼 수도 있습니다.

위에서 설명한 빼기와 정규화(normalization) 단계를 거친 후에도  $z_j$  가 여전히 매우 작은 음수값이 될 가능성이 있습니다. 따라서,  $e^{z_j}$  가 0과 매우 근접해지거나 유한한 프리시전(finite precision)(즉, underflow) 때문에 0으로 반올림될 수도 있습니다. 이렇게 되면,  $\hat{y}_j$  는 0이 되고,  $\log(\hat{y}_j)$  는 -inf 가 됩니다. 역전파(backpropagation)를 몇 번 거치면, 화면에 not-a-number (nan) 결과가 출력되는 것을 보게 될 것입니다.

이 문제에 대한 해결책은 지수 함수를 계산함에도 불구하고, 크로스-엔트로피(cross-entropy) 함수에서 이 값에 대한 로그(log) 값을 취하도록 합니다. 이 두 연산 softmax 와 cross\_entropy 를 함께 사용해서 수치 안정성 문제를 해결하고, 역전파(backpropagation) 과정에서 위 문제를 만나지 않게 할 수 있습니다. 아래 공식에서 보이는 것처럼,  $\log(\exp(\cdot))$  를 사용해서  $e^{z_j}$  를 계산하는 것을 피하고,  $z_j$

를 직접 사용할 수 있습니다.

$$\begin{aligned}\log(\hat{y}_j) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}\right) \\ &= \log(e^{z_j}) - \log\left(\sum_{i=1}^n e^{z_i}\right) \\ &= z_j - \log\left(\sum_{i=1}^n e^{z_i}\right)\end{aligned}$$

모델의 확률 결과에 대한 평가를 해야하는 경우에 사용되는 이런 전형적인 softmax 함수를 간편하게 만들고 싶습니다. 하지만, softmax 확률들을 새로운 손실 함수(loss function)에 대입하는 것 보다는,  $\hat{y}$  만 전달하면, softmax와 log 값들이 모두 softmax\_cross\_entropy 손실 함수(loss function)에서 계산되도록 하겠습니다. 이는 log-sum-exp 트릭 (see on Wikipedia) 과 같이 스마트한 것을 하는 것과 같다고 볼 수 있습니다.

```
[4]: loss = gloss.SoftmaxCrossEntropyLoss()
```

### 5.7.3 최적화 알고리즘

최적화 알고리즘으로 학습 속도(learning rate)를 0.1로 하는 미니 배치 확률적 경사 하강법(stochastic gradient descent)을 사용하겠습니다. 이는 선형 회귀에서도 동일하게 사용했는데, 옵티마이저(optimizer)들의 이식성 보여줍니다.

```
[5]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

### 5.7.4 학습

다음으로, 앞 절에서 정의된 학습 함수를 이용해서 모델을 학습 시킵니다.

```
[6]: num_epochs = 5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
              None, trainer)

epoch 1, loss 0.7895, train acc 0.747, test acc 0.805
epoch 2, loss 0.5731, train acc 0.812, test acc 0.820
epoch 3, loss 0.5297, train acc 0.823, test acc 0.833
```

(continues on next page)

```
epoch 4, loss 0.5060, train acc 0.828, test acc 0.836
epoch 5, loss 0.4889, train acc 0.834, test acc 0.840
```

이전과 같이 이 알고리즘은 매우 쓸만한 정확도인 83.7%로 수렴하는데, 이전보다 훨씬 더 적은 코드를 가지고 가능합니다. Gluon은 단순하게 구현할 경우 만날 수 있는 수치 안정성을 넘어서 특별한 예방을 포함하고 있기에, 모델을 직접 구현할 때 만날 수 있는 많은 일반적인 위험을 피할 수 있게 해줍니다.

### 5.7.5 문제

1. 배치 크기(batch size), 에포크(epoch), 학습 속도(learning rate)와 같은 하이퍼파라미터(hyper-parameter)를 변경하면서 어떤 결과가 나오는지 보세요.
2. 학습이 진행될 때 어느 정도 지나면 테스트 정확도가 감소할까요? 어떻게 고칠 수 있나요?

### 5.7.6 Scan the QR Code to Discuss



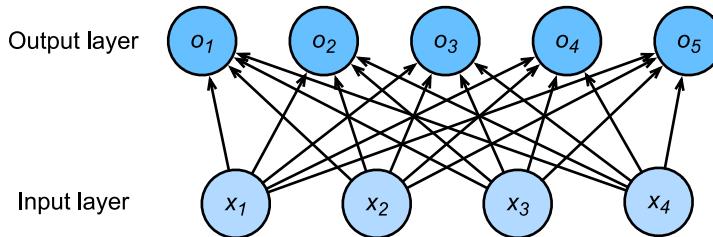
## 5.8 다층 퍼셉트론 (Multilayer Perceptron)

이전 절들에서 옷 이미지를 10개의 카테고리 중에 어디에 속하는지를 예측하는 멀티 클래스 로지스틱 리그레션(multiclass logistic regression) (또는 softmax regression)을 구현해봤습니다. 여기서부터 재미 있는 것들이 시작됩니다. 데이터를 다루고, 출력값을 유효한 확률 분포로 바꾸고, 적합한 loss 함수를 적용하고, 파라미터를 최적화하는 방법에 대해서 알아봤습니다. 기본적인 것들을 익혔으니, 이제 딥 뉴럴 네트워크를 포함하도록 우리의 도구 상자를 확장해보겠습니다.

### 5.8.1 은닉층(hidden layer)

이전의 것을 기억해보면, 단일 선형 변환 (single linear transformation)을 통해서 입력들을 바로 출력으로 매핑을 했고, 이는 아래과 같이 표현할 수 있습니다.

$$\hat{o} = \text{softmax}(Wx + b)$$

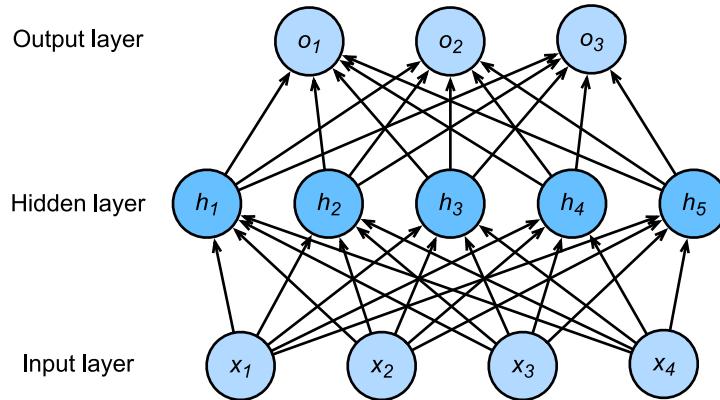


만약 레이블(label)들이 대략적인 선형 함수로 입력 데이터와 연관을 지을 수 있다면, 이 방법은 적절 할 수 있습니다. 하지만, 이 선형성은 너무 강한 가정입니다. 선형성은 각 입력에 대해서, 입력값이 증가하면 다른 입력값과는 상관없이 결과값이 커지거나 작아지는 것을 의미합니다.

### 5.8.2 하나에서 여러개로

검정색이나 희색 이미지를 이용해서 강아지나 고양이를 분류하는 케이스를 생각해봅시다. 각 픽셀의 값을 증가시키면 강아지라고 판별할 확률값을 높이거나 내려가는 경우를 생각해봅시다. 이는 합당하지 않습니다. 왜냐하면, 이렇게 된다면 결국 강아지는 모두 검정색이고 고양이는 모두 흰색이거나 그 반대라는 것을 의미하기 때문입니다.

이미지에 무엇이 있는지 알아내기 위해서는 입력과 출력 간의 매우 복잡한 관계와 이를 위해서는 패턴이 많은 특성(feature)들 사이의 관계를 통해서 특정 지어지는 가능성을 고려해야합니다. 이런 경우에는, 선형 모델의 정확도는 낮을 것입니다. 우리는 한 개 이상의 은닉층(hidden layer)을 함께 사용해서 더 일반적인 함수들을 이용한 모델을 만들 수 있습니다. 이를 구현하는 가장 쉬운 방법은 각 층 위에 다른 층들을 쌓는 것입니다. 각 층의 결과는 그 위의 층의 입력으로 연결되는데, 이는 마지막 출력층까지 반복됩니다. 이런 아키텍쳐는 일반적으로 “다층 퍼셉트론(multilayer perceptron)”이라고 불립니다. 즉, MLP는 여러 층을 연속해서 쌓아올립니다. 예를 들면 다음과 같습니다.



위 다층 퍼셉트론(multilayer perceptron)에서는 입력이 4개, 출력이 3개, 중간의 은닉층(hidden layer)은 5개의 은닉 유닛(hidden unit)이 있습니다. 입력층은 어떤 연산을 수행하지 않기 때문에, 이 다층 퍼셉트론(multilayer perceptron)은 총 2개의 층(layer)을 갖습니다. 은닉층(hidden layer)의 뉴런(neuron)들은 입력층의 입력들과 모두 연결되어 있습니다. 출력층(output layer)의 뉴런과 은닉층의 뉴런들도 모두 연결되어 있습니다. 따라서, 이 다층 퍼셉트론의 은닉층과 출력층은 모두 완전 연결층(fully connected layer)입니다.

### 5.8.3 선형에서 비선형으로

다중 클래스 분류의 경우 위 그림이 수학적으로 어떻게 정의되는지 보겠습니다.

$$\mathbf{h} = \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1$$

$$\mathbf{o} = \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$$

위 방법의 문제점은 은닉층을  $\mathbf{W} = \mathbf{W}_2 \mathbf{W}_1$  과  $\mathbf{b} = \mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2$ 를 사용해서 단일층 퍼셉트론(single layer perceptron) 식으로 재구성할 수 있기 때문에, 다층(multilayer)이 아닌 단순한 단일층 퍼셉트론(single layer perceptron)이라는 점입니다.

$$\mathbf{o} = \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2 (\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2 \mathbf{W}_1) \mathbf{x} + (\mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W} \mathbf{x} + \mathbf{b}$$

이를 해결하는 방법은 모든 층의 다음에  $\max(x, 0)$  와 같은 비선형 함수  $\sigma$  를 추가하는 것입니다. 이렇게 하면, 층들을 합치는 것이 더 이상 불가능해집니다. 즉,

$$\mathbf{h} = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{o} = \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$$

이렇게 하면, 여러 개의 은닉층들을 쌓는 것도 가능합니다. 즉,  $\mathbf{h}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$  과  $\mathbf{h}_2 = \sigma(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)$  를 각각 연결해서 진짜 다층 퍼셉트론을 만들 수 있습니다.

은닉 뉴런들이 각 입력의 값에 의존하고 있기 때문에 다층 퍼셉트론은 입력들 사이의 복잡한 상호 작용을 설명할 수 있습니다. 예를 들면, 입력들에 대한 논리 연산과 같이 임의의 연산을 수행하는 은닉 노드(hidden node)를 만드는 것도 쉽습니다. 다층 퍼셉트론이 보편적인 approximator라는 것이 잘 알려져 있습니다. 이것의 의미는 다음과 같습니다. 한 개의 은닉층을 갖는 다층 퍼셉트론이라도, 충분히 많은 노드와, 정확한 가중치를 설정할 수 있다면 모든 함수에 대한 모델을 만들 수 있다는 것입니다. 사실 그 함수를 학습하는 것은 매우 어려운 부분입니다. 그리고, 더 깊은(또는 더 넓은) 뉴럴 네트워크를 이용한다면 함수를 더욱 더 간결하게 추정할 수도 있습니다. 수학적인 부분은 이어질 장들에서 다루겠고, 이 장에서는 MLP를 실제로 만들어 보겠습니다. 아래 예제에서는 2개의 은닉층과 1개의 출력층을 갖는 다층 퍼셉트론을 구현해보겠습니다.

#### 5.8.4 벡터화와 미니 배치

샘플들이 미니 배치로 주어지는 경우에는 벡터화를 통해서 구현의 효율성을 높일 수 있습니다. 요약하면, 벡터를 행렬로 바꿀 것입니다.  $\mathbf{X}$  는 미니 배치의 입력 행렬에 대한 표기입니다. 2개의 은닉층(hidden layer)를 갖는 MLP는 다음과 같이 표현됩니다.

$$\mathbf{H}_1 = \sigma(\mathbf{W}_1 \mathbf{X} + \mathbf{b}_1)$$

$$\mathbf{H}_2 = \sigma(\mathbf{W}_2 \mathbf{H}_1 + \mathbf{b}_2)$$

$$\mathbf{O} = \text{softmax}(\mathbf{W}_3 \mathbf{H}_2 + \mathbf{b}_3)$$

위 MLP는 구현을 쉽게 할 수 있고, 최적화도 쉽게 할 수 있습니다. 표기법을 조금 남용해서, 비선형(nonlinearity)  $\sigma$  를 정의하고, 이를 행 단위로 입력에 적용하겠습니다. 즉, 한번에 하나의 관찰 또는 한번에 한 좌표씩 적용합니다. 실제 대부분 활성화 함수는 이렇게 적용합니다. (batch normalization 은 이 규칙에 대한 예외 중에 하나입니다.)

## 5.8.5 활성화 함수(activation function)

활성화 함수(activation function)의 예를 더 알아보겠습니다. 결국 딥 네트워트를 동작시키는 것은 선형(linear) 항목과 비선형(nonlinear) 항목들을 서로 교차시키는 것입니다. 구현하기 간단하고 좋은 효과로 유명한 ReLU 함수가 있습니다.

## 5.8.6 ReLU 함수

ReLU (rectified linear unit) 함수는 아주 간단한 비선형 변환입니다. 주어진  $x$ 에 대해서, ReLU 함수는 다음과 같이 정의됩니다.

$$\text{ReLU}(x) = \max(x, 0).$$

ReLU 함수는 양수만 그대로 두고, 음수는 버리고 0으로 바꿔주는 역할을 합니다. 이해를 돋기 위해서 도표로 그려보겠습니다. 간단한 방법으로, 도표를 그리는 함수 `xypplot`를 정의하겠습니다.

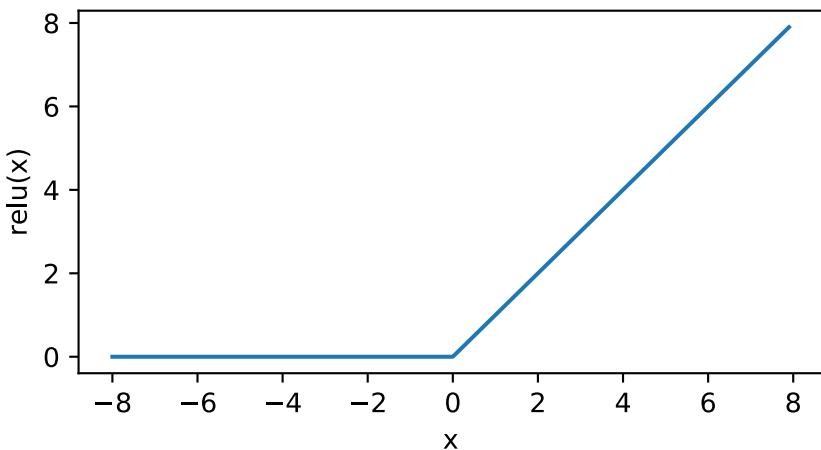
```
[1]: import sys
sys.path.insert(0, '..')

%matplotlib inline
import d2l
from mxnet import autograd, nd

def xypplot(x_vals, y_vals, name):
    d2l.set_figsize(figsize=(5, 2.5))
    d2l.plt.plot(x_vals.asnumpy(), y_vals.asnumpy())
    d2l.plt.xlabel('x')
    d2l.plt.ylabel(name + '(x)')
```

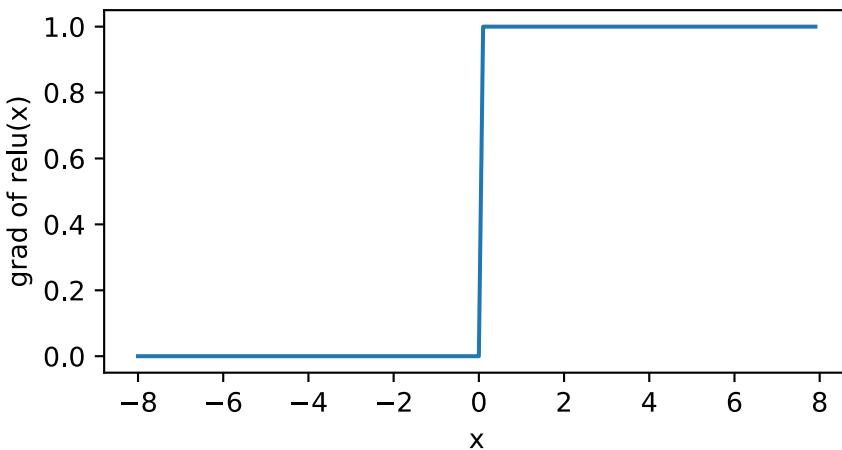
그리고, 이를 사용해서 ReLU 함수를 NDArray에서 제공하는 `relu` 함수를 이용해서 도식화합니다. 보이는 것처럼 활성화 함수(activation function)은 두 개의 선형 함수로 보입니다.

```
[2]: x = nd.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = x.relu()
xypplot(x, y, 'relu')
```



당연히 입력이 음수일 경우 ReLU 함수의 미분 값은 0이고, 입력이 양수면 ReLU 함수의 미분 값이 1이됩니다. 하지만, 입력이 0일 경우에는 ReLU 함수는 미분이 불가능하기 때문에, 입력이 0일 때는 left-hand-side (LHS) 미분인 0으로 선택합니다. ReLU 함수의 미분은 다음과 같습니다.

```
[3]: y.backward()
xyplot(x, x.grad, 'grad of relu')
```



ReLU 함수는 다양한 변형이 있는데, 예를 들면 He et al., 2015. parameterized ReLU (pReLU)가 있습니다. 이는 ReLU 선형 항목을 추가해서, 입력이 음수일 경우에도 정보가 전달될 수 있도록 만들고

있습니다.

$$\text{pReLU}(x) = \max(0, x) - \alpha x \text{ff}$$

ReLU를 사용하는 이유는 값이 사라지거나 그대로 전달하게 하는 식으로 미분이 아주 잘 작동하기 때문입니다. 이러한 특징으로 인해 최적화가 더 잘 되고, (나중에 설명할) vanishing gradient 문제를 줄여줍니다.

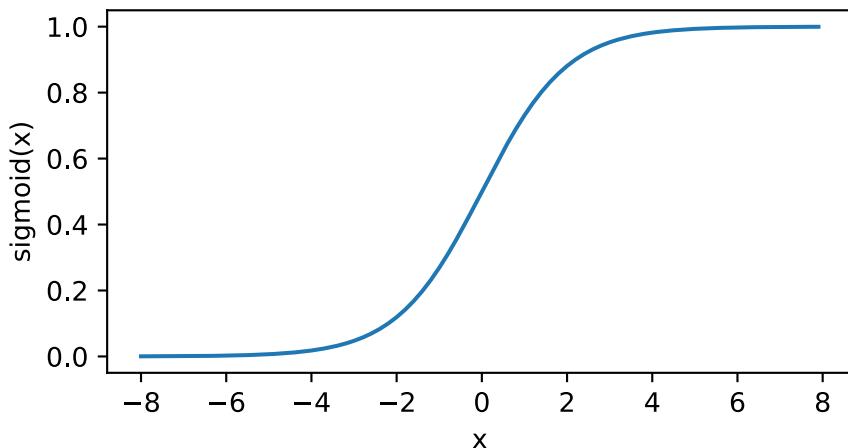
### 5.8.7 Sigmoid 함수

sigmoid 함수는 실수 값을 (0,1) 사이의 값으로 변환해줍니다.

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.\text{ff}$$

sigmoid 함수는 이전의 뉴럴 네트워크에서 일반적으로 사용되었으나, 현재는 더 간단한 ReLU 함수로 대체되었습니다. “Recurrent Neural Network”장에서는 이 함수가 0과 1사이의 값으로 변환해주는 특징을 이용해서 뉴럴 네트워크에서 정보의 전달을 제어하는데 어떻게 사용하는지 보겠습니다. Sigmoid 함수의 미분은 아래 그림과 같습니다. 입력이 0에 가까워지면, Sigmoid 함수는 선형 변환에 가까워집니다.

```
[4]: with autograd.record():
    y = x.sigmoid()
xyplot(x, y, 'sigmoid')
```

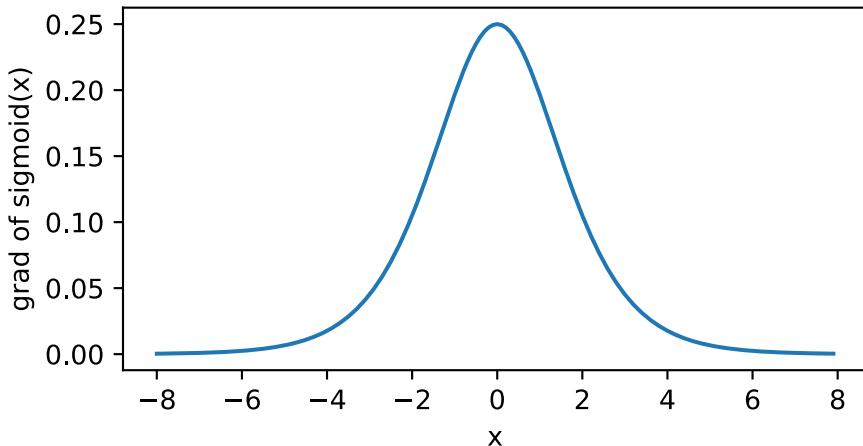


Sigmoid 함수의 미분은 아래와 같습니다.

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

Sigmoid 함수의 미분은 아래와 같이 생겼습니다. 입력이 0이면, Sigmoid 함수의 미분의 최대값인 0.25가 됩니다. 입력값이 0에서 멀어지면, Sigmoid 함수의 미분값은 0으로 접근합니다.

```
[5]: y.backward()
xyplot(x, x.grad, 'grad of sigmoid')
```



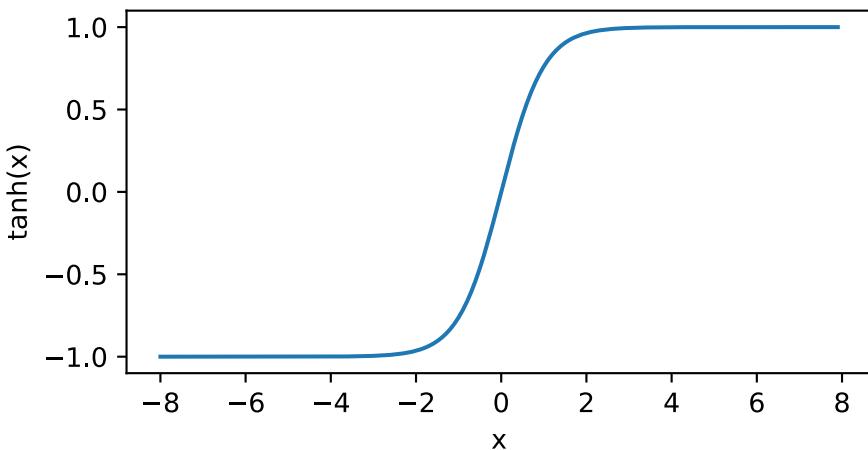
### 5.8.8 Tanh 함수

Tanh (Hyperbolic Tangent) 함수는 값을 -1과 1사이 값으로 변환합니다.

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

Tanh 함수를 도식화를 아래와 같이 할 수 있습니다. 입력이 0에 가까워지면, Tanh 함수는 선형 변환에 가까워집니다. 생긴 모양이 Sigmoid 함수와 비슷하지만, Tanh 함수는 좌표의 원점을 기준으로 대칭인 형태를 띕니다.

```
[6]: with autograd.record():
    y = x.tanh()
xyplot(x, y, 'tanh')
```

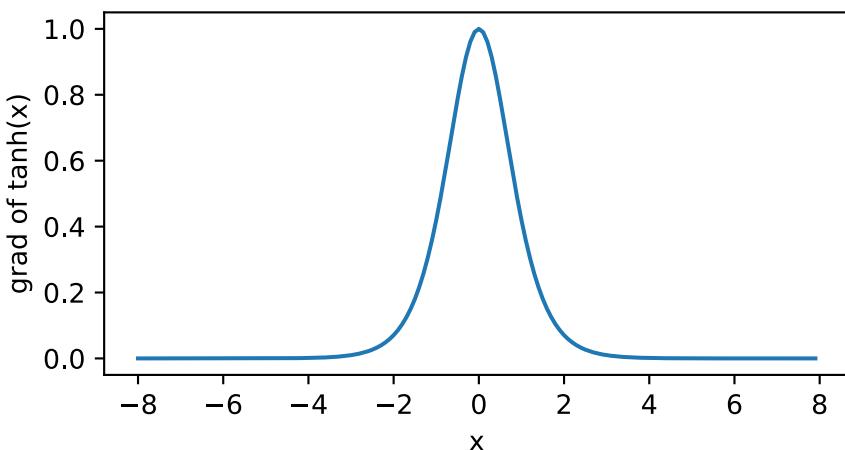


Tanh 함수의 미분은 다음과 같습니다.

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x).$$

Tanh 함수의 미분은 아래와 같이 그려지는데, 입력이 0과 가까우면 Tanh 함수의 미분은 최대값이 1에 근접하게 됩니다. 입력값이 0에서 멀어지면, Tanh 함수의 미분은 0에 접근합니다.

```
[7]: y.backward()
xyplot(x, x.grad, 'grad of tanh')
```



요약하면, 여러 종류의 비선형을 살펴봤고, 아주 강력한 네트워크 아키텍처를 만들기 위해서 어떻게 사용해야하는지도 알아봤습니다. 부수적으로 말하자면, 여기까지의 소개한 기법을 사용하면 1990년대의 최첨단의 딥러닝을 만들 수 있습니다. 이전과 다른 점은 예전에는 C 또는 Fortran 언어를 이용해서 수천 줄의 코드로 만들어야 했던 모델을 지금은 강력한 딥러닝 프레임워크가 있어서 몇 줄의 코드로 만들 수 있다는 점입니다.

### 5.8.9 요약

- 다층 퍼셉트론은 입력과 출력층에 한 개 이상의 완전 연결 은닉층(hidden layer)를 추가하고, 각 은닉층(hidden layer)의 결과에 활성화 함수(activation function)를 적용하는 것입니다.
- 일반적으로 사용되는 활성화 함수(activation function)는 ReLU 함수, Sigmoid 함수, Tanh 함수가 있습니다.

### 5.8.10 문제

1. Tanh, pReLU 활성화 함수(activation function)의 미분을 구하시오.
2. ReLU(또는 pReLU) 만을 사용해서 만든 multlayer perceptron은 연속된 piecewise linear function 임을 증명하세요.
3.  $\tanh(x) + 1 = 2\text{sigmoid}(2x)$  임을 증명하세요.
4. 층들 사이에 비선형없이 만든 다층 퍼셉트론이 있다고 가정합니다.  $d$  입력 차원,  $d$  출력 차원, 그리고 다른 layer는  $d/2$  차원을 찾는다고 했을 때, 이 네트워크는 단층 퍼셉트론(single layer perceptron) 보다 강하지 않다는 것을 증명하세요.
5. 한번에 하나의 미니배치에 적용하는 비선형이 있다고 가정합니다. 이렇게 해서 발생하는 문제가 무엇이 있을까요?

### 5.8.11 Scan the QR Code to Discuss



## 5.9 다층 퍼셉트론(multilayer perceptron)을 처음부터 구현하기

다층 퍼셉트론(multilayer perceptron, MLP)가 어떻게 작동하는지 이론적으로 배웠으니, 직접 구현해 보겠습니다. 우선 관련 패키지와 모듈을 import 합니다.

```
[1]: import sys  
sys.path.insert(0, '..')  
  
%matplotlib inline  
import d2l  
from mxnet import nd  
from mxnet.gluon import loss as gloss
```

이 예제에서도 Fashion-MNIST 데이터셋을 사용해서, 이미지를 분류하는데 다층 퍼셉트론(multilayer perceptron)을 사용하겠습니다.

```
[2]: batch_size = 256  
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

### 5.9.1 모델 파라미터 초기화하기

이 데이터셋은 10개의 클래스로 구분되어 있고, 각 이미지는  $28 \times 28 = 784$  픽셀의 해상도를 가지고 있습니다. 따라서, 입력은 784개이고, 출력은 10개가 됩니다. 우리는 한 개의 은닉층(hidden layer)을 갖는 MLP를 만들어보겠는데, 이 은닉층(hidden layer)은 256개의 은닉 유닛(hidden unit)을 갖도록 하겠습니다. 만약 원한다면 하이퍼파라미터인 은닉 유닛(hidden unit) 개수를 다르게 설정할 수도 있습니다. 일반적으로, 유닛(unit)의 개수는 메모리에 잘 배치될 수 있도록 2의 지수승의 숫자로 선택합니다.

```
[3]: num_inputs, num_outputs, num_hiddens = 784, 10, 256  
  
W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens))  
b1 = nd.zeros(num_hiddens)  
W2 = nd.random.normal(scale=0.01, shape=(num_hiddens, num_outputs))  
b2 = nd.zeros(num_outputs)  
params = [W1, b1, W2, b2]  
  
for param in params:  
    param.attach_grad()
```

## 5.9.2 활성화 함수(activation function)

여기서 ReLU 를 직접 호출하는 대신, ReLU를 maximum 함수를 이용해서 정의합니다.

```
[4]: def relu(X):
    return nd.maximum(X, 0)
```

## 5.9.3 모델

Softmax 회귀(regression)에서 그랬던 것처럼, reshape 함수를 이용해서 원래 이미지를 num\_inputs 크기의 벡터로 변환한 다음에 앞에서 설명한 대로 MLP를 구현합니다.

```
[5]: def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(nd.dot(X, W1) + b1)
    return nd.dot(H, W2) + b2
```

## 5.9.4 손실 함수(loss function)

더 나은 계산의 안정성을 위해서, softmax 계산과 크로스-엔트로피 손실(cross-entropy loss) 계산은 Gluon 함수를 이용하겠습니다. 왜 그런지는 [앞 절](#)에서 이 함수의 구현에 대한 복잡성을 이야기했으니 참고 바랍니다. Gluon 함수를 이용하면 코드를 안전하게 구현하기 위해서 신경을 써야하는 많은 세밀한 것들을 간단하게 피할 수 있습니다. (자세한 내용이 궁금하다면 [소스](#) 코드를 살펴보기 바랍니다. 소스 코드를 보면 다른 관련 함수를 구현하는데 유용한 것들을 배울 수도 있습니다.)

```
[6]: loss = gloss.SoftmaxCrossEntropyLoss()
```

## 5.9.5 학습

다층 퍼셉트론(multilayer perceptron)을 학습시키는 단계는 softmax 회귀(regression) 학습과 같습니다. g21 패키지에서 제공하는 train\_ch3 함수를 직접 호출합니다. 이 함수의 구현은 [여기](#)를 참고하세요. 총 에포크(epoch) 수는 10으로 학습 속도(learning rate)는 0.5로 설정합니다.

```
[7]: num_epochs, lr = 10, 0.5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
              params, lr)
```

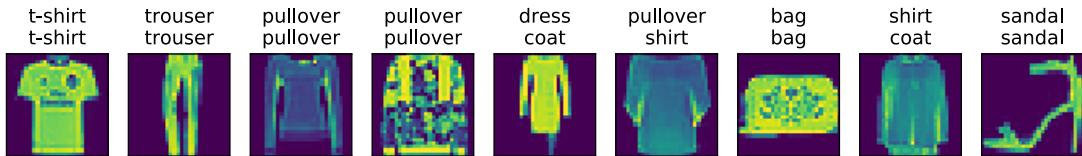
```
epoch 1, loss 0.8048, train acc 0.699, test acc 0.835
epoch 2, loss 0.4924, train acc 0.817, test acc 0.844
epoch 3, loss 0.4329, train acc 0.840, test acc 0.855
epoch 4, loss 0.4001, train acc 0.853, test acc 0.862
epoch 5, loss 0.3713, train acc 0.864, test acc 0.871
epoch 6, loss 0.3534, train acc 0.870, test acc 0.873
epoch 7, loss 0.3400, train acc 0.876, test acc 0.878
epoch 8, loss 0.3295, train acc 0.878, test acc 0.880
epoch 9, loss 0.3157, train acc 0.882, test acc 0.882
epoch 10, loss 0.3077, train acc 0.886, test acc 0.880
```

학습이 잘 되었는지 확인하기 위해서, 모델을 테스트 데이터에 적용해 보겠습니다. 이 모델의 성능이 궁금하다면, 동일한 분류를 수행하는 *linear 모델*의 결과와 비교해보세요.

```
[8]: for X, y in test_iter:
    break

true_labels = d2l.get_fashion_mnist_labels(y.asnumpy())
pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1).asnumpy())
titles = [truelabel + '\n' + predlabel
          for truelabel, predlabel in zip(true_labels, pred_labels)]

d2l.show_fashion_mnist(X[0:9], titles[0:9])
```



이전보다 조금 성능이 좋아 보이는 것으로 보아 MLP를 사용하는 것이 좋은 것임을 알 수 있습니다.

### 5.9.6 요약

간단한 MLP는 직접 구현하는 것이 아주 쉽다는 것을 확인했습니다. 하지만, 많은 수의 층을 갖는 경우에는 굉장히 복잡해질 수 있습니다. (예를 들면 모델 파라미터 이름을 정하는 것 등)

### 5.9.7 문제

1. num\_hiddens 하이퍼파라미터를 변경해서 결과가 어떻게 영향을 받는지 확인해보세요.
2. 새로운 은닉층(hidden layer)를 추가해서 어떤 영향을 미치는지 확인해보세요.
3. 학습 속도(learning rate)를 변경하면 결과가 어떻게 되나요?
4. 모든 하이퍼파라미터(학습 속도(learing rate), 에포크(epoch) 수, 은닉층(hidden layer) 개수, 각 층의 은닉 유닛(hidden unit) 개수)의 조합을 통해서 얻을 수 있는 가장 좋은 결과는 무엇인가요?

### 5.9.8 Scan the QR Code to Discuss



## 5.10 다층 퍼셉트론(multilayer perceptron)의 간결한 구현

다층 퍼셉트론(multilayer perceptron, MLP)가 어떻게 작동하는지 이론적으로 배웠으니, 이제 직접 구현해보겠습니다. 우선 관련 패키지와 모듈을 import 합니다.

```
[1]: import sys
sys.path.insert(0, '..')

import d2l
from mxnet import gluon, init
from mxnet.gluon import loss as gloss, nn
```

### 5.10.1 모델

Softmax 회귀(regression)와 유일하게 다른 점은 은닉 층(hidden layer)으로 완전 연결층(fully connected layer)을 추가한다는 점입니다. 이 은닉층(hidden layer)은 256개의 은닉 유닛(hidden unit)을 갖고, 활성화 함수(activation function)로 ReLU를 사용합니다.

```
[2]: net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

`net.add()` 를 호출할 때 세밀하게 살펴봐야 할 점이 있는데, 이 함수를 이용하면 한 개 이상의 층을 네트워크에 추가할 수 있다는 것입니다. 즉, 위 코드들로 정의된 뉴럴 네트워크는 `net.add(nn.Dense(256, activation='relu'))`, `nn.Dense(10)` 코드 한 줄로 정의되어 네트워크와 동일합니다. 또한, Gluon은 명시되지 않은 파라미터들을 자동으로 알아냅니다. 예를 들면, 두번째 층은  $256 \times 10^{ff}$  크기의 행렬이 필요한데, 이는 네트워크가 처음 실행될 때 자동으로 찾아내 집니다.

Softmax 회귀(regression) 학습과 거의 같은 절차로 데이터를 읽고 모델을 학습 시킵니다.

