

Deep Learning from Scratch 2

밑바닥부터 시작하는 딥러닝 2



| 표지 설명 |



표지 그림은 팽이상어(Japanese bullhead shark)다. 팽이상어는 팽이상어목 팽이상어과에 딸린 물고기다. 태평양 북서부 일본, 한반도, 중국 해안에 서식한다. 수심 6~37미터 가량의 바위투성이 밀바닥 또는 켈프림에 사는 저생성 어류다. 몸길이는 1.2미터까지 자란다. 머리는 짧고 무디며 등지느러미는 두 개다. 진한 갈색과 연한 갈색이 수직 줄무늬를 이룬다. 온순하며, 연체동물이나 작은 경골어류를 잡아먹는다. 어업적 측면에서는 거의 관심 밖인 종이다. 학명은 헤테로돈투스 자포니쿠스(Heterodontus japonicus). (출처: 위키백과)

밀바닥부터 시작하는 딥러닝 2

파이썬으로 직접 구현하며 배우는 순환 신경망과 자연어 처리

초판 1쇄 발행 2019년 5월 1일

지은이 사이토 고키 / 옮긴이 개얏맵사(이복연) / 펴낸이 김태현

펴낸곳 한빛미디어(주) / 주소 서울시 서대문구 연희로2길 62 한빛미디어(주) IT출판사업부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 6월 24일 제 25100-2017-000058호 / ISBN 979-11-6224-174-5 93000

총괄 전태호 / 책임편집 이상복 / 기획·편집 이미연

디자인 표지 고충열, 박정화 내지 김연정 조판 이경숙

영업 김형진, 김진불, 조유미 / 마케팅 송경석, 김나예, 이행은 / 제작 박성우, 김정우

이 책에 대한 의견이나 오타 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오. 잘못된 책은 구입하신 서점에서 교환해 드립니다. 책값은 뒷표지에 표시되어 있습니다.

한빛미디어 홈페이지 www.hanbit.co.kr / 이메일 ask@hanbit.co.kr

Copyright © Hanbit Media, Inc. 2019

Authorized translation of the Japanese edition of 'Deep Learning from Scratch2' © 2018 O'Reilly Japan, Inc. This translation is published and sold by permission of O'Reilly Japan, Inc., the owner of all rights to publish and sell the same.

이 책의 저작권은 오라일리재판과 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

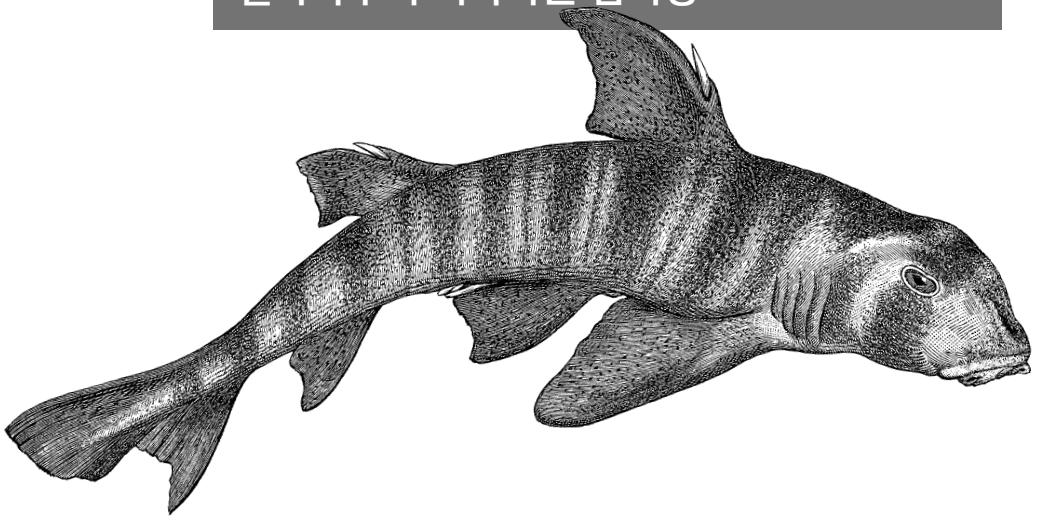
책으로 펴내고 싶은 아이디어나 원고를 메일(writer@hanbit.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

Deep Learning

from Scratch 2

밑바닥부터 시작하는 딥러닝 2



O'REILLY®

HB 한빛미디어
Hanbit Media, Inc.

자은이 **사이트 고키**(斎藤 康毅)

1984년 나가사키 현 쓰시마 태생. 도쿄공업대학교 공학부를 졸업하고 도쿄대학대학원 학제정보학부 석사 과정을 수료했다. 현재는 기업에서 인공지능 관련 연구·개발에 매진하고 있다. 오라일리 재팬에서 『밑바닥부터 시작하는 딥러닝』을 집필했으며, 『실천 파이썬 3』, 『컴퓨터 시스템의 이론과 구현』, 『실천 기계학습 시스템』 등을 번역했다.

옮긴이 **개앞맵시(이복연)** wegra.lee@gmail.com



고려대학교 컴퓨터학과를 졸업하고 삼성소프트웨어멤버십을 거쳐, 삼성전자 소프트웨어센터와 미디어솔루션센터에서 자바 가상 머신, 바다 플랫폼, 챗온 메신저 서비스 등을 개발했다. 주 업무 외에 분산 빌드, 지속적 통합, 앱 수명주기 관리 도구, 애자일 도입 등 동료 개발자들에게 실질적인 도움을 주는 일에 적극적이었다. 그 후 창업전선에 뛰어들어 소셜 서비스, 금융 거래 프레임워크 등을 개발하다가, 무슨 바람이 불어 서인지 책을 만들겠다고 기획·편집자(자칭 Wisdom Compiler)로 변신했다.

한빛미디어에서 『밑바닥부터 시작하는 딥러닝』(2017)과 『Effective Unit Testing』(2013)을 번역했고, 인사이트에서 『이펙티브 자바, 3판』(2018)과 『JUnit 인 액션』(2011)을 번역했다.

‘개앞맵시’는 ‘개발자의 앞길에 맵핵 시전’을 줄인 말로, 어려서부터 꿈꿔온 후학 양성의 의지를 담은 필명이다. 번역 외에도 게임, 서버, 웹 등 주요 직군별 개발자에게 필요한 기술과 역량을 다루는 책들로 로드맵을 만들어 공개하거나 프로그래머를 위한 책쓰기를 안내하는 등 재미난 아이디어가 생길 때마다 짬짬이 무언가를 만들어 공유하는 걸 좋아한다.

• **페이스북** <https://facebook.com/dev.loadmap>

• **IT 전문서 로드맵 모음** <https://www.mindmeister.com/ko/users/channel/86528>

• **프로그래머의 책쓰기** <https://github.com/WegraLee/Writing-IT-Books>

추천사

『밑바닥부터 시작하는 딥러닝』에 이어 널리 사용되는 딥러닝 모델을 직접 구현하면서 기본 원리를 체득하는 체험형 입문서입니다. 전편이 딥러닝의 기본 이론을 밑바닥부터 구현할 수 있도록 안내했다면 이번에는 자신만의 딥러닝 프레임워크를 구축할 수 있는 토대를 만들어줍니다. 파이썬과 넘파이의 활용까지 책임지는 훌륭한 지침서입니다. 딥러닝 프레임워크를 연구하는 모든 분께 이 책을 추천합니다.

윤영선, 한남대학교 정보통신공학과 교수

머신러닝 라이브러리를 이용하여 딥러닝 분야에 뛰어들게 되면 네트워크의 내부 구조와 디테일한 부분들을 자세하게 알 수 없어 한없이 답답합니다. 이 책은 큰 부분부터 디테일한 부분까지 밑바닥부터 직접 구현해 네트워크에 대한 이해를 돕고, 답답함을 한 방에 날려줍니다. 그동안 국내에 출판된 책들이 RNN 계열이나 자연어 처리 계열을 잘 다루지 않는 것에 비해, 이 책은 기초적인 자연어 처리부터 응용, 심화 과정까지 설명합니다. 자연어 처리 분야에 뛰어들 분들에게 추천합니다.

박동주, 광주과학기술원 석사과정

앞으로 자연어 처리 기본서는 이 책의 전과 후로 나뉘 겠니다. 전편에서와 마찬가지로 직관적이고 논리적으로 설명해주는 부분에서 감탄했습니다. 자연어 처리를 처음 접하는 입문자부터 단순히 프레임워크를 활용해본 경험자까지, 자연어 처리 과정이 어떻게 이루어지는지 알고 싶은 모든 분께 추천합니다. 특히 통계 기반 기법과 추론 기반 기법의 장단점과 차이점을 설명하는 부분이 명쾌해서 내용을 이해하는 데 큰 도움이 되었습니다.

성민석, 홍익대학교 컴퓨터공학과 4학년

딥러닝을 이용한 자연어 처리의 기본을 밑바닥부터 공부할 수 있습니다. 전편과 마찬가지로 RNN, LSTM, 어텐션 등을 구현하면서 각각의 작동 원리를 쉽게 공부할 수 있습니다. 저는 작년에 일본어 판을 먼저 읽어보았는데, 번역 또한 전편과 마찬가지로 아주 좋습니다.

김선웅, (주)스캐터랩 머신러닝 엔지니어

이 책은 RNN 기반 신경망에 대한 개념을 쉽게 설명해주고, 복잡한 응용 모델들을 그림과 예제를 통해 알려줍니다. 시계열을 공부하는 분, RNN 공부를 시작하는 분께 큰 도움이 될 것입니다.

김동성, 삼성 리서치(Samsung Research) 연구원

쉽게 풀어쓴 이론과 친절하게 설명한 코드를 따라가다 보면 자연어 처리 분야의 발전 과정을 자연스럽게 이해할 수 있습니다. 딥러닝 기초가 필요한 분은 전편부터 공부하시는 걸 추천합니다.

양민혁, 현대모비스 빅데이터팀

밑바닥부터 시작하는 딥러닝 2편이 나오다니! 너무나 신납니다. 저처럼 어린 학생들도 자연어 처리에 맞추어 무리 없이 쉽게 이해하며 따라 할 수 있는 책입니다. 저와 같은 어린 학생들이 과연 할 수 있을까 망설이고 있다면, 주저 없이 이 책을 집어 드세요.

김경수, 봉일천고등학교 2학년

수많은 분을 딥러닝의 세계로 안내하며 사랑받고 있는 『밑바닥부터 시작하는 딥러닝』이 ‘자연어 처리’라는 새로운 주제를 들고 다시 찾아왔습니다. 단순 비교는 할 수 없지만, 이번 편은 더 짜임새 있게 조직되었으며 세련미가 더해졌습니다. 그리고 여전히 쉽고 친절합니다.

그리고 원고를 검토해주신 여러 리뷰어 분들 덕에 헛갈리거나 아쉬운 부분을 상당히 보강할 수 있었습니다. 어려운 시간 내주신 윤영선 교수님, 박동주 님, 성민석 님, 김선웅 님, 김동성 님, 양민혁 님, 김정수 님 모두께 독자 분들을 대신해 감사드립니다. 특히 감수해주시듯 꼼꼼하고 깊이 있는 조언을 주신 윤영선 교수님의 기여로, 이번에도 원서보다 나은 번역서가 되었다고 조심스럽게 주장해 봅니다.

물론 다루는 내용이 전편과 다르고 전편보다 더 깊이 들어가기 때문에 살짝 더 어렵습니다. 하지만 전편을 무리 없이 독파하신 분이나 전편이 어려웠더라도 그동안 경험을 더 쌓으신 분은 이번 책도 어렵지 않게 따라오실 겁니다. 특히 원리를 직관적으로 보여주는 그림들이 많이 준비돼 있어서, 꼭 코드 수준에서 다 분석하며 따라오지 않아도 개념을 이해하는 데는 문제없을 겁니다. 물론 ‘밑바닥부터’ 만들어본다는 책 취지에 동감한다면 조금 느리더라도 직접 코딩하고 돌려보기를 바랍니다.

이 책은 ‘자연어 처리’를 다루지만, 본질적으로는 순환 신경망(RNN)을 활용한 ‘시계열 데이터 처리’를 배우게 됩니다. 실제로 “164+57=” 같은 덧셈을 학습시키는 예제도 등장한답니다. 호기심이 살짝 커졌겠죠? 긴 말 필요 없이, 지금 바로 뛰어들어 경험해봅시다!

내가 만들어낼 수 없다면, 난 그것을 이해하지 못한 것이다.

- 리처드 파인만

딥러닝이 세계를 크게 변화시키고 있습니다. 스마트폰의 음성 인식도, 웹의 실시간 번역도, 환율 예측도 이제 딥러닝을 빼고는 말할 수 없습니다. 신약 개발, 환자 진단, 자율주행 자동차도 딥러닝 덕분에 현실이 되고 있습니다. 이밖에도 첨단 기술 뒤에는 거의 딥러닝이 숨어 있습니다. 그리고 앞으로도 딥러닝으로 세계는 더욱 전진할 것입니다.

이 책은 『밑바닥부터 시작하는 딥러닝』의 속편입니다. 전편에 이어 계속 딥러닝 기술을 다룹니다. 특히 이번에는 자연어 처리와 시계열 데이터 처리에 초점을 맞춰 딥러닝을 사용해 다양한 문제에 도전합니다. 그리고 전편과 똑같이 ‘밑바닥부터 만든다’는 기치 아래, 딥러닝을 활용한 고급 기술들을 차분히 만끽해갈 것입니다.

이 책의 특징

저는 딥러닝과 같은 고도의 기술을 더 깊이 이해하려면 ‘밑바닥부터 만들어보는 경험’이 중요하다고 생각합니다. 여기서 ‘밑바닥부터’라 함은 자신이 이해할 수 있는 지점부터 시작하며, 되도록 기존의 외부 라이브러리나 도구를 사용하지 않고 목표하는 기술을 완성한다는 뜻입니다. 이러한 경험을 통해서 피상적이 아닌, 제대로 딥러닝에 정통한 것! 그것이 이 책의 지향점입니다.

기술을 깊게 이해하려면, 결국에는 그 기술을 만들 수 있을 만큼의 지식과 기량이 필요합니다. 이 책에서는 딥러닝을 밑바닥부터 만듭니다. 이를 위해 다양한 코드를 작성해가며 여러 가지 실험을 수행합니다. 시간이 걸리는 작업이고, 때때로 골치를 썩이는 일도 있겠지요. 그러나 오히려 이러한 과정에야말로 기술을 깊게 이해하는 데 중요한 핵심이 많이 담겨 있습니다. 이렇게 얻은 지식은 기존 라이브러리를 사용하거나 최첨단 논문을 읽을 때, 혹은 오리지널 시스템을 만드는 데도 반드시 도움이 될 것입니다. 그리고 무엇보다 딥러닝의 구조와 원리를 하나씩 풀면서 이해한다는 것 자체가 순수하게 즐거운 경험입니다.

자연어 처리의 세계로

이 책의 주요 주제는 딥러닝을 활용한 자연어 처리입니다. 자연어 처리란, 간단히 말해 우리가 평소 말하는 언어를 컴퓨터가 이해하도록 만드는 기술입니다. 매우 어려운 문제인 동시에 중요한 주제이기도 합니다. 실제로 이 자연어 처리 기술 덕에 우리의 생활은 크게 바뀌었습니다. 웹 검색, 기계 번역, 음성 비서 등 세상에 큰 영향을 준 기술의 근간에는 자연어 처리가 활용되고 있습니다.

이처럼 우리 생활에 빠뜨릴 수 없는 자연어 처리 기술 분야에도 딥러닝은 지극히 중요한 위치를 차지하고 있습니다. 사실 딥러닝 덕에 지금까지의 자연어 처리 성능이 크게 향상됐습니다. 예를 들어 구글의 기계 번역이 딥러닝 기법을 적용해 최근 큰 진전을 이뤘다는 소식을 들은 기억이 생생할 것입니다.

이 책에서는 자연어 처리와 시계열 데이터 처리에 초점을 맞춰 딥러닝에서 중요한 기술들을 배웁니다. 구체적으로는 word2vec과 RNN, LSTM과 GRU, seq2seq와 어텐션 같은 기술입니다. 이 책은 이 기술들을 가능한 한 쉬운 말로 설명하고 실제로 만들어보면서 확실한 내 것이 되도록 안내합니다. 독자분들은 직접 실험해보며 그 가능성을 몸소 느껴주셨으면 합니다.

이 책은 딥러닝을 중심으로 자연어 처리를 탐험하는 모험 책입니다. 총 8장으로 구성했으며 전체가 하나의 이야기처럼 처음부터 순서대로 읽도록 꾸몄습니다. 문제가 가로막고, 그것을 해결하는 새로운 기법을 떠올리고, 한층 더 개선합니다. 이런 흐름으로 다양한 자연어 처리 문제를 딥러닝이라는 무기를 휘둘러 하나씩 해결해 나가겠습니다. 그리고 여러분은 이 모험을 함께하며 딥러닝의 중요 기법들을 높은 수준으로 습득해, 그 재미를 온전히 느껴주셨으면 합니다.

누구를 위한 책인가?

이 책은 『밑바닥부터 시작하는 딥러닝』의 속편이라 전편에서 배운 지식을 토대로 합니다. 하지만 전편의 내용을 요약한 신경망 복습을 첫 장에 배치하여, 신경망과 파이썬 지식을 어느 정도 갖춘 분이라면 전편을 읽지 않았더라도 무리 없이 따라올 수 있도록 배려했습니다.

딥러닝을 깊이 이해하기 위해 이 책도 전편과 똑같이 ‘만든다’와 ‘동작시켜본다’에 중점을 두어 이야기를 풀어갑니다. ‘모르는 것은 쓰지 않는다’, ‘이해한 것만 사용한다’라는 자세로 딥러닝의 세계를, 그리고 자연어 처리의 세계를 탐색해갈 것입니다. ‘누구를 위한 책인가’를 더 명확하게 하기 위해 이 책에서 수행하는 것과 이 책의 특징을 적어봤습니다.

- 외부 라이브러리에 의지하지 않고, 밑바닥부터 딥러닝 프로그램을 구현합니다.
- 『밑바닥부터 시작하는 딥러닝』의 속편으로서 자연어 처리와 시계열 데이터 처리에 사용하는 딥러닝 기술에 초점을 맞춥니다.
- 실제로 동작하는 파이썬 소스 코드와 독자가 직접 실행해볼 수 있는 학습 환경을 제공합니다.
- 가능한 한 쉬운 말로, 명확한 그림을 많이 동원하여 설명합니다.
- 수식도 사용하지만 그 이상으로 소스 코드에 기초한 설명을 중시합니다.
- ‘왜 그 기법이 뛰어난가?’, ‘왜 그 방식이 먹히는가?’, ‘왜 그것이 문제인가?’ 등 ‘왜’를 소중히 합니다.

다음은 이 책에서 배우는 기술들입니다.

- 파이썬을 이용한 텍스트 처리
- 딥러닝 등장 이전의 단어 표현 방법
- 단어 벡터를 얻기 위한 word2vec(CBOW 모델과 skip-gram 모델)
- 대규모 데이터의 학습을 고속화하는 네거티브 샘플링
- 시계열 데이터를 처리하는 RNN, LSTM, GRU
- 시계열 데이터의 오차역전파법인 BPTT
- 문장을 생성하는 신경망
- 시계열 데이터를 다른 시계열 데이터로 변환하는 seq2seq
- 중요 정보에 주목하는 어텐션

이 책에서는 이러한 기술을 친절하고 알기 쉽게 설명하고, 구현 수준까지 습득할 수 있게끔 이야기를 풀어갑니다. 나아가 이러한 기술을 단순히 사실들만을 나열하지 않고, 마치 하나의 잘 짜여진 이야기처럼 연결해 설명했습니다.

누구를 위한 책이 아닌가?

‘누구를 위한 책이 아닌가’도 중요하다고 생각합니다. 다음은 이 책에서 다루지 않는 주제입니다.

- 딥러닝 분야의 최신 연구에 대해서는 자세한 다루지 않습니다.
- 카페Caffe, 텐서플로TensorFlow, 체이너Chainer 등 딥러닝 프레임워크 사용법은 설명하지 않습니다.
- 딥러닝 이론을 아주 상세한 수준까지는 담지 않았습니다.
- 이 책은 주로 자연어 처리를 다룹니다. 영상, 음성, 강화 학습 등의 주제는 다루지 않습니다.

이처럼 이 책은 최신 연구나 아주 자세한 이론은 다루지 않습니다. 그러나 이 책을 읽고 나면, 그 다음 단계로 최신 논문과 자연어 처리 관련 최첨단 기법을 학습하기가 한결 수월해질 것입니다.

구현 환경

이 책의 예제 소스는 파이썬 3로 작성했습니다. 소스 코드를 이용하면 여러분의 컴퓨터에서 직접 구동해볼 수 있습니다. 코드를 읽으면서 생각해보고, 자신의 아이디어대로 코드를 수정해 시험해보세요. 그러면 지식을 더 확실하게 자기 것으로 만들 수 있습니다. 덧붙여 이 책에서 사용하는 소스 코드는 다음 깃허브 저장소에서 받을 수 있습니다.

- <https://github.com/WegraLee/deep-learning-from-scratch-2>

이 책의 목표는 딥러닝을 밑바닥부터 구현하는 것입니다. 그래서 외부 라이브러리는 가능한 한 사용하지 않는 것이 기본 방침입니다만, 다음 두 라이브러리는 예외로 하겠습니다. 하나는 넘파이numpy, 다른 하나는 맷플롯립matplotlib입니다. 이 두 라이브러리를 사용하는 이유는 딥러닝을 효율적으로 구현하기 위해서입니다.

넘파이는 수치 계산용 라이브러리입니다. 넘파이에는 고도의 수학 알고리즘과 배열(행렬)을 조작하기 위한 편의 메서드가 많이 준비되어 있습니다. 이 메서드들을 이용하면 이 책의 딥러닝 구현을 훨씬 효율적으로 진행할 수 있습니다.

맷플롯립은 그래프를 그려주는 라이브러리입니다. 맷플롯립을 이용하면 실험 결과를 시각화하거나 딥러닝 학습 과정을 시각적으로 확인할 수 있습니다. 이 책에서는 이 두 라이브러리를 사용해 딥러닝 알고리즘을 구현해갈 것입니다.

덧붙여 이 책의 소스 코드 대부분은 일반적인 PC로도 빠르게 실행할 수 있도록 배려했습니다. 그러나 일부 코드는, 특히 큰 신경망의 학습은 시간이 오래 걸릴 수밖에 없습니다. 그래서 이처럼 시간이 걸리는 처리를 빠르게 수행하기 위해 GPU를 활용할 수 있는 코드(구조)도 함께 실었습니다. GPU 활용에는 쿠파이^{CuPy} 라이브러리를 사용합니다(쿠파이에 관해서는 ‘1장. 신경망의 복습’에서 설명합니다). 엔비디아^{NVIDIA} GPU를 장착한 컴퓨터를 사용하는 독자는 쿠파이를 설치하면 이 책의 해당 예제 코드를 GPU에서 고속으로 처리할 수 있습니다.

NOTE 이 책에서는 아래와 같은 프로그래밍 언어와 라이브러리를 사용합니다.

- 파이썬 3
- 넘파이
- 맷플롯립
- 쿠파이 (선택사항)

다시 만드는 여행으로

기술이 발달하고 무엇이든 쉽게 복제하는 시대가 되었습니다. 사진도 동영상도 소스 코드도 라이브러리도 모두 쉽게 복제할 수 있는 편리한 세상입니다. 하지만 아무리 기술이 발달하고 생활이 편리해져도 경험은 복제할 수 없습니다. 손가락을 움직여 만들고 시간을 들여 생각해보는 경험은 결코 복제할 수 없습니다. 이런 복제할 수 없는 것이야말로 어느 시대에서나 변함없는 가치를 지닐 것입니다.

이것으로 서론은 끝입니다. 그럼 딥러닝을 만드는 여행을 다시 시작해볼까요?!

감사의 말

이 책이 존재할 수 있는 것은 지금까지 딥러닝과 인공지능, 나아가 자연과학을 연구해온 위대한 선인들이 있었기 때문입니다. 우선 그 분들에게 감사의 말씀을 드립니다. 그리고 제 주변 사람들의 지지와 협력에도 진심으로 감사드립니다. 그들은 직설적으로, 때론 간접적으로 저를 격려해주었습니다. 고맙습니다.

이 책을 집필하면서는 ‘공개 리뷰’라는 새로운 방식을 시도했습니다. 책의 원고를 웹에 공개하고, 누구나 열람하고 댓글을 남길 수 있도록 했습니다. 그 결과, 불과 1개월 정도의 리뷰 기간에 1,500건이 넘는 유용한 피드백을 받았습니다. 리뷰에 참여하신 분들께도 진심으로 감사를 드립니다. 이 책이 더욱 다듬어질 수 있었던 것은 틀림없이 그러한 리뷰어 분들의 도움 덕분입니다. 정말 감사합니다. 당연하지만, 여전히 미비한 점이나 잘못된 부분은 모두 제 책임이며, 리뷰어 분들께는 전혀 책임이 없습니다.

마지막으로, 이 책이 존재하는 것은 전 세계 사람들 덕분입니다. 저는 이름 모를 많은 사람으로부터 매일 영향을 받으며 살고 있습니다. 누군가 한 사람만 없었어도 이런 책은 (적어도 지금 상태 그대로는) 존재하지 않았을 것입니다. 제 주변 자연에도 똑같이 말할 수 있습니다. 강과 나무에, 땅과 하늘에 제 일상이 있습니다. 이름 없는 사람께, 이름 없는 자연에 진심으로 감사드립니다.

2018년 6월 1일
사이토 고키

CONTENTS

지은이 · 옮긴이 소개	4
추천사	5
옮긴이의 말	7
들어가며	8
감사의 말	13

CHAPTER 1 신경망 복습

1.1 수학과 파이썬 복습	23
1.1.1 벡터와 행렬	24
1.1.2 행렬의 원소별 연산	26
1.1.3 브로드캐스트	26
1.1.4 벡터의 내적과 행렬의 곱	27
1.1.5 행렬 형상 확인	29
1.2 신경망의 추론	30
1.2.1 신경망 추론 전체 그림	30
1.2.2 계층으로 클래스화 및 순전파 구현	35
1.3 신경망의 학습	39
1.3.1 손실 함수	40
1.3.2 미분과 기울기	42
1.3.3 연쇄 법칙	44
1.3.4 계산 그래프	45
1.3.5 기울기 도출과 역전파 구현	56
1.3.6 가중치 갱신	60
1.4 신경망으로 문제를 푼다	62
1.4.1 스파이럴 데이터셋	62
1.4.2 신경망 구현	64
1.4.3 학습용 코드	66

1.4.4 Trainer 클래스	69
1.5 계산 가속화	71
1.5.1 비트 정밀도	71
1.5.2 GPU(쿠파이)	73
1.6 정리	74

CHAPTER 2 자연어와 단어의 분산 표현

2.1 자연어 처리란	78
2.1.1 단어의 의미	79
2.2 시소러스	79
2.2.1 WordNet	81
2.2.2 시소러스의 문제점	81
2.3 통계 기반 기법	82
2.3.1 파인션으로 말뭉치 전처리하기	83
2.3.2 단어의 분산 표현	86
2.3.3 분포 가설	87
2.3.4 동시발생 행렬	88
2.3.5 벡터 간 유사도	92
2.3.6 유사 단어의 랭킹 표시	94
2.4 통계 기반 기법 개선하기	97
2.4.1 상호정보량	97
2.4.2 차원 감소	101
2.4.3 SVD에 의한 차원 감소	104
2.4.4 PTB 데이터셋	106
2.4.5 PTB 데이터셋 평가	109
2.5 정리	111

CONTENTS

CHAPTER 3 word2vec

3.1 추론 기반 기법과 신경망	114
3.1.1 통계 기반 기법의 문제점	114
3.1.2 추론 기반 기법 개요	115
3.1.3 신경망에서의 단어 처리	116
3.2 단순한 word2vec	121
3.2.1 CBOW 모델의 추론 처리	121
3.2.2 CBOW 모델의 학습	126
3.2.3 word2vec의 가중치와 분산 표현	129
3.3 학습 데이터 준비	131
3.3.1 맥락과 타깃	131
3.3.2 원핫 표현으로 변환	134
3.4 CBOW 모델 구현	135
3.4.1 학습 코드 구현	139
3.5 word2vec 보충	141
3.5.1 CBOW 모델과 확률	142
3.5.2 skip-gram 모델	143
3.5.3 통계 기반 vs. 추론 기반	146
3.6 정리	147

CHAPTER 4 word2vec 속도 개선

4.1 word2vec 개선 ①	150
4.1.1 Embedding 계층	152
4.1.2 Embedding 계층 구현	153

4.2 word2vec 개선 ②	157
4.2.1 은닉층 이후 계산의 문제점	157
4.2.2 다중 분류에서 이진 분류로	159
4.2.3 시그모이드 함수와 교차 엔트로피 오차	161
4.2.4 다중 분류에서 이진 분류로 (구현)	164
4.2.5 네거티브 샘플링	168
4.2.6 네거티브 샘플링의 샘플링 기법	171
4.2.7 네거티브 샘플링 구현	174
4.3 개선판 word2vec 학습	176
4.3.1 CBOW 모델 구현	177
4.3.2 CBOW 모델 학습 코드	179
4.3.3 CBOW 모델 평가	181
4.4 word2vec 남은 주제	185
4.4.1 word2vec을 사용한 애플리케이션의 예	185
4.4.2 단어 벡터 평가 방법	187
4.5 정리	189

CHAPTER 5 순환 신경망(RNN)

5.1 확률과 언어 모델	192
5.1.1 word2vec을 확률 관점에서 바라보다	192
5.1.2 언어 모델	194
5.1.3 CBOW 모델을 언어 모델로?	196
5.2 RNN이란	199
5.2.1 순환하는 신경망	199
5.2.2 순환 구조 펼치기	201
5.2.3 BPTT	203

CONTENTS

5.2.4 Truncated BPTT	204
5.2.5 Truncated BPTT의 미니배치 학습	208
5.3 RNN 구현	210
5.3.1 RNN 계층 구현	211
5.3.2 Time RNN 계층 구현	215
5.4 시계열 데이터 처리 계층 구현	220
5.4.1 RNNLM의 전체 그림	220
5.4.2 Time 계층 구현	223
5.5 RNNLM 학습과 평가	225
5.5.1 RNNLM 구현	225
5.5.2 언어 모델의 평가	229
5.5.3 RNNLM의 학습 코드	231
5.5.4 RNNLM의 Trainer 클래스	234
5.6 정리	235

CHAPTER 6 게이트가 추가된 RNN

6.1 RNN의 문제점	238
6.1.1 RNN 복습	238
6.1.2 기울기 소실 또는 기울기 폭발	239
6.1.3 기울기 소실과 기울기 폭발의 원인	241
6.1.4 기울기 폭발 대책	246
6.2 기울기 소실과 LSTM	247
6.2.1 LSTM의 인터페이스	247
6.2.2 LSTM 계층 조립하기	249
6.2.3 output 게이트	251
6.2.4 forget 게이트	253