```
[3]: batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)

loss = gloss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
              None, trainer)

epoch 1, loss 0.7845, train acc 0.707, test acc 0.781
epoch 2, loss 0.4881, train acc 0.818, test acc 0.842
epoch 3, loss 0.4291, train acc 0.841, test acc 0.860
epoch 4, loss 0.3920, train acc 0.855, test acc 0.868
epoch 5, loss 0.3727, train acc 0.863, test acc 0.870
epoch 6, loss 0.3538, train acc 0.870, test acc 0.868
epoch 7, loss 0.3357, train acc 0.876, test acc 0.879
epoch 8, loss 0.3288, train acc 0.878, test acc 0.875
epoch 9, loss 0.3166, train acc 0.883, test acc 0.882
epoch 10, loss 0.3070, train acc 0.886, test acc 0.886
```

## 5.10.2 문제

1. 은닉층(hidden layer)들을 더 추가해서 결과가 어떻게 변하는지 확인하세요.
2. 다른 활성화 함수(activation function)를 적용해보세요. 어떤 것이 가장 좋게 나오나요?
3. 가중치에 대한 초기화를 다르게 해보세요.

### 5.10.3 Scan the QR Code to Discuss



## 5.11 모델 선택, 언더피팅(underfitting), 오버피팅(overfitting)

머신러닝에서 우리의 목표는 일반적인 패턴을 발견하는 것입니다. 예를 들면, 유전자 표지와 성인기의 치매 발병간의 관련성을 배우기를 원할 수 있습니다. 여기서 우리의 원하는 것은 전인류에 대한 위험을 평가하기 위해서 성공적으로 적용될 수 있는 패턴을 발견하는 것입니다.

하지만, 우리가 모델을 학습시킬 때, 전체 인구에 대한 정보를 사용하는 것은 불가능합니다. 대신, 우리는 작은 유한한 샘플만을 사용할 수 있습니다. 대형 병원 시스템에서 조차도 수십만건의 의료 기록 정도를 얻을 수도 있습니다. 이렇게 한정된 표본의 크기를 감안하면, 보이지 않은 데이터에 존재하는 않는 가짜 연관성을 발견할 수 있습니다.

극단적인 병리학적 사례를 생각해보겠습니다. 어떤 사람들이 대출금을 상환할 것인지를 예측하는 것을 배우기를 원한다고 상상해보십시오. 대출 기관이 케이스를 조사하기 위해서 당신을 데이터 과학자로 고용해서, 100명의 지원자에 대한 완전한 파일을 제공합니다. 100명의 지원자 중에 5명이 3년간 채무 불이행을 했습니다. 제공된 파일에는 수입, 직업, 신용점수, 취업 기간 등을 포함한 100여개의 특성(feature)들이 있습니다. 더불어 대출기관과의 인터뷰 비디오를 추가로 제공한다고 상상해보겠습니다. 이것들이 아주 많은 데이터처럼 보일 수 있습니다!

엄청난 양의 특성(feature) 세트를 만든 후, 채무 불이행을 한 5명이 모두 인터뷰 중에 파란 셔츠를 입었다는 것을 발견했다고 가정하겠습니다. 반면에 일반 인구의 40%만이 파란 셔츠를 입었습니다. 여러분이 학습시킨 모델이 이 신호를 포착해서, 학습한 패턴의 중요한 부분으로 사용할 가능성이 큽니다.

채무 불이행자가 더 이상 파란 셔츠를 입지 않을지라도, 모든 5명의 채무 불이행자가 파란 셔츠를 입을 것이라고 관찰할 확률이 1%입니다. 수백 또는 수천개의 특성(feature)들을 가지고 있으면서 샘플의 크기를 작게 유지한다면, 아주 많은 가짜 상관 관계를 관찰할 것입니다. 수조개의 학습 샘플이 주어진다면, 이 잘못된 연관성을 사라질 것입니다. 하지만, 실제 그렇게 많은 데이터를 얻을 수 있는 경우가 드뭅니다.

모델이 실제 분포보다 학습 샘플들 분포에 더 근접하게 학습되는 현상을 오버피팅(overfitting)이라고

하며, 오버피팅(overfitting)을 피하는 방법을 정규화(regularization)라고 합니다. 더 정확하게는 이전 절의 Fashion-MNIST 데이터셋에서 이 현상이 나왔었습니다. 실험 중에 모델의 구조나 하이퍼파라미터(hyperparameter)들을 바꾸면, 어떤 조합에서는 모델이 학습 데이터와 비교해서 테스팅 데이터셋을 사용했을 때 정확하지 않게 나오는 현상을 찾아냈을 수 있습니다.

### 5.11.1 학습 오류와 일반화 오류

이 현상에 대해서 설명하기 전에, 학습 오류와 일반화 오류에 대해서 구분할 필요가 있습니다. Layman 의 용어에 의하면, 학습 오류는 학습 데이터셋을 사용했을 때 나오는 오류이고, 일반화 오류는 기본 데이터 분포에서 추가로 데이터를 뽑아낸 가상의 스트림에 모델을 적용할 때 예상되는 오류를 의미합니다. 종종 일반화 오류는 테스트 셋에 모델을 적용해서 추정합니다. 예를 들면, 이전에 논의한 손실 함수(loss function)들 중에 선형회귀에 사용된 제곱 손실 함수(squared loss function)나 softmax 회귀(regression)에 사용된 크로스-엔트로피 손실 함수(cross-entropy loss function)를 이용해서 학습 오류와 일반화 오류 비율을 계산할 수 있습니다.

다음 세가지 사고 실험이 이 상황을 더 잘 설명하는데 도움이 될 것입니다. 학기말 시험을 준비하는 대학생을 생각해봅시다. 근면한 학생은 준비를 잘 하고, 이전 년도의 시험 문제를 통해서 본인의 능력을 테스트하는 등의 노력할 것입니다. 하지만, 이전 시험을 잘 푸는 것이 꼭 그 학생이 실제 시험을 잘 본다는 것을 보장하지는 못합니다. 예를 들면, 그 학생은 시험 문제의 답을 기계적으로 학습하면서 준비하려고 노력 할지도 모릅니다. 이렇게 하면 그 학생은 많은 것을 외워야 합니다. 이렇게 해서 그 학생은 이전 시험의 답을 완벽하게 암기할 수도 있습니다. 반면에, 다른 학생은 문제에 대한 특정 답이 나오게되는 이유를 이해하려고 노력하면서 준비했습니다. 대부분의 경우에는, 후자의 경우에 실제 시험에서 더 좋은 성적을냅니다.

마찬가지로, 질문에 대한 답을 테이블에서 조회하는 역할을 수행하는 모델을 생각해 보겠습니다. 입력이 이산적(discrete)인 경우, 많은 샘플을 보는 것을 통해서 잘 동작할 수 있습니다. 하지만, 이 모델은 데이터가 실수값이거나 우리가 원하는 것보다 더 부족한 경우 실제 상황에서는 잘 동작하지 않을 가능성이 높습니다. 더군다나, 우리는 모델을 저장할 수 있는 한정된 양의 메모리만 가지고 있습니다.

마지막으로, 간단한 분류 문제를 고려해보겠습니다. 공정한 동전을 던져서 앞면이 나오면 0, 뒷면이 나오면 1로 분류한 테이블을 갖는 학습 데이터가 있습니다. 우리가 무엇을 하던지 상관없이, 일반화 오류는 항상  $\frac{1}{2}$ 입니다. 하지만, 학습 오류는 동전을 던지는 운에 따라서 더 작아질 수 있습니다. 예를 들어 {0, 1, 1, 1, 0, 1}인 학습 데이터를 사용하는 경우에는, 1을 예측한다면,  $\frac{1}{3}$ 의 오류가 발생하는데, 이는 실제 보다 더 좋은 값입니다. 데이터 양을 늘릴 수록, 확률의 편차가  $\frac{1}{2}$ 로 부터 감소하고, 학습 오류도 이에 근접하게 될 것입니다. 그 이유는 우리의 모델이 데이터에 오버핏(overfit) 되어 있고, 데이터의 양을 늘리면 모든것이 평균상태가 되기 때문입니다.

## 통계적 학습 이론(statistical learning theory)

이 현상에 대한 공식 이론이 있습니다. Glivenko와 Cantelli는 그들의 eponymous theorem에서 학습 오류가 일반화 오류로 수렴하는 비율을 도출했습니다. Vapnik와 Chervonenkis는 여러 논문을 통해서 이것을 더 일반적인 함수의 클래스들로 확장했고, Statistical Learning Theory의 기초가 되었습니다.

특별히 이야기하지 않으면, 학습 데이터셋과 테스트 데이터셋은 동일한 분포로 부터 독립적으로 추출되었다고 가정합니다. 즉, 이 분포로부터 추출을 할 때, 추출들 간의 어떤 기억도 없다는 것을 의미합니다. 더 나아가, 두 경우 모두 동일한 분포를 사용하는 것을 의미합니다. 이를 위반하는 명확한 사례로 초등학교 학생들의 얼굴 데이터로 학습한 얼굴 인식 모델을 이용해서 일반 인구에 적용하는 것을 들 수 있습니다. 초등학교 학생들과 일반 사람들은 아주 다르게 보일 것이기 때문에 잘 동작하지 않을 가능성이 높습니다. 학습을 통해서 우리는 학습 데이터에 잘 맞는 함수를 찾으려고 노력합니다. 만약 학습 데이터의 자세한 것에 아주 잘 적응할 정도로 유연하다면, 지나치게 잘하게 될 것입니다. 이것이 바로 우리가 피하려고 또는 통제하려는 것입니다. 대신, 우리는 일반화 오류를 줄이는 모델을 찾는 것을 원합니다. 딥러닝의 많은 튜닝은 이런 것이 일어나지 않도록 하는데 이용됩니다.

## 모델 복잡도

우리는 간단한 모델들과 많은 데이터가 있을 경우, 일반화 오류가 학습 오류와 비슷해 지기를 예상합니다. 반면에 모델이 복잡하고 데이터가 적을 때는, 학습 오류는 작아지지만, 일반화 오류는 커질 것을 예상합니다. 무엇이 정확하게 모델의 복잡성을 구성하는지는 복잡한 문제입니다. 모델이 일반화를 잘 할 수 있을지는 많은 것들에 의해서 영향을 받습니다. 예를 들면, 더 많은 파라미터를 갖는 모델이 더 복잡하다고 여기질 수 있고, 값의 범위가 더 넓은 파라미터를 갖는 모델이 더 복잡하다고 여겨질 수도 있습니다. 뉴럴 네트워크의 경우에는 학습을 더 오래한 모델이 더 복잡한 것이라고 생각될 수도 있고, 일찍 학습을 종료한 모델은 덜 복잡하다고 생각될 수도 있습니다.

다양한 모델 종류들 간의 복잡성을 비교하는 것은 어려운 일 수 있습니다. 예를 들면 결정 트리(decision tree)와 뉴럴 네트워크의 복잡성을 비교하는 것은 어렵습니다. 이런 경우, 간단한 경험의 법칙을 적용하는 것이 유용합니다. 통계학자들은 임의의 사실을 잘 설명하는 모델을 복잡하다고 하고, 제한적인 설명을 하는 능력을 갖으나 데이터를 여전히 잘 설명하는 모델은 진실에 좀 더 가깝다고 합니다. 철학에서 이것은 포퍼의 과학 이론의 허위 진술성(falsifiability)과 밀접한 관련이 있습니다. 어떤 이론이 데이터에 적합하고, 오류를 입증할 수 있는 특정 테스트가 있다면, 그 이론을 좋다고 합니다. 모든 통계적 추정이 post-hoc이기에 이는 매우 중요합니다. 즉, 우리는 어떤 사실을 관찰한 후에 추정을 합니다. 따라서, 관련 오류에 취약하게 됩니다. 자, 철학에 대해서는 충분히 이야기했으니, 더 구체적인 이슈를 살펴보겠습니다.

여러분이 이 장에 대한 직관을 가질 수 있도록, 모델 클래스의 일반화에 영향을 줄 수 있는 몇 가지 요소들에 집중하겠습니다.

- 튜닝이 가능한 파라미터의 개수. 자유도라고 불리기도 하는 튜닝 가능한 파라미터의 수가 많을 경우, 모델이 오버피팅(overfitting)에 더 취약한 경향이 있습니다.
- 파라미터에 할당된 값. 가중치들이 넓은 범위의 값을 갖을 경우, 모델은 오버피팅(overfitting)에 더 취약할 수 있습니다.
- 학습 예제의 개수. 모델이 간단할 지라도 학습 데이터가 한 개 또는 두 개인 경우에는 오버핏(overfit) 되기가 아주 쉽습니다. 하지만, 수백만개의 학습 데이터를 이용해서 모델을 오버피팅(overfitting) 시키기 위해서는 모델이 아주 복잡해야 합니다.

## 5.11.2 모델 선택

머신러닝에서, 우리는 보통 여러 후보 모델들의 성능을 평가해서 모델을 선정합니다. 이 과정을 모델 선택(model selection)이라고 합니다. 후보 모델들은 다른 하이퍼파라미터(hyper-parameter)들을 적용한 간단한 모델들일 수 있습니다. 다층 퍼셉트론(multilayer perceptron)을 예로 들면, 우리는 은닉층(hidden layer)의 개수, 은닉 유닛(hidden unit)의 개수, 각 은닉층(hidden layer)의 활성화 함수(activation function)를 선택할 수 있습니다. 효과적인 모델을 찾기 위해서는 모델 선택에 상당한 노력이 필요합니다. 다음 절에서 모델 선택에 종종 사용되는 검증 데이터셋(validation data set)에 대해서 설명하겠습니다.

### 검증 데이터셋

엄밀하게 이야기하면, 테스트 셋은 모든 하이퍼파라미터(hyper-parameter)들과 모델 파라미터들이 선택된 후에만 사용되어야 합니다. 특히, 테스트 데이터셋은 하이퍼파라미터(hyper-parameter) 선택과 같은 모델 선택 과정에서 사용되서는 안됩니다. 모델 선택 과정에서 학습 데이터에만 의존해서도 안됩니다. 그 이유는 일반화 오류율이 학습 오류율로 예상될 수 없기 때문입니다. 이를 고려해서, 학습 데이터와 테스트 데이터 이외의 데이터를 확보해서 모델 선택에 사용할 수 있습니다. 이렇게 확보한 데이터는 검증 데이터 셋(validation data set) 또는 검증셋(validation set)이라고 합니다. 예를 들면, 학습 데이터에서 임의로 선택한 일부의 데이터를 검증 셋으로 사용하고, 나머지를 실제 학습 데이터로 사용할 수 있습니다.

하지만, 실제 응용의 경우에는 테스트 데이터를 구하기 어렵기 때문에 한 번 사용하고 버리는 경우가 드뭅니다. 따라서, 실제의 경우에는 검증 데이터와 테스트 데이터 셋의 구분이 명확하지 않을 수도 있습니다. 명시적으로 별도로 언급하지 않는 경우 이 책에서 실험으로 사용하는 테스트 데이터셋은 검증 데이터 셋이라고 하고, 실험 결과의 테스트 정확도는 검증 정확도를 의미하겠습니다. 좋은 소식은 검증 셋에 아주 많은 데이터가 필요하지 않다는 것입니다. 우리의 예측의 불명확성은  $O(n^{-\frac{1}{2}})$  오더로 보여질 수 있습니다.

## K-겹 교차 검증(K-fold Cross-Validation)

학습 데이터가 충분하지 않을 경우 검증 데이터를 많이 확보하는 것은 과하다고 간주됩니다. 왜냐하면, 검증 데이터는 모델 학습에 어떤 역할도 할 수 없기 때문입니다. 이에 대한 해결책으로 K-겹 교차 검증(K-Fold Cross-Validation) 방법이 있습니다. 예 K-겹 교차 검증(K-Fold Cross-Validation)에서는 원래 학습 데이터를 겹치지 않는 K개의 부분 데이터셋으로 나누고, 모델 학습과 검증을 K 번 반복합니다. 검증이 수행될 때마다 K – 1 개의 부분 데이터셋으로 학습을 하고, 1개의 부분 데이터셋으로 검증을 수행합니다. 모델을 검증하는데 사용하는 부분 데이터셋은 계속 바꾸면서 K 번 학습과 검증을 수행하게 됩니다. 마지막으로, K 번의 학습과 검증 오류에 대한 평균을 각각 구합니다.

### 5.11.3 언더피팅(underfitting)과 오버피팅(overfitting)

다음으로는 모델 학습을 진행하면서 만나게 되는 일반적인 두가지 문제에 대해서 살펴보겠습니다. 첫 번째 문제는 모델이 너무 간단하기 때문에 학습 오류가 줄어들지 않는 것입니다. 이 현상을 언더피팅(underfitting)이라고 합니다. 두번째 문제는 앞에서 이야기했던 오버피팅(overfitting)으로, 이는 학습 오류가 테스트 데이터셋에 대한 오류보다 아주 작은 경우입니다. 실제로 이 두 문제는 가능한 경우 항상 동시에 해결이 되어야 합니다. 이 두 문제의 원인은 여러 요소들이 있지만, 여기서는 두 가지 요소에 대해서 집중하겠습니다. 이 두 가지는 모델 복잡성과 학습 데이터셋의 크기입니다.

#### 모델 복잡도

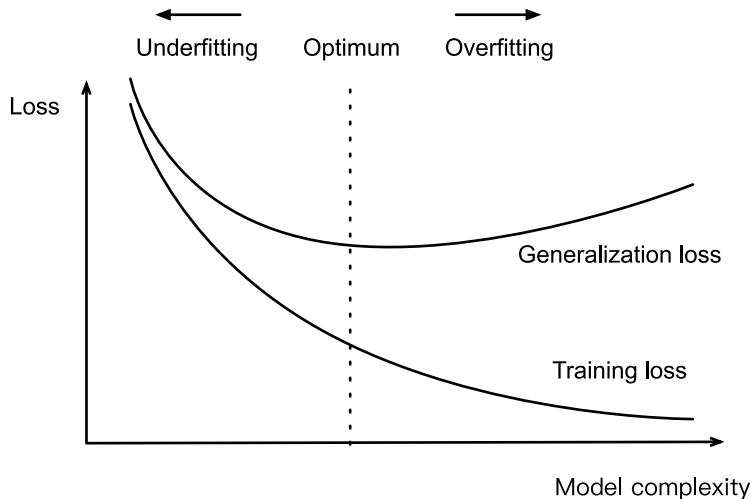
이 이슈를 설명하기 위해서 다항식을 예로 들겠습니다. 스칼라 데이터 특성(feature)  $x$  와 이에 대한 스칼라 레이블(label)  $y$ 로 구성된 학습 데이터가 주어진 경우,  $y$ 를 추정하는  $d$  차원 다항식을 찾는다고 하겠습니다.

$$\hat{y} = \sum_{i=0}^d x^i w_i$$

여기서  $w_i$ 는 모델의 가중치 파라미터를 의미하고, 편향(bias)은  $x^0 = 1$ 이기 때문에  $w_0$ 이 됩니다. 간단하게 하기 위해서, 선형 회귀(linear regression)의 경우와 같이 제곱 손실(squared loss)을 사용하겠습니다. (사실  $d = 1$ 인 경우 이 모델은 선형 회귀(linear regression)입니다.)

고차원의 다항 함수는 저차원의 다항 함수보다 더 복잡합니다. 이유는 차원이 더 높아지면, 더 많은 파라미터를 갖게 되고, 모델 함수의 선택 범위가 더 넓어지기 때문입니다. 따라서, 같은 학습 데이터셋을 사용하는 경우, 더 높은 차원의 다항 함수에 대한 학습 오류는 그보다 낮은 차원의 다항 함수의 오류보다 낮을 것입니다. 이를 염두하면, 학습 데이터셋이 고정되어 있을 때 모델의 복잡도와 오류의 일반적인 상관관계는 아래 그림으로 설명됩니다. 데이터에 비해서 모델이 너무 간단하면, 언더피팅

(underfitting)이 발생하고, 모델을 너무 복잡하게 선택하면 오버피팅(overfitting)이 발생합니다. 데이터에 대한 모델을 적절한 복잡성을 선택하는 것이 오버피팅(overfitting)과 언더피팅(underfitting) 문제를 피하는 방법 중에 하나입니다.



## 데이터셋의 크기

다른 원인은 학습 데이터의 양입니다. 일반적으로 학습 데이터셋의 샘플 개수가 충분하지 않은 경우, 특히 모델의 파라미터 개수보다 적은 수의 샘플을 사용하는 경우, 오버피팅(overfitting)이 쉽게 발생합니다. 학습 데이터의 양을 늘리면, 일반화 오류는 일반적으로 줄어듭니다. 즉, 더 많은 데이터는 모델 학습에 나쁜 영향을 미치지 않다는 것을 의미합니다. 더 나아가서, 이는 충분한 데이터가 있다면, 일반적으로 많은 층들을 갖는 복잡한 모델을 사용해야 한다는 것을 의미합니다.

### 5.11.4 다항식 회귀(Polynomial Regression)

데이터를 이용해서 다항식을 학습시켜 보면서 이것이 어떻게 동작하는지 보겠습니다. 우선 몇 가지 모듈을 import 합니다.

```
[1]: import sys  
sys.path.insert(0, '..')  
  
%matplotlib inline  
import d2l
```

(continues on next page)

```
from mxnet import autograd, gluon, nd
from mxnet.gluon import data as gdata, loss as gloss, nn
```

## 데이터셋 생성하기

우선 데이터가 필요합니다. 주어진  $x$ 에 대해서, 다음 3차원 방정식을 사용해서 학습 데이터와 테스트 데이터로 사용할 레이블(label) 만들겠습니다.

$$y = 5 + 1.2x - 3.4 \frac{x^2}{2!} + 5.6 \frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1)$$

노이즈 항인  $\epsilon$ 은 평균이 0이고 표준 편차가 0.1인 정규 분포를 따릅니다. 학습과 테스트 데이터셋의 샘플의 개수는 각각 100개, 1000개로 하겠습니다.

```
[2]: maxdegree = 20 # Maximum degree of the polynomial
n_train, n_test = 100, 1000 # Training and test data set sizes
true_w = nd.zeros(maxdegree) # Allocate lots of empty space
true_w[0:4] = nd.array([5, 1.2, -3.4, 5.6])

features = nd.random.normal(shape=(n_train + n_test, 1))
features = nd.random.shuffle(features)
poly_features = nd.power(features, nd.arange(maxdegree).reshape((1, -1)))
poly_features = poly_features / (
    nd.gamma(nd.arange(maxdegree) + 1).reshape((1, -1)))
labels = nd.dot(poly_features, true_w)
labels += nd.random.normal(scale=0.1, shape=labels.shape)
```

최적화를 위해서, 그래디언트(gradient), 손실(loss) 등이 큰 값을 갖는 것을 피해야합니다. `poly_features`에 저장되는 단항들이  $x^i$ 에서  $\frac{1}{i!}x^i$ 로 스케일을 조정하는 이유입니다. 이렇게 하면 큰 차원  $i$ 의 값들이 아주 커지는 것을 방지할 수 있습니다. 팩토리얼은 Gluon의 Gamma 함수를 이용해서 구현합니다. ( $n! = \Gamma(n+1)$ )

생성된 데이터 셋에서 처음 두 샘플을 확인해봅니다. 값 1도 기술적으로 보면 하나의 특성(feature)으로, bias에 대한 상수 특성(feature)라고 볼 수 있습니다.

```
[3]: features[:2], poly_features[:2], labels[:2]
[3]: (
  [[-0.5095612 ]
   [ 0.34202248]]
```

(continues on next page)

```
<NDArray 2x1 @cpu(0)>,
[[ 1.0000000e+00 -5.09561181e-01  1.29826277e-01 -2.20514797e-02
  2.80914456e-03 -2.86286173e-04  2.43133891e-05 -1.76987987e-06
  1.12732764e-07 -6.38269260e-09  3.25237282e-10 -1.50662070e-11
  6.39762874e-13 -2.50767950e-14  9.12725858e-16 -3.10059752e-17
  9.87465261e-19 -2.95984643e-20  8.37901598e-22 -2.24716890e-23]
[ 1.0000000e+00  3.42022479e-01  5.84896803e-02  6.66826218e-03
  5.70173899e-04  3.90024616e-05  2.22328640e-06  1.08630559e-07
  4.64426186e-09  1.76493528e-10  6.03647601e-12  1.87691835e-13
  5.34956872e-15  1.40744067e-16  3.43840286e-18  7.84007342e-20
  1.67592618e-21  3.37178999e-23  6.40682210e-25  1.15330363e-26]]
```

<NDArray 2x20 @cpu(0)>, [3.8980482 5.3267784]

<NDArray 2 @cpu(0)>

## 모델 정의, 학습, 그리고 테스트

우선 그래프를 그리는 함수 `semilogy`를 정의합니다.  $y$  축은 로그(logarithm) 단위를 사용합니다.

```
[4]: # This function has been saved in the d2l package for future use
def semilogy(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
              legend=None, figsize=(3.5, 2.5)):
    d2l.set_figsize(figsize)
    d2l.plt.xlabel(x_label)
    d2l.plt.ylabel(y_label)
    d2l.plt.semilogy(x_vals, y_vals)
    if x2_vals and y2_vals:
        d2l.plt.semilogy(x2_vals, y2_vals, linestyle=':')
        d2l.plt.legend(legend)
```

선형 회귀(Linear regression)와 비슷하게, 다항 함수 학습에 제곱 손실 함수(squared loss function)를 이용하겠습니다. 생성된 데이터를 이용해서 여러 복잡도를 갖는 모델들을 학습시킬 것이기 때문에, 모델 정의를 `fit_and_plot` 함수에 전달하도록 하겠습니다. 다항 함수에 대한 학습과 테스트 단계는 softmax 회귀(regression)와 비슷합니다.

```
[5]: num_epochs, loss = 200, gloss.L2Loss()

def fit_and_plot(train_features, test_features, train_labels, test_labels):
    net = nn.Sequential()
    # Switch off the bias since we already catered for it in the polynomial
```

(continues on next page)

```

# features
net.add(nn.Dense(1, use_bias=False))
net.initialize()
batch_size = min(10, train_labels.shape[0])
train_iter = gdata.DataLoader(gdata.ArrayDataset(
    train_features, train_labels), batch_size, shuffle=True)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': 0.01})
train_ls, test_ls = [], []
for _ in range(num_epochs):
    for X, y in train_iter:
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    train_ls.append(loss(net(train_features),
                         train_labels).mean().asscalar())
    test_ls.append(loss(net(test_features),
                         test_labels).mean().asscalar())
print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
          range(1, num_epochs + 1), test_ls, ['train', 'test'])
print('weight:', net[0].weight.data().asnumpy())

```

### 3차 다항 함수 피팅(Third-order Polynomial Function Fitting (Normal))

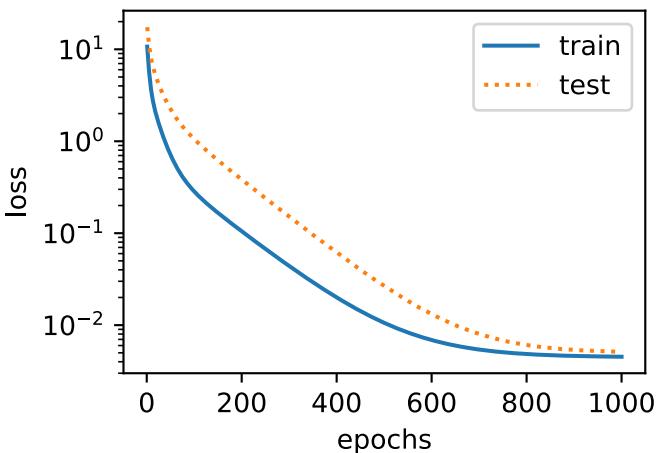
우선, 데이터를 생성한 것과 같은 3차원 다항함수를 이용해보겠습니다. 테스트 데이터를 이용해서 얻은 모델의 오류는 낮게 나오는 것이 보여집니다. 학습된 모델 파라미터 역시 실제 값  $w = [5, 1.2, -3.4, 5.6]$  과 비슷합니다.

```

[6]: num_epochs = 1000
# Pick the first four dimensions, i.e. 1, x, x^2, x^3 from the polynomial
# features
fit_and_plot(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
             labels[:n_train], labels[n_train:])

final epoch: train loss 0.004533171 test loss 0.0051389267
weight: [[ 4.9952755  1.2208132 -3.3927262  5.5615196]]

```

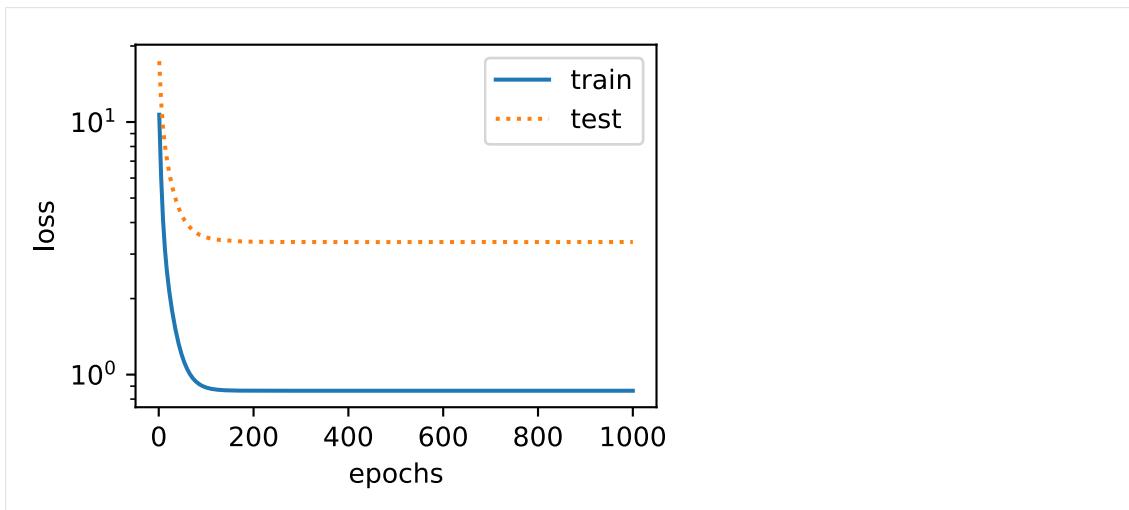


### 선형 함수 피팅 (언더피팅, underfitting)

선형 함수의 경우를 보겠습니다. 초기 에포크(epoch)를 수행하면서 학습 오류가 감소한 후로 더 이상 모델 학습의 오류가 감소하지 않는 것은 자연스러운 현상입니다. 마지막 epoch까지 마친 후에도 학습 오류는 여전히 높습니다. 선형 모델은 비선형 모델(3차 다항 함수)로 만들어진 데이터 셋에 대해서 언더피팅(underfitting)에 민감합니다.

```
[7]: num_epochs = 1000
# Pick the first four dimensions, i.e. 1, x from the polynomial features
fit_and_plot(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
             labels[:n_train], labels[n_train:])

final epoch: train loss 0.86323035 test loss 3.349208
weight: [[ 4.926944   3.3796043 -2.6735153]]
```



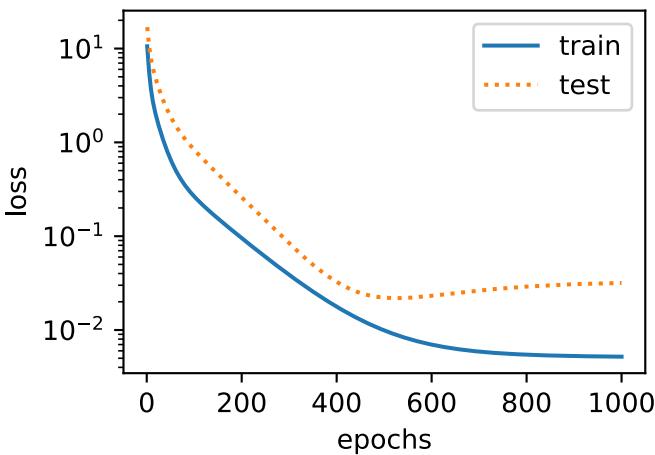
## 부족한 학습 (오버피팅, overfitting)

실제 상황에서, 데이터를 생성할 때 사용한 것과 같은 3차 다항 함수를 이용할 경우에도 학습을 충분히 오래하지 않은 경우에는 오버핏(overfit)이 쉽게 발생할 수 있습니다. 아주 높은 차원의 다항식을 사용해서 모델을 학습시켜보겠습니다. 모든 높은 차수의 계수들이 0에 가깝다는 사실을 학습하기에는 데이터가 너무 적습니다. 이 경우에는 모델이 너무 복잡해서 학습 데이터의 노이즈에 쉽게 영향을 받는 결과가 나옵니다. 학습 오류가 낮을지라도, 테스트 오류는 여전히 높습니다.

다른 모델의 복잡도 (`n_degreee`)와 학습 셋 크기(`n_subset`)를 적용해서 어떤 일이 발생하는지에 대한 직감을 얻어보세요.