6.2.5 새로운 기억 셀	254
6.2.6 input 게이트	255
6.2.7 LSTM의 기울기 흐름	256
6.3 LSTM 구현	257
6.3.1 Time LSTM 구현	262
6.4 LSTM을 사용한 언어 모델	265
6.5 RNNLM 추가 개선	272
6.5.1 LSTM 계층 다중화	272
6.5.2 드롭아웃에 의한 과적합 억제	273
6.5.3 가중치 공유	278
6.5.4 개선된 RNNLM 구현	279
6.5.5 첨단 연구로	284
6.6 정리	286

CHAPTER 7 RNN을 사용한 문장 생성

7.1 언어 모델을 사용한 문장 생성	290
7.1.1 RNN을 사용한 문장 생성의 순서	290
7.1.2 문장 생성 구현	294
7.1.3 더 좋은 문장으로	298
7.2 seq2seq	299
7.2.1 seq2seq의 원리	300
7.2.2 시계열 데이터 변환용 장난감 문제	303
7.2.3 가변 길이 시계열 데이터	304
7.2.4 덧셈 데이터셋	306
7.3 seq2seq 구현	308
7.3.1 Encoder 클래스	308

CONTENTS

7.3.2 Decoder 클래스	311
7.3.3 Seq2seq 클래스	316
7.3.4 seq2seq 평가	317
7.4 seq2seq 개선	321
7.4.1 입력 데이터 반전(Reverse)	321
7.4.2 엿보기(Peeky)	323
7.5 seq2seq를 이용하는 애플리케이션	328
7.5.1 챗봇	329
7.5.2 알고리즘 학습	330
7.5.3 이미지 캡셔닝	331
7.6 정리	333

CHAPTER 8 어텐션

8.1 어텐션의 구조	335
8.1.1 seq2seq의 문제점	336
8.1.2 Encoder 개선	337
8.1.3 Decoder 개선 ①	339
8.1.4 Decoder 개선 ②	347
8.1.5 Decoder 개선 ③	352
8.2 어텐션을 갖춘 seq2seq 구현	358
8.2.1 Encoder 구현	358
8.2.2 Decoder 구현	359
8.2.3 seq2seq 구현	361
8.3 어텐션 평가	361
8.3.1 날짜 형식 변환 문제	362
8.3.2 어텐션을 갖춘 seq2seq의 학습	363

8.3.3 어텐션 시각화	367
8.4 어텐션에 관한 남은 이야기	369
8.4.1 양방향 RNN	369
8.4.2 Attention 계층 사용 방법	372
8.4.3 seq2seq 심층화와 skip 연결	374
8.5 어텐션 응용	376
8.5.1 구글 신경망 기계 번역(GNMT)	376
8.5.2 트랜스포머	379
8.5.3 뉴럴 튜링 머신(NTM)	382
8.6 정리	387

APPENDIX A 시그모이드 함수와 tanh 함수의 미분

A.1 시그모이드 함수	389
A.2 tanh 함수	392
A.3 정리	394

APPENDIX B WordNet 맛보기

B.1 NLTK 설치	395
B.2 WordNet에서 동의어 얻기	396
B.3 WordNet과 단어 네트워크	398
B.4 WordNet을 사용한 의미 유사도	399

CONTENTS

APPENDIX **C** GRU

C.1 GRU의 인터페이스	401
C.2 GRU의 계산 그래프	402
마지막으로	405
참고 문헌	406
찾아보기	413

신경망 복습

한 가지 이상의 방법을 알아내기 전에는
제대로 이해한 것이 아니다.

– 마빈 민스키(컴퓨터과학자이자 인지과학자)

이 책은 『밑바닥부터 시작하는 딥러닝』의 속편으로, 전편에 이어 딥러닝의 가능성을 한층 더 깊게 탐험할 것입니다. 물론 전편과 똑같이, 라이브러리나 프레임워크 등 기존 제품은 사용하지 않고 ‘밑바닥부터’ 만드는 데 초점을 뒀습니다. 직접 만들어보면서 딥러닝 관련 기술의 재미와 깊이를 탐구해봅시다.

이번 장에서는 신경망을 복습합니다. 전편의 내용을 요약한 장이라고 할 수 있겠죠. 한 가지 더, 이 책에서는 효율을 높이하고자 전편에서의 구현 규칙을 일부 변경했습니다(예컨대 메서드 이름이나 매개변수를 선언하는 규칙 등). 달라진 점도 이번 장에서 확인해보겠습니다.

1.1 수학과 파이썬 복습

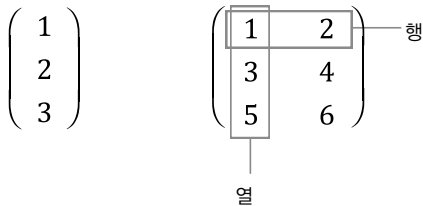
먼저 수학부터 복습해봅시다. 정확하게는 신경망 계산에 필요한 ‘벡터(vector)’나 ‘행렬(matrix)’ 등에 관한 이야기입니다. 또, 신경망을 원활하게 구현하기 위한 파이썬 코드, 특히 넘파이를 사용한 코드도 되새겨볼 것입니다.

1.1.1 벡터와 행렬

신경망에서는 ‘벡터’와 ‘행렬’(또는 ‘텐서^{tensor}’)이 도처에서 등장합니다. 그러니 먼저 이 용어들을 가볍게 정리하여 이 책을 읽어나가기 위한 준비를 해봅시다.

‘벡터’부터 시작해보죠. 벡터는 크기와 방향을 가진 양^量입니다. 벡터는 숫자가 일렬로 늘어선 집합으로 표현할 수 있으며, 파이썬에서는 1차원 배열로 취급할 수 있습니다. 그에 반해 ‘행렬’은 숫자가 2차원 형태(사각형 형상)로 늘어선 것입니다. [그림 1-1]에서 벡터와 행렬의 예를 살펴보세요.

그림 1-1 벡터와 행렬의 예



[그림 1-1]처럼 벡터는 1차원 배열로, 행렬은 2차원 배열로 표현할 수 있습니다. 또 행렬에서는 가로줄을 행^{row}이라 하고, 세로줄을 열^{column}이라 합니다. 그래서 [그림 1-1]의 행렬은 ‘3행 2열의 행렬’이라 하고 ‘3×2 행렬’이라고 씁니다.

NOTE 벡터와 행렬을 확장하여 숫자 집합을 N 차원으로 표현한 것도 생각할 수 있습니다. 이를 일반적으로 텐서라고 합니다.

벡터는 단순한 개념이지만 이를 표현하는 방법이 두 가지이므로 주의해야 합니다. 하나는 숫자들을 세로로 나열하는 방법(열벡터)이고, 다른 하나는 가로로 나열하는 방법(행벡터)입니다.

그림 1-2 벡터의 표현법



수학과 딥러닝 등 많은 분야에서 ‘열벡터’ 방식을 선호하지만, 이 책에서는 구현 편의를 고려해 ‘행벡터’로 다루겠습니다(매번 행벡터임을 명시합니다). 또한 수식에서의 벡터나 행렬은 \mathbf{x} 와 \mathbf{W} 처럼 굵게 표기하여 단일 원소로 이뤄진 스칼라 값과 구별했습니다(소스 코드에서의 변수를 가리키는 x 와 W 는 일반 글꼴로 표기했습니다).

WARNING_ 파이썬으로 구현할 때 벡터를 ‘행벡터’로 취급할 경우, 벡터를 가로 방향 ‘행렬’로 변환해 사용하면 명확해집니다. 예컨대 원소 수가 N 개인 벡터라면 $1 \times N$ 형상의 행렬로 처리합니다. 구체적인 예는 나중에 살펴보겠습니다.

그럼 파이썬을 대화형 모드로 실행하고 벡터와 행렬을 생성해봅시다. 물론 여기에서는 행렬을 취급할 때의 단골 라이브러리인 넘파이를 이용합니다.

```
>>> import numpy as np

>>> x = np.array([1, 2, 3])
>>> x.__class__          # 클래스 이름 표시
<class 'numpy.ndarray'>
>>> x.shape
(3,)
>>> x.ndim
1

>>> W = np.array([[1, 2, 3], [4, 5, 6]])
>>> W.shape
(2, 3)
>>> W.ndim
2
```

이 코드에서 보듯 벡터와 행렬은 `np.array()` 메서드로 생성할 수 있습니다. 이 메서드는 넘파이의 다차원 배열 클래스인 `np.ndarray` 클래스를 생성합니다. `np.ndarray` 클래스에는 다양한 편의 메서드와 인스턴스 변수가 준비되어 있으며, 앞의 예에서는 인스턴스 변수 중 `shape`와 `ndim`을 이용했습니다. `shape`는 다차원 배열의 형상을, `ndim`은 차원 수를 담고 있습니다. 앞의 결과를 보면 `x`는 1차원 배열이며 원소 수가 3개인 벡터임을 알 수 있습니다. 그리고 `W`는 2차원 배열이며, 2×3 (2행 3열) 행렬임을 알 수 있습니다.

1.1.2 행렬의 원소별 연산

앞에서는 수의 집합을 벡터나 행렬로 표현하는 방법을 알아보았습니다. 이번에는 이렇게 표현한 벡터와 행렬을 사용해 간단한 계산을 해봅시다. 먼저 ‘원소별^{element-wise} 연산’을 살펴봅시다.

```
>>> W = np.array([[1, 2, 3], [4, 5, 6]])
>>> X = np.array([[0, 1, 2], [3, 4, 5]])
>>> W + X
array([[ 1,  3,  5],
       [ 7,  9, 11]])
>>> W * X
array([[ 0,  2,  6],
       [12, 20, 30]])
```

다차원 넘파이 배열의 사칙연산 중 더하기(+)와 곱하기(*)를 해보았습니다. 이렇게 하면 피연산자인 다차원 배열들에서 서로 대응하는 원소끼리(각 원소가 독립적으로) 연산이 이뤄집니다. 이것이 넘파이 배열의 ‘원소별 연산’입니다.

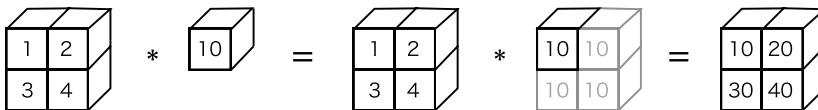
1.1.3 브로드캐스트

넘파이의 다차원 배열에서는 형상이 다른 배열끼리도 연산할 수 있습니다. 예컨대 다음과 같은 계산이 가능합니다.

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A * 10
array([[10, 20],
       [30, 40]])
```

이 계산에서는 2×2 행렬 A에 10이라는 스칼라 값을 곱했습니다. 이렇게 하면 [그림 1-3]처럼 스칼라 값 10이 2×2 행렬로 확장된 후에 원소별 연산을 수행합니다. 이 영리한 기능을 브로드캐스트^{broadcast}라 합니다.

그림 1-3 브로드캐스트의 예: 스칼라 값인 10이 2×2 행렬로 처리된다.

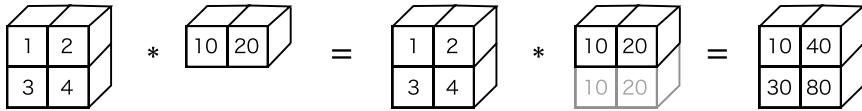


브로드캐스트의 또 다른 예로 다음 계산을 살펴봅시다.

```
>>> A = np.array([[1, 2], [3, 4]])
>>> b = np.array([10, 20])
>>> A * b
array([[10, 40],
       [30, 80]])
```

이 계산에서는 [그림 1-4]처럼 1차원 배열인 b가 2차원 배열 A와 형상이 같아지도록 ‘영리하게’ 확장됩니다.

그림 1-4 브로드캐스트의 예 2



이처럼 넘파이는 브로드캐스트라는 기능을 제공하여 형상이 다른 배열끼리의 연산을 영리하게 수행할 수 있습니다.

WARNING_ 넘파이의 브로드캐스트가 효과적으로 동작하려면 다차원 배열의 형상이 몇 가지 규칙을 충족해야 합니다. 브로드캐스트의 자세한 규칙은 문헌 [1]을 참고하세요.

1.1.4 벡터의 내적과 행렬의 곱

계속해서 ‘벡터의 내적’과 ‘행렬의 곱셈’에 관해 살펴보겠습니다. 우선은 벡터의 내적으로, 수식으로는 다음과 같습니다.

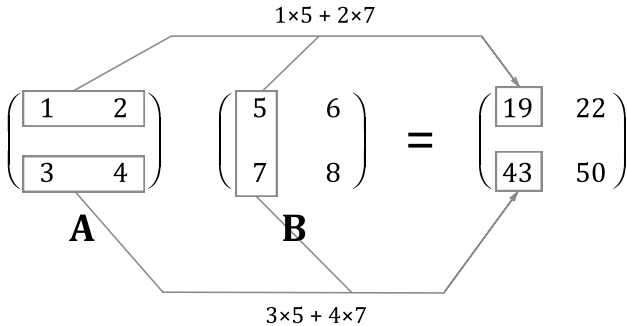
$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \quad [\text{식 1.1}]$$

여기에서는 2개의 벡터 $\mathbf{x} = (x_1, \dots, x_n)$ 과 $\mathbf{y} = (y_1, \dots, y_n)$ 이 있다고 가정합니다. 그리고 [식 1.1]에서 보듯, 벡터의 내적은 두 벡터에서 대응하는 원소들의 곱을 모두 더한 것입니다.

NOTE _ 벡터의 내적은 직관적으로는 ‘두 벡터가 얼마나 같은 방향을 향하고 있는가’를 나타냅니다. 벡터의 길이가 1인 경우로 한정하면, 완전히 같은 방향을 향하는 두 벡터의 내적은 1이 됩니다. 반대로, 반대 방향을 향하는 두 벡터의 내적은 -1입니다.

계속해서 ‘행렬의 곱’도 살펴보겠습니다. 행렬의 곱은 [그림 1-5]의 순서로 계산합니다.

그림 1-5 행렬의 곱셈 방법



[그림 1-5]처럼 행렬의 곱은 ‘왼쪽 행렬의 행벡터(가로 방향)’와 ‘오른쪽 행렬의 열벡터(세로 방향)’의 내적(원소별 곱의 합)으로 계산합니다. 그리고 계산 결과는 새로운 행렬의 대응하는 원소에 저장되죠. 예를 들어 **A**의 1행과 **B**의 1열의 계산 결과는 1행 1열 위치의 원소가 되고, **A**의 2행과 **B**의 1열의 계산 결과는 2행 1열 위치의 원소가 되는 식입니다.

그럼 벡터의 내적과 행렬의 곱을 파이썬으로 구현해봅시다. 넘파이의 `np.dot()`과 `np.matmul()` 메서드를 이용하면 쉽게 구현할 수 있습니다.