```
[8]: num_epochs = 1000
n_subset = 100 # Subset of data to train on
n_degree = 20 # Degree of polynomials
fit_and_plot(poly_features[1:n_subset, 0:n_degree],
             poly_features[n_train:, 0:n_degree], labels[1:n_subset],
             labels[n_train:])

final epoch: train loss 0.0051964703 test loss 0.031715635
weight: [[ 4.9595799e+00  1.2577659e+00 -3.2017481e+00  5.2781754e+00
   -6.3637513e-01  1.3321723e+00 -1.6429817e-02  2.0419054e-01
   -6.1866865e-02  5.8488604e-02 -3.7163254e-02 -6.7995429e-02
   3.7113726e-02 -1.9592566e-02  6.2177785e-02  3.2198679e-02
   3.4999892e-02 -4.5971844e-02 -2.2483468e-02  2.9451251e-03]]
```



다음 장들에서 오버피팅(overfitting) 문제들을 계속 논의하고, 이를 해결하기 위한 가중치 감쇠(weight decay)와 드롭아웃(dropout)과 같은 방법을 알아보겠습니다.

### 5.11.5 요약

- 일반화 오류율은 학습 오류율을 이용해서 추정될 수 없기 때문에, 단순히 학습 오류율을 줄이는 것이 일반화 오류를 줄이는 것을 의미하지 않습니다. 머신 러닝 모델은 일반화 오류를 줄이기를 통해서 오버피팅(overfitting)에 조심스럽게 대비 해야합니다.
- 검증 셋은 모델 선택에 사용됩니다. (너무 남용되지 않는다는 가정에서)
- 언더피팅(underfitting)은 모델이 학습 오류를 줄이지 못하는 상황을 의미하고, 오버피팅(overfitting)은 모델 학습 오류가 테스트 데이터의 오류보다 훨씬 작은 경우를 의미합니다.
- 우리는 적절한 모델의 복잡성을 선택해야하고, 부족한 학습 샘플을 이용하는 것을 피해야합니다.

### 5.11.6 문제

- 다항 회귀 문제를 정확하게 풀 수 있나요? 힌트 - 선형대수를 이용합니다.
- 다항식에 대한 모델 선택에 대해서
  - 학습 오류와 모델 복잡도(다항식의 차원 수)를 도식화해보세요. 무엇이 관찰되나요?

- 이 경우 테스트 오류를 도식화해보세요.
  - 같은 그래프를 데이터 양에 따라서 그려보세요.
3. 다항식의 특성(feature)  $x^i$ 에 적용한 정규화  $1/i!$ 를 제거하면 어떤 일이 일어날까요? 다른 방법으로 이를 해결할 수 있나요?
  4. 학습 오류를 0으로 줄이기 위해서 몇 차원을 사용하나요?
  5. 일반화 오류를 0으로 줄이는 것이 가능한가요?

### 5.11.7 Scan the QR Code to Discuss



## 5.12 가중치 감쇠 (weight decay)

앞 절에서 우리는 오버피팅(overfitting)에 대해서 알아봤고, 이를 해결하기 위해서 용량 제어(capacity control)의 필요성에 대해서도 이야기했습니다. 학습 데이터셋의 양을 늘리는 것은 오버피팅(overfitting) 문제를 해결할 수도 있지만, 학습 데이터를 추가로 확보하는 것은 일반적으로 어려운 일입니다. 그렇기 때문에, 사용하는 함수의 복잡도를 조정하는 것을 더 선호합니다. 구체적으로는 차수를 조정해서 다항식의 복잡도를 조절할 수 있는 것을 확인했습니다. 이 방법은 일차원 데이터를 다루는 문제에 대해서는 좋은 전략이 될 수 있지만, 이 방법은 쉽게 복잡해지기 때문에 관리가 어려워질 수 있고, 너무 투박한 방법입니다. 예를 들면,  $D$  차원 벡터의 경우,  $d$  차수에 대한 단항의 개수는  $\binom{D-1+d}{D-1}$  가 됩니다. 따라서, 여러 함수에 대한 제어를 하는 것보다는 함수의 복잡도를 조절할 수 있는 보다 정교한 틀이 필요합니다.

### 5.12.1 제곱 놈 정규화(squared norm regularization)

가장 많이 사용하는 기법 중에 하나로 가중치 감쇠(weight decay)가 있습니다. 이 방법은 모든 함수  $f$  들 중에서  $f = 0$  이 가장 간단한 형태라는 것에 작안하고 있습니다. 따라서, 0과 얼마나 가까운가를 이용해서 함수에 대한 측정을 할 수 있습니다. 이를 측정하는 방법은 다양한데 별도의 수학 분야가 존

재하기까지 합니다. 예를 들면, 이 문제에 대한 답을 찾는 것에 목적을 두고 있는 함수 분석과 Banach 공간 이론 (the theory of Banach spaces)를 들 수 있습니다.

우리의 목적을 위해서는 아주 간단한 것을 사용해도 충분합니다:

선형 함수  $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ 에서 가중치 벡터(weight vector)가 작을 경우 “이 함수는 간단하다”라고 간주합니다. 이것은  $\|\mathbf{w}\|^2$ 로 측정될 수 있습니다. 가중치 벡터(weight vector)를 크게 유지하는 방법 중에 하나는 손실(loss)을 최소화하는 문제에 이 값을 패널티(penalty)로 더하는 것입니다. 이렇게 하면, 가중치 벡터(weight vector)가 너무 커지면, 학습 알고리즘은 학습 오류를 최소화하는 것보다  $\mathbf{w}^\top \mathbf{x} + b$ 를 최소화하는데 우선 순위를 둘 것입니다. 이것이 바로 우리가 원하는 것입니다. 코드에서 이를 설명하기 위해서, 앞 절의 “*Linear Regression*”를 고려해보면, 손실(loss)은 다음과 같이 주어집니다.

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

위 수식에서  $\mathbf{x}^{(i)}$ 는 관찰들이고,  $y^{(i)}$ 는 label,  $(\mathbf{w}, b)$ 는 가중치와 편향(bias) 파라미터들입니다. 가중치 벡터(weight vector)의 크기에 대한 패널티를 주는 새로운 손실 함수(loss function)를 만들기 위해서,  $\|\mathbf{w}\|^2$ 를 더합니다. 하지만, 얼마나 더해야 할까요? 이를 조절하는 정규화 상수(regularization constant)인  $\lambda$  하이퍼파라미터(hyperparameter)가 그 역할을 합니다.

$$l(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

$\lambda \geq 0$ 는 정규화(regularization)의 정도를 조절합니다.  $\lambda = 0$ 인 경우, 원래의 손실 함수(loss function)가 되고,  $\lambda > 0$ 이면,  $\mathbf{w}$ 가 너무 커지지 않도록 강제합니다. 통찰력이 있는 분은 가중치 벡터(weight vector)를 왜 제곱을 하는지 의아해할 것입니다. 이는 두 가지 이유 때문인데, 하나는 미분 계산이 쉬워지기 때문에 연산의 편의성을 위함이고, 다른 하나는 작은 가중치 벡터(weight vector)들 보다 큰 가중치 벡터(weight vector)에 더 많은 패널티를 부여하는 것으로 통계적인 성능 향상을 얻기 위하는 것입니다. 확률적 경사 하강법(Stochastic gradient descent) 업데이트는 다음과 같이 이뤄집니다.

$$\mathbf{w} \leftarrow \left(1 - \frac{\eta \lambda}{|\mathcal{B}|}\right) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left( \mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right),$$

이전과 같이, 관찰된 값과 예측된 값의 차이에 따라서  $\mathbf{w}$ 를 업데이트합니다. 하지만,  $\mathbf{w}$ 의 크기를 0과 가까워지게 줄이고 있습니다. 즉, 가중치를 감쇠하게(decay) 만듭니다. 이것은 다행식에 파라미터 개수를 선택하는 것보다 더 편한 방법입니다. 특히,  $f$ 의 복잡도를 조절하는 연속성이 있는 방법을 갖게 되었습니다. 작은  $\lambda$  값은  $\mathbf{w}$ 를 적게 제약하는 반면, 큰 값은  $\mathbf{w}$ 를 많이 제약합니다. 편향(bias) 항 역시 큰 값을 갖기를 원하지 않기 때문에,  $b^2$ 를 패널티로 더하기도 합니다.

## 5.12.2 고차원 선형 회귀

고차원 회귀(regression)에서 생략할 정확한 차원을 선택하기 어려운데, 가중치 감쇠 정규화(weight-decay regularization)는 아주 간편한 대안이 됩니다. 왜 그런지를 지금부터 설명하겠습니다.. 우선, 아래 공식을 사용해서 데이터를 생성합니다.

$$y = 0.05 + \sum_{i=1}^d 0.01x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01)$$

즉, 이 식에서는 평균이 0이고 표준편차가 0.01인 가우시안(Gaussian) 노이즈를 추가했습니다. 오버피팅(overfitting)을 더 잘 재현하기 위해서, 차원  $d$  가 200인 고차원 문제를 선택하고, 적은 양의 학습 데이터(20개)를 사용하겠습니다. 이전과 같이 필요한 패키지를 import 합니다.

```
[1]: import sys
sys.path.insert(0, '..')

%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import data as gdata, loss as gloss, nn

n_train, n_test, num_inputs = 20, 100, 200
true_w, true_b = nd.ones((num_inputs, 1)) * 0.01, 0.05

features = nd.random.normal(shape=(n_train + n_test, num_inputs))
labels = nd.dot(features, true_w) + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
train_features, test_features = features[:n_train, :], features[n_train:, :]
train_labels, test_labels = labels[:n_train], labels[n_train:]
```

## 5.12.3 처음부터 구현하기

다음으로는 가중치 감쇠(weight decay)를 직접 구현해보겠습니다. 이를 위해서, 간단하게 타켓(target) 함수 다음에  $\ell_2$  패널티를 추가 손실 항목으로 더합니다. 제곱 놈(squared norm) 패널티라는 이름은 제곱수를 더하는 것,  $\sum_i x_i^2$ , 으로 부터 왔습니다. 이 외에도 여러가지 패널티들이 있습니다.  $\ell_p$  놈(norm)은 다음과 같이 정의됩니다.

$$\|\mathbf{x}\|_p^p := \sum_{i=1}^d |x_i|^p$$

## 파라미터 초기화하기

우선 모델 파라미터를 임의로 초기화하는 함수를 정의합니다. 이 함수는 각 파라미터에 그래디언트(gradient)를 붙입니다.

```
[2]: def init_params():
    w = nd.random.normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

## $\ell_2$ 놈 패널티(Norm Penalty) 정의하기

이 패널티를 정의하는 간단한 방법은 각 항을 모두 제곱하고 이를 더하는 것입니다. 수식이 멋지고 간단하게 보이기 위해서 2로 나눕니다.

```
[3]: def l2_penalty(w):
    return (w**2).sum() / 2
```

## 학습 및 테스트 정의하기

아래 코드는 학습 데이터셋과 테스트 데이터셋을 이용해서 모델을 학습시키고 테스트하는 함수를 정의합니다. 이전 절의 예와는 다르게, 여기서는  $\ell_2$  놈 패널티(norm penalty)를 최종 손실 함수(loss function)를 계산할 때 더합니다. 선형 네트워크와 제곱 손실(squared loss)은 이전과 같기 때문에, d2l.linreg 와 d2l.squared\_loss 를 import 해서 사용하겠습니다.

```
[4]: batch_size, num_epochs, lr = 1, 100, 0.003
net, loss = d2l.linreg, d2l.squared_loss
train_iter = gdata.DataLoader(gdata.ArrayDataset(
    train_features, train_labels), batch_size, shuffle=True)

def fit_and_plot(lambd):
    w, b = init_params()
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                # The L2 norm penalty term has been added
                pass
```

(continues on next page)

```

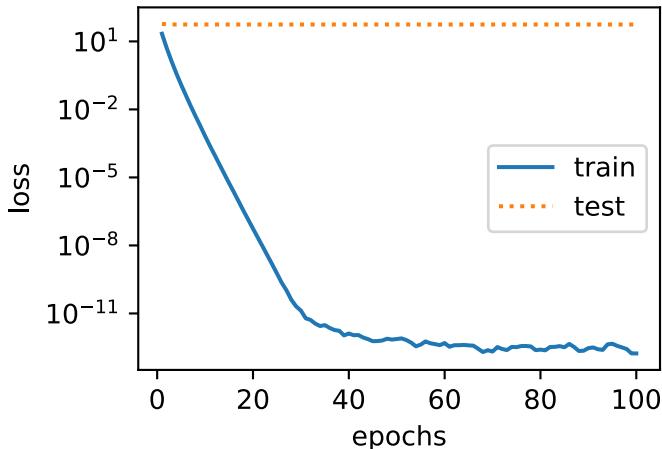
l = loss(net(X, w, b), y) + lambd * l2_penalty(w)
l.backward()
d2l.sgd([w, b], lr, batch_size)
train_ls.append(loss(net(train_features, w, b),
                     train_labels).mean().asscalar())
test_ls.append(loss(net(test_features, w, b),
                     test_labels).mean().asscalar())
d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
              range(1, num_epochs + 1), test_ls, ['train', 'test'])
print('l2 norm of w:', w.norm().asscalar())

```

## 정규화(regularization) 없이 학습하기

자 이제 고차원의 선형 회귀(linear regression) 모델을 학습시키고 테스트해봅니다.  $\text{lambd} = 0$ 인 경우에는 가중치 감쇠(weight decay)를 사용하지 않습니다. 그 결과로, 학습 오류가 줄어드는 반면, 테스트 오류는 줄어들지 않게 됩니다. 즉, 오버피팅(overfitting)의 완벽한 예제가 만들어졌습니다.

[5]: fit\_and\_plot(lambd=0)

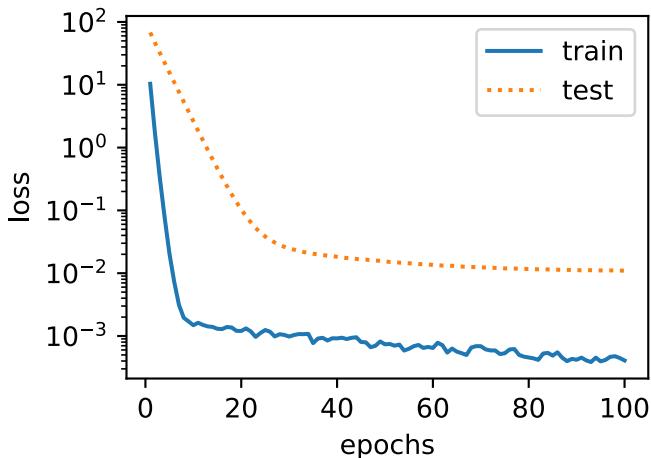


l2 norm of w: 11.611942

## 가중치 감쇠(weight decay) 사용하기

아래 예는 학습 오류는 증가하는 반면, 테스트 오류는 감소하는 것을 보여줍니다. 이것은 가중치 감쇠(weight decay)를 사용하면서 예상한 개선된 결과입니다. 완벽하지는 않지만, 오버피팅(overfitting) 문제가 어느정도 해결되었습니다. 추가로, 가중치  $w$ 에 대한  $\ell_2$  놈(norm)도 가중치 감쇠(weight decay)를 사용하지 않을 때보다 작아졌습니다.

```
[6]: fit_and_plot(lambda=3)
```



```
12 norm of w: 0.041329514
```

### 5.12.4 간결한 구현

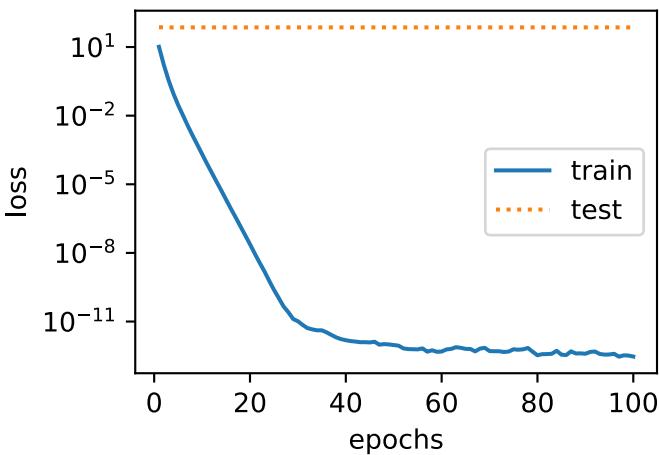
Gluon에는 최적화 알고리즘에 가중치 감쇠(weight decay)가 통합되어 있어 더 편하게 적용할 수 있습니다. 그 이유는 옵티마이저(optimizer)가 모든 파라미터를 직접 다루기 때문에, 옵티마이저(optimizer)가 가중치 감쇠(weight decay)를 직접 관리하고, 관련된 것을 최적화 알고리즘에서 다루는 것이 실행 속도면에서 더 빠르기 때문입니다.

아래 예제에서는 Trainer 인스턴스를 생성할 때, wd 파라미터를 통해서 가중치 감쇠(weight decay) 하이퍼파라미터(hyperparameter)를 직접 지정합니다. Gluon의 기본 설정은 가중치와 편향(bias)을 모두 감쇠(decay) 시킵니다. 다른 종류의 파라미터에 대해서 다른 옵티마이저(optimizer)를 사용할 수 있습니다. 예를 들면,  $w$ 에는 가중치 감쇠(weight decay)를 적용하는 Trainer를 하나 만들고,  $b$ 에는 가중치 감쇠(weight decay)를 적용하지 않은 다른 Trainer를 각각 만들 수 있습니다.

```
[7]: def fit_and_plot_gluon(wd):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    # The weight parameter has been decayed. Weight names generally end with
    # "weight".
    trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                               {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                               {'learning_rate': lr})
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
                l.backward()
            # Call the step function on each of the two Trainer instances to
            # update the weight and bias separately
            trainer_w.step(batch_size)
            trainer_b.step(batch_size)
        train_ls.append(loss(net(train_features),
                             train_labels).mean().asscalar())
        test_ls.append(loss(net(test_features),
                            test_labels).mean().asscalar())
    d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                 range(1, num_epochs + 1), test_ls, ['train', 'test'])
    print('L2 norm of w:', net[0].weight.data().norm().asscalar())
```

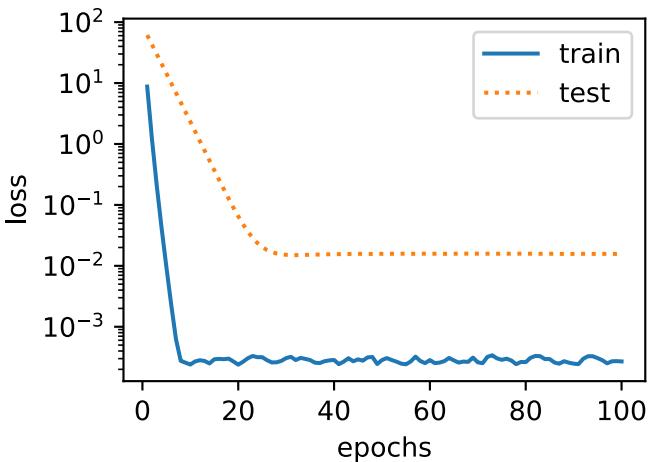
그래프는 가중치 감쇠(weight decay)를 직접 구현해서 얻었던 것과 아주 비슷하게 생겼습니다. 하지만, 더 빠르고 더 구현하기 쉬웠고, 큰 문제의 경우에는 더욱 그렇습니다.

```
[8]: fit_and_plot_gluon(0)
```



L2 norm of w: 13.311798

[9]: fit\_and\_plot\_gluon(3)



L2 norm of w: 0.033094507

지금까지 우리는 간단한 선형 함수를 구성하는 것들만을 다뤘습니다. 비선형 함수에 대해서 이것들을 다루는 것은 훨씬 더 복잡합니다. 예를 들어, [Reproducing Kernel Hilbert Spaces](#)라는 것이 있는데, 이를 이용하면 선형 함수에서 사용한 많은 도구들을 비선형에서 사용할 수 있게 해줍니다. 하지만 안타깝게도, 사용되는 알고리즘들이 데이터가 매우 많은 경우 잘 동작하지 않는 확장성 문제가 있습니다. 따라서, 이 책의 목적을 위해서 우리는 각 층의 가중치들을 단순히 더하는 방법,  $\sum_l \|w_l\|^2$  을 사용하

겠습니다. 이렇게 하는 것은 전체 층들에 가중치 감쇠(weight decay)를 적용하는 것과 같습니다.

### 5.12.5 요약

- 정규화(regularization)은 오버피팅(overfitting)을 다루는 일반적인 방법입니다. 학습된 모델의 복잡도를 줄이기 위해서 학습 데이터에 대한 손실 함수(loss function)의 값에 패널티 항목을 더합니다.
- 모델을 간단하게 유지하는 방법으로  $\ell_2$  놈 패널티(norm penalty)를 사용하는 가중치 감쇠(weight decay)를 선택했습니다. 이를 통해서, 학습 알고리즘의 업데이트 단계에서 가중치 감쇠(weight decay)가 적용됩니다.
- Gluon은 옵티마이저(optimizer)에 하이퍼파라미터(hyperparameter) `wd` 를 설정하는 것으로 가중치 감쇠(weight decay) 기능을 자동으로 추가할 수 있습니다.
- 같은 학습에서 파라미터마다 다른 옵티마이저(optimizer)를 적용할 수 있습니다.

### 5.12.6 문제

1. 이 장의 예측 문제에서  $\lambda$  값을 실험해보세요.  $\lambda$ 에 대한 함수의 형태로 학습 정확도와 테스트 정확도를 도식화해보세요. 어떤 것이 관찰되나요?
2. 검증 데이터셋을 이용해서 최적의  $\lambda$  값을 찾아보세요. 찾은 값이 진짜 최적값인가요? 진짜 값을 찾는 것이 중요한가요?
3. 패널티 항목으로  $\|\mathbf{w}\|^2$  대신  $\sum_i |w_i|$  를 사용하면 업데이트 공식이 어떻게 될까요? ( $i$ 는  $\ell_1$  정규화(regularization)라고 합니다.)
4.  $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$  입니다. 행렬에서 비슷한 공식을 찾아볼 수 있나요? (수학자들은 이를 Frobenius norm 이라고 합니다)
5. 학습 오류와 일반화 오류의 관계를 복습해보세요. 가중치 감쇠(weight decay), 학습 데이터셋 늘리기, 적당한 복잡도를 갖는 모델 사용하기 외에, 오버피팅(overfitting)을 다룰 수 있는 방법이 어떤 것들이 있을까요?
6. 베이시안 통계에서, prior 와 likelihood 곱을 이용해서 posterior를 구할 수 있습니다.  $p(w|x) \propto p(x|w)p(w)$ .  $p(w)$  가 정규화(regularization)와 어떻게 동일할까요?

### 5.12.7 Scan the QR Code to Discuss



## 5.13 드롭아웃(dropout)

앞에서 우리는 통계적인 모델을 정규화(regularize)하는 전통적인 방법을 알아봤습니다. 가중치의 크기 ( $\ell_2$  norm)을 패널티로 사용해서, 가중치 값이 강제로 작아지도록 했습니다. 확률적인 용어로 말하자면, 가우시안 프리어(Gaussian prior)를 가중치 값에 적용한다고 할 수 있습니다. 하지만, 더 직관적인 함수의 용어를 사용하면, 가중치 값들이 다른 특성(feature)들 사이에 더 확산되도록 하고, 임자적으로 가짜 연관들에 더 적게 의존되도록 모델을 학습시킨다고 할 수 있습니다.

### 5.13.1 오버피팅 다시 살펴보기

뛰어난 유연성은 오버피팅(overfitting)에 대한 책임이 따릅니다.

샘플들 보다 더 많은 특성(feature)들이 주어지면, 선형 모델은 오버핏(overfit) 될 수 있습니다. 반면에 특성(feature) 수 보다 샘플이 더 많은 경우에는 선형 모델은 일반적으로 오버핏(overfit) 되지 않습니다. 아쉽게도, 일반화를 잘하기 위해서는 그에 따른 비용이 들어갑니다. 매 특성(feature)에 대해서, 선형 모델은 양수 또는 음수의 가중치를 할당 해야합니다. 선형 모델은 특성(feature)들 사이의 미묘한 상호 작용을 설명하지 못 합니다. 좀 더 공식적인 용어로 이야기하면, 편향-분산 트레이드오프(bias-variance tradeoff)로 논의되는 현상을 볼 것입니다. 선형 모델은 높은 편향(bias) (표현할 수 있는 함수의 개수가 적습니다)를 보이나, 분산(variance)은 낮습니다 (다른 랜덤 샘플 데이터에 대해서 비슷한 결과를 줍니다)

반면에 딥 뉴럴 네트워크는 편향-분산(bias-variance) 스펙트럼에서 반대의 현상을 보입니다. 뉴럴 네트워크는 각 특성(feature)을 독립적으로 보는 제약이 없기 때문에 유연합니다. 대신, 특성(feature)들의 그룹들에서 복잡한 상관관계를 학습할 수 있습니다. 예를 들면, 뉴럴 네트워크는 “Nigeria”와 “Western Union”이 이메일에 함께 나오면 그 이메일을 스팸으로 판단하고, “Western Union”이라는 단어가 없이 “Nigeria”가 등장하는 이메일은 스팸이 아니라고 판단할 수 있습니다.

특성(feature)들의 개수가 적은 경우에도, 딥 뉴럴 네트워크는 오버피팅(overfitting) 될 수 있습니다. 뉴럴 네트워크의 뛰어난 유연성을 보여주는 예로, 연구자들은 임의로 레이블(label)이 할당된 데이터

를 완벽하게 분류하는 것을 입증했습니다. 이것이 무엇을 뜻하는지 생각해봅시다. 10개의 분류로 된 레이블들이 균일하게 임의로 부여되어 있는 경우, 어떤 분류기도 10% 이상의 정확도를 얻을 수 없습니다. 이렇게 패턴을 학습할 수 없는 경우에도, 뉴럴 네트워크는 학습 레이블들에 완벽하게 맞춰질 수 있습니다.

### 5.13.2 변화를 통한 견고함

좋은 통계적인 모델로 부터 무엇을 기대할 수 있는지에 대해서 간단히 알아보겠습니다. 당연하게 우리는 이 모델이 보지 않은 테스트 데이터에 대해서 잘 작동하기를 기대합니다. 이를 달성하는 방법 중에 하나로 어떤 것이 “간단한” 모델을 만드는지를 묻는 것입니다. 단항 함수 (monomial basis function) 을 사용해서 모델을 학습시키면서 언급했던 것처럼 차원의 수가 적은 것으로부터 간단함이 유도될 수 있습니다. 또한 간단함은 기본이 되는 함수의 작은 놈(norm)의 형태로 만들어질 수도 있습니다. 즉, 가중치 감쇠(weight decay)와  $\ell_2$  정규화(regularization)이 그런 예입니다. 간단함을 만드는 또 다른 요소는 입력의 완만한 변화에도 큰 영향을 받지 않는 함수를 들 수 있습니다. 예를 들어 이미지를 분류할 때, 몇개의 픽셀들의 변경으로 인해서 결과에 영향을 미치지 않기를 기대하는 것입니다.

사실 이 개념은 1995년 Bishop<sup>10</sup> [Training with Input Noise is Equivalent to Tikhonov Regularization](#) 를 증명하면서 공식화 되었습니다. 즉, 그는 부드러운 (따라서 간단한) 함수의 개념을 입력의 변화에 탄력적인 것과 연관을 시켰습니다. 2014년으로 훌러가서, 여러층을 갖는 딥 네트워크의 복잡도가 주어졌을 때, 입력에 부드러움을 강제하는 것은 꼭 다음 층들에서도 보장되지는 않습니다. Srivastava et al., 2014 에서 발표된 독창적인 아이디어는 Bishop의 아이디어를 네트워크의 내부층들에 적용했습니다. 이는, 학습 과정에 네트워크 연산 경로에 노이즈를 집어넣는 것입니다.

여기서 주요 과제는 지나친 편향(bias)을 추가하지 않으면서 어떻게 노이즈를 추가하는지입니다. 입력  $\mathbf{x}$ 에 대해서는 노이즈를 추가하는 것은 상대적으로 간단합니다. 즉,  $\epsilon \sim \mathcal{N}(0, \sigma^2)$  노이즈를 입력에 더한 후  $\mathbf{x}' = \mathbf{x} + \epsilon$  이 것을 학습 데이터로 사용하면 됩니다. 이렇게 했을 때 주요 특징은  $\mathbf{E}[\mathbf{x}'] = \mathbf{x}$  을 갖는 것입니다. 하지만, 중간층들에서는 이 노이즈의 스캐일이 적절하지 않을 수 있기 때문에 이 특징을 기대하기 어렵습니다. 대안은 다음과 같이 좌표를 뒤틀어 놓는 것입니다.

$$h' = \begin{cases} 0 & \text{확률 } p \text{ 인 경우} \\ \frac{h}{1-p} & \text{그 외의 경우} \end{cases}$$

설계상으로는 기대값이 변하지 않습니다. 즉,  $\mathbf{E}[h'] = h$ 입니다. 중간 레이어들에 적용되는 활성화(activation)  $h$  를 같은 기대값을 갖는 랜덤 변수  $h'$ 로 바꾸는 것이 드롭아웃(dropout)의 핵심 아이디어입니다. ‘드롭아웃(dropout)’이라는 이름은 마지막 결과를 계산하기 위해서 사용되는 연산의 몇몇 뉴런들을 누락(drop out) 시킨다는 개념에서 왔습니다. 학습 과정에서, 중간의 활성화(activation)들을 활률 변수로 바꿉니다.

### 5.13.3 드롭아웃(dropout) 실제 적용하기

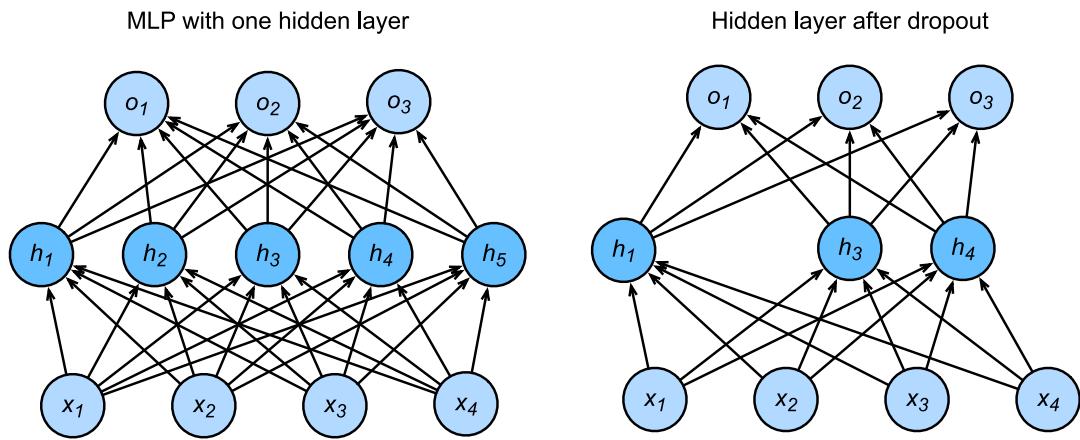
5개의 은닉 유닛(hidden unit)을 갖는 한개의 은닉층을 사용하는 [다층 퍼셉트론\(multilayer perceptron\)](#)의 예를 다시 들어보겠습니다. 이 네트워크의 아키텍처는 다음과 같이 표현됩니다.

$$h = \sigma(W_1x + b_1)$$

$$o = W_2h + b_2$$

$$\hat{y} = \text{softmax}(o)$$

은닉층에 드롭아웃(dropout)을 확률  $p$ 로 적용하는 경우, 은닉 유닛들을  $p$  확률로 제거하는 것이 됩니다. 이유는, 그 확률을 이용해서 출력을 0으로 설정하기 때문입니다. 이를 적용한 네트워크는 아래 그림과 같습니다. 여기서  $h_2$  와  $h_5$  가 제거되었습니다. 결과적으로  $y$ 를 계산할 때,  $h_2$  와  $h_5$  는 사용되지 않게 되고, 역전파(backprop)을 수행할 때 이 것들에 대한 그래디언트(gradient)들도 적용되지 않습니다. 이렇게 해서 출력층으로 계산할 때  $h_1, \dots, h_5$  중 어느 하나에 전적으로 의존되지 않게 합니다. 이것이 오버피팅(overfitting) 문제를 해결하는 정규화(regularization) 목적을 위해서 필요한 것입니다. 테스트 시에는, 더 확실한 결과를 얻기 위해서 드롭아웃(dropout)을 사용하지 않는 것이 일반적입니다.



### 5.13.4 직접 구현하기

드롭아웃(dropout)를 구현하기 위해서는 입력 개수 만큼의 확률 변수를 균일한 분포  $U[0, 1]$ 에서 추출해야합니다. 드롭아웃(dropout)의 정의에 따르면, 이를 간단하게 구현할 수 있습니다. 다음 드롭아웃(dropout) 함수는 NDArray 입력  $x$ 의 원소들을  $\text{drop\_prob}$  확률로 누락시킵니다.

```
[1]: import sys
sys.path.insert(0, '..')

import d2l
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import loss as gloss, nn

def dropout(X, drop_prob):
    assert 0 <= drop_prob <= 1
    # In this case, all elements are dropped out
    if drop_prob == 1:
        return X.zeros_like()
    mask = nd.random.uniform(0, 1, X.shape) > drop_prob
    return mask * X / (1.0-drop_prob)
```

몇 가지 예제에 적용해서 어떻게 동작하는지 살펴보겠습니다. 드롭아웃(dropout) 확률을 각각 0, 0.5, 그리고 1로 설정해봅니다.

```
[2]: X = nd.arange(16).reshape((2, 8))
print(dropout(X, 0))
print(dropout(X, 0.5))
print(dropout(X, 1))

[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
<NDArray 2x8 @cpu(0)>

[[ 0.  0.  0.  0.  8. 10. 12.  0.]
 [16.  0. 20. 22.  0.  0.  0. 30.]]
<NDArray 2x8 @cpu(0)>

[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
<NDArray 2x8 @cpu(0)>
```

### 5.13.5 모델 파라미터 정의하기

“Softmax 회귀(regression)를 처음부터 구현하기” 절에서 사용한 Fashion-MNIST 데이터셋을 다시 사용합니다. 두개의 은닉층들을 갖는 다층 퍼셉트론(multilayer perceptron)을 정의하는데, 각 은닉층은

256개의 결과를 출력합니다.

```
[3]: num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens1))
b1 = nd.zeros(num_hiddens1)
W2 = nd.random.normal(scale=0.01, shape=(num_hiddens1, num_hiddens2))
b2 = nd.zeros(num_hiddens2)
W3 = nd.random.normal(scale=0.01, shape=(num_hiddens2, num_outputs))
b3 = nd.zeros(num_outputs)

params = [W1, b1, W2, b2, W3, b3]
for param in params:
    param.attach_grad()
```

## 5.13.6 모델 정의하기

정의하는 모델은 각 활성화 함수(activation function)의 결과에 드롭아웃(dropout)을 적용하면서 완전 연결층(fully connected layer)과 활성화 함수(activation function) ReLU를 연결하도록 되어 있습니다. 각 층에 서로 다른 드롭아웃(dropout) 확률을 설정할 수 있습니다. 일반적으로는 입력층에 가까울수록 낮은 드롭아웃(dropout) 확률값을 사용하는 것을 권장합니다. 아래 모델에서는 첫번째 층에는 0.2를 두번째 층에는 0.5를 적용하고 있습니다. “Autograd” 절에서 정의한 `is_training` 을 사용하면, 학습할 때만 드롭아웃(dropout) 이 적용될 수 있게 할 수 있습니다.

```
[4]: drop_prob1, drop_prob2 = 0.2, 0.5

def net(X):
    X = X.reshape((-1, num_inputs))
    H1 = (nd.dot(X, W1) + b1).relu()
    # Use dropout only when training the model
    if autograd.is_training():
        # Add a dropout layer after the first fully connected layer
        H1 = dropout(H1, drop_prob1)
    H2 = (nd.dot(H1, W2) + b2).relu()
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
        H2 = dropout(H2, drop_prob2)
    return nd.dot(H2, W3) + b3
```

### 5.13.7 학습 및 테스트

다층 퍼셉트론(multilayer perceptron)의 학습과 테스트는 이전에 설명한 것과 비슷합니다.

```
[5]: num_epochs, lr, batch_size = 10, 0.5, 256
loss = gloss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
              params, lr)

epoch 1, loss 1.1395, train acc 0.557, test acc 0.804
epoch 2, loss 0.5754, train acc 0.786, test acc 0.837
epoch 3, loss 0.4932, train acc 0.820, test acc 0.855
epoch 4, loss 0.4438, train acc 0.840, test acc 0.858
epoch 5, loss 0.4192, train acc 0.847, test acc 0.859
epoch 6, loss 0.3984, train acc 0.854, test acc 0.865
epoch 7, loss 0.3826, train acc 0.861, test acc 0.866
epoch 8, loss 0.3699, train acc 0.863, test acc 0.876
epoch 9, loss 0.3595, train acc 0.869, test acc 0.877
epoch 10, loss 0.3497, train acc 0.871, test acc 0.873
```

### 5.13.8 간결한 구현

Gluon을 이용하면, 완전 연결층(fully connected layer) 다음에 드롭아웃(dropout) 확률값을 주면서 드롭아웃(dropout) 층을 추가하기만 하면 됩니다. 모델을 학습시킬 때 드롭아웃(dropout) 층은 명시된 드롭아웃(dropout) 확률에 따라서 결과 원소들을 임의로 누락시켜주고, 테스트를 수행할 때는 데이터를 그냥 통과 시킵니다.

```
[6]: net = nn.Sequential()
net.add(nn.Dense(256, activation="relu"),
       # Add a dropout layer after the first fully connected layer
       nn.Dropout(drop_prob1),
       nn.Dense(256, activation="relu"),
       # Add a dropout layer after the second fully connected layer
       nn.Dropout(drop_prob2),
       nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

다음으로 모델을 학습시키고 테스트를 수행합니다.

```
[7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
              None, trainer)

epoch 1, loss 1.2567, train acc 0.519, test acc 0.746
epoch 2, loss 0.6010, train acc 0.776, test acc 0.830
epoch 3, loss 0.5051, train acc 0.815, test acc 0.844
epoch 4, loss 0.4591, train acc 0.831, test acc 0.848
epoch 5, loss 0.4304, train acc 0.843, test acc 0.855
epoch 6, loss 0.4078, train acc 0.853, test acc 0.866
epoch 7, loss 0.3908, train acc 0.859, test acc 0.867
epoch 8, loss 0.3792, train acc 0.862, test acc 0.870
epoch 9, loss 0.3650, train acc 0.867, test acc 0.876
epoch 10, loss 0.3547, train acc 0.870, test acc 0.877
```

### 5.13.9 요약

- 차원 수를 조절하고 가중치 벡터(weight vector)의 크기를 제어하는 것 이외에, 드롭아웃(dropout)은 오버피팅(overfitting)을 해결하는 또 다른 방법입니다. 이 세가지는 종종 함께 사용됩니다.
- 드롭아웃(dropout)은  $h$ 를 같은 기대값  $h$ 를 갖는 확률 변수  $h'$ 로 드롭아웃(dropout) 확률  $p$  만큼 바꾸는 것입니다.
- 드롭아웃(dropout)은 학습에만 적용합니다.

### 5.13.10 문제

- 층 1과 2에서 드롭아웃(dropout) 확률값을 바꾸면서 그 결과를 관찰해보세요. 특히, 두 층에 대한 드롭아웃(dropout) 확률을 동시에 바꾸면 어떻게 될까요?
- 에포크(epoch) 수를 늘리면서 드롭아웃(dropout)을 적용할 때와 적용하지 않을 때의 결과를 비교해보세요.
- 드롭아웃(dropout)을 적용한 후, 활성화(activation) 확률 변수의 편차를 계산해보세요.
- 왜 일반적으로 드롭아웃(dropout)을 사용하지 않아야 하나요?
- 은닉층 유닛(hideen layer unit)을 추가하는 것처럼 모델의 복잡도를 높이는 변경을 할 때, 드롭아웃(dropout)을 사용하는 효과가 오버피팅(overfitting) 문제를 해결하는 더 확실한가요?

6. 위 예제를 이용해서 드롭아웃(dropout)과 가중치 감쇠(weight decay) 효과를 비교해보세요.
7. 활성화 결과가 아니라 가중치 행렬(weight matrix)의 각 가중치에 적용하면 어떻게 될까요?
8.  $[0, \gamma/2, \gamma]$ 에서 추출한 값을 갖도록 드롭아웃(dropout)을 바꿔보세요. 이진 드롭아웃(binary dropout) 함수보다 더 좋은 것을 만들어볼 수 있나요? 왜 그런 방법을 사용할 것인가요? 왜 아닌가요?

### 5.13.11 참고자료

[1] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). JMLR

### 5.13.12 Scan the QR Code to Discuss



## 5.14 순전파(forward propagation), 역전파(back propagation), 연산 그래프

앞에서 우리는 모델을 학습 시키는 방법으로 미니 배치 확률적 경사 강하법(stochastic gradient descent) 최적화 알고리즘을 사용했습니다. 이를 구현할 때, 우리는 모델의 순전파(forward propagation)을 계산하면서 입력에 대한 모델의 결과만을 계산했습니다. 그리고, 자동으로 생성된 backward 함수를 호출함으로 autograd 을 이용해서 gradient를 계산합니다. 역전파(back-propagation)을 이용하는 경우 자동으로 그래디언트(gradient)를 계산하는 함수를 이용함으로 딥러닝 학습 알고리즘 구현이 굉장히 간단해졌습니다. 이 절에서는 순전파(forward propagation)와 역전파(back propagation)를 수학적이고 연산적인 그래프를 사용해서 설명하겠습니다. 더 정확하게는 한개의 은닉층(hidden layer)를 갖는 다층 퍼셉트론(multilayer perceptron)에  $\ell_2$  놈 정규화(norm regularization)를 적용한 간단한 모델을 이용해서 순전파(forward propagation)와 역전파(back propagation)를 설명합니다. 이 절은 딥러닝을 수행할 때 어떤 일이 일어나고 있는지에 대해서 더 잘 이해할 수 있도록 해줄 것입니다.

### 5.14.1 순전파(forward propagation)

순전파(forward propagation)은 뉴럴 네트워크 모델의 입력층부터 출력층까지 순서대로 변수들을 계산하고 저장하는 것을 의미합니다. 지금부터 한개의 은닉층(hidden layer)을 갖는 딥 네트워크를 예로 들어 단계별로 어떻게 계산되는지 설명하겠습니다. 다소 지루할 수 있지만, backward 를 호출했을 때, 어떤 일이 일어나는지 논의할 때 도움이 될 것입니다.

간단하게 하기 위해서, 입력은  $d$  차원의 실수 공간  $\mathbf{x} \in \mathbb{R}^d$  으로 부터 선택되고, 편향(bias) 항목은 생략하겠습니다. 중간 변수는 다음과 같이 정의됩니다.

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$$

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  은 은닉층(hidden layer)의 가중치 파라미터입니다. 중간 변수  $\mathbf{z} \in \mathbb{R}^h$  를 활성화 함수(activation function)  $\phi$  에 입력해서 벡터 길이가  $h$  인 은닉층(hidden layer) 변수를 얻습니다.

$$\mathbf{h} = \phi(\mathbf{z}).$$

은닉 변수  $\mathbf{h}$  도 중간 변수입니다. 출력층의 가중치  $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$  만을 사용한다고 가정하면, 벡터 길이가  $q$  인 출력층의 변수를 다음과 같이 계산할 수 있습니다.

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

손실 함수(loss function)를  $l$  이라고 하고, 샘플 레이블을  $y$  라고 가정하면, 하나의 데이터 샘플에 대한 손실(loss) 값을 다음과 같이 계산할 수 있습니다.

$$L = l(\mathbf{o}, y).$$

$\ell_2$  놈 정규화(norm regularization)의 정의에 따라서, 하이퍼파라미터(hyper-parameter)  $\lambda$  가 주어졌을 때, 정규화(regularization) 항목은 다음과 같습니다.

$$s = \frac{\lambda}{2} \left( \|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right),$$

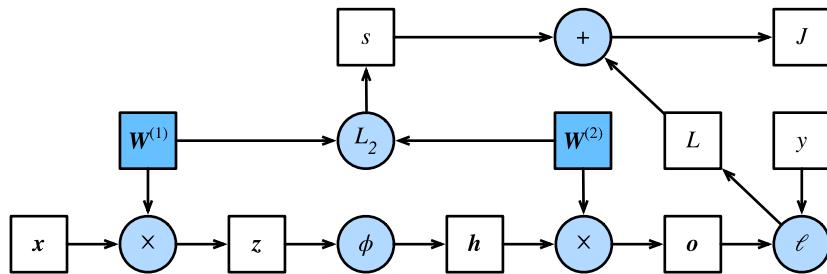
여기서 행렬의 Frobenius norm은 행렬을 벡터로 바꾼 후 계산하는  $L_2$  놈(norm)과 같습니다. 마지막으로, 한개의 데이터 샘플에 대한 모델의 정규화된 손실(regularized loss) 값을 계산합니다.

$$J = L + s.$$

$J$  를 주어진 데이터 샘플에 대한 목표 함수(objective function)라고 하며, 앞으로 이를 '목표 함수(objective function)'라고 하겠습니다.

### 5.14.2 순전파(forward propagation)의 연산 그래프

연산 그래프를 도식화하면 연산에 포함된 연산자와 변수들 사이의 관계를 시각화하는데 도움이 됩니다. 아래 그림은 위에서 정의한 간단한 네트워크의 그래프입니다. 왼쪽 아래는 입력이고, 오른쪽 위는 출력입니다. 데이터의 흐름을 표시하는 화살표의 방향이 오른쪽과 위로 향해 있습니다.



### 5.14.3 역전파(back propagation)

역전파(back propagation)는 뉴럴 네트워크의 파라미터들에 대한 그래디언트(gradient)를 계산하는 방법을 의미합니다. 일반적으로는 역전파(back propagation)은 뉴럴 네트워크의 각 층과 관련된 목적 함수(objective function)의 중간 변수들과 파라미터들의 그래디언트(gradient)를 출력층에서 입력층 순으로 계산하고 저장합니다. 이는 미적분의 '체인룰(chain rule)'을 따르기 때문입니다. 임의의 모양을 갖는 입력과 출력 텐서(tensor)  $X, Y, Z$  를 이용해서 함수  $Y = f(X)$  와  $Z = g(Y) = g \circ f(X)$  를 정의했다고 가정하고, 체인룰(chain rule)을 사용하면,  $X$  에 대한  $Z$  의 미분은 다음과 같이 정의됩니다.

$$\frac{\partial Z}{\partial X} = \text{prod} \left( \frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right).$$

여기서  $\text{prod}$  연산은 전치(transposition)나 입력 위치 변경과 같이 필요한 연산을 수행한 후 곱을 수행하는 것을 의미합니다. 벡터의 경우에는 이것은 직관적입니다. 단순히 행렬-행렬 곱셈이고, 고차원의 텐서의 경우에는 새로 대응하는 원소들 간에 연산을 수행합니다.  $\text{prod}$  연산자는 이 모든 복잡한 개념을 감춰주는 역할을 합니다.

하나의 은닉층(hidden layer)를 갖는 간단한 네트워크의 파라미터는  $W^{(1)}$  와  $W^{(2)}$  이고, 역전파(back propagation)는 미분값  $\partial J / \partial W^{(1)}$  와  $\partial J / \partial W^{(2)}$  를 계산하는 것입니다. 이를 위해서 우리는 체인룰(chain rule)을 적용해서 각 중간 변수와 파라미터에 대한 그래디언트(gradient)를 계산합니다. 연산

그래프의 결과로부터 시작해서 파라미터들에 대한 그래디언트(gradient)를 계산해야하기 때문에, 순전파(forward propagation)와는 반대 방향으로 연산을 수행합니다. 첫번째 단계는 손실(loss) 항목  $L$  과 정규화(regularization) 항목  $s$ 에 대해서 목적 함수(objective function)  $J = L + s$ 의 그래디언트(gradient)를 계산하는 것입니다.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1$$

그 다음, 출력층  $o$ 의 변수들에 대한 목적 함수(objective function)의 그래디언트(gradient)를 체인룰(chain rule)을 적용해서 구합니다.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left( \frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}} \right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$$

이제 두 파라미터에 대해서 정규화(regularization) 항목의 그래디언트(gradient)를 계산합니다.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$$

이제 우리는 출력층과 가장 가까운 모델 파라미터들에 대해서 목적 함수(objective function)의 그래디언트(gradient)  $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ 를 계산할 수 있습니다. 체인룰(chain rule)을 적용하면 다음과 같이 계산됩니다.

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}} \right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

$\mathbf{W}^{(1)}$ 에 대한 그래디언트(gradient)를 계산하기 위해서, 출력층으로부터 은닉층까지 역전파(back propagation)를 계속 해야합니다. 은닉층(hidden layer) 변수에 대한 그래디언트(gradient)  $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$ 는 다음과 같습니다.

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}.$$

활성화 함수(activation function)  $\phi$ 는 각 요소별로 적용되기 때문에, 중간 변수  $\mathbf{z}$ 에 대한 그래디언트(gradient)  $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$ 를 계산하기 위해서는 요소별 곱하기(element-wise multiplication) 연산자를 사용해야합니다. 우리는 이 연산을  $\odot$ 로 표현하겠습니다.

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}).$$

마지막으로, 입력층과 가장 가까운 모델 파라미터에 대한 그래디언트(gradient)  $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ 를

체인룰(chain rule)을 적용해서 다음과 같이 계산합니다.

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}} \right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

#### 5.14.4 모델 학습시키기

네트워크를 학습시킬 때, 순전파(forward propagation)과 역전파(backward propagation)은 서로 의존하는 관계입니다. 특히 역전파(forward propagation)는 연관되는 관계를 따라서 그래프를 계산하고, 그 경로의 모든 변수를 계산합니다. 이것들은 연산이 반대 방향인 역전파(back propagation)에서 다시 사용됩니다. 그 결과 중에 하나로 역전파(back propagation)을 완료할 때까지 중간 값들을 모두 가지고 있어야하는 것이 있습니다. 이것이 역전파(back propagation)가 단순 예측을 수행할 때보다 훨씬 더 많은 메모리를 사용하는 이유들 중에 하나입니다. 즉, 체인룰(chain rule)을 적용하기 위해서 모든 중간 변수를 저장하고 있어야, 그래디언트(gradients)인 텐서(tensor)들을 계산할 수 있습니다. 메모리를 더 많이 사용하는 다른 이유는 모델을 학습 시킬 때 미니 배치 형태로 하기 때문에, 더 많은 중간 활성화(activation)들을 저장해야하는 것입니다.

#### 5.14.5 요약

- 순전파(forwards propagation)은 뉴럴 네트워크의 그래프를 계산하기 위해서 중간 변수들을 순서대로 계산하고 저장합니다. 즉, 입력층부터 시작해서 출력층까지 처리합니다.
- 역전파(back propagation)은 중간 변수와 파라미터에 대한 그래디언트(gradients)를 반대 방향으로 계산하고 저장합니다.
- 딥러닝 모델을 학습시킬 때, 순전파(forward propagation)과 역전파(back propagation)는 상호 의존적입니다.
- 학습은 상당히 많은 메모리와 저장 공간을 요구합니다.

#### 5.14.6 문제

1. 입력  $\mathbf{x}$  가 행렬이라고 가정하면, 그래디언트(gradients)의 차원이 어떻게 되나요?
2. 이 절에서 설명한 모델의 은닉층(hidden layer)에 편향(bias)을 추가하고,
  - 연산 그래프를 그려보세요
  - 순전파(forward propagation)와 역전파(backward propagation) 공식을 유도해보세요.

3. 이 절에 사용한 모델에 대해서 학습과 예측에 사용되는 메모리 양을 계산해보세요.
4. 2차 미분을 계산을 해야한다고 가정합니다. 그래프 연산에 어떤 일이 생길까요? 좋은 아이디어 인가요?
5. 연산 그래프가 사용 중인 GPU에 비해서 너무 크다고 가정합니다.
  - 한개 이상의 GPU로 나눌 수 있나요?
  - 작은 미니배치로 학습을 할 경우 장점과 단점이 무엇인가요?

#### 5.14.7 Scan the QR Code to Discuss



### 5.15 수치 안정성(numerical stability) 및 초기화

지금까지 우리는 다층 퍼셉트론(multilayer perception)을 구현하는데 필요한 도구, 회귀와 분류의 문제를 어떻게 풀 수 있는지, 그리고 모델의 용량을 어떻게 제어해야하는지에 대해서 다뤘습니다. 하지만, 파라미터의 초기화는 당연한 것으로 간주하면서, 특별히 중요하지 않은 것으로 단순하게 가정했습니다. 이 절에서는 이것들에 대해서 자세히 살펴보고, 유용한 경험적 방법론에 대해서 논의하겠습니다.

두번째로 우리는 활성화 함수(activation function) 선택에 큰 관심을 두지 않았습니다. 실제로 얇은 네트워크에서는 크게 중요하지 않지만, 딥 네트워크 (deep network)에서는 비선형성과 초기화의 선택이 최적화 알고리즘을 빠르게 수렴시키는데 중요한 역할을 합니다. 이 이슈들을 중요하게 생각하지 않으면 그래디언트 소실(vanishing) 또는 폭발(exploding)이 발생할 수 있습니다.

#### 5.15.1 그래디언트 소실(vanishing)과 폭발(exploding)

입력이  $\mathbf{x}$ , 출력이  $\mathbf{o}$ 이고  $d$  층을 갖는 딥 네트워크를 예로 들겠습니다. 각 층은 다음을 만족합니다.

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{이고, 따라서 } \mathbf{o} = f_d \circ \dots \circ f_1(\mathbf{x})$$

모든 활성화(activation)들과 입력들이 벡터인 경우,  $t$  번째 층의 함수  $f_t$  와 관련된 파라미터  $\mathbf{W}_t$  의 임의의 세트에 대한  $\mathbf{o}$  의 그래디언트(gradient)는 다음과 같이 표현됩니다.

$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \dots \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:=\mathbf{v}_t}.$$

다르게 말하면, 위 공식은  $d - t$  개의 행렬  $\mathbf{M}_d \cdot \dots \cdot \mathbf{M}_t$  과 그래디언트(gradient) 벡터  $\mathbf{v}_t$  의 곱입니다. 너무 많은 확률을 곱할 때 산술적인 언더플로우(underflow)를 경험할 때와 비슷한 상황이 발생합니다. 이 문제를 로그 공간으로 전환시켜서, 즉 문제를 가수(mantissa)에서 수치 표현의 지수로 이동시켜서 완화할 수 있었습니다. 처음에 행렬들  $M_t$ 은 다양한 고유값(eigenvalue)들을 갖을 것입니다. 어떤 것들은 작을 수도, 어떤 것은 클 수도 있습니다. 특히 그것들의 곱이 아주 크거나 아주 작을 수도 있습니다. 이것은 수치적인 표현의 문제일 뿐만 아니라 최적화 알고리즘이 수렴되지 않을 수 있다는 것을 의미합니다. 아주 큰 그래디언트(gradient)가 되거나, 너무 조금씩 업데이트가 되기도 합니다. 앞의 경우에는, 파라미터가 너무 커질 것이고, 후자의 경우에는 그래디언트 소실(vanishing gradient)이 되어버려서 더 이상 의미 있는 진척을 만들어 내지 못하게 됩니다.

## 그래디언트 폭발(exploding gradient)

좀 더 자세히 설명해 보겠습니다. 하나의 행렬을 선택한 후 100개의 가우시안 랜덤 행렬을 선택해서 모두 곱합니다. 우리가 선택한 스케일링으로 인해서 행렬의 곱은 너무 커지게 됩니다. 이러한 일이 딥 네트워크에서 발생한다면, 알고리즘을 수렴하게 만들기 어려워집니다.

```
[1]: %matplotlib inline
import mxnet as mx
from mxnet import nd, autograd
from matplotlib import pyplot as plt

M = nd.random.normal(shape=(4, 4))
print('A single matrix', M)
for i in range(100):
    M = nd.dot(M, nd.random.normal(shape=(4, 4)))

print('After multiplying 100 matrices', M)

A single matrix
[[ 2.2122064   0.7740038   1.0434405   1.1839255 ]
 [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
```

(continues on next page)

```
<NDArray 4x4 @cpu(0)>
After multiplying 100 matrices
[[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
 [-5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20]
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19]]
```

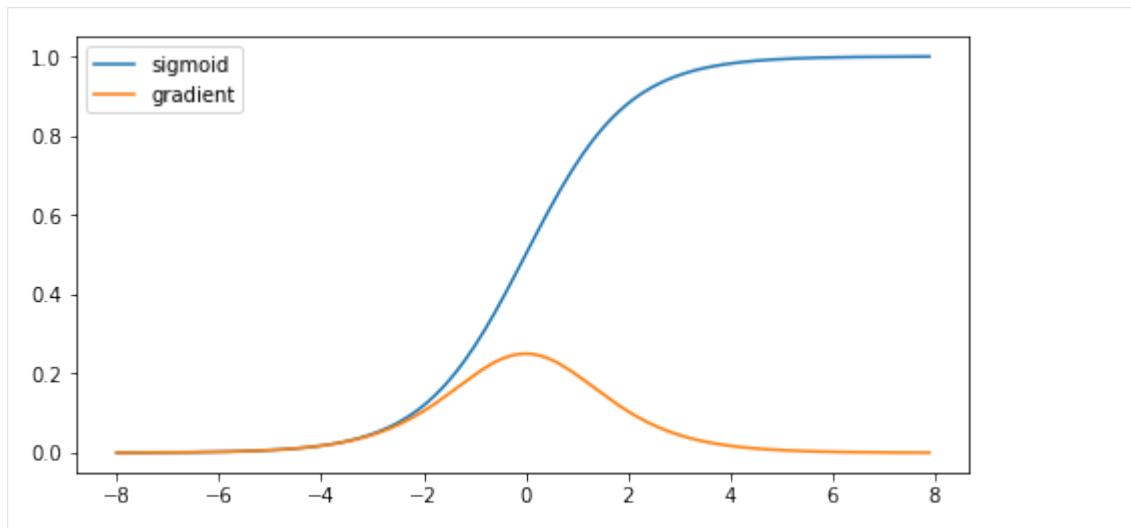
<NDArray 4x4 @cpu(0)>

## 그래디언트 소실(vanishing gradient)

반대의 문제인 그래디언트 소실(vanishing gradient)도 나쁜 경우입니다. 주요 원인 중의 하나는 각 층의 선형 연산과 함께 엮이는 활성화 함수(activation function)  $\sigma$ 입니다. 역사적으로는 *Multilayer Perceptrons* 절에서 소개했던 sigmoid 함수 ( $1 + \exp(-x)$ ) 가 유명한 활성화 함수(activation function)였습니다. 이 함수를 비선형 활성화 함수(activation function)로 사용했을 때 문제가 될 수 있는지를 보기 위해서, 이 함수에 대해서 간단하게 살펴보겠습니다.

```
[2]: x = nd.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = x.sigmoid()
y.backward()