```

# 벡터의 내적
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> np.dot(a, b)
32

# 행렬의 곱
>>> A = np.array([[1, 2], [3, 4]])
>>> B = np.array([[5, 6], [7, 8]])
>>> np.matmul(A, B)
array([[19, 22],
       [43, 50]])

```

사실 벡터의 내적과 행렬의 곱 모두에 `np.dot()`을 사용할 수 있습니다. `np.dot(x, y)`의 인수가 모두 1차원 배열이면 벡터의 내적을 계산하고, 2차원 배열이면 행렬의 곱을 계산합니다. 다만, 가능하면 둘을 구분하여 코드의 논리와 의도를 명확히 해주는 게 좋습니다.

`np.dot()`과 `np.matmul()` 외에도 넘파이에는 행렬 계산을 도와주는 편의 메서드가 많이 준비되어 있습니다. 그 메서드들을 잘 다루면 신경망 구현도 막힘없이 진행할 수 있을 것입니다.

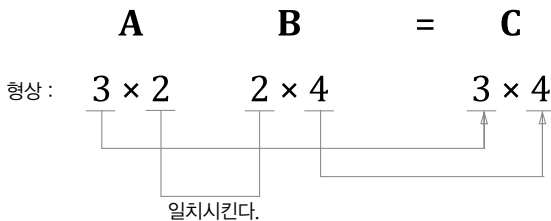
NOTE_ 배우기보다 익숙해져라

넘파이를 익히는 데는 실제로 코딩해보며 연습해보는 방법이 가장 좋습니다. 넘파이 경험을 쌓고 싶은 분께는 ‘100 numpy exercises’ 사이트를 추천합니다. 넘파이 연습 문제가 100개나 있으니 꼭 도전해보세요.

1.1.5 행렬 형상 확인

행렬이나 벡터를 사용해 계산할 때는 그 ‘형상(shape)’에 주의해야 합니다. 이번 절에서는 ‘행렬의 곱’을 형상에 주목해 다시 확인해보려 합니다. 행렬 곱의 계산 순서는 앞서 설명했습니다만, 이때 [그림 1-6]처럼 ‘형상 확인’이 중요합니다.

그림 1-6 형상 확인: 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시킨다.



[그림 1-6]은 3×2 행렬 **A**와 2×4 행렬 **B**를 곱하여 3×4 행렬 **C**를 만드는 예입니다. 이때 이 그림처럼 행렬 **A**와 **B**가 대응하는 차원의 원소 수가 같아야 합니다. 그리고 결과로 만들어진 행렬 **C**의 형상은 **A**의 행 수와 **B**의 열 수가 됩니다. 이것이 행렬의 ‘형상 확인’입니다.

NOTE_ 행렬의 곱 등 행렬을 계산할 때는 형상 확인이 중요합니다. 그래야 신경망 구현을 부드럽게 진행할 수 있습니다.

1.2 신경망의 추론

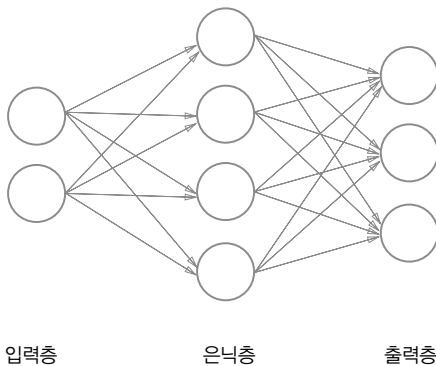
이제 신경망을 복습할 차례입니다. 신경망에서 수행하는 작업은 두 단계로 나눌 수 있습니다. 바로 ‘학습’과 ‘추론’이죠. 이번 절에서는 신경망 ‘추론’에 집중하고, ‘학습’은 다음 절에서 다루겠습니다.

1.2.1 신경망 추론 전체 그림

신경망은 간단히 말하면 단순한 ‘함수’라 할 수 있습니다. 함수란 무엇인가를 입력하면 무엇인가를 출력하는 변환기죠. 다시 말해 신경망도 함수처럼 입력을 출력으로 변환합니다.

이번 절에서는 2차원 데이터를 입력하여 3차원 데이터를 출력하는 함수를 예로 들겠습니다. 이 함수를 신경망으로 구현하려면 **입력층** input layer에는 뉴런 2개를, **출력층** output layer에는 3개를 각각 준비합니다. 그리고 **은닉층** hidden layer (혹은 **중간층**)에도 적당한 수의 뉴런을 배치합니다. 이번 예에는 은닉층에 뉴런 4개를 두기로 합니다. 그러면 우리의 신경망은 [그림 1-7]처럼 그릴 수 있습니다.

그림 1-7 신경망의 예



[그림 1-7]에서는 뉴런을 ○로, 그 사이의 연결을 화살표로 나타냈습니다. 이때 화살표에는 **가중치** weight가 존재하여, 그 가중치와 뉴런의 값을 각각 곱해서 그 합이 다음 뉴런의 입력으로 쓰입니다(정확하게는 그 합에 활성화 함수 activation function를 적용한 값이 다음 뉴런의 입력이 됩니다). 또, 이때 각 층에서는 이전 뉴런의 값에 영향받지 않는 ‘정수’도 더해집니다. 이 정수는

편향^{bias}이라고 합니다. 덧붙여 [그림 1-7]의 신경망은 인접하는 층의 모든 뉴런과 연결(화살표로 이어짐)되어 있다는 뜻에서 **완전연결계층**^{fully connected layer}이라고 합니다.

NOTE [그림 1-7]의 신경망은 총 3층 구성입니다만, 가중치를 지니는 층은 사실 2개뿐입니다. 그래서 이 책에서는 이러한 신경망을 '2층 신경망'이라고 부르겠습니다. 문헌에 따라서는 [그림 1-7]과 같은 구성을 '3층 신경망'이라고 하는 경우도 있으니 주의하세요.

그럼 [그림 1-7]의 신경망이 수행하는 계산을 수식으로 나타내봅시다. 여기에서는 입력층의 데이터를 (x_1, x_2) 로 쓰고, 가중치는 w_{11} 과 w_{21} 으로, 편향은 b_1 으로 쓰겠습니다. 그러면 [그림 1-7]의 은닉층 중 첫 번째 뉴런은 다음과 같이 계산할 수 있습니다.

$$h_1 = x_1 w_{11} + x_2 w_{21} + b_1 \quad \text{[식 1.2]}$$

[식 1.2]와 같이 은닉층의 뉴런은 가중치의 합으로 계산됩니다. 이런 식으로 가중치와 편향의 값을 바꿔가며 [식 1.2]의 계산을 뉴런의 수만큼 반복하면 은닉층에 속한 모든 뉴런의 값을 구할 수 있지요.

가중치와 편향에는 첨자(인덱스)가 붙습니다. 이 첨자를 붙이는 규칙('11'이나 '12' 등을 어떻게 설정하느냐 하는 규칙)은 중요하지 않습니다. 중요한 것은 그것이 '가중치 합'으로 계산된다는 것, 그리고 그 값은 행렬의 곱으로 한꺼번에 계산할 수 있다는 사실입니다. 실제로 완전연결계층이 수행하는 변환은 행렬의 곱을 이용해 다음처럼 정리해 쓸 수 있습니다.

$$(h_1, h_2, h_3, h_4) = (x_1, x_2) \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + (b_1, b_2, b_3, b_4) \quad \text{[식 1.3]}$$

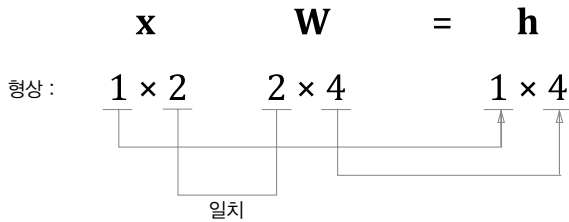
여기에서 은닉층의 뉴런들은 (h_1, h_2, h_3, h_4) 로 정리되며, 1×4 행렬로 간주할 수 있습니다(혹은 '행벡터'로 취급합니다). 또 입력 (x_1, x_2) 는 1×2 행렬이며, 가중치는 2×4 행렬, 편향은 1×4 행렬에 대응합니다. 그러면 [식 1.3]은 다음처럼 간소화할 수 있습니다.

$$\mathbf{h} = \mathbf{xW} + \mathbf{b} \quad \text{[식 1.4]}$$

여기서 \mathbf{x} 는 입력, \mathbf{h} 는 은닉층의 뉴런, \mathbf{W} 는 가중치, \mathbf{b} 는 편향을 뜻합니다. 이 기호 각각은 모두 행렬입니다. 그리고 [식 1.4]의 각 행렬의 형상을 잘 보면 [그림 1-8]처럼 변환된다는

사실을 알 수 있죠.

그림 1-8 형상 확인: 대응하는 차원의 원소 수가 일치함(편향은 생략)

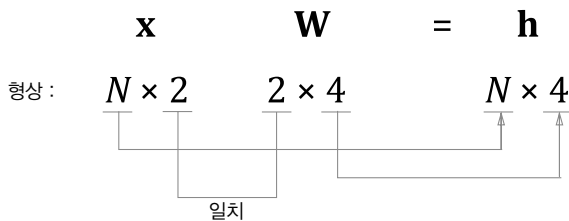


[그림 1-8]에서 보듯, 행렬의 곱에서는 대응하는 차원의 원소 수가 일치해야 합니다. 이처럼 행렬의 형상을 살펴보면 올바른 변환인지를 확인할 수 있습니다.

NOTE 행렬의 곱 계산에서는 행렬의 형상 확인이 중요합니다. 형상을 보면 그 계산이 올바른 계산인지, 적어도 계산이 성립하는지 여부를 확인할 수 있습니다.

이것으로 완전연결계층에 의한 변환을 행렬로 정리하고 계산해봤습니다. 그런데 지금까지 수행한 변환은 하나의 샘플 데이터(입력 데이터)만을 대상으로 했습니다. 하지만 신경망의 추론이나 학습에서는 다수의 샘플 데이터(미니배치^{minibatch}*)를 한꺼번에 처리합니다. 이렇게 하려면 행렬 **x**의 행 각각에 샘플 데이터를 하나씩 저장해야 합니다. 예컨대 *N*개의 샘플 데이터를 미니배치로 한꺼번에 처리한다면 [그림 1-9]처럼 됩니다(행렬의 형상에 주목).

그림 1-9 형상 확인: 미니배치 버전의 행렬 곱(편향은 생략)



[그림 1-9]와 같이 형상 확인을 통해 각 미니배치가 올바르게 변환되었는지를 알 수 있습니다. 이때 *N*개의 샘플 데이터가 한꺼번에 완전연결계층에 의해 변환되고, 은닉층에는 *N*개 분의

* 옮긴이_ 전체 데이터를 작은 그룹으로 나눠 그룹 단위로 반복 학습하는 방식을 미니배치 학습이라 하며, 이때 각각의 그룹을 미니배치라 합니다. 이 책에서의 신경망 학습은 기본적으로 미니배치 방식을 가정합니다.

뉴런이 함께 계산됩니다. 자, 그럼 완전연결계층에 의한 변환의 미니배치 버전을 파이썬으로 구현해봅시다.

```
>>> import numpy as np
>>> W1 = np.random.randn(2, 4) # 가중치
>>> b1 = np.random.randn(4)    # 편향
>>> x = np.random.randn(10, 2) # 입력
>>> h = np.matmul(x, W1) + b1
```

이 예에서는 10개의 샘플 데이터 각각을 완전연결계층으로 변환시켰습니다. 이때 x 의 첫 번째 차원이 각 샘플 데이터에 해당합니다. 예컨대 $x[0]$ 는 0번째 입력 데이터, $x[1]$ 은 첫 번째 입력 데이터가 되는 식입니다. 마찬가지로 $h[0]$ 는 0번째 데이터의 은닉층 뉴런, $h[1]$ 은 1번째 데이터의 은닉층 뉴런이 저장됩니다.

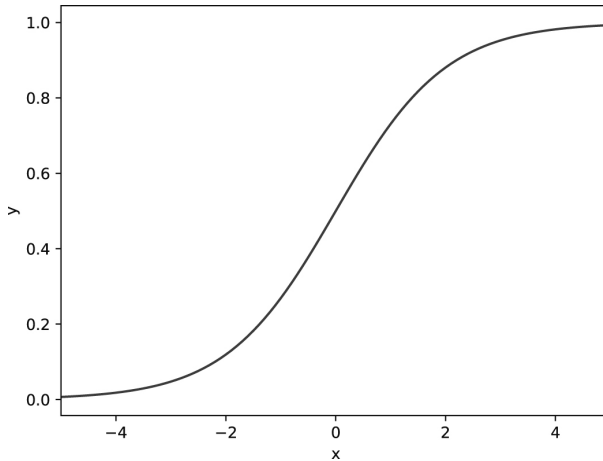
WARNING_ 이 코드의 마지막 줄에서 편향 $b1$ 의 덧셈은 브로드캐스트됩니다. $b1$ 의 형상은 (4,)이지만 자동으로 (10, 4)로 복제되는 것이죠.

그런데 완전연결계층에 의한 변환은 ‘선형’ 변환입니다. 여기에 ‘비선형’ 효과를 부여하는 것이 바로 활성화 함수입니다. 더 정확하게 말하면, 비선형 활성화 함수를 이용함으로써 신경망의 표현력을 높일 수 있습니다. 활성화 함수는 아주 다양합니다만, 여기에서는 [식 1.5]의 **시그모이드 함수** sigmoid function를 사용하기로 하죠.

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \quad \text{[식 1.5]}$$

시그모이드 함수는 [그림 1-10]과 같은 알파벳 ‘S’자 모양의 곡선 함수입니다.

그림 1-10 시그모이드 함수의 그래프



시그모이드 함수는 임의의 실수를 입력받아 0에서 1 사이의 실수를 출력합니다. 곧바로 이 시그모이드 함수를 파이썬으로 구현해보겠습니다.

```
def sigmoid(x):  
    return 1 / (1 + np.exp(-x))
```

[식 1.5]를 그대로 구현한 코드이니 특별히 어려운 점은 없을 겁니다. 이 시그모이드 함수를 사용하여 방금 전의 은닉층 뉴런을 변환하겠습니다.

```
>>> a = sigmoid(h)
```

이로써 시그모이드 함수에 의해 비선형 변환이 가능했습니다. 계속해서 이 활성화 함수의 출력인 a (이를 **활성화**^{activation}라고 합니다)를 또 다른 완전연결계층에 통과시켜 변환합니다. 지금 예에서는 은닉층의 뉴런은 4개, 출력층의 뉴런은 3개이므로 완전연결계층에 사용되는 가중치 행렬은 4×3 형상으로 설정해야 합니다. 이것으로 출력층의 뉴런을 얻을 수 있습니다. 이상이 신경망의 추론입니다. 그럼 지금까지의 이야기를 종합해 파이썬으로 작성해볼까요?