plt.figure(figsize=(8, 4))
plt.plot(x.asnumpy(), y.asnumpy())
plt.plot(x.asnumpy(), x.grad.asnumpy())
plt.legend(['sigmoid', 'gradient'])
plt.show()
```



위 그림에서 보이는 것처럼 sigmoid의 그래디언트는 아주 큰 수나 아주 작은 수에서 소멸합니다. 체인룰(chain rule)로 인해서, 활성화(activation)들이  $[-4, 4]$  범위에 들어가지 않지 않으면 전체 곱의 그래디언트(gradient)는 소멸될 수 있다는 것을 의미합니다. 층을 많이 사용하는 경우, 이 현상이 어떤 층에서 일어날 가능성이 높습니다. ReLU  $\max(0, x)$  가 소개되기 전까지는 이 문제가 딥 네트워크 학습의 단점이었습니다. 그 결과 ReLU 가 활성화(activation)로 설계할 때 기본 선택이 되었습니다.

## 대칭성

딥 네트워크 디자인의 마지막 문제는 파라미터화에 내재된 대칭입니다. 두 개의 은닉 유닛(hidden unit),  $h_1$  and  $h_2$  을 갖는 한 개의 은닉층(hidden layer)를 가지고 있는 딥 네트워크를 가정하겠습니다. 이 경우, 첫번째 층의 가중치  $\mathbf{W}_1$  를 뒤집고, 두번째 층의 결과도 뒤집을 경우, 동일한 함수를 얻게 됩니다. 좀 더 일반적으로 각 층의 은닉 유닛(hidden unit) 간에는 치환 대칭성(permutation symmetry)이 존재합니다. 이것은 이론적으로만 말썽이되는 것이 아닙니다. 어떤 층의 파라미터를 모두 0으로 초기화를 하거나 ( $\mathbf{W}_l = 0$ ),  $\mathbf{W}_l$  의 모든 값을 동일하게 설정한다고 가정합니다. 이 경우, 모든 차원의 그래디언트(gradient)들이 같게 되고, 주어진 층에 내재된 표현력을 전혀 사용할 수 없게 됩니다. 사실, 그 은닉층(hidden layer)은 단일 유닛(single unit)처럼 동작합니다.

### 5.15.2 파라미터 초기화

위 문제를 해결하거나 최소한 완화시키는 방법은 가중치 벡터의 초기화를 조심하게 하는 것입니다. 이렇게 해서 적어도 초기의 그래디언트(gradient)가 소멸되지 않게 하고, 네트워크 가중치들이 너무

커지지 않게 합리적인 범위에 존재하게 할 수 있습니다. 최적화에서의 추가적 조치나 적합한 정규화(regularization)을 통해서 너무 나빠지는 것을 막을 수 있습니다. 이제 방법들에 대해서 알아보겠습니다.

## 기본 초기화

“*Concise Implementation of Linear Regression*” 절에서 우리는 `net.initialize(init.Normal(sigma=0.01))` 을 이용해서 가중치의 초기값으로 정규 분포에서 임의의 수 선택하는 방법을 사용했습니다. 초기화 방법을 명시하지 않은 경우, 즉, `net.initialize()` 를 호출하는 경우에 MXNet은 기본 랜덤 초기화 방법을 적용합니다. 이는, 가중치의 각 원소는  $U[-0.07, 0.07]$  범위의 균일 분포에서 선택된 값을 갖고, 편향(bias) 파라미터는 모두 0으로 설정됩니다. 일반적인 문제의 크기에서 이 두 방법은 상당히 잘 작동합니다.

## Xavier 초기화

어떤 층의 은닉 유닛(hidden unit)  $h_i$ 에 적용된 활성화(activation)의 범위 분포를 살펴보겠습니다. 이 값들은 다음과 같이 계산됩니다.

$$h_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j$$

가중치  $W_{ij}$  들은 같은 분포에서 서로 독립적으로 선택됩니다. 이 분포는 평균이 0이고 분산이  $\sigma^2$  라고 가정하겠습니다. (하지만, 이 분포가 가우시안(Gaussian) 이어야 한다는 것은 아니고, 단지 평균과 분산이 필요할 뿐입니다.) 층의 입력  $x_j$  를 제어할 수 있는 방법이 없지만, 그 값들의 평균이 0이고 분산이  $\gamma^2$  이고,  $\mathbf{W}$  과는 독립적이라는 다소 비현실적인 가정을 하겠습니다. 이 경우,  $h_i$  의 평균과 분산을 다음과 같이 계산할 수 있습니다.

$$\begin{aligned}\mathbf{E}[h_i] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij} x_j] = 0 \\ \mathbf{E}[h_i^2] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2 x_j^2] \\ &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2] \mathbf{E}[x_j^2] \\ &= n_{\text{in}} \sigma^2 \gamma^2\end{aligned}$$

$n_{\text{in}}\sigma^2 = 1$  을 적용하면 분산을 고정시킬 수 있습니다. 이제 back propagation을 고려해봅니다. 가장 상 위층들로부터 전달되는 그래디언트(gradient) 와 함께 비슷한 문제를 만나게 됩니다. 즉,  $\mathbf{W}\mathbf{w}$  대신에  $\mathbf{W}^\top \mathbf{g}$  를 다뤄야합니다. 여기서  $\mathbf{g}$  는 상위층으로부터 전달되는 그래디언트(gradient)를 의미합니다. 포워드 프로퍼게이션(forward propagation)에서와 같은 논리로,  $n_{\text{out}}\sigma^2 = 1$  이 아닐 경우에는 그래디언트(gradient)의 분산이 너무 커질 수 있습니다. 이 상황이 우리를 다음과 같은 딜레마에 빠지게 합니다. 즉, 우리는 두 조건을 동시에 만족시킬 수 없습니다. 대신, 다음은 조건은 쉽게 만족시킬 수 있습니다.

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ 또는 동일하게 } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

이것이 2010년에 Xavier Glorot and Yoshua Bengio 이 제안한 Xavier 초기화의 기본이 되는 논리입니다. 이 방법은 실제로 충분이 잘 작동합니다. 가우시안(Gaussian) 확률 변수에서 Xavier 초기화는 평균이 0이고 분산이  $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$  인 정규 분포에서 값을 선택합니다.  $U[-a, a]$  에 균등하게 분포한 확률 변수의 경우, 분산이  $a^2/3$  이 됩니다.  $a^2/3$  을  $\sigma^2$  에 대한 조건에 대입하면 다음과 같은 분포의 초기화를 할 수 있습니다.

$$U \left[ -\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})} \right].$$

## 그 외의 것들

위 내용은 아주 일부입니다. MXNet의 `mxnet.initializer` 모듈은 10가지 이상의 경험적 방법들을 제공합니다. 이 방법들은 슈퍼 해상도(superresolution), 시퀀스(sequence) 모델 또는 관련된 문제들에서 파라미터들이 연관된 경우에 사용할 수 있습니다. 이 모듈이 제공하는 것들을 살펴보는 것을 권장합니다.

### 5.15.3 요약

- 그래디언트 소멸(Vanishing gradient)과 그래디언트 폭발(exploding gradient)은 아주 깊은 네트워크에서 발생하는 흔한 문제입니다. 이를 위해서 그래디언트(gradient)와 파라미터가 잘 통제 되도록 하는 것이 중요합니다.
- 초기화 방법은 최소한 초기의 그래디언트(gradient)들이 너무 커지거나 너무 작아지지 않도록 하는데 필요합니다.
- ReLU는 그래디언트 소멸(vanishing gradient) 문제 중에 하나를 해결합니다. 즉, 매우 큰 입력에 대해서 그래디언트(gradient) 가 사라지는 것을 해결합니다. 이는 수렴을 아주 빠르게 가속화해 줍니다.

- 랜덤 초기화는 최적화를 수행하기 전 대칭을 깨 주는데 중요합니다.

#### 5.15.4 문제

1. 치환 대칭성(permuation symmetry) 이외에 대칭성(symmetry)를 깨는 다른 사례를 디자인할 수 있나요?
2. 선형 회귀나 softmax 회귀에서 모든 가중치 파라미터를 같은 값으로 초기화할 수 있나요?
3. 두 행렬의 곱에서 고유값(eigenvalue)들의 해석적 범위(analytic bound)를 찾아보세요. 그래디언트(gradients)들을 잘 제어되도록 하는 것에 대해서 어떤 것을 알 수 있나요?
4. 어떤 항들의 값이 커지고 있는 것을 알게 된 경우, 이 후에 이를 고칠 수 있나요? You, Gitman and Ginsburg, 2017 의 LARS 논문을 참고해보세요.

#### 5.15.5 Scan the QR Code to Discuss



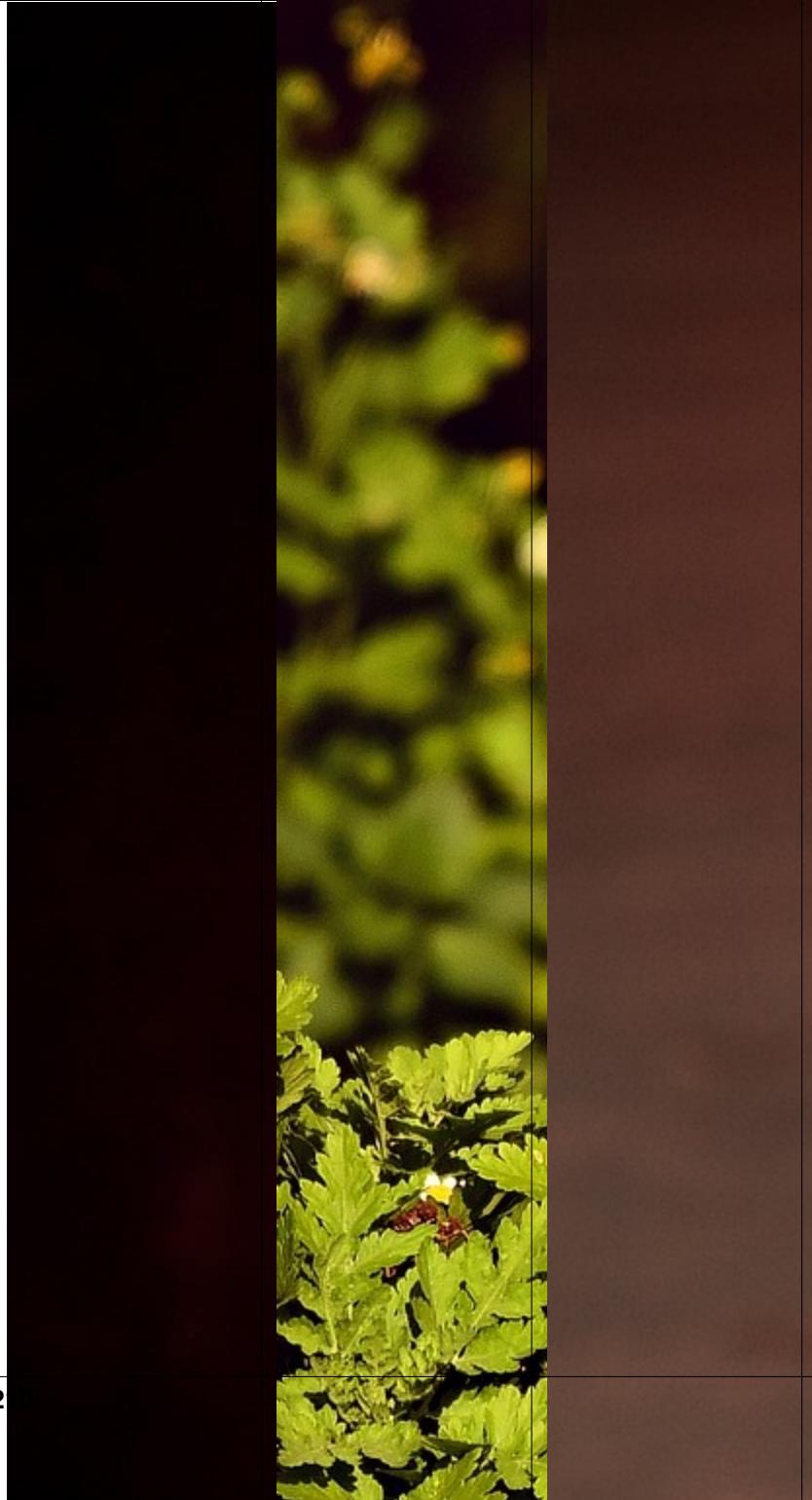
### 5.16 환경

지금까지 우리는 데이터가 어디서 왔는지 모델이 어떻게 배포되는지에 대해서는 걱정하지 않았습니다. 하지만, 이것들을 고려하지 않는 것은 문제가 됩니다. 실패한 많은 머신러닝 배포들의 원인을 추적해보면 이런 상황이 원인이 됩니다. 이 절에서는 이 상황을 초기에 발견하고, 완화하는 방법을 알아봅니다. 상황에 따라서, 정확한 데이터를 사용하면 되는 다소 간단한 문제일 수 있기도 하지만, 강화학습 시스템을 만드는 것과 같이 어려운 문제이기도 합니다.

#### 5.16.1 공변량 변화(covariate shift)

이해하는 것은 쉽지만, 놓치기 쉬운 문제가 있습니다. 강아지와 고양을 구분하는 문제를 생각해봅시다. 학습 데이터는 다음과 같이 주어졌습니다.

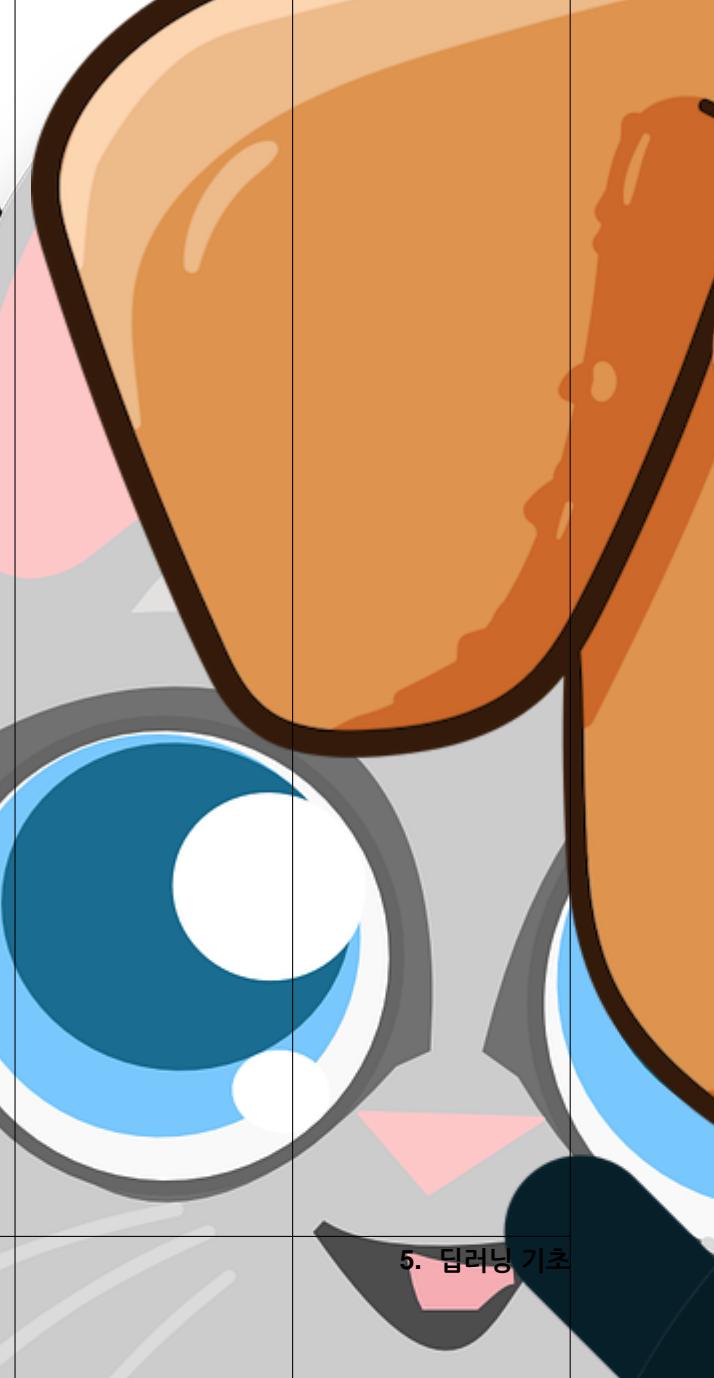
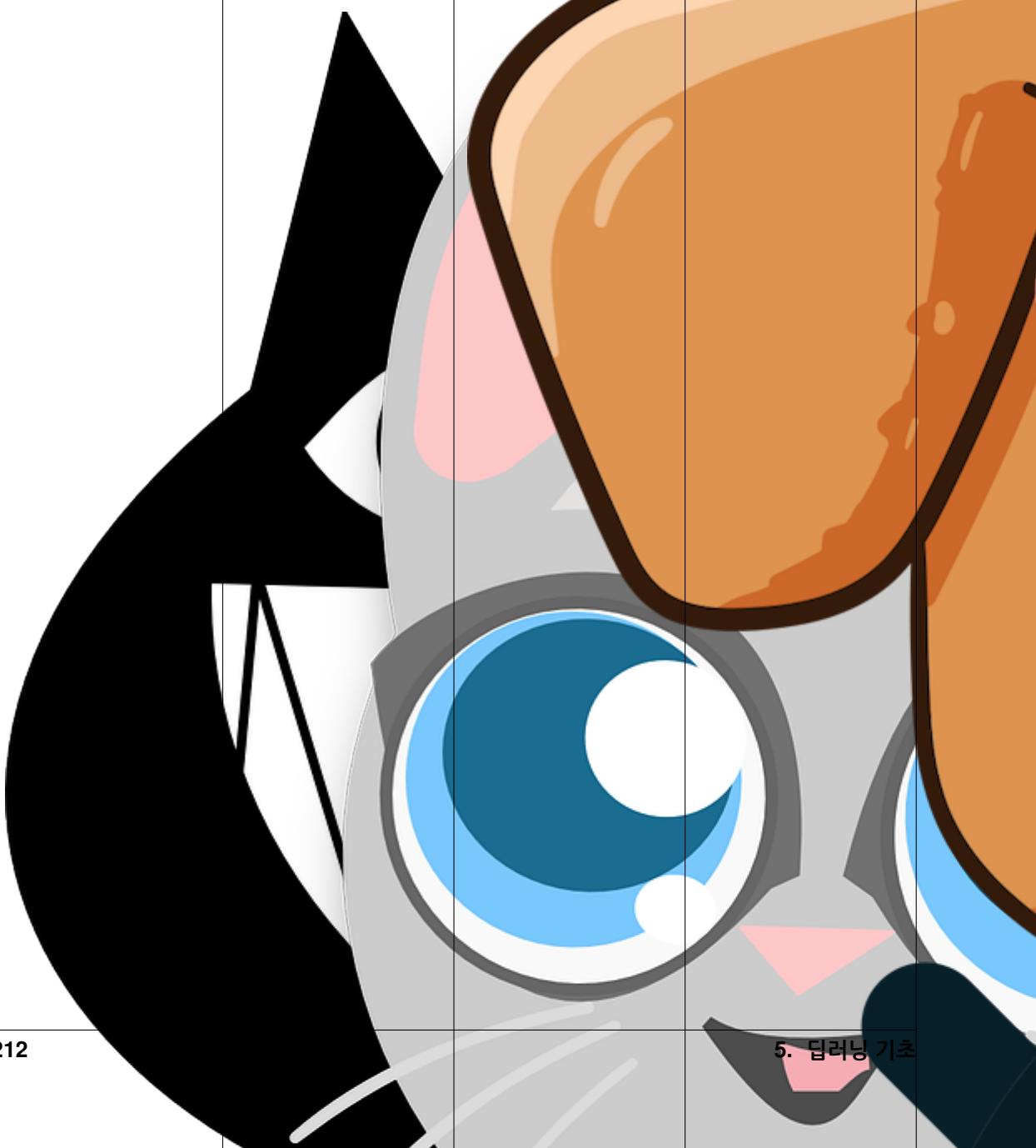
고양이	고양이	강아지	강아지
-----	-----	-----	-----



2

5. 딥러닝 기초

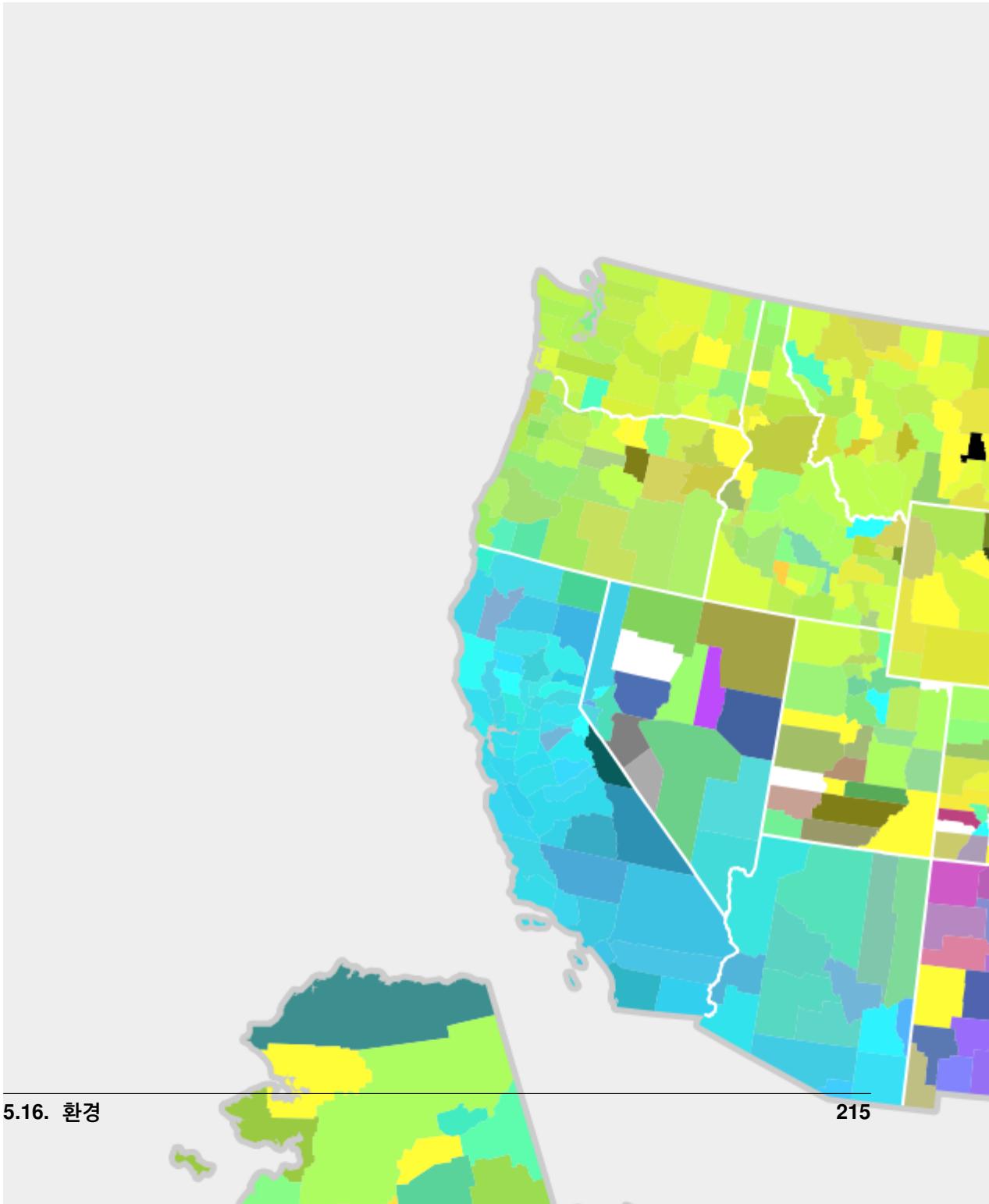
테스트에서는 다음 그림을 분류하도록 요청 받습니다.

고양이	고양이	강아지	강아지
			

당연하게 이것은 잘 작동하지 않습니다. 학습 데이터는 실제 사진으로 구성되어 있지만, 테스트셋은 만화 그림으로 되어있습니다. 색상도 정확하지 않습니다. 새로운 도메인에 어떻게 적용할지 계획이 없이 테스트셋과 다른 데이터로 학습을 시키는 것은 나쁜 아이디어입니다. 불행하게도 이것은 흔한 함정입니다. 통계학자들은 이것을 **공변량 변화(covariate shift)** 라고 합니다. 즉, 공변량(covariates)(학습 데이터)의 분포가 테스트 데이터의 분포가 다른 상황을 의미합니다. 수학적으로 말하자면,  $p(x)$  는 변화하는데,  $p(y|x)$  는 그대로 있는 경우를 의미합니다.



## 5.16.2 개념 변화(concept shift)



기계 번역 시스템을 만든다면, 분포  $p(y|x)$  는 지역에 따라서 다를 수 있습니다. 이 문제를 집어 내 기에는 상당이 까다롭습니다. 다행한 것은 많은 경우에  $p(y|x)$  는 조금씩만 변화한다는 것입니다. 더 자세히 살펴보기 전에, 공변량 변화(covariate shift)와 개념 변화(concept shift)가 명백하게 드러나지 않는 많은 상황에 대해서 살펴보겠습니다.

### 5.16.3 예제

#### 의학 분석

암을 진단하는 알고리즘을 설계하는 것을 상상해보세요. 건강한 사람과 아픈 사람의 데이터를 얻은 후, 알고리즘을 학습시킵니다. 학습된 모델이 높은 정확도를 보여주면서 잘 동작합니다. 당신은 이제 의료 분석 분야에서 성공적인 경력을 시작할 수 있다고 판단합니다. 하지만 너무 이릅니다.

많은 것들이 잘못될 수 있습니다. 특히, 학습에 사용한 분포와 실제 분포는 상당히 다를 수 있습니다. 실제로 수년 전에 스타트업 회사를 컨설팅하면서 겪었던 일입니다. 이 회사는 주로 나이 많은 남성에서 발견되는 질병에 대한 혈액 테스트를 개발하고 있었습니다. 이를 위해서 환자들로부터 상당히 많은 샘플을 수집할 수 있었습니다. 하지만, 윤리적인 이유로 건강한 남자의 혈액 샘플을 구하는 것은 상당히 어려웠습니다. 이를 해결하기 위해서, 캠퍼스의 학생들에게 혈액을 기증 받아서 테스트를 수행했습니다. 그리고, 그 회사는 나에게 질병을 분류하는 모델을 만드는 것에 대한 도움을 요청했습니다. 거의 완벽한 정확도의 확률로 두 데이터 셋을 분류하는 것은 아주 쉽다고 알려졌습니다. 결국, 모든 테스트 대상은 나이, 호르몬 레벨, 신체 활동, 식이 상태, 알콜 섭취, 그리고 질병과 연관이 없는 아주 많은 요소들이 달랐습니다. 하지만, 이는 실제 환자의 경우와 차이가 있습니다. 이들이 사용한 샘플링 절차는 아주 심한 공변량 변화(covariate shift)를 가지고 와서, 어떤 전통적인 방법으로 고쳐질 수가 없었습니다. 달리 말하면, 학습 데이터와 테스트 데이터가 너무나 달라서 어떤 유용한 일도 할 수 없었고, 결국 상당히 많은 돈을 낭비만 했습니다.

#### 자율 주행 자동차

자율 주행차를 위한 머신러닝 시스템을 만들고자 하는 한 회사가 있습니다. 도로를 탐지하는 것이 중요한 컴포넌트 중에 하나입니다. 실제 답을 다는 것이 너무 비싸기 때문에, 게임 렌더링 엔진을 사용해서 생성한 데이터를 추가 학습 데이터로 사용하기로 했습니다. 이렇게 학습된 모델은 렌더링 엔진으로 만들어진 '테스트 데이터'에는 잘 동작했습니다. 하지만, 실제 차에서는 재앙이었습니다. 이유는 렌더링 된 도로가 너무 단순한 텍스처를 사용했기 때문이었습니다. 더 중요한 것은 모든 도로 경계가 같은 텍스터로 렌더되었기에, 도로 탐지기는 이 '특징'을 너무 빨리 배워버렸습니다.

미국 군대에서 숲 속에서 있는 탱크를 탐지하는 것을 하려고 했을 때도 비슷한 문제가 발생했습니다. 탱크가 없는 숲의 항공 사진을 찍고, 탱크를 숲으로 몰고 가서 다른 사진을 찍었습니다. 이렇게 학습

된 분류기는 아주 완벽하게 동작했습니다. 하지만 불행히도 이 모델은 그늘이 있는 나무들과 그늘이 없는 나무를 구분하고 있었습니다. 이유는 첫번째 사진은 이른 아침에 찍었고, 두번째 사진은 정오에 찍었기 때문이었습니다.

### 정적이지 않은 분포(nonstationary distribution)

더 알아내기 힘든 상황은 분포가 천천히 변화하는 상황에서 모델을 적절하게 업데이트를 하지 않는 경우입니다. 전형적인 사례로는 다음과 같은 경우가 있습니다.

- 광고 모델을 학습시킨 후, 자주 업데이트하는 것을 실패한 경우. (예를 들면, iPad라는 새로운 디바이스가 막 출시된 것을 반영하는 것을 잊은 경우)
- 스팸 필터를 만들었습니다. 이 스팸 필터는 우리가 봤던 모든 스팸을 모두 잘 탐지합니다. 하지만, 스팸을 보내는 사람들이 이를 알고 이전에 봐왔던 것과는 아무 다른 새로운 메시지를 만듭니다.
- 상품 추천 시스템을 만들었습니다. 겨울에는 잘 동작합니다. 하지만, 크리스마스가 지난 후에도 산타 모자를 계속 추천하고 있습니다.

### 더 많은 예제들

- “업무에 부적합 또는 안전한 (Not suitable/safe for work (NSFW))” 이미지 판별기를 만들고 있습니다. 쉽게하기 위해서, Subreddit에서 이미지를 수집합니다. 불행하게도 실제 생활 데이터에 대한 정확도는 낮게 나옵니다. (Reddit에 올라와 있는 사진은 전문 사진가가 찍은 품질이 좋은 사진들인 반면에 실제 NSFW 이미지는 품질이 좋지 않습니다.)
- 얼굴 인식기를 만듭니다. 모든 밴치마크에서 잘 동작합니다. 하지만, 테스트 데이터에서는 그렇지 못합니다. 실패한 이미지를 보니 이미지 전체를 얼굴이 차지하는 클로즈업 사진들입니다.
- 미국 마켓을 위한 웹 검색 엔진을 만들어서 영국에 배포하고 싶습니다.

요약하면, 학습 데이터의 분포와 테스트 데이터의 분포가 다른 다양한 사례가 있습니다. 어떤 경우에는 운이 좋아서 covariate shift가 있음에도 불구하고 모델이 잘 동작할 수 있습니다. 자 지금부터 원칙적인 해결 전략에 대해서 이야기하겠습니다. 경고 - 약간의 수학과 통계가 필요합니다.

#### 5.16.4 공변량 변화(covariate shift) 교정

레이블을 달아놓은 데이터  $(x_i, y_i)$ 에 대한 의존도  $p(y|x)$ 를 추정하는 것을 한다고 가정합니다. 그런데,  $x_i$ 가 올바른 분포인  $p(x)$ 가 아닌 다른 분포  $q(x)$ 를 갖는 곳에서 추출됩니다. 먼저, 우리는 학습 과정에 정확하게 어떤 일이 일어나는지에 대해서 잘 생각해볼 필요가 있습니다. 즉, 학습 데이터와

연관된 레이블을 반복하면서, 매 미니 배치 이후에 모델의 가중치 벡터(weight vector)들을 업데이트 합니다.

경우에 따라서 우리는 파라미터에 가중치 감쇠(weight decay), 드롭아웃(dropout), 존아웃(zoneout) 또는 유사한 패널티를 적용합니다. 즉, 학습은 대부분 손실(loss)을 최소화하는 것을 의미합니다.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^n l(x_i, y_i, f(x_i)) + \text{some penalty}(w)$$

통계학자들은 첫번째 항을 경험적인 평균(empirical average)이라고 합니다. 즉, 이것은  $p(x)p(y|x)$  확률로 선택된 데이터에 구해진 평균을 의미합니다. 만약 데이터가 잘못된 분포  $q$ 에서 선택된다면, 다음과 같이 간단한 아이덴티티(identity)를 사용해서 수정할 수 있습니다.

$$\begin{aligned} \int p(x)f(x)dx &= \int p(x)f(x)\frac{q(x)}{p(x)}dx \\ &= \int q(x)f(x)\frac{p(x)}{q(x)}dx \end{aligned}$$

다르게 설명해보면, 데이터가 추출 되어야하는 올바른 분포에 대한 확률의 비율을 곱( $\beta(x) := p(x)/q(x)$ )해서 각 샘플의 가중치를 조절하면 됩니다. 하지만 안타깝게도 이 비율을 알지 못 합니다. 따라서, 우선 해야하는 일은 이 값을 추정하는 것입니다. 이를 추정하는 다양한 방법이 존재합니다. 예로는 다소 멋진 이론적인 연산 방법이 있습니다. 이는 예상치를 계산하는 연산을 재조정하는 것으로, 이는 최소-њ(minimum-norm)이나 최대 엔트로피(maximum entropy) 원칙을 직접 이용하는 방법입니다. 이런 방법들은 두 분포에서 샘플들을 수집해야하는 것을 염두해 두세요. 즉, 학습 데이터를 이용해서 진짜  $p$ , 그리고 학습 데이터셋을  $q$ 를 만드는데 사용한 분포를 의미합니다.

이 경우 좋은 결과를 주는 효과적인 방법이 있는데, 그것은 바로 로지스틱 회귀(logistic regression)입니다. 로지스틱 회귀를 이용하면 확률 비율을 계산해낼 수 있습니다.  $p(x)$ ff로부터 추출된 데이터와  $q(x)$ ff로부터 추출된 데이터를 구분하기 위한 분리 모델을 학습 시키실 수 있습니다. 두 분포를 구별하는 것이 불가능하다면, 샘플들은 두 분포 중에 하나에서 나왔다는 것을 의미합니다. 반면에 분류가 잘 되는 샘플들은 오버웨이트(overweighted) 되었거나 언더웨이트(underweight)되어 있을 것입니다. 간단하게 설명하기 위해서, 두 분포로부터 같은 개수만큼 샘플을 추출했다고 가정하겠습니다. 이를 각각  $x_i \sim p(x)$ ff와  $x_i2 \sim q(x)$ ff로 표기합니다.  $p$ ff로부터 추출된 경우  $z_i$ ff를 1로,  $q$ ff로부터 추출된 경우에는 -1로 값을 할당합니다. 그러면, 섞인 데이터셋의 확률은 다음과 같이 표현됩니다.

$$p(z = 1|x) = \frac{p(x)}{p(x) + q(x)} \quad \text{이고 따라서} \quad \frac{p(z = 1|x)}{p(z = -1|x)} = \frac{p(x)}{q(x)}$$

따라서,  $p(z = 1|x) = \frac{1}{1+\exp(-f(x))}$  를 만족시키는 로지스틱 회귀(logistic regression) 방법을 사용하면,

이 비율은 아래와 같은 수식으로 계산됩니다.

$$\beta(x) = \frac{1/(1 + \exp(-f(x)))}{\exp(-f(x)/(1 + \exp(-f(x))))} = \exp(f(x))$$

결론적으로 우리는 두 문제를 풀어야합니다. 첫번째 문제는 두 분포에서 추출된 데이터를 구분하는 것이고, 두번째는 가중치를 다시 적용한 최소화 문제입니다. 가중치 조정은  $\beta$  를 이용하는데, 이는 헤드 그라디언트(head gradient)를 이용합니다. 레이블이 없는 학습셋  $X$  와 테스트셋  $Z$  을 사용하는 프로토타입의 알고리즘은 아래와 같습니다.

1. 학습셋  $\{(x_i, -1) \dots (z_j, 1)\}$  을 생성합니다.
2. 로지스틱 회귀(Logistic regression)를 이용해서 이진(binary) 분류기를 학습시킵니다. 이를 함수  $f$  라고 하겠습니다.
3.  $\beta_i = \exp(f(x_i))$  또는  $\beta_i = \min(\exp(f(x_i)), c)$  를 이용해서 학습 데이터에 가중치를 적용합니다.
4. 데이터  $X$  와 이에 대한 레이블  $Y$  에 대한 학습을 수행할 때, 가중치  $\beta_i$  를 이용합니다.

**Generative Adversarial Networks** 는 위에서 설명한 아이디어를 이용해서, 참조 데이터 셋과 구분이 어려운 데이터를 만드는 데이터 생성기(*data generator*)를 만듭니다. 네트워크  $f$  는 진짜와 가짜 데이터는 구분하고, 다른 네트워크  $g$  는 판정하는 역할을 하는  $f$  를 속이는 역할, 즉 가짜 데이터를 진짜라고 판별하도록하는 역할을 수행합니다. 이에 대한 자세한 내용은 다시 다루겠습니다.

### 5.16.5 개념 변화(concept shift) 교정

개념 변화(concept shift)는 개념적으로 해결하기 훨씬 어렵습니다. 예를 들면, 고양이와 강아지를 구분하는 문제에서 흰색과 검은색 동물을 구분하는 문제로 갑자기 바뀌었다고 하면, 새로운 레이블을 이용해서 새로 학습을 시키는 것보다 더 잘 동작시키는 것을 기대하는 것은 무리일 것입니다. 다행히, 실제 상황에서는 이렇게 심한 변화는 발생하지 않습니다. 대신, 변화가 천천히 일어나는 것이 보통의 경우입니다. 더 정확하게 하기 위해서, 몇 가지 예를 들어보겠습니다.

- 광고에서 새로운 상품이 출시되고, 이전 상품의 인기는 떨어집니다. 즉, 광고의 분포와 인기도는 서서히 변화되기 때문에, click-through rate 예측 모델은 그에 따라서 서서히 바뀌어야 합니다.
- 교통 카메라 렌즈는 환경의 영향으로 서서히 성능이 떨어지게 되고, 그 결과 이미지 품질에 영향을 미칩니다.
- 뉴스 내용이 서서히 바뀝니다. (즉, 대부분의 뉴스는 바뀌지 않지만, 새로운 이야기가 추가됩니다.)

이런 경우에 네트워크 학습에 사용한 것과 같은 방법을 데이터의 변화에 적응시키는 데 사용할 수 있습니다. 즉, 네트워크를 처음부터 다시 학습시키는 것이 아니라, 현재 가중치 값을 갖는 네트워크에 새로이 추가된 데이터를 이용해서 학습시키는 것입니다.

### 5.16.6 학습 문제의 분류

$p(x)$  과  $p(y|x)$  이 바뀔 때 어떻게 다뤄야하는지에 대해서 알아봤으니, 머신러닝을 이용해서 풀 수 있는 여러가지 문제들에 대해서 알아보겠습니다.

- **배치 러닝.** 학습 데이터와 레이블 쌍  $\{(x_1, y_1), \dots, (x_n, y_n)\}$  을 사용해서 네트워크  $f(x, w)$  를 학습시킨다고 생각해봅니다. 모델을 학습시킨 후, 학습 데이터와 같은 분포에서 새로운 데이터  $(x, y)$  를 뽑아서 이 모델에 적용합니다. 우리가 여기서 논의하는 대부분의 문제는 이 기본적인 가정을 포함하고 있습니다. 예를 들면, 고양이와 강아지 사진을 사용해서 고양이 탐지 모델을 학습시킵니다. 모델을 학습시킨 후, 고양이만 들어올 수 있도록 하는 컴퓨터 비전을 이용한 고양이 전용 문 시스템에 이 모델을 사용합니다. 이 시스템을 고객의 가정에 설치한 후에 모델을 다시 업데이트하지 않습니다.
- **온라인 러닝.** 데이터  $(x_i, y_i)$  가 한번에 하나씩 들어오는 것을 가정합니다. 조금 더 명확하게 말하자면, 우선  $x_i$  가 관찰되면,  $f(x_i, w)$  를 통해서 추측을 수행 한 이후에만  $y_i$  를 알 수 있는 경우를 가정합니다. 이 후, 추측 결과에 대한 보상 또는 loss 를 계산합니다. 많은 실제 문제가 이러한 분류에 속합니다. 예를 들면, 다음 날의 주식 가격을 예측하는 경우를 생각해보면, 예측된 주가에 근거해서 거래를 하고, 그날의 주식시장이 끝나면 예측이 수익을 가져다 줬는지 알 수 있습니다. 달리 말하면, 새로운 관찰을 통해서 모델을 지속적으로 발전시키는 다음과 같은 사이클을 만들 수 있습니다.

model  $f_t \rightarrow$  data  $x_t \rightarrow$  estimate  $f_t(x_t) \rightarrow$  observation  $y_t \rightarrow$  loss  $l(y_t, f_t(x_t)) \rightarrow$  model  $f_{t+1}$

- **반딧.** 반딧은 위 문제의 특별한 경우입니다. 대부분의 학습 문제는 연속된 값을 출력하는 함수  $f$  의 파라미터(예를 들면 딥 네트워크)를 학습하는 경우이지만, 반딧 문제는 선택할 수 있는 종류가 유한한 경우 (즉, 취할 수 있는 행동이 유한한 경우)입니다. 이 간단한 문제의 경우, 최적화의 측면에서 강력한 이론적인 보증을 얻을 수 있다는 것이 당연합니다. 이 문제를 별도로 분류한 이유는 이 문제를 종종 *distinct learning* 과 혼동하기 때문입니다.
- **제어 (그리고 비대립적 강화 학습).** 많은 경우에 환경은 우리가 취한 행동을 기억합니다. 적의적인 의도가 아닌 경우에도, 단순히 기억하고, 이전에 일어난 일에 근거해서 반응하는 환경들이 있습니다. 즉, 커피 포트 제어기의 경우 이전에 데웠는지 여부에 따라서 다른 온도를 감지하기도 합니다. PID (Propotional Integral Derivative) 제어 알고리즘도 유명한 예입니다. 비슷한 예로, 뉴스 사이트에 대한 사용자의 행동은 이전에 무엇을 무엇이었는지 영향을 받습니다. 이런 종류

의 많은 알고리즘들은 그 결정들이 임의의 선택으로 보이지 않도록 모델을 만들어냅니다. (즉, 분산을 줄이는 방향으로)

- **강화 학습.** 기억을 하는 환경의 더 일반적인 예로 우리와 협력을 시도하는 환경(non-zero-sum 게임과 같이 협력적인 게임)이나 이기려고 하는 환경이 있습니다. 체스나, 바둑, 서양주사위놀이(Backgammon) 또는 스타크래프트가 경쟁하는 환경의 예들입니다. 마찬가지로, 자율주행차를 위한 좋은 제어기를 만드는 것도 생각해볼 수 있습니다. 이 경우 다른 차량들은 자율주행차의 운전 스타일에 여러가지로 반응을 합니다. 때로는 피하려고하거나, 사고를 내려고하거나, 같이 잘 주행하려고 하는 등 여러 반응을 보일 것입니다.

위에 설명한 다양한 상황들 간의 주요 차이점은 안정적인 환경에서 잘 작동하는 전략이 환경이 변화는 상황에서는 잘 작동하지 않을 수 있다는 것입니다. 예를 들면, 거래자가 발견한 차익 거래 기회는 한번 실행되면 사라질 가능성이 높습니다. 환경이 변화하는 속도나 형태는 계속해서 사용할 수 있는 알고리즘의 형태를 많이 제약합니다. 예를 들면, 어떤 것이 천천히 변화할 것이라고 알고 있을 경우, 예측 모델 또한 천천히 바꿔도록 할 수 있습니다. 만약, 환경이 불규칙으로 순간적으로 바뀐다고 알고 있는 경우에는, 이에 대응하도록 만들 수 있습니다. 이런 종류의 지식은 풀고자 하는 문제가 시간에 따라서 바뀌는 상황, 즉 개념 변화(concept shift)를 다루는 야심 찬 데이터 사이언티스트에게 아주 중요합니다.

### 5.16.7 요약

- 많은 경우에 학습셋과 테스트셋은 같은 분포로부터 얻어지지 않습니다. 이런 상황을 우리는 공변량 변화(covariate shift)라고 합니다.
- 공변량 변화(covariate shift)는 변화가 아주 심하지 않을 경우에 탐지하고 이를 교정할 수 있습니다. 만약 그렇게 하지 못하면, 테스트 시점에 좋지 않은 결과가 나옵니다.
- 어떤 경우에는 환경이 우리가 취한 것을 기억하고, 예상하지 못한 방법으로 결과를 줄 수도 있습니다. 모델을 만들에 이 점을 유의해야합니다.

### 5.16.8 문제

1. 검색 엔진의 행동을 바꾸면 어떤 일이 일어날까요? 사용자는 어떻게 반응할까요? 광고주는 어떨까요?
2. 공변량 변화(covariate shift) 탐기지를 구현해보세요. 힌트 - 분류기를 만들어 봅니다.
3. 공변량 변화(covariate shift) 교정기를 구현해보세요.

- 학습 세트와 테스트 세트가 많이 다를 경우 무엇이 잘못될 수 있을까요? 샘플 weight들에는 어떤 일이 일어날까요?

### 5.16.9 Scan the QR Code to Discuss



## 5.17 Kaggle의 주택 가격 예측하기

앞 절들에서 딥 네트워크를 만들고 차원과 가중치 감쇠(weight decay) 그리고 드롭아웃(dropout)을 사용해서 용량을 제어하는 다양한 기본적인 도구들을 소개했습니다. 이제 앞에서 배운 내용들을 잘 활용해서 Kaggle 대회에 참여해보겠습니다. 집 가격 예측 문제는 상당히 일반적이고, 텍스트나 이미지 데이터처럼 규칙적인 구조도 없기 때문에 시작하기 좋은 문제입니다. 사용할 데이터는 1978년의 Harrison과 Rubinfeld의 [보스턴 집 데이터 세트](#)가 아니고, 더 크고 더 많은 특성(feature)을 지닌 데이터 세트로 2006년부터 2010년까지의 Ames, IA의 집 가격 데이터입니다. 이 데이터는 2011년에 [Bart de Cock](#)이 수집한 것입니다. 더 크기 때문에 조금 더 흥미 있는 예측 문제를 다루게 됩니다.

이 절에서는 우리가 배운 것들을 적용해 볼 예정입니다. 특히, 데이터 전처리, 모델 설계, 하이퍼파라미터(hyperparameter) 선택과 튜닝에 대한 자세한 내용들을 살펴봅니다. 직접 수행하면서 용량 제어, 특성(feature) 추출 등의 영향이 어떻게 되는지 실제로 알아보게 되는데, 이 경험은 숙련된 데이터 사이언티스트가 되기 위해서는 꼭 필요한 것입니다.

### 5.17.1 Kaggle

Kaggle 은 머신 러닝 대회로 유명한 플랫폼으로, 데이터와 코드를 사용해서 사용자간에 협력을 하거나 경쟁을 하는 곳입니다. 예를 들어, 경쟁자가 제출한 코드를 볼 수 있고, 다른 참여자들과 대비해서 여러분이 얼마나 잘하고 있는지를 볼 수도 있습니다. 대회 중에 하나에 참여하기 위해서는, 계정을 등록해야 합니다. 자 그럼 지금 계정을 만들어보겠습니다.

# Kaggle is the place to do data science projects

[See how it works](#) 



5.17. Kaggle의 주택 가격 예측하기

집 가격 예측 페이지에서, 데이터 탭을 눌러보면 데이터 세트를 찾을 수 있고, 예측을 제출해서 여러분의 순위를 확인할 수 있습니다. 아래 URL을 방문해보세요.

<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>



# House Prices: A

Predict sales prices and

5,012 teams · Ongoing

[Overview](#)

[Data](#)

[Kernels](#)

[Discussion](#)

[Leaderboard](#)

[Overview](#)

[Description](#)

[Evaluation](#)

[Frequently Asked  
Questions](#)

[Tutorials](#)

**Start here if...**

You have some experience with Python and machine learning, and you're looking for data science students who want to expand their skill set before taking on more advanced challenges.

**Competition Description**

## 5.17.2 데이터셋에 접근하고 읽기

대회 데이터는 학습셋과 테스트셋으로 나눠져 있습니다. 각 레코드는 집에 대한 특징 값들과 도로 종류, 지어진 연도, 지붕 형태, 지하실 상태 등에 대한 속성이 포함되어 있습니다. 데이터는 다양한 데이터 형으로 구성되어 있습니다. 예를 들면, 지어진 연도는 정수, 지붕 형태는 이산 레이블(discrete label), 다른 속성들은 실수 등으로 되어 있습니다. 어떤 데이터는 누락된 경우가 있는데 이는 ‘na’로 표기되어 있습니다. 각 집의 가격 (즉, 레이블)은 학습 데이터 세트(경진 대회이기 때문에)에만 포함되어 있습니다. ‘Competition’ 탭의 ‘Data’ 탭을 눌러보면 데이터를 다운로드할 수 있는 링크를 찾을 수 있습니다.

데이터를 읽고 처리하는 데 효과적인 데이터 분석 툴킷인 pandas를 이용하겠습니다. 이 절을 수행하기 위해서 pandas를 우선 설치하세요.

```
[1]: # If pandas is not installed, please uncomment the following line:  
# !pip install pandas  
  
import sys  
sys.path.insert(0, '..')  
  
%matplotlib inline  
import d2l  
from mxnet import autograd, gluon, init, nd  
from mxnet.gluon import data as gdata, loss as gloss, nn  
import numpy as np  
import pandas as pd
```

편의를 위해서 데이터를 미리 다운로드해서 ../data 디렉토리 저장해 놓았습니다. 학습 데이터와 테스트 데이터가 담겨있는 두 개의 CSV(Comma Separated Values) 파일을 각각 Pandas를 이용해서 읽습니다.