```
import numpy as np  
  
def sigmoid(x):  
    return 1 / (1 + np.exp(-x))
```

```

x = np.random.randn(10, 2)
W1 = np.random.randn(2, 4)
b1 = np.random.randn(4)
W2 = np.random.randn(4, 3)
b2 = np.random.randn(3)

h = np.matmul(x, W1) + b1
a = sigmoid(h)
s = np.matmul(a, W2) + b2

```

여기에서 x 의 형상은 (10, 2)입니다. 2차원 데이터 10개가 미니배치로 처리된다는 뜻이죠. 그리고 최종 출력인 s 의 형상은 (10, 3)이 됩니다. 다시 말하지만, 이것은 10개의 데이터가 한꺼번에 처리되었고, 각 데이터는 3차원 데이터로 변환되었다는 뜻입니다.

그런데 이 신경망은 3차원 데이터를 출력합니다. 따라서 각 차원의 값을 이용하여 3 클래스 분류를 할 수 있습니다. 이 경우, 출력된 3차원 벡터의 각 차원은 각 클래스에 대응하는 ‘점수^{score}’가 됩니다(첫 번째 뉴런이 첫 번째 클래스, 두 번째 뉴런이 두 번째 클래스, ...). 실제로 분류를 한다면 출력층에서 가장 큰 값을 내뱉는 뉴런에 해당하는 클래스가 예측 결과가 되는 것이죠.

NOTE 점수란 ‘확률’이 되기 전의 값입니다. 점수가 높을수록 그 뉴런에 해당하는 클래스의 확률도 높아집니다. 덧붙여 (나중에 설명합니다만) 점수를 소프트맥스 함수 `softmax function`에 입력하면 확률을 얻을 수 있습니다.

이상으로 신경망의 추론을 구현해봤습니다. 다음 절에서는 지금까지 수행한 처리를 하나의 ‘계층’으로 추상화한 파이썬 클래스를 구현해볼 것입니다.

1.2.2 계층으로 클래스화 및 순전파 구현

그럼 신경망에서 하는 처리를 계층^{layer}으로 구현해봅시다. 여기에서는 완전연결계층에 의한 변환을 Affine 계층으로, 시그모이드 함수에 의한 변환을 Sigmoid 계층으로 구현할 겁니다. 참고로 완전연결계층에 의한 변환은 기하학에서의 아핀^{affine} 변환에 해당하기 때문에 Affine 계층이라고 이름 지었습니다. 또한 각 계층은 파이썬 클래스로 구현하며, 기본 변환을 수행하는 메서드의 이름은 `forward()`로 하겠습니다.

NOTE 신경망 추론 과정에서 하는 처리는 신경망의 **순전파**(forward propagation)에 해당합니다. 순전파란 말 그대로 입력층에서 출력층으로 향하는 전파입니다. 순전파 때는 신경망을 구성하는 각 계층이 입력으로부터 출력 방향으로 처리 결과를 차례로 전파해갑니다. 나중에 살펴볼 신경망 학습에서는 데이터(기울기)를 순전파와는 반대 방향으로 전파합니다. 이런 **역전파**(backward propagation)라고 합니다.

신경망에는 다양한 계층이 등장하는데, 우리는 이 계층들을 모두 파이썬 클래스로 구현할 겁니다. 이렇게 모듈화를 해두면 레고 블록을 조합하듯 신경망을 구축할 수 있습니다. 이 책에서는 이러한 계층을 구현할 때 다음의 ‘구현 규칙’을 따르겠습니다.

- 모든 계층은 forward()와 backward() 메서드를 가진다.
- 모든 계층은 인스턴스 변수인 params와 grads를 가진다.

자, 이 구현 규칙을 간단히 설명해보죠. forward()와 backward() 메서드는 각각 순전파와 역전파를 수행합니다. params는 가중치와 편향 같은 매개변수를 담은 리스트입니다(매개변수는 여러 개가 있을 수 있어서 리스트에 보관합니다). 마지막으로 grads는 params에 저장된 각 매개변수에 대응하여, 해당 매개변수의 기울기를 보관하는 리스트입니다(기울기에 관해서는 뒤에서 설명합니다). 이것이 이 책의 ‘구현 규칙’입니다.

NOTE 이 구현 규칙에 따라 구현하면 일관되고 확장성이 좋아집니다. 왜 이 규칙을 따르는지, 또 이 규칙이 얼마나 유효한지는 나중에 밝혀집니다.

이번 절에서는 순전파만 구현할 것이므로 앞의 구현 규칙 중 다음 두 사항만 적용하겠습니다. 첫째, 각 계층은 forward() 메서드만 가집니다. 둘째, 매개변수들은 params 인스턴스 변수에 보관합니다. 이 구현 규칙에 따라 계층들을 구현해볼까요? 가장 먼저 구현할 계층은 Sigmoid 계층입니다(☞ ch01/forward_net.py).

```
import numpy as np

class Sigmoid:
    def __init__(self):
        self.params = []

    def forward(self, x):
        return 1 / (1 + np.exp(-x))
```

이와 같이 시그모이드 함수를 클래스로 구현했으며, 주 변환 처리는 `forward(x)` 메서드가 담당합니다. Sigmoid 계층에는 학습하는 매개변수가 따로 없으므로 인스턴스 변수인 `params`는 빈 리스트로 초기화합니다. 그럼 계속해서 완전연결계층인 Affine 계층의 구현을 봅시다 (☞ `ch01/forward_net.py`).

```
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]

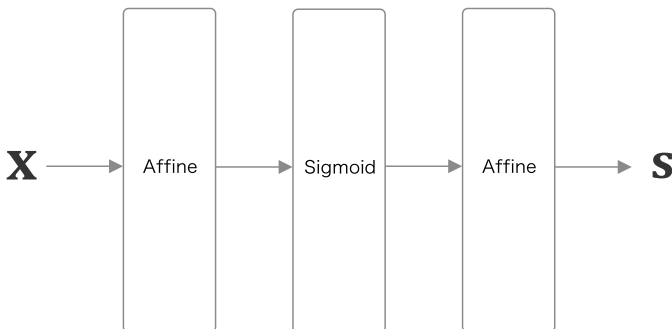
    def forward(self, x):
        W, b = self.params
        out = np.matmul(x, W) + b
        return out
```

Affine 계층은 초기화될 때 가중치와 편향을 받습니다. 즉, 가중치와 편향은 Affine 계층의 매개변수이며(이 두 매개변수는 신경망이 학습될 때 수시로 갱신됩니다), 리스트인 `params` 인스턴스 변수에 보관합니다. 다음으로 `forward(x)`는 순전파 처리를 구현합니다.

NOTE 이 책의 예제 코드는 앞의 '구현 규칙'을 따르므로, 모든 계층에는 학습해야 하는 매개변수가 반드시 인스턴스 변수인 `params`에 존재하게 됩니다. 이 덕분에 신경망의 모든 매개변수를 간단하게 정리할 수 있고, 자연스럽게 매개변수 갱신 작업이나 매개변수를 파일로 저장하는 일이 쉬워집니다.

그러면 앞에서 구현한 계층을 사용해 신경망의 추론 처리를 구현해보죠. [그림 1-11]처럼 구성된 신경망을 구현할 것입니다.

그림 1-11 구현해볼 신경망의 계층 구성



[그림 1-11]에서 보듯, 이번 예에서는 입력 \mathbf{x} 가 Affine 계층, Sigmoid 계층, Affine 계층을 차례로 거쳐 점수인 \mathbf{s} 를 출력하게 됩니다. 이 신경망을 TwoLayerNet이라는 클래스로 추상화 하고, 주 추론 처리는 predict(\mathbf{x}) 메서드로 구현하겠습니다.

NOTE 신경망을 그릴 때, 지금까지는 [그림 1-7]과 같은 '뉴런 관점'의 그림을 이용했습니다만, 이후로는 [그림 1-11]처럼 '계층 관점'으로 표현하겠습니다.

그러면 TwoLayerNet의 구현을 살펴보죠(☞ ch01/forward_net.py).

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 가중치와 편향 초기화
        W1 = np.random.randn(I, H)
        b1 = np.random.randn(H)
        W2 = np.random.randn(H, O)
        b2 = np.random.randn(O)

        # 계층 생성
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]

        # 모든 가중치를 리스트에 모은다.
        self.params = []
        for layer in self.layers:
            self.params += layer.params

    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x
```

이 클래스의 초기화 메서드(__init__)는 먼저 가중치를 초기화하고 3개의 계층을 생성합니다. 마지막으로 학습해야 할 가중치 매개변수들을 params 리스트에 저장합니다. 모든 계층은 자신의 학습 매개변수들을 인스턴스 변수인 params에 보관하고 있으므로, 이 변수들을 더

해주기만 하면 됩니다. 이로써 TwoLayerNet의 params 변수에는 모든 학습 매개변수가 담기게 되었습니다. 이처럼 매개변수들을 하나의 리스트에 보관하면 ‘매개변수 갱신’과 ‘매개변수 저장’을 손쉽게 처리할 수 있습니다.

참고로, 파이썬에서 + 연산자는 리스트들을 결합해줍니다. 간단한 예를 보시죠.

```
>>> a = ['A', 'B']
>>> a += ['C', 'D']
>>> a
['A', 'B', 'C', 'D']
```

보다시피 리스트끼리 더하면 리스트들이 결합됩니다. 앞서의 TwoLayerNet 구현에서는 각 계층의 params 리스트를 더해주는 것만으로, 모든 학습 매개변수를 하나의 리스트에 담은 것입니다. 그럼 TwoLayerNet 클래스를 이용해 신경망의 추론을 수행해봅시다.

```
x = np.random.randn(10, 2)
model = TwoLayerNet(2, 4, 3)
s = model.predict(x)
```

이상으로 입력 데이터 x에 대한 점수(s)를 구할 수 있었습니다. 이처럼 계층을 클래스로만 들어두면 신경망을 쉽게 구현할 수 있습니다. 또한 학습해야 할 모든 매개변수가 model.params라는 하나의 리스트에 모여 있으므로, 이어서 설명할 신경망 학습이 한결 수월해집니다.

1.3 신경망의 학습

학습되지 않은 신경망은 ‘좋은 추론’을 해낼 수 없습니다. 그래서 학습을 먼저 수행하고, 그 학습된 매개변수를 이용해 추론을 수행하는 흐름이 일반적입니다. 추론이란 앞 절에서 본 것 같은 다중 클래스 분류 등의 문제에 답을 구하는 작업입니다. 한편, 신경망의 학습은 최적의 매개변수 값을 찾는 작업입니다. 이번 절에서는 신경망의 학습에 대해 살펴보겠습니다.

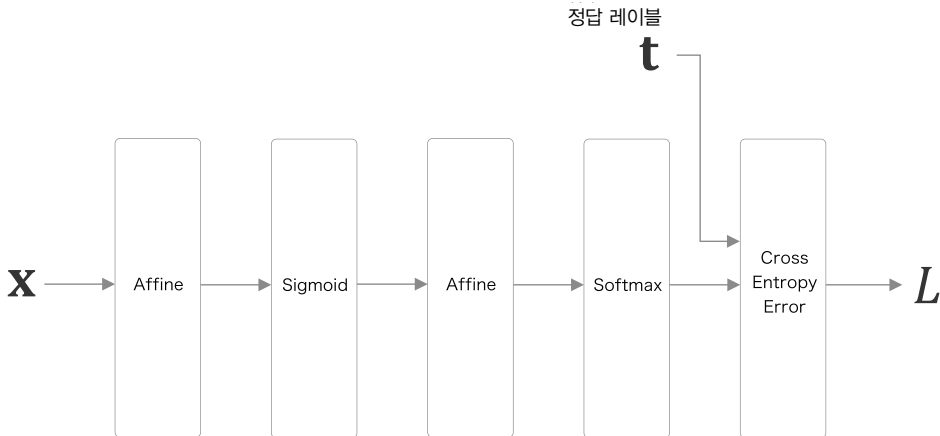
1.3.1 손실 함수

신경망 학습에는 학습이 얼마나 잘 되고 있는지를 알기 위한 ‘척도’가 필요합니다. 일반적으로 학습 단계의 특정 시점에서 신경망의 성능을 나타내는 척도로 **손실**(loss)을 사용합니다. 손실은 학습 데이터(학습 시 주어진 정답 데이터)와 신경망이 예측한 결과를 비교하여 예측이 얼마나 나쁜가를 산출한 단일 값(스칼라)입니다.

신경망의 손실은 **손실 함수**(loss function)를 사용해 구합니다. 다중 클래스 분류(multi-class classification) 신경망에서는 손실 함수로 흔히 **교차 엔트로피 오차**(Cross Entropy Error)를 이용합니다. 교차 엔트로피 오차는 신경망이 출력하는 각 클래스의 ‘확률’과 ‘정답 레이블’을 이용해 구할 수 있습니다.

그럼 우리가 지금까지 다뤄 온 신경망에서 손실을 구해보죠. 우선 앞 절의 신경망에 Softmax 계층과 Cross Entropy Error 계층을 새로 추가합니다(Softmax 계층은 소프트맥스 함수를, Cross Entropy Error 계층은 교차 엔트로피 오차를 구하는 계층입니다). 이 신경망의 구성을 ‘계층 관점’에서 그리면 [그림 1-12]처럼 됩니다.

그림 1-12 손실 함수를 적용한 신경망의 계층 구성



[그림 1-12]의 \mathbf{x} 는 입력 데이터, \mathbf{t} 는 정답 레이블, L 은 손실을 나타냅니다. 이때 Softmax 계층의 출력은 확률이 되어, 다음 계층인 Cross Entropy Error 계층에는 확률과 정답 레이블이 입력됩니다.

이어서 소프트맥스 함수와 교차 엔트로피 오차에 관해 알아보시다. 우선은 소프트맥스 함수를 식으로 쓰면 다음과 같습니다.

$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^n \exp(s_i)} \quad [\text{식 1.6}]$$

[식 1.6]은 출력이 총 n 개일 때, k 번째의 출력 y_k 를 구하는 계산식입니다. y_k 는 k 번째 클래스에 해당하는 소프트맥스 함수의 출력입니다. 이 식에서 보듯 소프트맥스 함수의 분자는 점수 s_k 의 지수 함수이고, 분모는 모든 입력 신호의 지수 함수의 총합입니다.

소프트맥스 함수의 출력의 각 원소는 0.0 이상 1.0 이하의 실수입니다. 그리고 그 원소들을 모두 더하면 1.0이 됩니다. 이것이 소프트맥스의 출력을 ‘확률’로 해석할 수 있는 이유죠. 소프트맥스의 출력인 이 ‘확률’이 다음 차례인 교차 엔트로피 오차에 입력됩니다. 이때 교차 엔트로피 오차의 수식은 다음과 같습니다.

$$L = -\sum_k t_k \log y_k \quad [\text{식 1.7}]$$

여기서 t_k 는 k 번째 클래스에 해당하는 정답 레이블입니다. \log 는 네이피어 상수(혹은 오일러의 수) e 를 밑으로 하는 로그입니다(정확하게는 \log_e 로 표기합니다). 정답 레이블은 $t = [0, 0, 1]$ 과 같이 원핫 벡터로 표기합니다.

NOTE 원핫 벡터(one-hot vector)란 단 하나의 원소만 1이고, 그 외에는 0인 벡터입니다. 여기서 1인 원소가 정답 클래스에 해당합니다. 따라서 [식 1.7]은 실질적으로 정답 레이블이 1의 원소에 해당하는 출력의 자연로그(\log)를 계산할 뿐입니다(다른 원소들은 t_k 가 0이므로 계산 결과에 영향을 주지 못합니다).

나아가 미니배치 처리를 고려하면 교차 엔트로피 오차의 식은 다음처럼 됩니다. 이 식에서 데이터는 N 개이며, t_{nk} 는 n 번째 데이터의 k 차원째의 값을 의미합니다. 그리고 y_{nk} 는 신경망의 출력이고, t_{nk} 는 정답 레이블입니다.

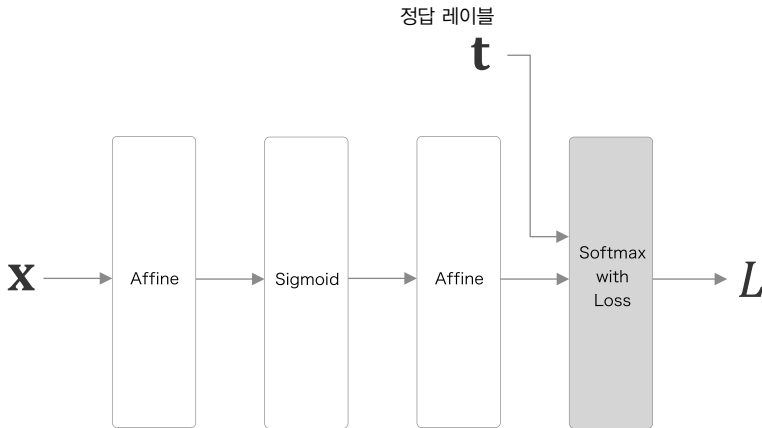
$$L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk} \quad [\text{식 1.8}]$$

[식 1.8]은 조금 복잡해 보이지만, 하나의 데이터에 대한 손실 함수를 나타낸 [식 1.7]을 단순히 데이터 N 개짜리로 확장했을 뿐입니다. 다만, [식 1.8]에서는 N 으로 나눠서 1개당의 ‘평균 손실 함수’를 구합니다. 이렇게 평균을 구함으로써 미니배치의 크기에 관계없이 항상 일관된

척도를 얻을 수 있습니다.

이 책에서는 소프트맥스 함수와 교차 엔트로피 오차를 계산하는 계층을 Softmax with Loss 계층 하나로 구현합니다(이 두 계층을 통합하면 역전파 계산이 쉬워집니다). 따라서 우리의 (학습 시) 신경망의 계층 구성은 [그림 1-13]처럼 됩니다.

그림 1-13 Softmax with Loss 계층을 이용하여 손실을 출력한다.



[그림 1-13]과 같이 이 책에서는 Softmax with Loss 계층을 이용합니다만, 그 구현에 대해서는 따로 설명하지 않겠습니다. 구현 파일은 `common/layers.py`에 있으니 궁금한 분은 참고하세요. 『밑바닥부터 시작하는 딥러닝』의 ‘4.2 손실 함수’ 절에서도 Softmax with Loss 계층을 자세히 설명했습니다.

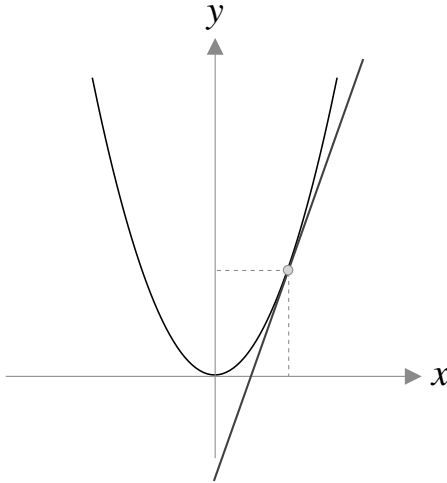
1.3.2 미분과 기울기

신경망 학습의 목표는 손실을 최소화하는 매개변수를 찾는 것입니다. 이때 중요한 것이 ‘미분’과 ‘기울기’입니다. 이번 절에서는 미분과 기울기에 대해 간략히 설명하겠습니다.

자, 어떤 함수 $y = f(x)$ 가 있다고 합시다. 이때 x 에 관한 y 의 미분은 $\frac{dy}{dx}$ 라고 씁니다. 이 $\frac{dy}{dx}$ 가 의미하는 것은 x 의 값을 ‘조금’ 변화시켰을 때(더 정확하게는 그 ‘조금의 변화’를 극한까지 줄일 때) y 값이 얼마나 변하는가 하는 ‘변화의 정도’입니다.

$y = x^2$ 이라는 함수를 예로 살펴보죠. 이 함수의 미분을 해석적으로 구하면 $\frac{dy}{dx} = 2x$ 가 됩니다. 이 미분 결과는 각 x 에서의 변화의 정도를 뜻하며, [그림 1-14]에서 보듯 함수의 '기울기'에 해당합니다.

그림 1-14 $y = x^2$ 의 미분은 각 x 에서의 기울기를 나타낸다.



[그림 1-14]에서는 x 라는 변수 하나에 대해 미분을 구했지만, 여러 개의 변수(다변수)라도 마찬가지로 미분할 수 있습니다. 예를 들어 L 은 스칼라, x 는 벡터인 함수 $L = f(x)$ 가 있습니다. 이때 (x 의 i 번째 원소인) x_i 에 대한 L 의 미분은 $\frac{\partial L}{\partial x_i}$ 로 쓸 수 있습니다. 그리고 벡터 x 의 다른 원소의 미분도 구할 수 있고, 이를 다음과 같이 정리할 수 있습니다.

$$\frac{\partial L}{\partial \mathbf{x}} = \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \dots, \frac{\partial L}{\partial x_n} \right) \quad \text{[식 1.9]}$$

이처럼 벡터의 각 원소에 대한 미분을 정리한 것이 **기울기**(gradient)입니다.

벡터와 마찬가지로, 행렬에서도 기울기를 생각할 수 있습니다. 예컨대 \mathbf{W} 가 $m \times n$ 행렬이라면, $L = g(\mathbf{W})$ 함수의 기울기는 다음과 같이 쓸 수 있습니다.

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial W_{m1}} & & \frac{\partial L}{\partial W_{mn}} \end{pmatrix} \quad [\text{식 1.10}]$$

[식 1.10]처럼 L 의 \mathbf{W} 에 대한 기울기를 행렬로 정리할 수 있습니다(정확하게는 행렬의 기울기를 위와 같이 정의합니다). 여기에서 중요한 점은 \mathbf{W} 와 $\frac{\partial L}{\partial \mathbf{W}}$ 의 형상이 같다는 것입니다. 그리고 ‘행렬과 그 기울기의 형상이 같다’라는 이 성질을 이용하면 매개변수 갱신과 연쇄 법칙을 쉽게 구현할 수 있습니다(연쇄 법칙은 다음 절에서 자세히 설명합니다).

WARNING_ 엄밀하게 말하면, 이 책에서 사용하는 ‘기울기’는 수학에서 말하는 기울기와는 다릅니다. 수학에서의 기울기는 벡터에 대한 미분으로 한정됩니다. 한편, 딥러닝에서는 행렬이나 텐서에 대해서도 미분을 정의하고, 그것을 기울기라 부르는 것이 일반적입니다.

1.3.3 연쇄 법칙

학습 시 신경망은 학습 데이터를 주면 손실을 출력합니다. 여기서 우리가 얻고 싶은 것은 각 매개변수에 대한 손실의 기울기입니다. 그 기울기를 얻을 수 있다면, 그것을 사용해 매개변수를 갱신할 수 있기 때문이죠. 그렇다면 신경망의 기울기는 어떻게 구할까요? 여기서 **오차역전파법** back-propagation이 등장합니다.

오차역전파법을 이해하는 열쇠는 **연쇄 법칙** chain rule입니다. 연쇄 법칙이란 합성함수에 대한 미분의 법칙이죠(합성함수란 여러 함수로 구성된 함수입니다).

연쇄 법칙을 자세히 알아봅시다. 예를 들어, $y = f(x)$ 와 $z = g(y)$ 라는 두 함수가 있습니다. 그러면 $z = g(f(x))$ 가 되어, 최종 출력 z 는 두 함수를 조합해 계산할 수 있죠. 이때 이 합성함수의 미분(x 에 대한 z 의 미분)은 다음과 같이 구할 수 있습니다.

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \quad [\text{식 1.11}]$$

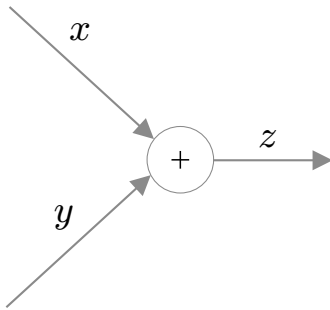
[식 1.11]이 말하듯, x 에 대한 z 의 미분은 $y = f(x)$ 의 미분과 $z = g(y)$ 의 미분을 곱하면 구해 집니다. 이것이 바로 연쇄 법칙입니다. 이 연쇄 법칙이 중요한 이유는 우리가 다루는 함수가 아무리 복잡하다 하더라도, 즉 아무리 많은 함수를 연결하더라도, 그 미분은 개별 함수의 미분들을 이용해 구할 수 있기 때문입니다. 달리 말하면, 각 함수의 국소적인 미분을 계산할 수 있다면 그 값들을 곱해서 전체의 미분을 구할 수 있습니다.

NOTE 신경망은 여러 '함수'가 연결된 것이라고 생각할 수 있습니다. 오차역전파법은 그 여러 함수(신경망)에 대해 연쇄 법칙을 효율적으로 적용하여 기울기를 구해냅니다.

1.3.4 계산 그래프

곧이어 오차역전파법을 살펴볼 텐데, 그 준비 단계로 '계산 그래프'부터 설명하겠습니다. 계산 그래프는 계산 과정을 시각적으로 보여줍니다. 먼저 [그림 1-15]처럼 아주 단순한 계산 그래프를 예로 들어 살펴봅시다.

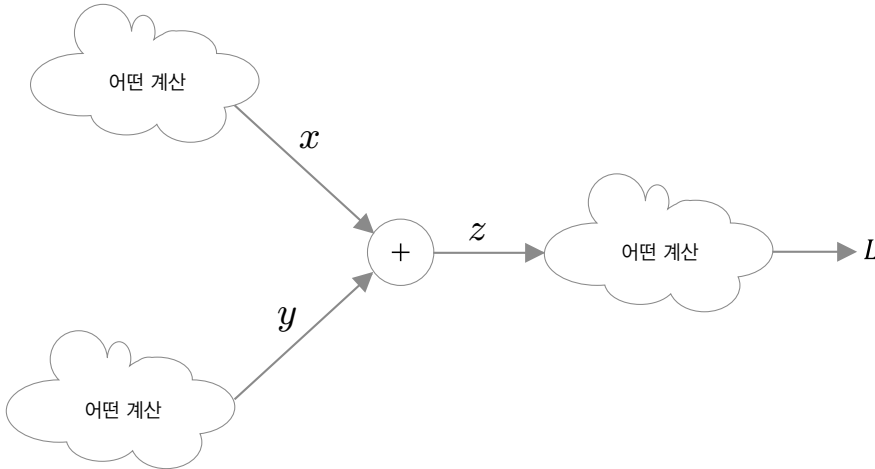
그림 1-15 $z = x + y$ 를 나타내는 계산 그래프



[그림 1-15]에서 보듯 계산 그래프는 노드와 화살표로 그림니다. 더하기를 '+' 노드로 나타냈고, 변수 x 와 y 를 해당 화살표 위에 썼습니다. 이처럼 계산 그래프에서는 연산을 노드로 나타내고 그 처리 결과가 순서대로(이 예에서는 왼쪽에서 오른쪽으로) 흐릅니다. 이것이 계산 그래프의 '순전파'입니다. 계산 그래프를 이용하면 계산을 시각적으로 파악할 수 있습니다. 게다가 그 기울기도 직관적으로 구할 수 있죠. 여기서 중요한 점은 기울기가 순전파와 반대 방향으로 전파된다는 사실인데, 이 반대 방향의 전파가 '역전파'입니다.

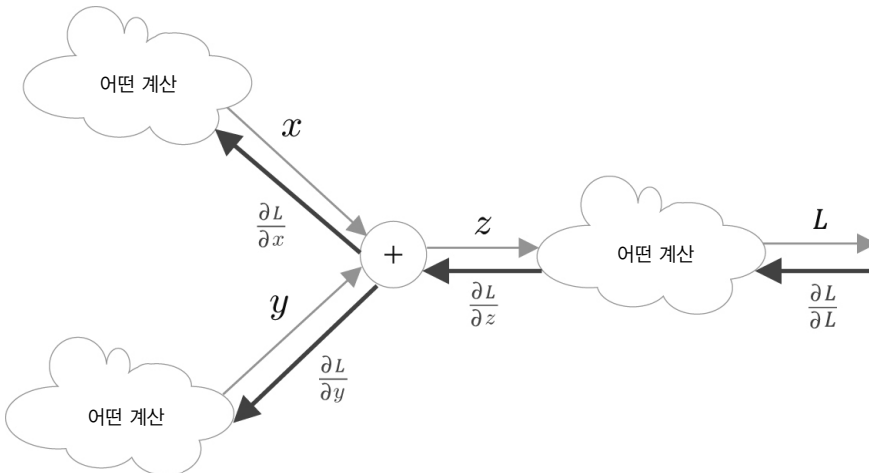
역전파를 설명하기에 앞서, 역전파가 이뤄지는 전체 그림을 더 명확하게 그려보는 게 좋겠습니다. 지금 우리는 $z = x + y$ 라는 계산을 다루고 있습니다만, 이 계산 앞뒤로도 ‘어떤 계산’이 있다고 가정합니다(그림 1-16). 그리고 최종적으로 스칼라 값인 L 이 출력된다고 가정합니다(신경망 학습에서 계산 그래프의 최종 출력은 손실이며, 그 값은 스칼라입니다).

그림 1-16 앞뒤로 추가된 노드는 ‘복잡한 전체 계산’의 일부를 구성한다.



우리 목표는 L 의 미분(기울기)을 각 변수에 대해 구하는 것입니다. 그러면 계산 그래프의 역전파는 [그림 1-17]과 같이 그릴 수 있습니다.

그림 1-17 계산 그래프의 역전파

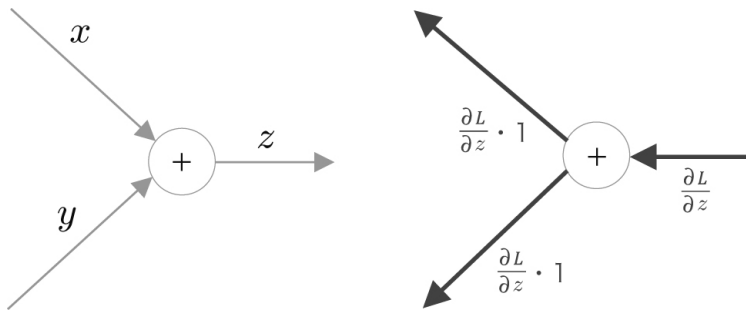


[그림 1-17]처럼 역전파는 두꺼운(붉은) 화살표로 그리고, 화살표 아래에 ‘전파되는 값’을 쓰도록 하겠습니다. 이때 ‘전파되는 값’은 최종 출력 L 의 각 변수에 대한 미분입니다. 이 예에서는 z 에 대한 미분은 $\frac{\partial L}{\partial z}$ 이고, x 와 y 에 대한 미분은 각각 $\frac{\partial L}{\partial x}$ 과 $\frac{\partial L}{\partial y}$ 입니다.

그리고 여기서 다시 연쇄 법칙이 등장합니다. 앞서 복습한 연쇄 법칙에 따르면 역전파로 흐르는 미분 값은 상류로부터 흘러온 미분과 각 연산 노드의 국소적인 미분을 곱해 계산할 수 있습니다(여기서 ‘상류’는 출력 쪽을 가리킵니다). 그러므로 이 예에서는 $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$ 이고, $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$ 가 됩니다.

그런데 우리는 지금 $z = x + y$ 의 덧셈 노드에서 이뤄지는 연산을 다루고 있습니다. 따라서 $\frac{\partial z}{\partial x} = 1$ 과 $\frac{\partial z}{\partial y} = 1$ 이라는 결과를 (해석적으로) 구할 수 있지요. 이 결과를 적용하면 덧셈 노드는 [그림 1-18]처럼 상류로부터 받은 값에 1을 곱하여 하류로 기울기를 전파합니다. 즉, 상류로부터의 기울기를 그대로 흘리기만 합니다.

그림 1-18 덧셈 노드의 순전파(왼쪽)와 역전파(오른쪽)



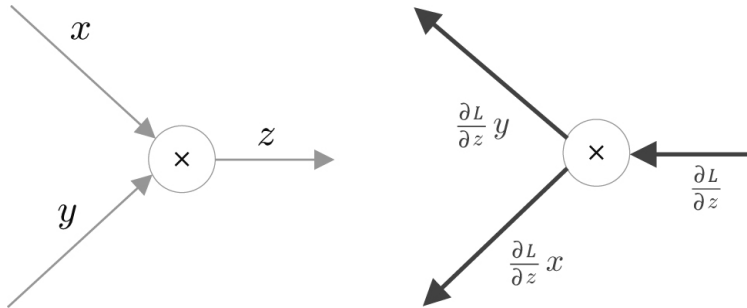
이처럼 계산 그래프는 계산을 시각적으로 보여줍니다. 그리고 역전파에 의한 기울기 흐름을 살펴봄으로써, 그 도출 과정을 이해하는 데도 도움을 주죠.

계산 그래프를 구축하는 연산 노드로는 여기서 본 ‘덧셈 노드’ 외에도 다양한 연산을 생각할 수 있습니다. 이어서 대표적인 연산 노드를 몇 가지 소개해드리겠습니다.

곱셈 노드

곱셈 노드는 $z = x \times y$ 계산을 수행합니다. 이때 $\frac{\partial z}{\partial x} = y$ 와 $\frac{\partial z}{\partial y} = x$ 라는 미분 결과를 각각 구할 수 있습니다. 따라서 곱셈 노드의 역전파는 [그림 1-19]처럼 ‘상류로부터 받은 기울기’에 ‘순전파 시의 입력을 서로 바꾼 값’을 곱합니다(즉, 순전파 시 입력이 x 면 y 를 곱하고, y 면 x 를 곱합니다).