```
[2]: train_data = pd.read_csv('../data/kaggle_house_pred_train.csv')  
test_data = pd.read_csv('../data/kaggle_house_pred_test.csv')
```

학습 데이터 셋은 1,460개의 샘플을 가지고 있고, 각 샘플은 80개의 특성(feature)들과 1개의 label을 가지고 있습니다. 테스트 데이터는 1,459개 샘플과 각 80개의 특성(feature)들이 있습니다.

```
[3]: print(train_data.shape)  
print(test_data.shape)  
  
(1460, 81)  
(1459, 80)
```

맨 앞 4개의 샘플들의 처음 4개의 특성(feature)들과 마지막 2개의 특성(feature)들, 그리고 판매가 레이블을 확인해봅시다.

```
[4]: train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]]
```

```
[4]:
```

	Id	MSSubClass	MSZoning	LotFrontage	SaleType	SaleCondition	SalePrice
0	1	60	RL	65.0	WD	Normal	208500
1	2	20	RL	80.0	WD	Normal	181500
2	3	60	RL	68.0	WD	Normal	223500
3	4	70	RL	60.0	WD	Abnorml	140000

각 샘플을 확인한 결과, 첫번째 특성(feature)은 ID라는 것을 확인할 수 있습니다. 이 값은 모델이 학습 데이터를 구분하는데 활용할 수 있겠습니다. 편리한 특성(feature)이긴 하지만, 예측이라는 목적에 어떤 정보를 제공하지는 않습니다. 따라서, 네트워크에 데이터를 넣기 전에 우리는 이 값을 데이터셋에서 제거하겠습니다.

```
[5]: all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

### 5.17.3 데이터 전처리하기

앞서 설명했듯이, 이 데이터는 다양한 데이터 형을 가지고 있습니다. 데이터를 딥 네트워크에 대입하기 전에, 상당한 처리를 해야합니다. 수치형의 특성(feature)부터 시작해봅시다. 누락된 값은 평균값으로 채워 넣는 것으로 시작합니다. 이는 특성(feature)이 규칙없이 누락된 경우에는 의미있는 전략입니다. 공통적인 스케일로 조정하기 위해서, 평균이 0이고 분산이 1이 되도록 조정을 하겠습니다. 이를 위한 방법은 다음과 같습니다.

$$x \leftarrow \frac{x - \mu}{\sigma}$$

이 변환이  $x$ 를 평균이 0이고 분산이 1인 데이터로 변환하는 방법은  $\mathbf{E}[(x - \mu)/\sigma] = (\mu - \mu)/\sigma = 0$  간단히 계산해보면 됩니다. 분산을 확인하기 위해서,  $\mathbf{E}[(x - \mu)^2] = \sigma^2$ 을 사용하면, 변환된 분산이 1을 갖는다는 것을 확인할 수 있습니다. 데이터를 표준화(normalizing)하는 이유는 모든 특성(feature) 값이 동일한 크기 정도로 변환 해주기 때문입니다. 결국에는 우리는 어떤 특성(feature)이 관련이 있는지에 대한 선형적 정보(*priori*)를 모릅니다. 따라서, 그 값들은 동일하게 다루는 것은 의미가 있습니다.

```
[6]: numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
```

(continues on next page)

```
# values to 0
all_features = all_features.fillna(0)
```

다음으로는, 불연속된 값(discrete value)들을 다뤄보겠습니다. 이것은 'MSZoning'과 같은 변수들을 포함합니다. 멀티클래스 분류 데이터를 0과 1의 벡터로 변환한 것과 같은 방법으로 이 값들을 원-핫 인코딩(one-hot-encoding)을 이용해서 변환합니다. 예를 들면, 'MSZoning'에 대한 값이 'RL'과 'RM'을 가질 수 있다고 하면, 이들은 각각 (1,0)과 (0,1) 벡터로 매핑하는 것을 의미합니다. Pandas는 이를 자동으로 해주는 기능을 제공합니다.

```
[7]: # Dummy_na=True refers to a missing value being a legal eigenvalue, and
# creates an indicative feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape

[7]: (2919, 354)
```

이 변환을 수행하면 특성(feature)의 개수가 79개에서 331개로 증가합니다. 마지막으로, values 속성을 통해서, Pandas dataframe을 NumPy 형태로 추출하고, 이를 학습에 사용하기 위해서 다시 MXNet의 기본 표현인 NDArray로 바꿉니다.

```
[8]: n_train = train_data.shape[0]
train_features = nd.array(all_features[:n_train].values)
test_features = nd.array(all_features[n_train:].values)
train_labels = nd.array(train_data.SalePrice.values).reshape((-1, 1))
```

## 5.17.4 학습하기

우선 제곱 손실(squared loss)을 사용해서 선형 모델을 학습시켜보겠습니다. 이 모델은 당연히 이 대회에서 우승을 할 정도로 좋은 모델이 될 수는 없지만, 데이터에 의미있는 정보가 있는지를 점검하는데에 도움을 줍니다. 또한, 이 모델은 더 멋진 모델이 되기 위해 얼마나 좋을 결과를 만들어내야 하는지에 대한 최소한의 기준점(baseline)을 주기도 합니다.

```
[9]: loss = gloss.L2Loss()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

집 가격은 주식과 같이 상대적입니다. 즉, 절대 오류 보다는 상대 오류  $\frac{y - \hat{y}}{y}$  가 더 의미가 있을 것입니다. 예를 들면, 실제 집 가격이 125,000 달러인 Rural Ohio에서 가격을 100,000 달러만큼 틀리게 예측하는 것은 아주 나쁜 예측이 되지만, 평균 집 값이 4백만 달러가 넘는 캘리포니아 Los Altos Hills의 집 가격을 같은 오차로 예산했다면, 이 모델을 충분히 정확한 것으로 간주될 것입니다.

이런 문제를 해결하는 방법 중에 하나는 예측된 가격에 로그(logarithm)를 취한 값의 차이로 측정하는 것입니다. 사실, 이 대회에서 품질을 측정하는 방법으로 사용되는 오류이기도 합니다. 결국,  $\log y - \log \hat{y}$  의 작은 값  $\delta$  는  $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$  로 해석되고, 다음과 같은 loss 함수를 정의할 수 있습니다.

$$L = \sqrt{\frac{1}{n} \sum_{i=1}^n (\log y_i - \log \hat{y}_i)^2}$$

```
[10]: def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the
    # value less than 1 as 1
    clipped_preds = nd.clip(net(features), 1, float('inf'))
    rmse = nd.sqrt(2 * loss(clipped_preds.log(), labels.log()).mean())
    return rmse.asscalar()
```

이전 절들과는 다르게, 아래 학습 함수에서는 Adam 최적화 알고리즘을 사용합니다. 앞에서 사용한 미니 배치 확률적 경사 하강법(stochastic gradient descent)과 비교하자면, Adam 최적화 알고리즘은 학습 속도(learning rate)에 상대적으로 덜 민감합니다. 이에 대한 자세한 설명은 [Optimization Algorithms](#) 장에서 하겠습니다.

```
[11]: def train(net, train_features, train_labels, test_features, test_labels,
            num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = gdata.DataLoader(gdata.ArrayDataset(
        train_features, train_labels), batch_size, shuffle=True)
    # The Adam optimization algorithm is used here
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate, 'wd': weight_decay})
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
                l.backward()
                trainer.step(batch_size)
            train_ls.append(log_rmse(net, train_features, train_labels))
    test_ls.append(log_rmse(net, test_features, test_labels))
```

(continues on next page)

```

if test_labels is not None:
    test_ls.append(log_rmse(net, test_features, test_labels))
return train_ls, test_ls

```

## 5.17.5 K-겹 교차 검증(K-fold cross-validation)

k-겹 교차 검증(k-fold cross-validation)은 “모델 선택, 언더피팅(*underfitting*), 오버피팅(*overfitting*)” 절에서 어떻게 다뤄야하는지를 소개한 개념입니다. 우리는 이 방법을 모델 디자인을 선택하고, 하이퍼파라미터(hyperparameter)를 조정하는데 사용하겠습니다. 우선, k-겹 교차 검증(k-fold cross-validation) 절차에 사용될, i-번째 데이터 겹(fold)을 반환하는 함수가 필요합니다. 데이터를 다루는 가장 효과적인 구현이 아님을 명시해주세요. 이 후에 우리는 아주 많은 데이터를 더 똑똑하게 다루는 방법을 사용할 예정이지만, 함수의 구현 코드를 간결하게 하기 위해서 지금은 사용하지 않습니다.

```
[12]: def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = nd.concat(X_train, X_part, dim=0)
            y_train = nd.concat(y_train, y_part, dim=0)
    return X_train, y_train, X_valid, y_valid
```

다음 함수는 k-겹 교차 검증(k-fold cross-validation)에서 학습을  $k$  번 수행했을 때 학습 오류의 평균과 검증(validation) 오류의 평균을 반환합니다.

```
[13]: def k_fold(k, X_train, y_train, num_epochs,
            learning_rate, weight_decay, batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
```

(continues on next page)

```

        weight_decay, batch_size)

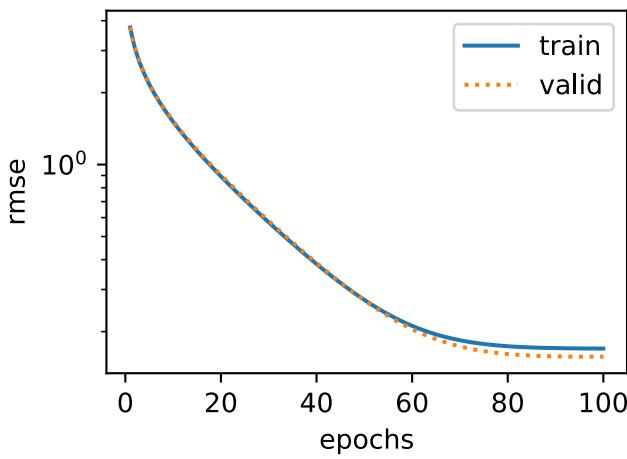
train_l_sum += train_ls[-1]
valid_l_sum += valid_ls[-1]
if i == 0:
    d2l.semilog(y(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse',
                  range(1, num_epochs + 1), valid_ls,
                  ['train', 'valid']))
print('fold %d, train rmse: %f, valid rmse: %f' % (
    i, train_ls[-1], valid_ls[-1]))
return train_l_sum / k, valid_l_sum / k

```

## 5.17.6 모델 선택하기

아래 하이퍼파라미터(hyperparameter)는 튜닝되지 않은 값을 사용했으니, 여러분이 이 값을 변경해서 모델의 성능을 높여보기를 바랍니다. 몇 개를 조정할 것인지에 따라 좋은 값들을 찾는데 상당히 많은 시간이 걸릴 수도 있습니다. 왜냐하면 k-겹 교차 검증(k-fold cross-validation) 방법은 테스트를 여러번 수행하는 것에도 영향을 받지 않기 때문입니다. 하지만, 너무 많은 오선들을 시도해볼려고 한다면, 실패할 수도 있습니다. 그 이유는 검증 데이터셋에 특정 하이퍼파라미터가 좋게 나오는 것이 있을 수 있기 때문입니다.

```
[14]: k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
      % (k, train_l, valid_l))
```



```

fold 0, train rmse: 0.169811, valid rmse: 0.156983
fold 1, train rmse: 0.162197, valid rmse: 0.189349
fold 2, train rmse: 0.163657, valid rmse: 0.168053
fold 3, train rmse: 0.167549, valid rmse: 0.154575
fold 4, train rmse: 0.162648, valid rmse: 0.182916
5-fold validation: avg train rmse: 0.165172, avg valid rmse: 0.170375

```

어떤 하이퍼파라미터(hyperparameter) 세트들을 사용하면 학습 오류가 상당히 크게 나오나,  $K$ -겹 교차 검증( $K$ -fold cross-validation) 오류는 상당히 크게 나오는 현상을 발견하게 될 것입니다. 이것은 대부분 오버피팅(overfitting)의 결과입니다. 따라서, 학습 오류를 줄일 때,  $K$ -겹 교차 검증( $K$ -fold cross-validation) 오류도 함께 감소하고 있는지를 확인하는 것이 필요합니다.

### 5.17.7 예측하고 제출하기

하이퍼파라미터(hyperparameter)의 좋은 조합을 찾았으면 (학습 데이터의  $1 - 1/k$  만큼 사용하는 것이 아니라) 모든 학습 데이터를 사용해서 모델을 학습시킵니다. 이렇게 학습된 모델을 테스트셋에 적용하고, 예측 결과를 CSV 파일에 저장해서 Kaggle에 업로드를 할 것입니다.

```
[15]: def train_and_pred(train_features, test_feature, train_labels, test_data,
                     num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse')
```

(continues on next page)

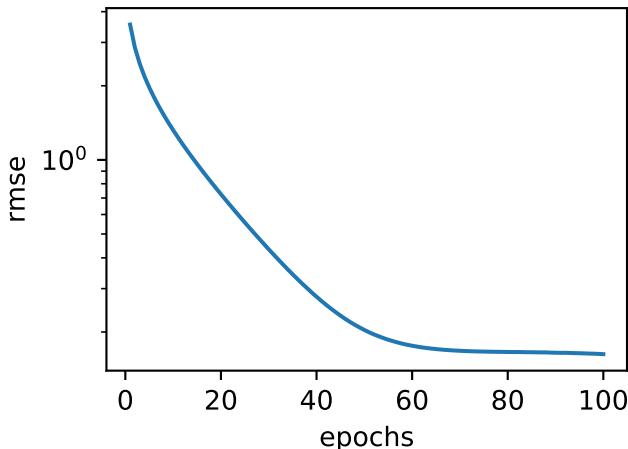
```

print('train rmse %f' % train_ls[-1])
# Apply the network to the test set
preds = net(test_features).asnumpy()
# Reformat it for export to Kaggle
test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
submission.to_csv('submission.csv', index=False)

```

자 모델을 수행해 보겠습니다. 잘되고 있는지 확인하는 좋은 방법은  $K$ -겹 교차 검증( $K$ -fold cross-validation)으로 예측한 것과 테스트셋에 대한 예측이 비슷하게 나오는지 확인해보는 것입니다. 만약 비슷하게 나온다면, 결과를 Kaggle에 업로드하세요.

```
[16]: train_and_pred(train_features, test_features, train_labels, test_data,
                  num_epochs, lr, weight_decay, batch_size)
```



```
train rmse 0.162497
```

위 코드를 수행하면 `submission.csv` 파일이 생성됩니다. (CSV는 Kaggle에서 결과 파일로 받는 형식 중에 하나임) 그 다음, Kaggle에 예측 값을 제출해서 테스트 데이터셋에 대한 실제 집 가격과 비교해서 오류를 확인해보는 것입니다. 방법은 아주 간단합니다.

- Kaggle 웹사이트에 로그인하고, 집 값 예측 대회 페이지를 방문합니다.
- 오른 쪽의 “Submit Predictions” 또는 “Late Submission”을 클릭합니다.
- 점선 박스 안의 “Upload Submission File” 버튼을 클릭하고, 업로드할 예측 파일을 선택합니다.

- 페이지 아래에 있는 “Make Submission” 버튼을 클릭해서 여러분의 결과를 보세요.

## Step 1

### Upload submission file

File Format

Your submission should be in  
You can upload this in a zip/gz  
archive, if you prefer.

## Step 2

### Describe submission

B I

Briefly describe your submission

## 5.17.8 요약

- 실제 데이터는 종종 다양한 데이터 타입의 값들을 갖고 있기 때문에, 전처리가 꼭 필요합니다.
- 실수 값을 평균이 0이고 분산이 1로 변환을 기본으로 선택하는 것은 좋은 방법이고, 누락된 값을 평균 값으로 채워 넣는 것도 그렇습니다.
- 카테고리 변수를 지표 변수(indicator variable)로 변환해서 이 값들을 벡터처럼 다룰 수 있습니다.
- 모델을 선택하고 하이퍼파라미터(hyper-parameter)를 선택하기 위해서  $K$ -겹 교차 검증( $K$ -fold cross-validation)을 사용할 수 있습니다.
- 로그(Logarithm)는 상대적인 손실(loss)를 구하는데 유용합니다.

## 5.17.9 문제

1. 여러분이 수행한 예측 결과를 Kaggle에 제출하세요. 여러분의 예측이 얼마나 좋은가요?
2. log 가격을 직접 최소화하는 방법으로 모델을 향상시킬 수 있나요? 가격이 아닌 log 가격을 예측하도록 하면 어떻게 될까요?
3. 누락된 값을 평균값으로 채우는 방법이 항상 좋은 아이디어일까요? 힌트 - 값들이 불규칙하게 누락되지 않은 경우를 생각해보세요.
4. 누락된 값을 다루는 더 좋은 표현법을 찾아보세요. 힌트 - 지표 변수를 추가하면 어떻게 될까요?
5.  $K$ -겹 교차 검증( $K$ -fold cross-validation)을 이용해서 하이퍼파라미터(hyper-parameter)를 튜닝하고 더 Kaggle에서 좋은 점수를 획득해보세요.
6. 층 추가, 정규화 적용, 드롭아웃(dropout) 적용 등을 통해서 모델을 향상시켜서 점수를 높여보세요.
7. 연속된 수치 특성(feature)이 절에서 한 것처럼 표준화하지 않은 경우 어떤 일이 일어날까요?

## 5.17.10 Scan the QR Code to Discuss



# 6

---

## 딥러닝 계산

---

앞 장에서는 간단한 딥러닝 모델을 위한 원칙과 구현에 대해서 알아봤습니다. 이 장에서 우리는 딥러닝 연산의 주요 요소들(모델 생성, 파라미터 접근, 초기화, 커스텀 층, 읽기/저장하기, GPU 사용)을 다룰 예정입니다. 이 장을 통해서 여러분은 모델 구현 및 연산 상세에 대한 중요한 통찰을 얻고, 다음 장들에서 설명한 더 복잡한 모델 구현을 위한 견고한 기본을 다질 수 있습니다.

### 6.1 층(layer)과 블럭(Block)

딥러닝이 유명해질 수 있었던 중요 요소들 중에 하나는 바로 강력한 소프트웨어입니다. 반도체 설계를 하는데 엔지니어들이 논리 회로를 트랜지스터로 구현하던 것에서 코드를 작성하는 것으로 넘어간 것과 같은 일이 딥 네트워크 설계에도 비슷하게 일어나고 있습니다. 앞 장들은 단일 뉴런으로 부터 뉴런으로 구성된 전체 층들로 옮겨가는 것을 보여줬습니다. 하지만, 컴퓨터 비전 문제를 풀기 위해서 2016년에 [He et al.](#)에 의해서 제안된 ResNet-152의 경우처럼 152개의 층들을 갖는 네트워크 층들을 사용한 네트워크 설계 방법 조차도 지루할 수 있습니다.

이런 네트워크는 많은 정도로 반복되는 부분을 갖고, 반복되는 (또는 비슷하게 설계된) 층들의 블럭들로 구성됩니다. 이들 블럭들은 더 복잡한 네트워크 디자인을 구성하는 기본 요소가 됩니다. 간략하게

말하면, 블럭은 하나 또는 그 이상의 층의 조합입니다. 마치 레고 공장이 만든 블럭을 이용해서 멋진 구조물을 만들 수 있는 것처럼, 이 디자인은 요청에 따라서 블럭을 생성하는 코드의 도움으로 만들어질 수 있습니다.

아주 간단한 블럭부터 살펴보겠습니다. 이 블럭은 앞 장에서 본 다층 퍼셉트론(multilayer perception)을 위한 것입니다. 일반적인 방법으로 두 개의 층을 갖는 네트워크를 다음과 같이 만들 수 있습니다.

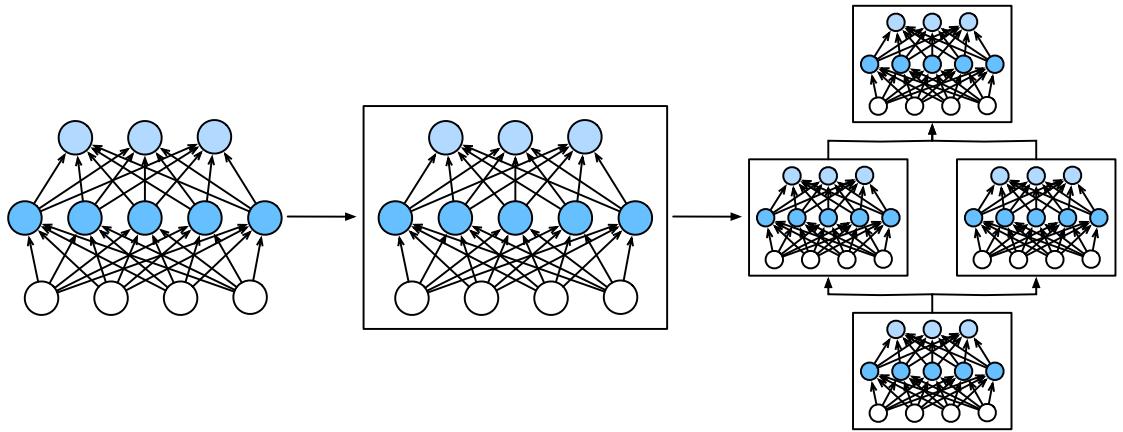
```
[1]: from mxnet import nd
from mxnet.gluon import nn

x = nd.random.uniform(shape=(2, 20))

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)

[1]:
[[ 0.09543004  0.04614332 -0.00286654 -0.07790349 -0.05130243  0.02942037
  0.08696642 -0.0190793  -0.04122177  0.05088576]
 [ 0.0769287   0.03099705  0.00856576 -0.04467199 -0.06926839  0.09132434
  0.06786595 -0.06187842 -0.03436673  0.04234694]]
<NDArray 2x10 @cpu(0)>
```

이 코드는 256개의 유닛(unit)들을 갖는 은닉층(hidden layer) 한 개를 포함한 네트워크를 생성합니다. 은닉층(hidden layer)은 ReLU 활성화(activation)로 연결되어 있고, 결과 층의 10개 유닛(unit)들로 연결되어 있습니다. 여기서 우리는 nn.Sequential 생성자를 사용해서 빈 네트워크를 만들고, 그 다음에 층들을 추가했습니다. 아직은 nn.Sequential 내부에서 어떤 일이 벌어지는 지는 미스테리로 남아있습니다. 아래 내용을 통해서 이것은 실제로 블럭을 생성하고 있는 것을 확인할 것입니다. 이 블럭들은 더 큰 결과물로 합쳐지는데 때로는 재귀적으로 합쳐지기도 합니다. 아래 그림은 이 것이 어떻게 일어나는지 보여줍니다.



층(layer)을 정의하는 것부터 (하나 또는 그 이상이 층들을 갖는) 블럭을 정의하는 데 필요한 다양한 절차에 대해서 설명하겠습니다. 블럭은 멋진 층과 비슷하게 동작합니다. 즉, 블럭은 아래 기능을 제공합니다.

1. 데이터(입력을)를 받아야 합니다.
2. 의미 있는 결과를 출력해야 합니다. 이는 `forward`라고 불리는 함수에서 처리합니다. 원하는 결과를 얻기 위해서 `net(x)`를 통해서 블럭을 수행할 수도 있는데, 실제로는 순전파(forward propagation)을 수행하는 `forward` 함수를 호출합니다.
3. `backward` 함수가 호출되면 입력에 대해서 그래디언트(gradient)를 생성해야 합니다. 일반적으로 이것은 자동으로 이뤄집니다.
4. 블럭에 속한 파라미터들을 저장해야 합니다. 예를 들면, 위 블럭은 두 개의 은닉층(hidden layer)을 갖는데, 파라미터를 저장할 공간이 있어야 합니다.

### 6.1.1 커스텀 블럭

`nn.Block` 클래스는 우리가 필요로 하는 기능들을 제공합니다. `nn` 모듈에서 제공하는 모델 생성자로, 우리가 원하는 모델을 정의하기 위해서 상속하는 클래스입니다. 아래 코드는 이 절을 시작할 때 언급한 다층 퍼셉트론(multilayer perceptron)을 생성하기 위해서 `Block` 클래스를 상속하고 있습니다. 여기서 `MLP` 클래스는 `Block` 클래스의 `__init__` 과 `forward` 함수를 오버라이드하고 있습니다. 이 함수들은 각각 모델 파라미터들을 생성하고 `forward` 계산을 정의하는 함수입니다. `Forward` 연산은 역전파(backward propagation)을 의미합니다.

```
[2]: from mxnet import nd
from mxnet.gluon import nn

class MLP(nn.Block):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers
    def __init__(self, **kwargs):
        # Call the constructor of the MLP parent class Block to perform the
        # necessary initialization. In this way, other function parameters can
        # also be specified when constructing an instance, such as the model
        # parameter, params, described in the following sections
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu') # Hidden layer
        self.output = nn.Dense(10) # Output layer

    # Define the forward computation of the model, that is, how to return the
    # required model output based on the input x
    def forward(self, x):
        return self.output(self.hidden(x))
```

조금 더 자세히 살펴보겠습니다. `forward` 메소드는 은닉층(hidden layer) `self.hidden(x)`를 계산하고, 그 값을 이용해서 결과층 `self.output(...)`을 계산합니다. 이것이 이 블럭의 `forward` 연산에서 해야하는 일입니다.

블럭이 어떤 값을 사용해서 계산을 수행해야하는지를 알기 위해서, 우리는 우선 층들을 정의 해야합니다. 이는 `__init__` 메소드가 하는 일입니다. 블럭과 관련된 모든 파라미터들을 초기화하고, 필요한 층을 생성합니다. 그리고, 관련 층들과 클래스에 필요한 파라미터들을 정의합니다. 시스템은 그래디언트(gradient)를 자동으로 계산해주는 `backward` 메소드를 자동으로 생성해줍니다. `initialize` 메소드도 자동으로 생성됩니다. 한번 수행해보겠습니다.

```
[3]: net = MLP()
net.initialize()
net(x)

[3]:
[[ 0.00362228  0.00633332  0.03201144 -0.01369375  0.10336449 -0.03508018
 -0.00032164 -0.01676023  0.06978628  0.01303309]
 [ 0.03871715  0.02608213  0.03544959 -0.02521311  0.11005433 -0.0143066
 -0.03052466 -0.03852827  0.06321152  0.0038594 ]]
<NDArray 2x10 @cpu(0)>
```

위에서 설명했듯이, 블럭 클래스는 무엇을 하는지에 따라서 아주 다르게 정의될 수 있습니다. 예를 들어, 그것의 하위 클래스가 (Gluon에서 제공하는 `Dense` 클래스와 같은) 층이 될 수도 있고, (우리가

막 정의한 MLP 클래스와 같은) 모델이 될 수도 있습니다. 또는 다른 모델의 일부가 될 수도 있습니다. 이는 아주 깊은 네트워크를 디자인할 때 사용되는 방법입니다. 이 장을 통해서 우리는 이것을 아주 유연하게 사용할 수 있는 방법에 대해서 알아보겠습니다.

## 6.1.2 Sequential 블럭

Block 클래스는 데이터흐름을 기술하는 일반 컴포넌트입니다. 사실 Sequential 클래스는 Block 클래스로부터 정의됩니다. 모델을 forward 연산은 각 층에 대한 연산의 단순한 연결이기 때문에, 우리는 모델을 아주 간단한 방법으로 정의할 수 있습니다. Sequential 클래스의 목적은 유용한 편의 함수들을 제공하는 것에 있습니다. 특히, add 메소드는 연결된 Block 하위클래스의 인스턴스를 하나씩 더할 수 있게 해주고, 모델의 forward 연산은 이 인스턴스들을 더하기 순서대로 계산합니다.

아래 코드에서 MySequential 클래스를 정의했는데, 이는 Sequential 클래스와 같은 기능을 제공합니다. 이를 통해서 Sequential 클래스가 어떻게 동작하는 이해하는데 도움이 될 것입니다.

```
[4]: class MySequential(nn.Block):
    def __init__(self, **kwargs):
        super(MySequential, self).__init__(**kwargs)

    def add(self, block):
        # Here, block is an instance of a Block subclass, and we assume it has
        # a unique name. We save it in the member variable _children of the
        # Block class, and its type is OrderedDict. When the MySequential
        # instance calls the initialize function, the system automatically
        # initializes all members of _children
        self._children[block.name] = block

    def forward(self, x):
        # OrderedDict guarantees that members will be traversed in the order
        # they were added
        for block in self._children.values():
            x = block(x)
        return x
```

add 메소드가 핵심입니다. 이 메소드는 순서가 있는 사전(dictionary)에 블럭을 추가하는 일을 합니다. 순전파(forward propagation)가 호출되면 이 블럭들은 순서대로 수행됩니다. MLP가 어떻게 구현되는지 보겠습니다.

```
[5]: net = MySequential()
net.add(nn.Dense(256, activation='relu'))
```

(continues on next page)

```
net.add(nn.Dense(10))
net.initialize()
net(x)

[5]: [[ 0.07787765  0.00216401  0.01682201  0.03059879 -0.00702019  0.01668714
       0.04822845  0.00394321 -0.09300036 -0.044943   ]
     [ 0.08891079 -0.00625484 -0.01619132  0.03807178 -0.01451489  0.02006172
       0.0303478   0.02463485 -0.07605445 -0.04389167]]
<NDArray 2x10 @cpu(0)>
```

실제로, “[다층 퍼셉트론\(multilayer perceptron\)의 간결한 구현](#)”에서 Sequential 클래스를 사용한 것과 MySequential 클래스를 사용한 것이 다르지 않다는 것을 볼 수 있습니다.

### 6.1.3 코드와 블록(Block)

Sequential 클래스가 모델 생성을 쉽게 해주고 forward 메소드를 별도로 구현할 필요가 없게 해주지만, Block 클래스를 직접 상속하면 더 유연한 모델 생성을 할 수 있습니다. 특히, forward 메소드에서 Python의 제어 흐름을 이용하는 것을 예로 들어보겠습니다. 설명하기에 앞서서 *constant* 파라미터라는 개념에 대해서 알아보겠습니다. 이 파라미터들은 역전파(back propagation)이 호출되었을 때 사용되지는 않습니다. 추상적으로 들릴 수 있지만, 실제 일어나는 일이 그렇습니다. 어떤 함수가 있다고 가정합니다.

$$f(\mathbf{x}, \mathbf{w}) = 3 \cdot \mathbf{w}^\top \mathbf{x}.$$

이 경우, 3이 상수(constant) 파라미터입니다. 우리는 3을 다른 값, 예를 들어  $c$ 로 바꿔서 다음과 같이 표현할 수 있습니다.

$$f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}.$$

$c$ 의 값을 조절할 수 있게 된 것 이외에는 바뀐 것이 없습니다.  $\mathbf{w}$  와  $\mathbf{x}$  만을 생각해보면 여전히 상수입니다. 하지만, Gluon은 이것을 미리 알지 못하기 때문에, 도움을 주는 것이 필요합니다. 이렇게 하는 것은 Gluon이 변하지 않는 파라미터에 대해서는 신경 쓰지 않도록 할 수 있기 때문에 코드가 더 빠르게 수행되게 해줍니다. `get_constant` 메소드를 이용하면 됩니다. 실제 어떻게 구현되는지 살펴보겠습니다.

```
[6]: class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        # Random weight parameters created with the get_constant are not
        # iterated during training (i.e. constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', nd.random.uniform(shape=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, x):
        x = self.dense(x)
        # Use the constant parameters created, as well as the relu and dot
        # functions of NDArray
        x = nd.relu(nd.dot(x, self.rand_weight.data()) + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        x = self.dense(x)
        # Here in Control flow, we need to call asscalar to return the scalar
        # for comparison
        while x.norm().asscalar() > 1:
            x /= 2
        if x.norm().asscalar() < 0.8:
            x *= 10
        return x.sum()
```

FancyMLP 모델에서 rand\_weight라는 상수 가중치를 정의했습니다. (이 변수는 모델 파라미터는 아니다라는 것을 알아두세요). 그리고, 행렬 곱하기 연산(nd.dot())을 수행하고, 같은 Dense 층을 재사용합니다. 서로 다른 파라미터 세트를 사용한 두 개의 텐스층(dense layer)를 사용했던 것과 다른 형태로 구현되었음을 주목하세요. 우리는 대신, 같은 네트워크를 두 번 사용했습니다. 네트워크의 여러 부분이 같은 파라미터를 공유하는 경우 딥 네트워크에서 이 것을 파라미터가 서로 묶여 있다(tied)라고 말하기도 합니다. 이 클래스에 대한 인스턴스를 만들어서 데이터를 입력하면 어떤 일이 일어나는지 보겠습니다.

```
[7]: net = FancyMLP()
net.initialize()
net(x)
```

```
[7]: [25.522684]
<NDArray 1 @cpu(0)>
```

네트워크를 만들 때 이런 방법을 섞어서 사용하지 않을 이유가 없습니다. 아래 예제를 보면 어쩌면 카메라와 닮아 보일 수도 있고 조금 다르게 말하면, **Rube Goldberg Machine**과 비슷하다고 할 수도 있

습니다. 즉, 개별적인 블럭을 합쳐서 블럭을 만들고 이렇게 만들어진 블럭이 다시 블럭으로 사용될 수 있는 것의 예제를 다음과 같이 만들어 볼 수 있습니다. 더 나아가서는 같은 forward 함수 안에서 여러 전략을 합치는 것도 가능합니다. 아래 코드가 그런 예입니다.