그림 1-19 곱셈 노드의 순전파(왼쪽)와 역전파(오른쪽)

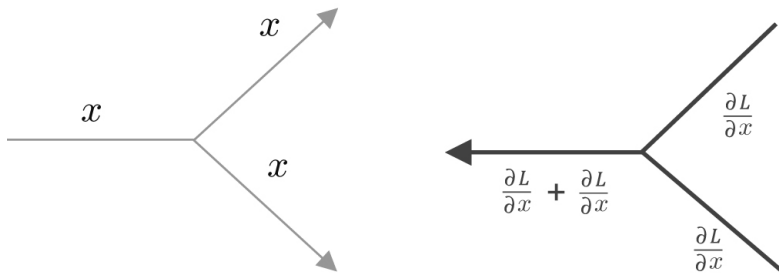


참고로, 지금까지 살펴본 덧셈 노드와 곱셈 노드 설명에서는 값이 하나짜리 데이터만 예로 들었습니다만, 벡터나 행렬 혹은 텐서 같은 다변수를 흘려도 문제없습니다. 덧셈 노드(또는 곱셈 노드)를 흐르는 데이터가 텐서라면 텐서의 각 원소를 독립적으로 계산할 뿐이죠. 다시 말해, 이 경우는 텐서의 다른 원소들과는 독립적으로, ‘원소별 연산’을 수행합니다.

분기 노드

분기 노드는 [그림 1-20]과 같이 분기하는 노드입니다.

그림 1-20 분기 노드의 순전파(왼쪽)와 역전파(오른쪽)

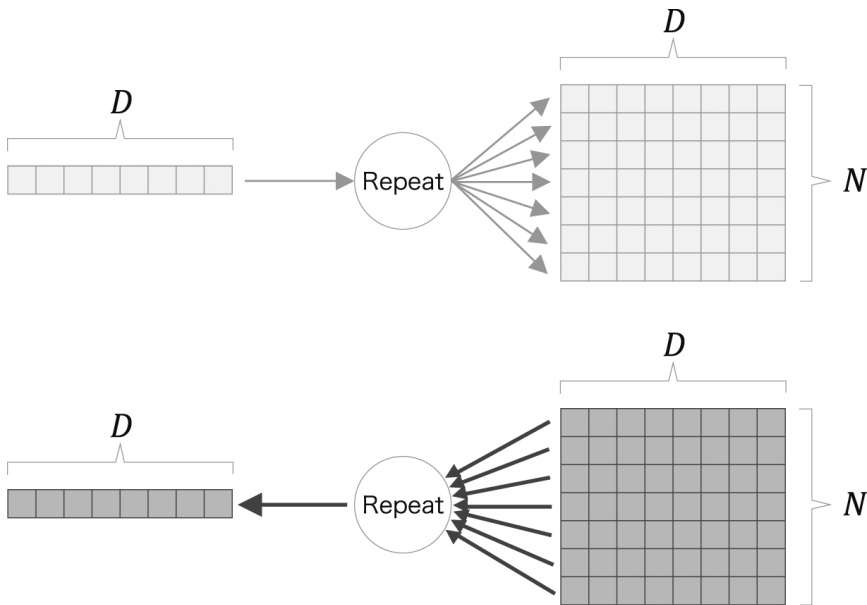


분기 노드는 따로 그리지 않고 단순히 선이 두 개로 나뉘도록 그리는데, 이때 같은 값이 복제되어 분기합니다. 따라서 분기 노드를 ‘복제 노드’라고 할 수도 있습니다. 그리고 그 역전파는 [그림 1-20]에서 보듯 상류에서 온 기울기들의 ‘합’이 됩니다.

Repeat 노드

2개로 분기하는 분기 노드를 일반화하면 N 개로의 분기(복제)가 되겠죠? 이를 Repeat 노드라고 합니다. Repeat 노드의 예를 계산 그래프로 그려봅시다.

그림 1-21 Repeat 노드의 순전파(위)와 역전파(아래)



[그림 1-21]은 길이가 D 인 배열을 N 개로 복제하는 예입니다. 이 Repeat 노드는 N 개의 분기 노드로 볼 수 있으므로, 그 역전파는 N 개의 기울기를 모두 더해 구할 수 있습니다. 코드로는 다음처럼 구현할 수 있죠.

```
>>> import numpy as np
>>> D, N = 8, 7
>>> x = np.random.randn(1, D)           # 입력
>>> y = np.repeat(x, N, axis=0)         # 순전파
```

```
>>> dy = np.random.randn(N, D)           # 무작위 기울기
>>> dx = np.sum(dy, axis=0, keepdims=True) # 역전파
```

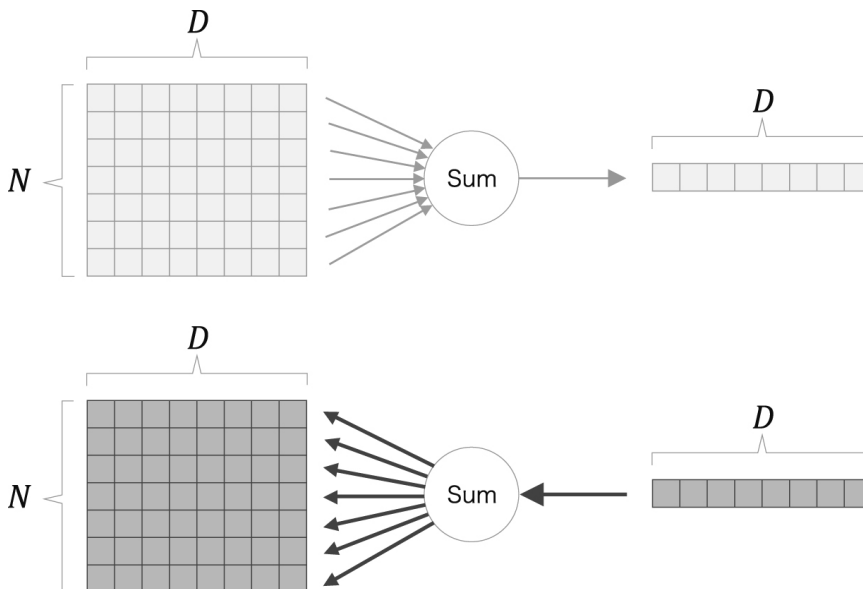
여기서 `np.repeat()` 메서드가 원소 복제를 수행합니다. 이 코드에서는 배열 `x`를 `N`번 복제하는데, 이때 `axis`를 지정하여 어느 축 방향으로 복제할지를 조정할 수 있습니다. 또, 역전파에서는 총합을 구해야 하므로 `np.sum()` 메서드를 이용합니다. 이때도 `axis` 인수를 설정하여 어느 축 방향으로 합을 구할지 지정합니다. 또한 인수로 `keepdims=True`를 설정하여 2차원 배열의 차원 수를 유지합니다. 이 예에서는 `keepdims`가 `True`면 `np.sum()`의 결과의 형상은 $(1, N)$ 이 되며, `False`면 $(N,)$ 이 됩니다.

NOTE 넘파이의 브로드캐스트는 배열의 원소를 복제하며, Repeat 노드를 사용하여 이 기능을 표현할 수 있습니다.

Sum 노드

Sum 노드는 범용 덧셈 노드입니다. 예컨대 $N \times D$ 배열에 대해 그 총합을 0축에 대해 구하는 계산을 생각해보죠. 이때 Sum 노드의 순전파(위)와 역전파(아래)는 [그림 1-22]처럼 됩니다.

그림 1-22 Sum 노드의 순전파(위)와 역전파(아래)



[그림 1-22]에서 보듯, Sum 노드의 역전파는 상류로부터의 기울기를 모든 화살표에 분배합니다. 덧셈 노드의 역전파를 자연스럽게 확장한 것이죠. 그럼 Repeat 노드와 마찬가지로 Sum 노드의 구현 예도 보여드리겠습니다.

```
>>> import numpy as np
>>> D, N = 8, 7
>>> x = np.random.randn(N, D)           # 입력
>>> y = np.sum(x, axis=0, keepdims=True) # 순전파

>>> dy = np.random.randn(1, D)          # 무작위 기울기
>>> dx = np.repeat(dy, N, axis=0)        # 역전파
```

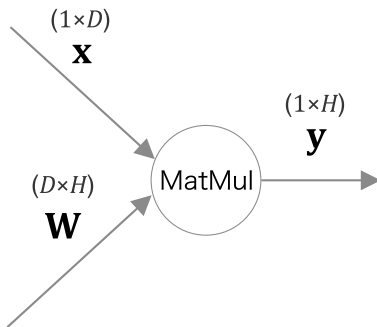
보다시피 Sum 노드의 순전파는 `np.sum()` 메서드로, 역전파는 `np.repeat()` 메서드로 구현할 수 있습니다. 여기서의 흥미로운 점은 Sum 노드와 Repeat 노드는 서로 ‘반대 관계’라는 것입니다. 반대 관계란 Sum 노드의 순전파가 Repeat 노드의 역전파가 되며, Sum 노드의 역전파가 Repeat 노드의 순전파가 된다는 뜻입니다.

MatMul 노드

이 책에서는 행렬의 곱셈을 MatMul 노드(‘Matrix Multiply’의 약자)로 표현합니다. MatMul 노드의 역전파는 다소 복잡하므로 여기에서는 일반적인 설명을 한 후에 직관적인 이해를 돕는 설명을 덧붙이겠습니다.

$y = xW$ 라는 계산을 예로 들어 MatMul 노드를 설명해보겠습니다. 여기서 x , W , y 의 형상은 각각 $1 \times D$, $D \times H$, $1 \times H$ 입니다(그림 1-23).

그림 1-23 MatMul 노드의 순전파: 각 변수 위에 형상을 표시함



이때 \mathbf{x} 의 i 번째 원소에 대한 미분 $\frac{\partial L}{\partial x_i}$ 은 다음과 같이 구합니다.

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad [\text{식 1.12}]$$

이 식의 $\frac{\partial L}{\partial x_i}$ 은 x_i 를 (조금) 변화시켰을 때 L 이 얼마나 변할 것인가라는 ‘변화의 정도’를 나타냅니다. 여기서 x_i 를 변화시키면 벡터 \mathbf{y} 의 모든 원소가 변하고, 그로 인해 최종적으로 L 이 변하게 됩니다. 따라서 x_i 에서 L 에 이르는 연쇄 법칙의 경로는 여러 개가 있으며, 그 총합은 $\frac{\partial L}{\partial x_i}$ 이 됩니다.

다시 [식 1.12]로 돌아와보죠. 이제 이 식을 쉽게 구할 수 있습니다. $\frac{\partial y_j}{\partial x_i} = w_{ij}$ 가 성립하므로, 이를 [식 1.12]에 대입하면 다음처럼 되는 것이죠.

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} w_{ij} \quad [\text{식 1.13}]$$

[식 1.13]에서 $\frac{\partial L}{\partial x_i}$ 은 ‘벡터 $\frac{\partial L}{\partial \mathbf{y}}$ ’과 ‘ \mathbf{W} 의 i 행 벡터’의 내적으로 구해짐을 알 수 있습니다. 그렇다면 이 관계로부터 다음 식을 유도할 수 있습니다.

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T \quad [\text{식 1.14}]$$

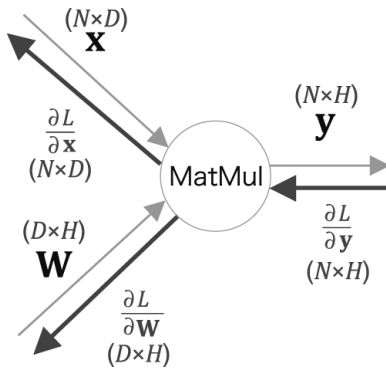
[식 1.14]에서 알 수 있듯, $\frac{\partial L}{\partial \mathbf{x}}$ 은 행렬의 곱을 사용해 단번에 구할 수 있습니다. 여기서 \mathbf{W}^T 의 \mathbf{T} 는 전치행렬이라는 뜻입니다. 그럼 [식 1.14]에 대해 ‘형상 확인’을 해보겠습니다. 그 결과는 [그림 1-24]와 같습니다.

그림 1-24 행렬 곱의 형상 확인

$$\begin{array}{l} \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T \\ \text{형상 : } \begin{array}{ccc} 1 \times D & 1 \times H & H \times D \end{array} \end{array}$$

[그림 1-24]에서 행렬의 형상이 올바른지 알 수 있습니다. 따라서 [식 1.14]가 올바른 계산임을 확인할 수 있죠. 또, 이를 역으로 취해(그 정합성이 성립되게 함으로써), 역전파의 수식(구현)을 유도할 수도 있는데, 이 방법도 $\mathbf{y} = \mathbf{x}\mathbf{W}$ 라는 행렬 곱 계산을 예로 설명해보겠습니다. 다만, 이번에는 미니배치 처리를 고려해 \mathbf{x} 에는 N 개의 데이터가 담겨 있다고 가정합니다. 그러면 \mathbf{x} , \mathbf{W} , \mathbf{y} 의 형상은 각각 $N \times D$, $D \times H$, $N \times H$ 가 되어, 역전파의 계산 그래프는 [그림 1-25]처럼 됩니다.

그림 1-25 MatMul 노드의 역전파



이제 $\frac{\partial L}{\partial \mathbf{x}}$ 은 어떻게 계산할지 생각해봅시다. 이때 $\frac{\partial L}{\partial \mathbf{x}}$ 과 관련된 변수(행렬)는 상류에서의 기울기 $\frac{\partial L}{\partial \mathbf{y}}$ 과 \mathbf{W} 입니다. 여기에서 \mathbf{W} 가 관여되는 이유는 뭘까요? 이는 곱셈의 역전파를 생각하면 이해하기 쉽습니다. 곱셈의 역전파에서는 ‘순전파 시의 입력을 서로 바꾼 값’을 사용했습니다. 이와 마찬가지로 행렬의 역전파에서도 ‘순전파 시의 입력을 서로 바꾼 행렬’을 사용하는 게 열쇠입니다. 그다음은 각 행렬의 형상에 주목하여 정합성이 유지되도록 행렬 곱을 조합합니다. 그러면 [그림 1-26]과 같이 행렬 곱의 역전파를 유도할 수 있습니다.

그림 1-26 행렬의 형상을 확인하여 역전파 식을 유도한다.

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T$$

형상: $N \times D$ $N \times H$ $H \times D$

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{x}^T \frac{\partial L}{\partial \mathbf{y}}$$

형상: $D \times H$ $D \times N$ $N \times H$

[그림 1-26]과 같이 행렬의 형상을 확인하여 행렬 곱의 역전파 식을 유도해냈습니다. 이것으로 MatMul 노드의 역전파까지 만들어봤으니, 이 노드를 하나의 계층으로 구현해봅시다 (common/layers.py).

```
class MatMul:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.x = None

    def forward(self, x):
        W, = self.params
        out = np.matmul(x, W)
        self.x = x
        return out

    def backward(self, dout):
        W, = self.params
        dx = np.matmul(dout, W.T)
        dW = np.matmul(self.x.T, dout)
        self.grads[0][...] = dW
        return dx
```

MatMul 계층은 학습하는 매개변수를 params에 보관합니다. 그리고 거기에 대응시키는 형태로, 기울기는 grads에 보관합니다. 역전파에서는 dx와 dW를 구해 가중치의 기울기를 인스턴

스 변수인 `grads`에 저장합니다.

참고로, 기울기 값을 설정하는 `grads[0][...] = dW` 코드에서 점 3개로 이뤄진 생략^{ellipsis} 기호 (...)를 사용했습니다. 이렇게 하면 넘파이 배열이 가리키는 메모리 위치를 고정시킨 다음, 그 위치에 원소들을 덮어씁니다.