```
[8]: class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                    nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, x):
        return self.dense(self.net(x))

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FancyMLP())

chimera.initialize()
chimera(x)

[8]: [30.518448]
<NDArray 1 @cpu(0)>
```

#### 6.1.4 컴파일

여러분이 관심이 많다면 이런 접근 방법의 효율에 대한 의심을 할 것입니다. 결국에는 많은 사전(dictionary) 참조, 코드 수행과 다른 Python 코드들 수행하면서 성능이 높은 딥러닝 라이브러리를 만들어야 합니다. Python의 Global Interpreter Lock은 아주 잘 알려진 문제로, 아주 성능이 좋은 GPU를 가지고 있을지라도 단일 CPU 코어에서 수행되는 Python 프로그램이 다음에 무엇을 해야 할지를 알려주기를 기다려야하기 때문에 딥러닝 환경에서 성능에 안 좋은 영향을 미칩니다. 당연하게도 아주 나쁜 상황이지만, 이를 우회하는 여러 방법들이 존재합니다. Python 속도를 향상시키는 방법은 이 모든 것을 모두 제거하는 것이 최선입니다.

Gluon은 Hybridization 기능을 통해서 해결하고 있습니다. Python 코드 블럭이 처음 수행되면 Gluon 런타임은 무엇이 수행되었는지를 기록하고, 이후에 수행될 때는 Python을 호출하지 않고 빠른 코드를 수행합니다. 이 방법은 속도를 상당히 빠르게 해주지만, 제어 흐름을 다루는데 주의를 기울여야 합니다. 하이브리드화(Hybridization)와 컴파일(compilation)에 대해서 더 관심이 있다면 이 장을 마치고, 해당 내용이 있는 절을 읽어보세요.

### 6.1.5 요약

- 층들은 블럭입니다.
- 많은 층들이 하나의 블럭이 될 수 있습니다.
- 많은 블럭들이 하나의 블럭이 될 수 있습니다.
- 코드도 블럭이 될 수 있습니다.
- 블럭은 파라미터 초기화, 역전파(back propagation) 또는 관련된 일을 대신 처리해줍니다.
- 층들과 블럭들을 순차적으로 연결하는 것은 Sequential 블럭에 의해서 처리됩니다.

### 6.1.6 문제

1. What kind of error message will you get when calling an `__init__` method whose parent class not in the `__init__` function of the parent class?
2. FancyMLP 클래스에서 `asscalar` 함수를 삭제하면 어떤 문제가 발생하나요?
3. NestMLP 클래스에서 Sequential 클래스의 인스턴스로 정의된 `self.net` 을 `self.net = [nn.Dense(64, activation='relu'), nn.Dense(32, activation='relu')]` 로 바꾸면 어떤 문제가 발생하나요?
4. 두 블럭 (`net1` 과 `net2`)를 인자로 받아서 forward pass의 두 네트워크의 결과를 연결해서 반환하는 블럭을 작성해보세요. (이는 parallel 블럭이라고 합니다)
5. 같은 네트워크의 여러 인스턴스를 연결하고자 가정합니다. 같은 블럭의 여러 인스턴스를 생성하는 factory 함수를 작성하고, 이를 사용해서 더 큰 네트워크를 만들어 보세요.

### 6.1.7 Scan the QR Code to Discuss



## 6.2 파라미터 관리

딥 네트워크 학습의 최종 목표는 주어진 아키텍처에 가장 잘 맞는 파라미터 값들을 찾는 것입니다. 일반적인 것 또는 표준에 준하는 것들을 다룰 때는 nn.Sequential 클래스가 이를 위한 완벽한 도구가 될 수 있습니다. 하지만, 소수의 모델이 완전히 표준이고, 대부분의 과학자들은 독창적인 것을 만들기를 원합니다. 이 절에서는 파라미터를 다루는 방법에 대해서 살펴보겠습니다. 좀 더 자세하게는 아래와 같은 것들을 포함합니다.

- 디버깅이나 분석을 위해서 파라미터를 접근하고, 그것들을 시각화하거나 저장하는 것을 통해서 커스텀 모델을 어떻게 만들어야 하는지 이해를 시작하겠습니다.
- 다음으로는 초기화 목적 등을 위해서 특별한 방법으로 파라미터들을 설정해야 하는데, 이를 위해서 파라미터 초기화 도구의 구조에 대해서 논의합니다.
- 마지막으로 일부 파라미터를 공유하는 네트워크를 만들면서 이 내용들이 어떻게 적용되는지 보겠습니다.

지금까지 그랬듯이 은닉층(hidden layer)을 갖는 다층 퍼셉트론(multilayer perceptron)으로부터 시작하겠습니다. 이를 이용해서 다양한 특징들을 살펴봅니다.

```
[1]: from mxnet import init, nd
from mxnet.gluon import nn

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize() # Use the default initialization method

x = nd.random.uniform(shape=(2, 20))
net(x) # Forward computation
```

```
[1]: [[ 0.09543004  0.04614332 -0.00286654 -0.07790349 -0.05130243  0.02942037
   0.08696642 -0.0190793  -0.04122177  0.05088576]
 [ 0.0769287   0.03099705  0.00856576 -0.04467199 -0.06926839  0.09132434
   0.06786595 -0.06187842 -0.03436673  0.04234694]]
<NDArray 2x10 @cpu(0)>
```

## 6.2.1 파라미터 접근

Sequential 클래스의 경우, 네트워크의 각 층의 인덱스를 사용해서 파라미터를 쉽게 접근할 수 있습니다. params 변수가 필요한 데이터를 가지고 있습니다. 자 그럼 첫번째 층의 파라미터를 조사하는 것을 직접해 보겠습니다.

```
[2]: print(net[0].params)
print(net[1].params)

dense0_ (
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
)
dense1_ (
    Parameter dense1_weight (shape=(10, 256), dtype=float32)
    Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

위 코드의 수행 결과는 많은 것을 우리에게 알려줍니다. 첫번째 정보는 예상대로 이 층은 파라미터들의 두 개의 세트, `dense0_weight` 와 `dense0_bias`,로 구성되어 있는 것을 확인할 수 있습니다. 이 값들은 모두 싱글 프리시전(single precision)이고, 입력 차원이 20이고 출력 차원이 256인 첫번째 층에 필요한 모양(shape)을 갖고 있습니다. 특히, 파라미터들의 이름이 주어지는데 이는 아주 유용합니다. 이름을 사용하면 간단하지 않은 구조를 갖는 수백개의 층들로 구성된 네트워크에서 파라미터를 쉽게 지정할 수 있기 때문입니다. 두 번째 층도 같은 방식으로 구성되어 있는 것을 확인할 수 있습니다.

## 지정된 파라미터

파라미터를 가지고 뭔가 유용한 일을 하기를 원한다면 이 값들을 접근할 수 있어야 합니다. 간단한 방법부터 일반적인 방법까지 다양한 방법이 있는데, 몇 가지를 살펴보겠습니다.

```
[3]: print(net[1].bias)
print(net[1].bias.data())

Parameter dense1_bias (shape=(10,), dtype=float32)

[0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 10 @cpu(0)>
```

첫번째 코드는 두번째 층의 편향(bias)를 출력합니다. 이는 데이터, 그래디언트(gradient) 그리고 추적적인 정보를 가지고 있는 객체이기에, 우리는 데이터를 명시적으로 접근해야 합니다. 우리는 편향

(bias)을 모두 0으로 초기화했기 때문에 편향(bias)이 모두 0임을 기억해두기 바랍니다. 이 값은 파라미터의 이름, `dense0_weight`, 을 이용해서 직접 접근할 수도 있습니다. 이렇게 할 수 있는 이유는 모든 레이어는 직접 접근할 수 있는 고유의 파라미터 사전(dictionary)를 갖고있기 때문입니다. 이 두 방법은 완전히 동일하나, 첫번째 방법이 조금 더 읽기 쉽습니다.

```
[4]: print(net[0].params['dense0_weight'])
print(net[0].params['dense0_weight'].data())

Parameter dense0_weight (shape=(256, 20), dtype=float32)

[[ 0.06700657 -0.00369488  0.0418822 ... -0.05517294 -0.01194733
-0.00369594]
[-0.03296221 -0.04391347  0.03839272 ...  0.05636378  0.02545484
-0.007007 ]
[-0.0196689   0.01582889 -0.00881553 ...  0.01509629 -0.01908049
-0.02449339]
...
[ 0.00010955  0.0439323 -0.04911506 ...  0.06975312  0.0449558
-0.03283203]
[ 0.04106557  0.05671307 -0.00066976 ...  0.06387014 -0.01292654
0.00974177]
[ 0.00297424 -0.0281784 -0.06881659 ... -0.04047417  0.00457048
0.05696651]]
<NDArray 256x20 @cpu(0)>
```

가중치들이 모두 0이 아닌 값으로 되어 있음을 주목하세요. 우리가 네트워크를 만들 때, 이 값들은 난수값으로 초기화했기 때문에 그렇습니다. `data` 함수만 있는 것이 아닙니다. 예를 들어 파라미터에 대해서 그래디언트(gradient)를 계산하고자 할 수도 있습니다. 이 결과는 가중치와 같은 모양(shape)을 갖게 됩니다. 하지만, 역전파(back propagation)을 아직 실행하지 않았기 때문에 이 값들은 모두 0으로 보여질 것입니다.

```
[5]: net[0].weight.grad()

[5]:
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

(continues on next page)

&lt;NDArray 256x20 @cpu(0)&gt;

## 한번에 모든 파라미터 지정

위 방법으로 파라미터를 접근하는 것은 다소 지루할 수 있습니다. 특히, 더 복잡한 블럭들을 갖거나, 블럭들로 구성된 블럭(심지어는 블럭들을 블럭들의 블럭)으로 구성된 네트워크인 경우, 블럭들이 어떻게 생성되었는지 알기 위해서 전체 트리를 모두 뒤져봐야 하는 경우가 그런 예입니다. 이를 피하기 위해서, 블럭은 `collect_params`라는 메소드를 제공하는데 이를 이용하면 네트워크의 모든 파라미터를 하나의 사전(dictionary)에 담아주고, 쉽게 조회할 수 있습니다. 이는 내부적으로 블럭의 모든 구성 요소들을 방문하면서 필요한 경우 서브블럭들에 `collect_params` 함수를 호출하는 식으로 동작합니다. 차이를 확인하기 위해서 아래 코드를 살펴 보겠습니다.

```
[6]: # parameters only for the first layer
print(net[0].collect_params())
# parameters of the entire network
print(net.collect_params())

dense0_ (
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
)
sequential0_ (
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
    Parameter dense1_weight (shape=(10, 256), dtype=float32)
    Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

이렇게 해서 네트워크의 파라미터를 접근하는 세번째 방법을 배웠습니다. 두번째 층의 편향(bias) 값을 확인하는 코드는 아래와 같이 간단하게 작성할 수 있습니다.

```
[7]: net.collect_params()['dense1_bias'].data()
[7]:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

이 책에서 설명을 계속하면서, 블럭들의 하위 블럭에 이름이 어떻게 부여되는지 보게될 것입니다.(그

중에, Sequential의 경우는 숫자를 할당합니다.) 이름 할당 규칙은 필요한 파라미터만 필터링하는 정규식을 사용할 수 있게해서 아주 편리합니다.

```
[8]: print(net.collect_params('.*weight'))
print(net.collect_params('dense0.*'))

sequential0_ (
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense1_weight (shape=(10, 256), dtype=float32)
)
sequential0_ (
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
)
```

## 루브 골드버그가 다시 공격하다.

블럭들이 중첩되어 있는 경우 파라미터의 이름이 어떤식으로 매겨지는지 보겠습니다. 이를 위해서 우리는 블럭들을 생성하는 함수(block factory 라고 불릴 수 있는)를 정의하고, 이를 이용해서 더 큰 블럭들이 블럭을 포함시켜보겠습니다.

```
[9]: def block1():
    net = nn.Sequential()
    net.add(nn.Dense(32, activation='relu'))
    net.add(nn.Dense(16, activation='relu'))
    return net

def block2():
    net = nn.Sequential()
    for i in range(4):
        net.add(block1())
    return net

rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(x)

[9]:
[[ 1.0116727e-08 -9.4839003e-10 -1.1526797e-08  1.4917443e-08
```

(continues on next page)

```
-1.5690811e-09 -3.9257650e-09 -4.1441655e-09  9.3013472e-09
 3.2393586e-09 -4.8612452e-09]
[ 9.0111598e-09 -1.9115812e-10 -8.9595842e-09  1.0745880e-08
 1.4963460e-10 -2.2272872e-09 -3.9153973e-09  7.0595711e-09
 3.4854222e-09 -4.5807327e-09]]
<NDArray 2x10 @cpu(0)>
```

네트워크를 설계했으니, 어떻게 구성되는지 확인해봅니다. `collect_params` 를 이용하면 이름과 논리적인 구조에 대한 정보를 얻을 수 있습니다.

```
[10]: print(rgnet.collect_params)
print(rgnet.collect_params())

<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(20 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (2): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (3): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
  )
  (1): Dense(16 -> 10, linear)
) >
sequential1_ (
  Parameter dense2_weight (shape=(32, 20), dtype=float32)
  Parameter dense2_bias (shape=(32,), dtype=float32)
  Parameter dense3_weight (shape=(16, 32), dtype=float32)
  Parameter dense3_bias (shape=(16,), dtype=float32)
```

(continues on next page)

```

Parameter dense4_weight (shape=(32, 16), dtype=float32)
Parameter dense4_bias (shape=(32,), dtype=float32)
Parameter dense5_weight (shape=(16, 32), dtype=float32)
Parameter dense5_bias (shape=(16,), dtype=float32)
Parameter dense6_weight (shape=(32, 16), dtype=float32)
Parameter dense6_bias (shape=(32,), dtype=float32)
Parameter dense7_weight (shape=(16, 32), dtype=float32)
Parameter dense7_bias (shape=(16,), dtype=float32)
Parameter dense8_weight (shape=(32, 16), dtype=float32)
Parameter dense8_bias (shape=(32,), dtype=float32)
Parameter dense9_weight (shape=(16, 32), dtype=float32)
Parameter dense9_bias (shape=(16,), dtype=float32)
Parameter dense10_weight (shape=(10, 16), dtype=float32)
Parameter dense10_bias (shape=(10,), dtype=float32)
)

```

총들이 계층적으로 생성되어 있으니, 우리도 총들을 그렇게 접근할 수 있습니다. 예를 들어서, 첫번째 큰 블럭의 두번째 하위 블럭의 첫번째 층의 편향(bias) 값은 다음과 같이 접근이 가능합니다.

```

[11]: rgnet[0][1][0].bias.data()

[11]:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 32 @cpu(0)>

```

## 6.2.2 파라미터 초기화

자 이제 파라미터를 어떻게 접근할 수 있는지 알게되었으니, 파라미터를 어떻게 적절하게 초기화할 수 있을지를 살펴볼 차례입니다. 이전 장에서 [초기화](#) 가 왜 필요한지를 설명했습니다. 기본 설명으로는 MXNet은 가중치 행렬은  $U[-0.07, 0.07]$  을 따르는 균일한 난수로, 편향(bias) 파라미터는 모두 0으로 설정합니다. 하지만, 때로는 가중치 값을 다르게 초기화 해야 할 필요가 있습니다. MXNet의 `init` 모듈은 미리 설정된 다양한 초기화 방법들을 제공하는데, 만약 특별한 방법으로 초기화하는 것이 필요하다면 몇 가지 추가적인 일이 필요합니다.

## 제공되는 초기화

빌트인 초기화 방법들을 우선 살펴보겠습니다. 아래 코드는 모든 파라미터를 Gaussian 확률 변수로 초기화하는 예제입니다.

```
[12]: # force_reinit ensures that the variables are initialized again, regardless of
# whether they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data() [0]
```

```
[12]: [-0.008166 -0.00159167 -0.00273115  0.00684697  0.01204039  0.01359703
       0.00776908 -0.00640936  0.00256858  0.00545601  0.0018105 -0.00914027
       0.00133803  0.01070259 -0.00368285  0.01432678  0.00558631 -0.01479764
       0.00879013  0.00460165]
<NDArray 20 @cpu(0)>
```

만약 파라미터들을 모두 1로 초기화하고 싶다면, 초기화 방법을 Constant (1)로 바꾸기만 하면 됩니다.

```
[13]: net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data() [0]
```

```
[13]: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
<NDArray 20 @cpu(0)>
```

만약 특정 파라미터만 다른 방법으로 초기화를 하고 싶다면, 해당하는 서브블럭에 초기화 함수를 지정하는 것으로 간단히 구현할 수 있습니다. 예를 들어, 아래 코드는 두번째 층을 42라는 값으로 초기화하고, 첫번째 층의 가중치들은 Xavier 초기화 방법을 적용하고 있습니다.

```
[14]: net[1].initialize(init=init.Constant(42), force_reinit=True)
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
print(net[1].weight.data() [0,0])
print(net[0].weight.data() [0])
```

```
[42.]
<NDArray 1 @cpu(0)>

[-0.14511706 -0.01173057 -0.03754489 -0.14020921  0.00900492  0.01712246
 0.12447387 -0.04094418 -0.12105145  0.00079902 -0.0277361 -0.10213967
 -0.14027238 -0.02196661 -0.04641148  0.11977354  0.03604397 -0.14493202
 (continues on next page)
```

```
-0.06514931  0.13826048]
<NDArray 20 @cpu(0)>
```

## 커스텀 초기화

때로는 우리가 필요한 초기화 방법이 `init` 모듈에 없을 수도 있습니다. 이 경우에는, `Initializer` 클래스의 하위 클래스를 정의해서 다른 초기화 메소드와 같은 방법으로 사용할 수 있습니다. 보통은, `_init_weight` 함수만 구현하면 됩니다. 이 함수는 입력 받은 `NDArray`를 원하는 초기값으로 바꿔줍니다. 아래 예제에서는 이를 잘 보여주기 위해서 다소 이상하고 특이한 분포를 사용해서 값을 초기화합니다.

$$w \sim \begin{cases} U[5, 10] & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U[-10, -5] & \text{with probability } \frac{1}{4} \end{cases}$$

```
[15]: class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = nd.random.uniform(low=-10, high=10, shape=data.shape)
        data *= data.abs() >= 5

    net.initialize(MyInit(), force_reinit=True)
    net[0].weight.data() [0]

    Init dense0_weight (256, 20)
    Init dense1_weight (10, 256)

[15]: [-5.44481   6.536484  -0.         0.         0.         7.7452965
       7.739216   7.6021366   0.         -0.        -7.3307705 -0.
       9.611603   0.         7.4357147   0.         0.         -0.
      8.446959   0.          ]
```

이 기능이 충분하지 않을 경우에는, 파라미터 값을 직접 설정할 수도 있습니다. `data()` 는 `NDArray`를 반환하기 때문에, 이를 이용하면 일반적인 행렬처럼 사용하면 됩니다. 고급 사용자들을 위해서 조금 더 설명하면, `autograd` 범위 안에서 파라미터를 조정하는 경우에는, 자동 미분 기능이 오작동하지

않도록 `set_data` 를 사용해야하는 것을 기억해두세요.

```
[16]: net[0].weight.data()[:] += 1
net[0].weight.data()[0, 0] = 42
net[0].weight.data()[0]

[16]:
[42.          7.536484   1.          1.          1.          8.7452965
 8.739216    8.602137   1.          1.         -6.3307705   1.
 10.611603    1.          8.435715   1.          1.          1.
 9.446959    1.          ]  
<NDArray 20 @cpu(0)>
```

### 6.2.3 묶인(Tied) 파라미터들

다른 어떤 경우에는, 여러 층들이 모델 파라미터를 공유하는 것이 필요하기도 합니다. 예를 들면, 좋은 단어 임베딩을 찾는 경우, 단어 인코딩과 디코딩에 같은 파라미터를 사용하도록 하는 결정할 수 있습니다. 이런 경우는 [Blocks](#)에서도 소개되었습니다. 이것을 보다 깔끔하게 구현하는 방법을 알아보겠습니다. 아래 코드에서는 텐스층(dense layer)을 하나 정의하고, 다른 층에 파라미터값을 동일하게 설정하는 것을 보여주고 있습니다.

```
[17]: net = nn.Sequential()
# We need to give the shared layer a name such that we can reference its
# parameters
shared = nn.Dense(8, activation='relu')
net.add(nn.Dense(8, activation='relu'),
        shared,
        nn.Dense(8, activation='relu', params=shared.params),
        nn.Dense(10))
net.initialize()

x = nd.random.uniform(shape=(2, 20))
net(x)

# Check whether the parameters are the same
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
# Make sure that they're actually the same object rather than just having the
# same value
print(net[1].weight.data()[0] == net[2].weight.data()[0])
```

```
[1. 1. 1. 1. 1. 1. 1.]  
<NDArray 8 @cpu(0)>
```

```
[1. 1. 1. 1. 1. 1. 1.]  
<NDArray 8 @cpu(0)>
```

위 예제는 두번째, 세번째 층의 파라미터가 묶여있는 것(tied)을 보여줍니다. 이 파라미터들은 값이 같은 수준이 아니라, 동일합니다. 즉, 하나의 파라미터를 바꾸면 다른 파라미터의 값도 함께 바뀝니다. 그레디언트(gradient)들에 일어나는 현상은 아주 독창적입니다. 모델은 파라미터는 그레디언트(gradient)를 갖고 있기 때문에, 두번째와 세번째 층의 그레디언트(gradient)들은 역전파(back propagation) 단계에서 `shared.params.grad()` 함수에 의해서 누적됩니다.

#### 6.2.4 요약

- 모델 파라미터를 접근하고, 초기화하고, 서로 묶는 다양한 방법이 있습니다.
- 커스텀 초기화를 사용할 수 있습니다.
- Gluon은 독특하고 계층적인 방법으로 파라미터에 접근하는 정교한 방법을 제공합니다.

#### 6.2.5 문제

1. 이전 절의 FancyMLP 정의를 사용해서, 다양한 레이어의 파라미터에 접근해보세요.
2. MXNet documentation의 다양한 초기화 방법들을 살펴보세요.
3. `net.initialize()` 수행 후와 `net(x)` 수행 전에 모델 파라미터를 확인해서, 모델 파라미터들의 모양(shape)를 관찰해보세요. 무엇 바뀌어 있고, 왜 그럴까요?
4. 파라미터를 공유하는 레이어를 갖는 다층 퍼셉트론(multilayer perceptron)을 만들어서 학습을 시켜보세요. 학습 과정을 수행하면서 모델 각 층의 파라미터들과 그레디언트(gradient) 값을 관찰해보세요.

## 6.2.6 Scan the QR Code to Discuss



## 6.3 초기화 지연(deferred Initialization)

앞의 예제들에서는 네트워크들을 빠르게 그리고 조금은 느슨하게 만들어왔습니다. 특히 다음과 같이 동작하지 않을 것처럼 보일 수 있는 것들을 했습니다.

- 입력의 차원을 고려하지 않고 네트워크 아키텍처를 정의했습니다.
- 이전 층(layer)의 출력 차원을 고려하지 않고 다음 층에 추가했습니다.
- 얼마나 많은 파라미터들이 있을지 모르는 상태에서 이 파라미터들을 초기화까지 했습니다.

이 모든 것이 불가능하게 들리고, 실제로도 불가능합니다. 사실 MXNet이나 다른 프레임워크들이 네트워크에 들어올 입력값의 차원을 예측할 수 있는 방법은 없습니다. 이후에 살펴 볼, 컨볼루션(convolutional) 네트워크나 이미지를 다룰 때 이 문제는 더욱 그렇게 보일 것입니다. 그 이유는 이미지의 해상도 같은 입력의 차원은 네트워크의 연속된 층들의 차원에 영향을 미치기 때문입니다. 따라서 코드를 작성할 때 차원이 무엇인지 미리 알 필요 없이 파라미터를 설정할 수 있는 능력은 통계적인 모델링을 아주 간단하게 해줄 수 있습니다. 지금부터 초기화를 예로 어떻게 동작하는지 살펴보겠습니다. 결국에는 존재하는지 모르는 변수를 초기화하는 것은 불가능합니다.

### 6.3.1 네트워크 생성하기

네트워크에 대한 인스턴스를 만들면 일어나는 일을 살펴보겠습니다. 앞에서와 같이 MLP를 사용합니다.

```
[1]: from mxnet import init, nd
from mxnet.gluon import nn

def getnet():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
```

(continues on next page)

```
net.add(nn.Dense(10))
return net

net = getnet()
```

이 시점에서 네트워크는 여러 파라미터들의 차원이 어떻게 되는지를 알 수 있는 방법이 없습니다. 이 상태에서 말할 수 있는 사실은 각 층의 차원이 무엇이 되던 가중치들과 편향들이 필요하다는 것입니다. 파라미터를 읽어보면 이것을 알 수 있습니다.

```
[2]: print(net.collect_params())
print(net.collect_params())

<bound method Block.collect_params of Sequential(
    (0): Dense(None -> 256, Activation(relu))
    (1): Dense(None -> 10, linear)
)>
sequential0_ (
    Parameter dense0_weight (shape=(256, 0), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
    Parameter dense1_weight (shape=(10, 0), dtype=float32)
    Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

`net[0].weight.data(0)` 을 수행하면 무언가를 하기 위해서는 네트워크가 초기화 되어야한다는 런타임 에러를 만나게 됩니다. 파라미터를 초기화한 후, 무엇인 바뀌는지를 확인해보겠습니다.

```
[3]: net.initialize()
net.collect_params()

[3]: sequential0_ (
    Parameter dense0_weight (shape=(256, 0), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
    Parameter dense1_weight (shape=(10, 0), dtype=float32)
    Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

결과에서 볼 수 있듯이, 아무것도 바뀐 것이 없습니다. 네트워크에 데이터를 입력하는 경우에 비로소 변화가 생기게 됩니다. 한번 해보겠습니다.

```
[4]: x = nd.random.uniform(shape=(2, 20))
net(x) # Forward computation

net.collect_params()

[4]: sequential0_(
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
    Parameter dense1_weight (shape=(10, 256), dtype=float32)
    Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

이전에 대비해서 주요 차이점은 입력에 대한 차원,  $x \in \mathbb{R}^{20}$  을 알게 되면, 첫번째 층의 가중치 행렬을 정의할 수 있다는 것입니다. 이렇게 되면, 우리는 두번째 층에 대한 차원을  $10 \times 256$  으로 결정할 수 있게 됩니다. 계속 연산 그래프를 따라서 차원이 결정되게 됩니다. 이것이 끝나면, 파라미터를 초기화하는 것을 진행할 수 있습니다. 이 방법이 위 세가지 문제의 해결책입니다.

### 6.3.2 초기화 지연의 적용

이론적으로 어떻게 동작하는지를 배웠으니, 초기화가 실제로 언제 일어나는지 보겠습니다. 이를 확인하기 위해서 어떤 파라미터를 초기화할지를 지정하면서 호출되면 아무것도 하지 않지만 디버그 메시지를 출력하는 초기화 클래스를 하나 정의합니다.

```
[5]: class MyInit(init.Initializer):
    def __init_weight(self, name, data):
        print('Init', name, data.shape)
        # The actual initialization logic is omitted here

net = getnet()
net.initialize(init=MyInit())
```

MyInit 은 호출되면 모델 파라미터에 대한 정보를 출력하게 만들어졌는데, initialize 함수를 호출해도 아무런 정보가 출력되지 않고 있습니다. 즉, initialize 함수가 호출이 되어도 실제 파라미터 초기화가 일어나지 않습니다. 이제 입력값을 정의하고 forward 연산을 수행해봅니다.

```
[6]: x = nd.random.uniform(shape=(2, 20))
y = net(x)

Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

이제야 모델 파라미터들에 대한 정보가 화면에 출력됩니다. 주어진 입력  $x$ 에 대한 forward 연산을 수행할 때, 시스템은 입력의 모양(shape)을 기반으로 모든 층의 가중치 파라미터의 모양(shape)을 추론해냅니다. 시스템이 이 파라미터들을 생성하고 나면, MyInit 인스턴스를 호출해서 파라미터들을 초기화한 후, forward 연산을 수행하게 됩니다.

물론 이 초기화는 최초 forward 연산을 수행할 때만 일어납니다. 즉, 이후에  $\text{net}(x)$  을 호출해서 forward 연산이 수행되면 재초기화가 수행되지 않고, 따라서 MyInit 인스턴스의 결과는 다시 출력되지 않습니다.

```
[7]: y = net(x)
```

이 절을 시작하면서 언급했듯이, 지연된 초기화는 혼동을 가져올 수도 있습니다. 예를 들면, 첫번째 forward 연산 전에는, 모델 파라미터를 직접 바꾸는 것이 불가능합니다. 즉, `data`나 `set_data` 함수를 호출해서 모델 파라미터의 값을 얻거나 바꾸는 것이 불가능합니다. 따라서, 필요한 경우에는 샘플 입력을 이용해서 네트워크를 강제 초기화하기도 합니다.

### 6.3.3 강제 초기화

`initialize` 함수가 수행되는 시점에 시스템에 모든 파라미터의 모양(shape)을 하는 경우에는 지연된 초기화가 일어나지 않습니다. 아래 두 가지가 그런 경우입니다.

- 이미 어떤 데이터를 봤고, 파라미터를 재설정하고 싶은 경우
- 네트워크를 정의할 때, 모든 입력과 출력의 차원을 알고 있는 경우

첫번째 경우는 아래 예제 코드처럼 간단히 할 수 있습니다.

```
[8]: net.initialize(init=MyInit(), force_reinit=True)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

두번째 경우는 레이어를 생성할 때 파라미터에 대한 정보를 명시 해줘야합니다. 예를 들어, 덴스층(dense layer)의 경우에는 `in_units`에 대한 값을 명시하면 `initialize` 가 호출되면 파라미터가 바로 초기화됩니다.

```
[9]: net = nn.Sequential()
net.add(nn.Dense(256, in_units=20, activation='relu'))
net.add(nn.Dense(10, in_units=256))

net.initialize(init=MyInit())
```

```
Init dense4_weight (256, 20)
Init dense5_weight (10, 256)
```

### 6.3.4 요약

- 지연된 초기화는 좋습니다. 지연된 초기화는 Gluon이 많은 것들을 자동으로 설정할 수 있게 해주고, 새로운 네트워크 아키텍처를 정의할 때 발생할 수 있는 많은 오류 요소를 제거해줍니다.
- 간접적으로 정의된 변수의 값을 할당 하면 이 기능은 우회할 수 있습니다.
- `force_reinit=True` 플래그를 사용해서 초기화를 강제하거나 다시 수행하게 할 수 있습니다.

### 6.3.5 문제

- 입력들의 일부만 차원을 명시하면 어떤 일이 일어날까요? 이 경우에도 즉시 초기화가 수행되나요?
- 잘못된 차원을 명시하면 어떻게 될까요?
- 차원이 변하는 입력이 있을 때 어떻게 해야할까요? 힌트 - 파라미터 묶기를 참조하세요.

### 6.3.6 Scan the QR Code to Discuss



## 6.4 커스텀 층(custom layer)

딥러닝의 성공 요인 중에 하나는 딥 네트워크에서 사용할 수 있는 다양한 종류의 층(layer)이 있다 는 점에서 찾아볼 수 있습니다. 즉, 다양한 형태의 층을 사용해서 많은 종류의 커스터마이징과 다양한 문제에 적용이 가능하게 되었습니다. 예를 들면, 과학자들이 이미지, 텍스트, 풀링, loop, 동적 프로그래밍, 그리고 심지어는 컴퓨터 프로그램을 위한 층을 발명해왔습니다. 앞으로도 Gluon에 현재 존재하지

않은 새로운 층을 만나게될 것이고, 어쩌면 여러분이 만난 문제를 해결하기 위해서 새로운 층을 직접 발명 할지도 모릅니다. 자 그럼 커스텀 층을 만들어 보는 것을 이 절에서 배워보겠습니다.

### 6.4.1 파라미터가 없는 층(layer)

커스텀 층을 만드는 것은 다소 복잡할 수 있기 때문에, 파라미터를 계승 받지 않는 커스텀 층(또는 Block)을 만드는 것부터 시작해보겠습니다. 첫번째 시작은 이전에 *introduced blocks* 에서 소개했던 것과 비슷합니다. 아래 CenteredLayer 클래스는 입력에서 평균을 빼는 것을 계산하는 층을 정의 합니다. 우리는 이것을 Block 클래스를 상속하고, forward 메소드를 구현해서 만듭니다.

```
[1]: from mxnet import gluon, nd
from mxnet.gluon import nn

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)

    def forward(self, x):
        return x - x.mean()
```

어떻게 동작하는지 보기 위해서, 데이터를 층에 입력해봅니다.

```
[2]: layer = CenteredLayer()
layer(nd.array([1, 2, 3, 4, 5]))
```

```
[2]: [-2. -1.  0.  1.  2.]
<NDArray 5 @cpu(0)>
```

우리는 이를 사용해서 더 복잡한 모델을 만들 수도 있습니다.

```
[3]: net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

그럼 이 가운데로 만들어주는 층이 잘 작동하는지 보겠습니다. 이를 위해서 난수 데이터를 생성하고, 네트워크에 입력한 후 평균만큼 값이 조정되는지 확인합니다. 우리가 다루는 변수가 실수형이기 때문에, 아주 작지만 0이 아닌 숫자를 보게될 것임을 염두하세요.

```
[4]: y = net(nd.random.uniform(shape=(4, 8)))
y.mean().asscalar()
```

```
[4]: -7.212293e-10
```

## 6.4.2 파라미터가 있는 층

층을 어떻게 정의하는지 원리를 알게 되었으니, 파라미터를 갖는 층을 정의해보겠습니다. 이 파라미터들은 학습을 통해서 조정될 값들입니다. 딥러닝 연구자들의 일을 편하게 만들어 주기 위해서, Parameter 클래스와 ParameterDict 사전(dictionary)은 많이 사용하는 기능을 제공하고 있습니다. 이 클래스들은 접근을 관리하고, 초기화를 하고, 공유를 하고, 모델 파라미터를 저장하고 로딩하는 기능을 관리해줍니다. 예를 들면, 새로운 커스텀 층을 만든 때 매번 직렬화(serialization) 루틴을 작성할 필요가 없습니다.

다른 예로는, Block 클래스와 함께 제공되는 ParameterDict 타입인 params 를 사용할 수도 있습니다. 이 사전(dictionary)는 문자 타입의 파라미터 이름을 Parameter 타입의 모델 파라미터로 매핑하는 기능을 제공합니다. ParameterDict 의 get 함수를 사용해서 Parameter 인스턴스를 생성하는 것도 가능합니다.

```
[5]: params = gluon.ParameterDict()
params.get('param2', shape=(2, 3))
params
[5]: (
      Parameter param2 (shape=(2, 3), dtype=<class 'numpy.float32'>)
    )
```

텐스층(dense layer)을 직접 구현해보겠습니다. 이 층은 두 파라미터, 가중치과 편향(bias)을 갖습니다. 약간 특별하게 만들기 위해서, ReLU 활성화 함수를 기본으로 적용하도록 만들어봅니다. 가중치와 편향 파라미터를 갖는 완전 연결층(fully connected layer)을 구현하고, ReLU를 활성화 함수로 추가합니다. in\_units와 units 는 각각 입력과 출력의 개수입니다.

```
[6]: class MyDense(nn.Block):
    # units: the number of outputs in this layer; in_units: the number of
    # inputs in this layer
    def __init__(self, units, in_units, **kwargs):
        super(MyDense, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units, units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = nd.dot(x, self.weight.data()) + self.bias.data()
        return nd.relu(linear)
```

파라미터에 이름을 부여하는 것은 이후에 사전(dictionary) 조회를 통해서 원하는 파라미터를 직접 접근할 수 있도록 해줍니다. 그렇기 때문에, 잘 설명하는 이름을 정하는 것이 좋은 생각입니다. 자 이제 MyDense 클래스의 인스턴스를 만들고 모델 파라미터들을 직접 확인해봅니다.

```
[7]: dense = MyDense(units=3, in_units=5)
dense.params

[7]: mydense0_ (
    Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
    Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
)
```

커스텀 층의 forward 연산을 수행합니다.

```
[8]: dense.initialize()
dense(nd.random.uniform(shape=(2, 5)))

[8]:
[[0.06917784 0.01627153 0.01029644]
 [0.02602214 0.0453731 0.        ]]
<NDArray 2x3 @cpu(0)>
```

커스텀 층을 이용해서 모델은 만들어 보겠습니다. 만들어진 모델은 기본으로 제공되는 텐스층(dense layer)처럼 사용할 수 있습니다. 하나 다른 점은 입력, 출력의 크기를 자동으로 계산하는 것이 없다는 점입니다. 어떻게 이 기능을 구현할 수 있는지는 [MXNet documentation](#) 을 참고하세요.

```
[9]: net = nn.Sequential()
net.add(MyDense(8, in_units=64),
       MyDense(1, in_units=8))
net.initialize()
net(nd.random.uniform(shape=(2, 64)))

[9]:
[[0.03820474]
 [0.04035058]]
<NDArray 2x1 @cpu(0)>
```

### 6.4.3 요약

- Block 클래스를 이용해서 커스텀 층(layer)를 만들 수 있습니다. 이 방법은 블럭 팩토리를 정의하는 것보다 더 강력한 방법인데, 그 이유는 다양한 컨텍스트(context)들에서 불려질 수 있기 때문입니다.

- 블럭들은 로컬 파라미터를 가질 수 있습니다.

#### 6.4.4 문제

1. 데이터에 대해서 affine 변환을 학습하는 층을 디자인하세요. 예를 들면, 평균 값을 빼고, 대신 더할 파라미터를 학습합니다.
2. 입력을 받아서 텐서 축소를 하는 층을 만들어 보세요. 즉,  $y_k = \sum_{i,j} W_{ijk}x_i x_j$  를 반환합니다.
3. 데이터에 대한 퓨리에 계수의 앞에서 반을 리턴하는 층을 만들어보세요. 힌트 - MXNet의 `fft` 함수를 참고하세요.

#### 6.4.5 Scan the QR Code to Discuss



i# 파일 입/출력

지금까지 우리는 데이터를 처리하고, 딥러닝 모델을 만들고, 학습시키고, 테스트하는 방법들을 알아봤습니다. 실험을 진행하다가 어느 시점에는 얻은 결과가 만족스러워서, 나중에도 활용하고 배포하기 위해서 결과를 저장할 필요가 생깁니다. 또한 긴 학습을 수행할 때 서버의 전원에 문제가 생겼을 때 며칠 동안 수행한 내용을 잃지 않기 위해서 중간 결과들을 저장하는 것이 가장 최선의 방법이기도 합니다. 예를 들면, 영어 단어 임베딩을 가지고, 멋진 스팸 분류기를 만들고자 하는 경우, 미리 학습된 (pretrained) 모델을 읽어야 하는 경우도 있습니다. 이 모든 경우를 수행하기 위해서는, 개별 가중치 벡터들 또는 전체 모델을 저장하고 읽어야 합니다. 이번 절에서는 이 두 가지에 대해서 알아보겠습니다.

## 6.5 NDArray

가장 간단한 방법은 `save` 와 `load` 함수를 직접 호출해서 NDArray를 하나씩 저장하고 읽을 수 있습니다. 이는 다음과 같이 저장하는 것을 간단히 구현할 수 있습니다.

```
[1]: from mxnet import nd
from mxnet.gluon import nn
```

(continues on next page)

```
x = nd.arange(4)
nd.save('x-file', x)
```

그리고, 우리는 이 파일을 메모리로 다시 읽습니다.

```
[2]: x2 = nd.load('x-file')
x2
```

```
[2]: [
    [0. 1. 2. 3.]
    <NDArray 4 @cpu(0)>]
```

하나의 NDArray 객체 뿐만 아니라, NDArray들의 리스트도 저장하고 다시 메모리로 읽기도 가능합니다.

```
[3]: y = nd.zeros(4)
nd.save('x-files', [x, y])
x2, y2 = nd.load('x-files')
(x2, y2)
```

```
[3]: (
    [0. 1. 2. 3.]
    <NDArray 4 @cpu(0)>,
    [0. 0. 0. 0.]
    <NDArray 4 @cpu(0)>)
```

문자를 NDArray로 매핑하는 사전(dictionary)를 저장하고 읽는 것도 가능합니다. 이 방법은 모델의 전체 가중치들을 한꺼번에 저장하고 읽을 때 유용합니다.

```
[4]: mydict = {'x': x, 'y': y}
nd.save('mydict', mydict)
mydict2 = nd.load('mydict')
mydict2
```

```
[4]: {'x':
    [0. 1. 2. 3.]
    <NDArray 4 @cpu(0)>, 'y':
    [0. 0. 0. 0.]
    <NDArray 4 @cpu(0)>}
```

## 6.6 Gluon 모델 파라미터들

가중치 벡터를 하나씩 (또는 NDArray 텐서들) 저장하는 것이 유용하지만, 모델 전체를 저장하고 이후에 읽는데는 매우 불편한 방법입니다. 왜냐하면 모델 전체에 걸쳐서 수백개의 파라미터 그룹들이 있을 수 있기 때문입니다. 만약 모든 값을 모아서 아키텍처에 매핑시키는 스크립트를 작성한다면 매우 많은 일을 해야합니다. 이런 이유로 Gluon은 개별 가중치 벡터를 저장하는 것보다는 전체 네트워크를 저장하고 읽을 수 있는 기능을 제공합니다. 유의해야 할 점은 전체 모델을 저장하는 것이 아니라, 모델의 파라미터들을 저장한다는 것입니다. 만약에 3개 층을 갖는 MLP가 있다면, 네트워크의 아키텍처는 별도로 명시해줘야 합니다. 이렇게 한 이유는 모델들 자체는 임의의 코드를 담고 있을 수 있는데 이 경우에는 코드가 쉽게 직렬화(serialization)되지 않을 수 있기 때문입니다. (단, 컴파일된 모델의 경우에는 방법이 있는데, 기술적인 자세한 내용은 [MXNet documentation](#) 을 참고하세요.) 결국, 모델을 다시 만들려면, 아키텍처를 코드 형태로 만들고, 디스크로부터 파라미터를 로딩해야합니다. [지연된 초기화](#) 는 실제 값을 할당할 필요 없이 모델을 정의할 수 있기 때문에 이런 방식에 아주 도움이 됩니다. 역시 우리의 MLP를 사용해서 설명하겠습니다.

```
[5]: class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()
x = nd.random.uniform(shape=(2, 20))
y = net(x)
```

모델 파라미터들을 'mlp.params'라는 이름의 파일에 저장합니다.

```
[6]: net.save_parameters('mlp.params')
```

모델을 복원할 수 있는지 확인하기 위해서, 원본 MLP 모델의 복사본을 만듭니다. 모델 파라미터를 난수로 초기화하는 것이 아니라, 파일에 저장했던 파라미터들을 직접 읽습니다.

```
[7]: clone = MLP()
clone.load_parameters('mlp.params')
```

두 모델의 인스턴스가 같은 모델 파라미터를 갖고 있기 때문에, 같은 입력 x 를 가지고 계산한 결과는

같아야합니다. 확인해보겠습니다.

```
[8]: yclone = clone(x)
yclone == y

[8]:
[[1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]]
<NDArray 2x10 @cpu(0)>
```

## 6.7 요약

- `save` 와 `load` 함수를 이용해서 `NDArray` 객체들을 파일에 저장하고 읽을 수 있습니다.
- `load_parameters` 와 `save_parameters` 함수는 Gluon의 네트워크 전체 파라미터를 저장하고 읽는데 사용됩니다.
- 아키텍처를 저장하는 것은 파라미터와는 별도로 코드로 해야합니다.

## 6.8 문제

1. 다른 디바이스에 학습된 모델을 배포할 필요가 없을 경우라도, 모델 파라미터를 저장할 수 있을 때 얻을 수 있는 실용적인 이점은 무엇인가요?
2. 네트워크의 일부를 다른 아키텍처의 네트워크에 포함해야 한다고 가정합니다. 예를 들어 이전 네트워크의 처음 두 개층을 새로운 네트워크에서 어떻게 사용할 수 있을까요?
3. 네트워크 아키텍처와 파라미터를 저장하는 방법이 무엇이 있을까요? 네트워크 아키텍처에 어떤 제약을 둬야 할까요?

## 6.9 Scan the QR Code to Discuss



## 6.10 GPU

이 책의 소개에서 우리는 지난 이십년간 연산 능력의 급격한 증가에 대해서 논의했습니다. 간단하게 말하면, GPU 성능이 2000년부터 10년마다 약 1000배씩 증가해왔습니다. 이런 것이 우리에게 엄청난 기회를 주기도 하고, 그러한 성능을 제공할 필요성을 제시하기도 합니다.

연대	데이터셋	메모리	초당 부동소수점 연산수
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

여러분의 연구를 위해서 이 컴퓨팅 성능을 활용하는 방법에 대해서 논의하는 것으로 시작해보겠습니다. 우선은 하나의 GPU를 사용해보겠고, 이후에는 여러 GPU 및 (여러 GPU를 갖는) 여러 서버를 사용하는 방법에 대해서 다루겠습니다. 이미 눈치 챘겠지만, MXNet NDArray는 NumPy와 거의 유사합니다. 하지만, 몇 가지 중요한 차이점들이 있습니다. MXNet를 NumPy와 다르게 만드는 중요한 특징 중 하나는 다양한 하드웨어 디바이스를 지원한다는 점입니다.

MXNet의 모든 배열은 컨텍스트(context)를 갖습니다. 사실, 설명을 하기는 않았지만 지금까지 NDArray를 출력할 때마다, @cpu (0) 라는 이상한 내용이 결과에 함께 출력 되었습니다. 이것이 의미하는 것은 해당 연산이 CPU에서 수행되었다는 것입니다. 다른 컨텍스트(context)들로는 다양한 GPU들이 될 수 있습니다. 작업을 여러 서버에 배포하는 경우에는 상황이 더 어려워질 수 있습니다. 배열을 컨텍스트(context)들에 지능적으로 할당하면, 디바이스간에 데이터가 전송되는 시간을 최소화할 수 있습니다. 예를 들면, GPU 하나를 가지고 있는 서버에서 뉴럴 네트워크를 학습시키는 경우, 모델 파라미터가 GPU에 상주하는 것이 유리합니다.

요약하면, 복잡한 뉴럴 네트워크나 큰 스케일의 데이터를 다룰 때, CPU만을 사용해서 연산을 수행하는 것은 비효율적일 수 있습니다. 이 절에서 우리는 하나의 NVIDIA GPU를 사용해서 연산을 수행하는 것을 설명하겠습니다. 우선, 여러분의 시스템에 적어도 한개의 NVIDIA GPU가 설치되어 있는지 확인하세요. 다음, CUDA를 [다운로드](#)하고 경로를 적절히 설정하세요. 준비가 끝났다면, nvidia-smi 명령을 사용해서 그래픽 카드 정보를 조회해볼 수 있습니다.

```
[1]: !nvidia-smi
```

```

Thu Aug 29 01:25:43 2019
+-----+
| NVIDIA-SMI 418.67      Driver Version: 418.67      CUDA Version: 10.1    |
|   |           |
|-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. |
|   | ECC          |           |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage  | GPU-Util  Compute M.
|   |   |           |           |           |           |
|-----+-----+-----+
|   0  Tesla V100-SXM2... Off  | 00000000:00:1B.0 Off  | |
|   |   0  |
| N/A   45C     P0    54W / 300W |      0MiB / 16130MiB |      0%    |
|   | Default  |
+-----+-----+-----+
|   1  Tesla V100-SXM2... Off  | 00000000:00:1C.0 Off  | |
|   |   0  |
| N/A   44C     P0    54W / 300W |      322MiB / 16130MiB |      0%    |
|   | Default  |
+-----+-----+-----+
|   2  Tesla V100-SXM2... Off  | 00000000:00:1D.0 Off  | |
|   |   0  |
| N/A   44C     P0    57W / 300W |      0MiB / 16130MiB |      0%    |
|   | Default  |
+-----+-----+-----+
|   3  Tesla V100-SXM2... Off  | 00000000:00:1E.0 Off  | |
|   |   0  |
| N/A   42C     P0    40W / 300W |      11MiB / 16130MiB |      0%    |
|   | Default  |
+-----+-----+-----+
+-----+
| Processes:                               GPU |
|   |Memory  |
| GPU  PID  Type  Process name            Usage  |
|   |   |           |
|-----+-----+-----+-----+
|   1  66377    C  /home/ubuntu/miniconda3/bin/python  |
|   | 311MiB  |                                         (continues on next page)

```

+-----+

그 다음, GPU 버전의 MXNet이 설치되어 있는지 확인하세요. 만약 CPU 버전의 MXNet이 이미 설치되어 있는 경우에는 우선 MXNet을 제거해야 합니다. 즉, `pip uninstall mxnet` 명령으로 제거하고, 시스템에 설치된 CUDA 버전에 대응하는 MXNet을 설치합니다. CUDA 9.0이 설치되어 있다고 가정하면, CUDA 9.0을 지원하는 MXNet 버전 설치는 `pip install mxnet-cu90` 명령으로 합니다. 이 절의 프로그램들을 수행하기 위해서는 최소 두 개 이상의 GPU들이 필요합니다.

대부분의 데스크탑 컴퓨터에 GPU 두 개가 설치된 경우는 드물지만, 클라우드에서는 이런 시스템을 구하기 쉽습니다. 예를 들면, AWS 클라우드의 멀티 GPU를 제공하는 EC2 인스턴스를 사용할 수 있습니다. 거의 모든 다른 절들에서는 다중 GPU를 필요로 하지는 않습니다. 여기서는 데이터가 서로 다른 디바이스 간에 어떻게 이동하는지를 설명하기 위해서 여러 GPU가 필요합니다.

### 6.10.1 컴퓨팅 디바이스

MXNet은 값의 저장과 연산에 사용할 CPU나 GPU와 같은 디바이스를 지정할 수 있습니다. 기본 설정으로 MXNet은 메인 메모리에 데이터를 생성하고, CPU를 사용해서 연산을 수행합니다. MXNet에서는 CPU와 GPU는 각각 `cpu()` 와 `gpu()` 로 표현됩니다. `mx.cpu()` (또는 팔호안에 아무 정수를 사용)는 모든 물리적인 CPU들과 메모리를 의미한다는 것을 기억해두세요. 즉, MXNet은 연산을 수행할 때 모든 CPU 코어를 사용하려고 합니다. 반면에 `mx.gpu()` 는 하나의 그래픽 카드와 그 카드의 메모리를 지정합니다. 만약 여러 GPU를 가지고 있다면,  $i$  번째 GPU를 ( $i$ 는 0부터 시작) 지정하는 방법은 `mx.gpu( $i$ )` 라고 명시하면 됩니다. 참고로 `mx.gpu(0)` 과 `mx.gpu()` 는 같은 표현입니다.

```
[2]: import mxnet as mx
from mxnet import nd
from mxnet.gluon import nn

mx.cpu(), mx.gpu(), mx.gpu(1)

[2]: (cpu(0), gpu(0), gpu(1))
```

### 6.10.2 NDArray와 GPU

앞에서도 말했듯이 기본 설정은 NDArray 객체를 CPU에 생성합니다. 따라서, NDArray를 출력할 때, `@cpu(0)` 라는 식별자를 보게 됩니다.

```
[3]: x = nd.array([1, 2, 3])
x
```

```
[3]:
[1. 2. 3.]
<NDArray 3 @cpu(0)>
```

NDArray의 context 속성을 사용해서 NDArray 객체가 위치한 디바이스를 확인할 수 있습니다. 여러 객체에 대한 연산을 수행할 때는 항상 그 객체이 모두 같은 컨텍스트(context)에 있어야 한다는 것을 명심하세요. 즉, 두 변수를 더하는 경우, 두 변수가 같은 디바이스에 있어야 한다는 의미입니다. 그렇지 않을 경우에는 MXNet은 결과를 어느 곳에 저장할지 또는 연산을 어느 곳에서 수행해야 할지를 알 수 없습니다.

```
[4]: x.context
```

```
[4]: cpu(0)
```

## GPU의 저장소

GPU에 NDArray를 저장하는 방법은 여러가지가 있습니다. NDArray 객체를 생성할 때, ctx 파라미터를 이용해서 저장할 디바이스 지정이 가능합니다. 예를 들어, gpu(0)에 NDArray 변수 a를 생성합니다. a를 출력하면, 디바이스 정보가 @gpu(0)으로 나오는 것을 확인해보세요. GPU에서 만들어진 NDArray는 그 GPU의 메모리만 사용합니다. GPU 메모리 사용량은 nvidia-smi 명령으로 확인이 가능합니다. 일반적 우리는 GPU 메모리 크기를 넘어서 데이터를 생성하지 않도록 해야합니다.

```
[5]: x = nd.ones((2, 3), ctx=mx.gpu())
x
```

```
[5]:
[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @gpu(0)>
```

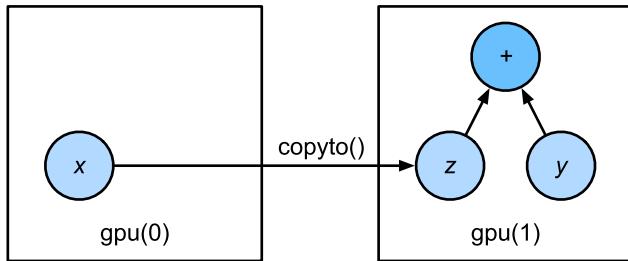
최소한 두 개의 GPU가 있다고 하면, 아래 코드는 난수 배열을 gpu(1)에 생성합니다.

```
[6]: y = nd.random.uniform(shape=(2, 3), ctx=mx.gpu(1))
y
```

```
[6]:
[[0.59119   0.313164   0.76352036]
 [0.9731786  0.35454726  0.11677533]]
<NDArray 2x3 @gpu(1)>
```

## 복사

$x + y$ 를 계산하고자 한다면, 이 연산을 어느 디바이스에서 수행할지를 결정해야합니다.  $x$ 를  $\text{gpu}(1)$ 로 옮기고, 연산을 거기서 수행할 수 있습니다. 단순히 “ $x + y$ “를 수행하지 마세요. 만약 그렇게 할 경우, 예외가 발생할 것입니다. 왜냐하면, 런타임 엔진은 무엇을 해야할지 모르고, 같은 디바이스에서 데이터를 찾을 수 없어서 연산이 실패하기 때문입니다.



`copyto` 메소드는 데이터를 다른 디바이스로 복사해서, 연산을 할 수 있도록 해줍니다.  $y$ 는 두번째 GPU에 있으니, 우리는 연산을 수행하기 전에  $x$ 를 그 디바이스로 옮겨야 합니다.

```
[7]: z = x.copyto(mx.gpu(1))
print(x)
print(z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @gpu(0)>

[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @gpu(1)>
```

자 이제 데이터가 모두 같은 GPU에 있으니, 두 값을 더할 수 있습니다. MXNet은 연산 결과를 다시 같은 디바이스에 저장합니다. 지금 예의 경우는  $@\text{gpu}(1)$ 입니다.

```
[8]: y + z
[8]: [[1.59119  1.313164  1.7635204]
 [1.9731786 1.3545473  1.1167753]]
<NDArray 2x3 @gpu(1)>
```

변수  $z$ 는 두번째 GPU,  $\text{gpu}(1)$ 에 있는데, 만약  $z.\text{copyto}(\text{gpu}(1))$ 을 수행하면 어떻게 될까요? 답

은 이미 같은 GPU에 값이 있더라도 새로운 메모리를 할당해서 값을 복사합니다. 프로그램이 수행되는 환경에 따라서 두 변수가 이미 같은 디바이스에 있는 경우도 있습니다. 우리는 변수가 다른 컨텍스트(context)에 있을 때만 복사를 수행하기 원합니다. 이 경우, `as_in_context()`를 이용하면 됩니다. 만약 변수가 지정된 컨텍스트(context)에 있는 경우라면, 아무 일이 일어나지 않습니다. 진짜로 데이터의 복제본을 만드는 경우가 아니라면, `as_in_context()`를 사용하세요.

```
[9]: z = x.as_in_context(mx.gpu(1))  
z
```

```
[9]:  
[[1. 1. 1.]  
 [1. 1. 1.]]  
<NDArray 2x3 @gpu(1)>
```

소스와 타겟 변수의 `context` 가 동일하다면, `as_in_context` 함수는 타겟 변수와 소스 변수가 소스 변수의 메모리를 공유한다는 사실을 기억해 두는게 중요합니다.

```
[10]: y.as_in_context(mx.gpu(1)) is y  
[10]: True
```

반면, `copyto` 함수는 타겟 변수를 위해서 항상 새로운 메모리를 만듭니다.

```
[11]: y.copyto(mx.gpu()) is y  
[11]: False
```

## 조심하세요

사람들은 빠른 속도를 기대하면서 머신러닝을 수행할 때 GPU들을 사용합니다. 컨텍스트(context)들 사이에 변수를 이동하는 것은 느립니다. 우리가 그렇게 하라고 하기 전에 이미 많은 경우 사람들은 느린 무언가를 수행합니다. 예를 들면, MXNet이 복사를 문제를 발생하지 않고 자동으로 수행했다면, 느리게 동작하는 코드를 작성했다는 것을 눈치채지 못할 것입니다.

디바이스 간(CPU, GPU, 다른 머신)에 데이터를 옮기는 것은 연산보다 훨씬 느립니다. 더군다나 병렬화(parallelization)를 더 어렵게 만듭니다. 연산을 계속 수행하기 전에 데이터가 보내지거나 받아지는 것이 끝날 때까지 대기해야하기 때문입니다. 그렇게 때문에 복사 연산은 아주 조심해서 수행해야 합니다. 경험적인 법칙으로 작은 연산을 많이 하는 것은 큰 연산보다 훨씬 나쁘고, 여러 연산을 동시에 수행하는 것은 하나의 연산을 여러 개를 수행하는 것보다 나쁩니다. 이런 경우들은 다른 무언가를 하기전에 한개의 디바이스가 다른 디바이스를 기다려야하는 예들입니다. 스마트폰으로 미리 주문한 후 도착하면 커피가 준비되어 있는 것이 아닌 줄을 서서 커피를 주문하는 것과 유사합니다.

마지막으로는 메인 메모리에 데이터가 있는 경우가 아닐 때, NDArray 데이터를 출력하거나 NDArray 를 NumPy 형태로 바꾸는 경우에 MXNet은 먼저 데이터를 메인 메모리에 복사합니다. 즉, 전송 오버헤드가 발생합니다. 더 나쁜 사실은 모든 것이 Python이 완료되기를 기다리는 글로벌 인터프린터 락에 종속된다는 것입니다.

### 6.10.3 Gluon과 GPU

NDArray와 비슷하게 Gluon의 모델도 초기화 과정중에 ctx 파라미터를 통해서 context를 지정할 수 있습니다. 아래 코드는 모델 파라미터를 GPU에서 초기화합니다.

```
[12]: net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=mx.gpu())
```

입력이 GPU에 있는 NDArray 객체라면, Gluon은 같은 GPU에서 연산을 수행합니다.

```
[13]: net(x)
[13]:
[[0.04995865]
 [0.04995865]]
<NDArray 2x1 @gpu(0)>
```

모델 파라미터들이 같은 GPU에 저장되어 있는지 확인해보겠습니다.

```
[14]: net[0].weight.data()
[14]:
[[0.0068339  0.01299825  0.0301265 ]]
<NDArray 1x3 @gpu(0)>
```

요약하면, 모든 데이터와 파라미터들이 같은 디바이스에 있어야 모델을 효과적으로 학습시킬 수 있습니다. 앞으로 그런 예제들을 여러개 보게 될 것입니다.

### 6.10.4 요약

- MXNet은 저장과 연산을 수행할 디바이스 (GPU, CPU)를 지정할 수 있습니다. 기본 설정으로 MXNet은 메인 메모리에 데이터를 생성하고, CPU를 사용해서 연산을 수행합니다.
- MXNet은 모든 입력 데이터가 동일한 디바이스 (CPU 또는 같은 GPU)에 있어야 연산을 수행할 수 있습니다.

- 데이터를 조심하게 옮기지 않을 경우 상단한 성능 손실이 발생합니다. 전형적인 실수는 다음과 같습니다. GPU를 이용해서 미니 배치의 손실(loss)을 계산하고, 매번 화면에 출력 (또는 NumPy 배열에 추가) 하는 경우. 이 경우, 글로벌 인터프린터 락이 필요하기 때문에 모든 GPU가 멈추어야 합니다. 권장하는 방법은 GPU에 로깅을 위한 메모리를 할당하고, 큰 로그를 옮기는 것입니다.

## 6.10.5 문제

- 큰 행렬의 곱같은 큰 연산을 수행하면서 CPU와 GPU의 속도 차이를 관찰해보세요. 작은 크기의 연산은 어떤가요?
- GPU에 파라미터를 읽고 쓰기를 어떻게 하나요?
- 100 × 100 행렬들의 행렬 곱 1000개를 수행하고, 행렬 놈(norm)  $\text{tr}MM^\top$  ff 매번 출력하는 것과 GPU에 로그를 저장한 후 마지막에 최종 결과만 옮길 때 각 수행 시간을 측정해보세요
- 두 개의 GPU에서 두 행렬 곱을 동시에 수행하는 것과, 하나의 GPU에서 순서대로 수행하면서 수행시간을 측정해보세요. 힌트 - 선형적인 성능 수치를 볼 것입니다.

## 6.10.6 참고자료

[1] CUDA download address. <https://developer.nvidia.com/cuda-downloads>

## 6.10.7 Scan the QR Code to Discuss



---

## Appendix

---

### 7.1 이 책에 기여하는 방법

이 오픈-소스 책의 기여자 목록을 [1]에서 볼 수 있습니다. 기여를 하고 싶다면, Git을 설치하고 이 책의 GitHub 코드 리포지토리에 pull request을 제출해주세요. 저자가 여러분의 pull request를 코드 리포지토리에 머지하면, 여러분은 기여자가 됩니다.

이 절에서는 이 책에 기여를 하는데 사용되는 기본적인 Git 절차를 설명하겠습니다. Git 사용법에 친숙하다면, 이 절을 넘어가도 됩니다.

아래 절차를 수행할 때, 기여자의 GitHub ID가 “astonzhang”이라고 가정하겠습니다.

1 단계: Git을 설치하세요. Git 오픈 소스 책[3]은 Git을 어떻게 설치하는지 상세하게 알려줍니다. 만약 아직 GitHub 계정이 없으면, 지금 가입하세요.[4]

2 단계: GitHub에 로그인 합니다. 웹 브라우저에 이 책의 코드 리포지토리 주소를 입력합니다.[2] 그림 11.20의 우측 상단에 빨간색으로 표시된 “Fork” 버튼을 클릭해서 이 책의 코드 리포지토리를 복제합니다.

Dive into Deep Learning, Berkeley STAT 157 (Spring 2019) textbook. With code, math, and discussions. <https://>

Branch: master [New pull request](#)

598 commits 4 branches 0 releases

mohamed-ali and astonzhang Improve sentence structure (#126)

build Update frontpage.html

chapter\_appendix Update buy-gpu.md

chapter\_computational-performance Update multiple-gpus-gluon.md

자 그럼 그림 11.21의 왼쪽 위에 보이는 “Your GitHub ID/d2l-ko”와 같이 여러분의 username에 이 책의 코드 리포지토리가 복사해졌을 것입니다.

Dive into Deep Learning, Berkeley STAT 157 (Spring 2019) textbook. With code, math, and discussions. <https://>

Manage topics

598 commits 4 branches 0 releases

Branch: master ▾ New pull request Create new file Upload files

This branch is even with d2l-ai:master.

mohamed-ali and astonzhang Improve sentence structure (d2l-ai#126)

build Update frontpage.html  
chapter\_appendix Update buy-gpu.md  
chapter\_computational-performance Update multiple-gpus-gluon.md

Clone with SSH ⓘ  
Use an SSH key and p...  
git@github.com:as...

Open in Desktop

3 단계: 그림 11.21의 오른쪽에 보이는 초록색 “Clone or download” 버튼을 클릭하고, 빨간색 박스로 표시한 버튼을 눌러서 여러분의 username 아래 있는 코드 리포지토리 주소를 복사합니다. “Acquiring and Running Codes in This Book”를 참고해서 명령행 모드로 들어가세요. 여기서는 로컬 디스크의 “~/repo” 경로에 코드 리포지토리를 복사한다고 가정하겠습니다. 그럼 이 경로로 이동하고, git clone을 적고, 그 뒤에 여러분의 username이 포함된 코드 리포지토리의 주소를 붙여놓습니다. 이제 명령을 수행하세요.

```
# Replace your_Github_ID with your GitHub username
git clone https://github.com/your_Github_ID/d2l-ko.git
```

이제 이 책의 코드 리포지토리의 모든 파일이 로컬 디스크의 ‘~/repo/d2l-ko’ 경로에 복사되었을 것입니다.

4 단계: 로컬 경로에 있는 코드 리포지토리를 수정하세요. ~/repo/d2l-ko/chapter\_deep-learning-basics/linear-regression.md 파일의 오타를 수정했다고 가정해보겠습니다. 명령행 모드에서, ~/repo/d2l-ko 경로로 이동한 후 아래 명령을 수행하세요.

```
git status
```

그러면 그림 11.22에서처럼 Git이 “chapter\_deep-learning-basics/linear-regression.md” 파일이 수정되었다고 알려줍니다.

```
aston@ubuntu:~/repo/d2l-en$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in work
               space)

        modified:   chapter_deep-learning-basics/linear-regression.md

no changes added to commit (use "git add" and/or "git commit")
```

변경을 제출할 파일이 정확하면, 아래 명령을 수행하세요.

```
git add chapter_deep-learning-basics/linear-regression.md
git commit -m 'fix typo in linear-regression.md'
git push
```

여기서 ‘fix typo in linear-regression.md’는 제출하는 변경에 대한 설명입니다. 여러분이 제출하는 변경 내용에 따라서 바꿔주세요.

5 단계: 다시 웹 브라우저에서 이 책의 코드 리포지토리 주소[2]를 입력하세요. 그럼 12.20의 좌측 하단의 빨간색 상자로 표시된 “New pull request” 버튼을 클릭하고, 이 후에 나오는 페이지에서 그림 11.12의 오른쪽에 빨간색 박스로 표시된 “compare across forks” 링크를 클릭하세요. 다음으로 그 아래 빨간 박스로 표시된 “head fork:d2l-ai/d2l-ko”를 클릭합니다. 그럼 11.23에서 보이는 것처럼, 팝업 텍스트 상자에 여러분의 GitHub ID를 입력하고, 드롭 다운 메뉴에서 ‘YourGitHub-ID/d2l-ko’를 선택합니다.

d2l-ai / d2l-en

Unwatch 57

Code Pull requests 0 Projects 0 Insights Settings

## Compare changes

Compare changes across branches, commits, tags, and more below. If you need to, you can also [compare across forks.](#)

base repository: d2l-ai/d2l-en base: master head repository: d2l-ai/d2l-en compare: master

Create pull request Choose different branches or fork

Choose a Head Repository

- astonzhang
- astonzhang/d2l-en

6 단계: 그림 11.24 처럼, 제목과 본문 텍스트 상자에 여러분이 제출하는 pull request에 대한 설명을 적어주세요. 그리고, 녹색 “Create pull request” 버튼을 눌러서 pull request를 제출합니다.

[Code](#)[Pull requests 0](#)[Projects 0](#)[Insights](#)[Settings](#)

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



base repository: d2l-ai/d2l-en

base: master



head repository: astonzhang/d2l-en

compare: master

✓ Able to merge. These branches can be automatically merged.



fix typo in linear-regression.md

[Write](#)[Preview](#)[AA](#) [B](#) [i](#) [“”](#) [↶](#) [☰](#) [☰](#) [☰](#) [@](#) [🔗](#) [↶](#)

Fix a typo in linear-regression.md: regression -> regression.

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Allow edits from maintainers. [Learn more](#)

[Create pull request](#)

요청이 제출되면, 그림 11.25과 같이 제출된 모든 pull request를 보여주는 페이지를 볼 수 있습니다.

[Code](#)[Pull requests 1](#)[Projects 0](#)[Insights](#)[Settings](#)

## fix typo in linear-regression.md #127

[Open](#)astonzhang wants to merge 1 commit into `d2l-ai:master` from `astonzhang:master`[Conversation 0](#)[Commits 1](#)[Checks 0](#)[Files changed 1](#)

astonzhang commented just now

Member

+ ...

Fix a typo in linear-regression.md: regression -&gt; regression.

fix typo in linear-regression.md

7e0318a

### 7.1.1 요약

- 여러분은 GitHub를 이용해서 이 책에 기여를 할 수 있습니다.

### 7.1.2 문제

- 이 책의 어떤 부분이 개선되어야 한다고 느낀다면, pull request를 제출해보세요.

### 7.1.3 참고 자료

- [1] 이 책(한글판)의 기여자 목록. <https://github.com/d2l-ai/d2l-ko/graphs/contributors>
- [2] 이 책의 코드 리포지토리 주소. <https://github.com/d2l-ai/d2l-ko>
- [3] Git 설치하기. <https://git-scm.com/book/zh/v2>
- [4] GitHub URL. <https://github.com/>

#### 7.1.4 Scan the QR Code to Discuss



## 7.2 d21 패키지 색인

함수 또는 클래스 이름	Relevant Chapter
bbox_to_rect	Object Detection and Bounding Boxes
Benchmark	Asynchronous Computation
corr2d	Two-dimensional Convolutional Layer
count_tokens	Sentiment Classification: Using Recurrent Neural Networks
data_iter	<i>Linear Regression Implementation from Scratch</i>
data_iter_consecutive	Language Model Data Set (Jay Chou album lyrics)
data_iter_random	Language Model Data Set (Jay Chou album lyrics)
download_imdb	Sentiment Classification: Using Recurrent Neural Networks
download_voc_pascal	Semantic Segmentation and Data Sets
evaluate_accuracy	Image Augmentation
get_data_ch7	Mini-batch Stochastic Gradient Descent
get_fashion_mnist_labels	<i>Image Classification Data Set(Fashion-MNIST)</i>
get_tokenized_imdb	Sentiment Classification: Using Recurrent Neural Networks
get_vocab_imdb	Sentiment Classification: Using Recurrent Neural Networks
grad_clipping	Recurrent Neural Network Implementation from Scratch
linreg	<i>Linear Regression Implementation from Scratch</i>
load_data_fashion_mnist	Deep Convolutional Neural Networks (AlexNet)
load_data_jay_lyrics	Language Model Data Set (Jay Chou album lyrics)
load_data_pikachu	Object Detection Data Set(Pikachu)
plt	<i>Linear Regression Implementation from Scratch</i>
predict_rnn	Recurrent Neural Network Implementation from Scratch
predict_rnn_gluon	Concise Implementation of Recurrent Neural Networks

일반 색인

Table 7.1 – 이전 페이지에서 계속

함수 또는 클래스 이름	Relevant Chapter
<code>predict_sentiment</code>	Sentiment Classification: Using Recurrent Neural Networks
<code>preprocess_imdb</code>	Sentiment Classification: Using Recurrent Neural Networks
<code>read_imdb</code>	Sentiment Classification: Using Recurrent Neural Networks
<code>read_voc_images</code>	Semantic Segmentation and Data Sets
<code>Residual</code>	Residual Networks (ResNet)
<code>resnet18</code>	Concise Implementation of Multi-GPU Computing
<code>RNNModel</code>	Concise Implementation of Recurrent Neural Networks
<code>semilogy</code>	<i>Model Selection</i>
<code>set_figsize</code>	<i>Linear Regression Implementation from Scratch</i>
<code>sgd</code>	<i>Linear Regression Implementation from Scratch</i>
<code>show_bboxes</code>	Anchor Boxes
<code>show_fashion_mnist</code>	<i>Image Classification Data Set(Fashion-MNIST)</i>
<code>show_images</code>	Image Augmentation
<code>show_trace_2d</code>	Gradient Descent and Stochastic Gradient Descent
<code>squared_loss</code>	<i>Linear Regression Implementation from Scratch</i>
<code>to_onehot</code>	Recurrent Neural Network Implementation from Scratch
<code>train</code>	Image Augmentation
<code>train_2d</code>	Gradient Descent and Stochastic Gradient Descent
<code>train_and_predict_rnn</code>	Recurrent Neural Network Implementation from Scratch
<code>train_and_predict_rnn_gluon</code>	Concise Implementation of Recurrent Neural Networks
<code>train_ch3</code>	<i>Softmax Regression Implementation from Scratch</i>
<code>train_ch5</code>	Convolutional Neural Networks (LeNet)
<code>train_ch7</code>	Mini-batch Stochastic Gradient Descent
<code>train_gluon_ch7</code>	Mini-batch Stochastic Gradient Descent
<code>try_all_gpus</code>	Image Augmentation
<code>try_gpu</code>	Convolutional Neural Networks (LeNet)
<code>use_svg_display</code>	<i>Linear Regression Implementation from Scratch</i>
<code>VOC_CLASSES</code>	Semantic Segmentation and Data Sets
<code>VOC_COLORMAP</code>	Semantic Segmentation and Data Sets
<code>voc_label_indices</code>	Semantic Segmentation and Data Sets
<code>voc_rand_crop</code>	Semantic Segmentation and Data Sets
<code>VOCSegDataset</code>	Semantic Segmentation and Data Sets