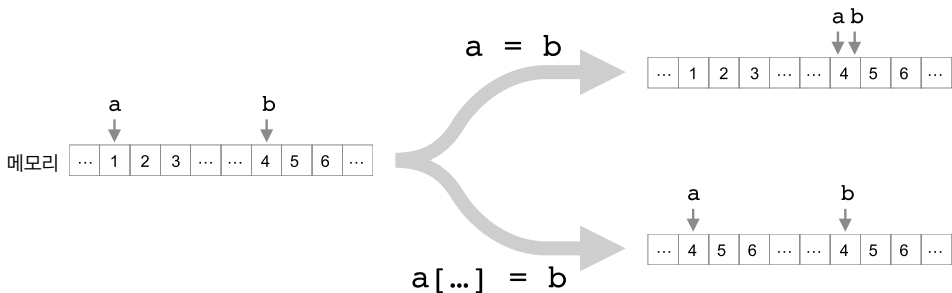
WARNING_ `grads[0] = dW`처럼 '할당'해도 되지만, '생략 기호'는 넘파이 배열의 '덮어쓰기'를 수행합니다. 결국 얇은 복사^{shallow copy}냐 깊은 복사^{deep copy}냐의 차이죠. `grads[0] = dW`처럼 그냥 할당하면 '얇은 복사'가 이뤄지고, `grads[0][...] = dW`처럼 덮어쓰면 '깊은 복사'가 이뤄집니다.

생략 기호가 등장하면서 이야기가 조금 복잡해졌으니 구체적인 예를 보며 설명하겠습니다. 여기 넘파이 배열 `a`와 `b`가 있습니다.

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
```

이 상태에서 `a = b`와 `a[...] = b` 모두 `a`에는 `[4, 5, 6]`이 할당됩니다. 그러나 두 경우에 `a`가 가리키는 메모리의 위치는 서로 다른데, 메모리를 단순화해 시각화하면 [그림 1-27]처럼 됩니다.

그림 1-27 `a = b`와 `a[...] = b`의 차이: 생략 기호는 데이터를 덮어쓰기 때문에 변수가 가리키는 메모리 위치는 변하지 않는다.



[그림 1-27]처럼 `a = b`에서는 `a`가 가리키는 메모리 위치가 `b`가 가리키는 위치와 같아집니다. 실제 데이터(4, 5, 6)는 복제되지 않는다는 뜻으로 이를 '얇은 복사'라고 합니다. 한편, `a[...] = b`일 때는 `a`의 메모리 위치는 변하지 않고, 대신 `a`가 가리키는 메모리에 `b`의 원소가 복제됩니다. 실제 데이터가 복제된다는 뜻에서 이 방식을 '깊은 복사'라고 하죠.

이상에서 '생략 기호'를 이용하여 변수의 메모리 주소를 고정할 수 있음을 알았습니다(앞의 예에서 `a`의 주소는 고정되어 있습니다). 우리의 경우 이처럼 메모리 주소를 고정함으로써 인스턴스 변수 `grads`를 다루기가 더 쉬워집니다.

NOTE `grads` 리스트에는 각 매개변수의 기울기를 저장합니다. 이때 `grads` 리스트의 각 원소는 넘파이 배열이며, 계층을 생성할 때 한 번만 생성합니다. 그 후로는 항상 '생략 기호'를 이용하므로, 이 넘파이 배열의 메모리 주소가 변하는 일 없이 항상 값을 덮어씹습니다. 이렇게 하면 기울기를 그룹화하는 작업을 최초에 한 번만 하면 된다는 이점이 생깁니다.

이상이 `MatMul` 계층의 구현이며, 이 구현 코드는 `common/layers.py`에 있으니 참고하세요.

1.3.5 기울기 도출과 역전파 구현

계산 그래프 설명도 끝났으니 이어서 실용적인 계층을 구현해보죠. 이번 절에서는 Sigmoid 계층, 완전연결계층의 Affine 계층, Softmax with Loss 계층을 구현합니다.

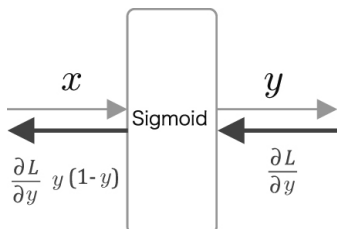
Sigmoid 계층

시그모이드 함수를 수식으로 쓰면 $y = \frac{1}{1 + \exp(-x)}$ 입니다. 그리고 그 미분은 다음과 같죠.

$$\frac{\partial y}{\partial x} = y(1 - y) \quad [\text{식 1.15}]$$

[식 1.15]로부터 Sigmoid 계층의 계산 그래프를 [그림 1-28]처럼 그릴 수 있습니다. 보다시피 출력 쪽 계층으로부터 전해진 기울기($\frac{\partial L}{\partial y}$)에 시그모이드 함수의 미분($\frac{\partial y}{\partial x}$), 즉 $y(1-y)$ 를 곱하고, 그 값을 입력 쪽 계층으로 전파합니다.

그림 1-28 Sigmoid 계층의 계산 그래프



NOTE 여기에서는 시그모이드 함수의 미분 도출 과정은 생략하겠습니다. 그 과정은 '부록 A. 시그모이드 함수와 tanh 함수의 미분'에서 계산 그래프를 사용해 자세히 설명하고 있으니 참고하세요.

그럼 Sigmoid 계층을 파이썬으로 구현해봅시다. [그림 1-28]을 토대로 다음과 같이 구현할 수 있습니다(`common/layers.py`).

```
class Sigmoid:
    def __init__(self):
        self.params, self.grads = [], []
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out
        return out

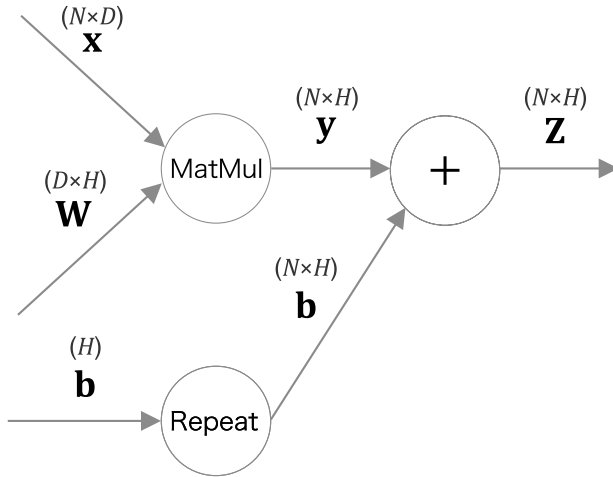
    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```

순전파 때는 출력을 인스턴스 변수 `out`에 저장하고, 역전파를 계산할 때 이 `out` 변수를 사용하는 모습을 볼 수 있습니다.

Affine 계층

앞에서와 같이 Affine 계층의 순전파는 $y = \text{np.matmul}(x, W) + b$ 로 구현할 수 있습니다. 여기서 편향을 더할 때는 넘파이의 브로드캐스트가 사용됩니다. 그 점을 명시적으로 나타내면 Affine 계층의 계산 그래프는 [그림 1-29]처럼 그릴 수 있습니다.

그림 1-29 Affine 계층의 계산 그래프



[그림 1-29]처럼 MatMul 노드로 행렬 곱을 계산합니다. 그리고 편향은 Repeat 노드에 의해 복제된 후 더해집니다(Repeat 노드가 수행하는 복제가 넘파이의 브로드캐스트 기능에 해당합니다). 다음은 Affine 계층의 파이썬 코드를 만나볼 차례군요(☞ `common/layers.py`).

```

class Affine:
    def __init__(self, W, b):
        self.params = [W, b]
        self.grads = [np.zeros_like(W), np.zeros_like(b)]
        self.x = None

    def forward(self, x):
        W, b = self.params
        out = np.matmul(x, W) + b
        self.x = x
        return out

    def backward(self, dout):
        W, b = self.params
        dx = np.matmul(dout, W.T)
        dW = np.matmul(self.x.T, dout)
        db = np.sum(dout, axis=0)

        self.grads[0][...] = dW
        self.grads[1][...] = db
        return dx

```

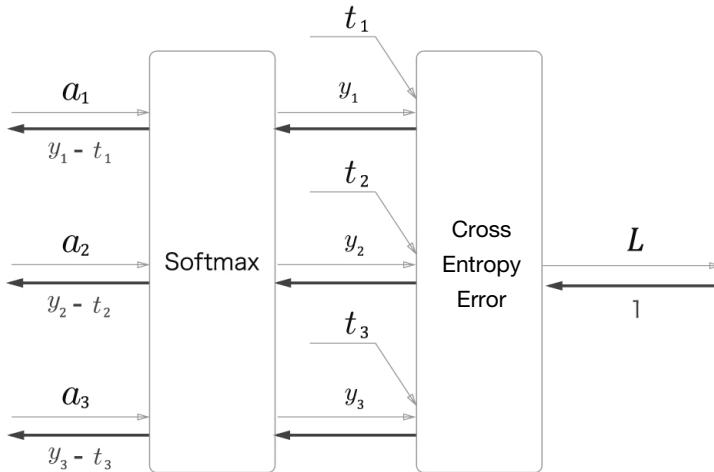
이 책의 구현 규칙에 따라 인스턴스 변수 `params`에는 매개변수를, `grads`에는 기울기를 저장합니다. Affine의 역전파는 MatMul 노드와 Repeat 노드의 역전파를 수행하면 구할 수 있습니다. Repeat 노드의 역전파는 `np.sum()` 메서드로 계산할 수 있는데, 이때 행렬의 형상을 잘 살펴보고 어느 축(axis)으로 합을 구할지를 명시해야 합니다. 마지막으로, 가중치 매개변수의 기울기를 인스턴스 변수 `grads`에 저장합니다. 이상이 Affine 계층의 구현입니다.

WARNING_ Affine 계층은 이미 구현한 MatMul 계층을 이용하면 더 쉽게 구현할 수 있습니다. 이번 절에서는 미리 구현해둔 MatMul 계층을 이용하지 않고, 넘파이의 메서드를 사용해 구현했습니다.

Softmax with Loss 계층

소프트맥스 함수와 교차 엔트로피 오차는 Softmax with Loss라는 하나의 계층으로 구현할 것입니다. [그림 1-30]은 이 계층의 계산 그래프입니다.

그림 1-30 Softmax with Loss 계층의 계산 그래프



이 계산 그래프에서는 소프트맥스 함수는 Softmax 계층으로, 교차 엔트로피 오차는 Cross Entropy Error 계층으로 표기했습니다. 그리고 3-클래스 분류를 가정하여 이전 계층(입력층에 가까운 계층)으로부터 3개의 입력을 받도록 했습니다.

[그림 1-30]처럼 Softmax 계층은 입력 (a_1, a_2, a_3) 를 정규화하여 (y_1, y_2, y_3) 를 출력합니다.

그리고 Cross Entropy Error 계층은 Softmax의 출력 (y_1, y_2, y_3)와 정답 레이블 (t_1, t_2, t_3)를 받고, 이 데이터로부터 손실 L 을 구해 출력합니다.

NOTE [그림 1-30]에서 주목할 부분은 역전파의 결과입니다. Softmax 계층의 역전파는 ($y_1 - t_1, y_2 - t_2, y_3 - t_3$)로 깔끔하게 떨어집니다. (y_1, y_2, y_3)는 Softmax 계층의 출력이고, (t_1, t_2, t_3)는 정답 레이블이므로, Softmax 계층의 역전파는 자신의 출력과 정답 레이블의 차이라는 뜻이죠. 이처럼 신경망의 역전파는 이 차이(오차)를 앞 계층에 전해주는 것으로, 신경망 학습에서 아주 중요한 성질입니다.

Softmax with Loss 계층의 구현에 관한 설명은 생략하겠습니다. 구현 코드는 common/layers.py에 있고, Softmax with Loss 계층의 역전파 유도 과정은 『밑바닥부터 시작하는 딥러닝』의 ‘부록 A. Softmax-with-Loss 계층의 계산 그래프’에서 자세히 설명했으니 관심 있는 분은 참고하세요.

1.3.6 가중치 갱신

오차역전파법으로 기울기를 구했으면, 그 기울기를 사용해 신경망의 매개변수를 갱신합니다. 이때 신경망의 학습은 다음 순서로 수행합니다.

- 1단계: 미니배치

훈련 데이터 중에서 무작위로 다수의 데이터를 골라낸다.

- 2단계: 기울기 계산

오차역전파법으로 각 가중치 매개변수에 대한 손실 함수의 기울기를 구한다.

- 3단계: 매개변수 갱신

기울기를 사용하여 가중치 매개변수를 갱신한다.

- 4단계: 반복

1~3단계를 필요한 만큼 반복한다.

이러한 단계를 거쳐 신경망 학습이 이뤄집니다. 우선 미니배치에서 데이터를 선택하고, 이어서 오차역전파법으로 가중치의 기울기를 얻습니다. 이 기울기는 현재의 가중치 매개변수에서 손실을 가장 크게 하는 방향을 가리킵니다. 따라서 매개변수를 그 기울기와 반대 방향으로 갱신하면 손실을 줄일 수 있습니다. 이것이 바로 **경사하강법** Gradient Descent입니다. 그런 다음 이상의 작업을 필요한 만큼 반복합니다.

3단계에서 수행하는 가중치 갱신 기법의 종류는 아주 다양한데, 여기에서는 그중 가장 단순한 **확률적경사하강법** Stochastic Gradient Descent (**SGD**)을 구현하겠습니다. 참고로, ‘확률적 Stochastic’은 무작위로 선택된 데이터(미니배치)에 대한 기울기를 이용한다는 뜻입니다.

SGD는 단순한 방법입니다. SGD는 (현재의) 가중치를 기울기 방향으로 일정한 거리만큼 갱신합니다. 수식으로는 다음과 같습니다.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}} \quad \text{[식 1.16]}$$

이 식에서 갱신하는 가중치 매개변수가 \mathbf{W} 이고, \mathbf{W} 에 대한 손실 함수의 기울기가 $\frac{\partial L}{\partial \mathbf{W}}$ 입니다. η 에타는 학습률 learning rate을 나타내며, 실제로는 0.01이나 0.001 같은 값을 미리 정해 사용합니다.

그럼 SGD를 파이썬으로 구현해볼까요? 여기에서는 모듈화를 고려하여 매개변수를 갱신할 클래스를 common/optimizer.py에 구현해놓겠습니다(이 파일에는 SGD 외에도 AdaGrad와 Adam 등의 구현도 들어 있습니다).

그리고 매개변수를 갱신하는 클래스는 update(params, grads)라는 공통 메서드를 갖도록 구현합니다. 이 메서드의 인수 params에는 신경망의 가중치가, grads에는 기울기가 각각 리스트로 저장되어 있어야 합니다. 그리고 params와 grads 리스트에는 대응하는 매개변수와 기울기가 같은 위치(인덱스)에 저장되어 있다고 가정합니다. 그러면 SGD는 다음과 같이 구현할 수 있습니다(☞ common/optimizer.py).

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for i in range(len(params)):
            params[i] -= self.lr * grads[i]
```

초기화 인수 lr은 학습률을 뜻하며, 그 값을 인스턴스 변수로 저장해둡니다. 그리고 update(params, grads) 메서드는 매개변수 갱신을 처리합니다.

이 SGD 클래스를 사용하면 신경망의 매개변수 갱신을 다음처럼 할 수 있습니다(실제로는 동작하지 않는 의사 코드입니다).

```
model = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # 미니배치 획득
    loss = model.forward(x_batch, t_batch)
    model.backward()
    optimizer.update(model.params, model.grads)
    ...
```

이처럼 최적화를 수행하는 클래스를 분리해 구현함으로써 기능을 쉽게 모듈화할 수 있게 했습니다. 이 책에서는 SGD 외에도 Momentum, AdaGrad, Adam 등의 기법을 구현하여 `common/optimizer.py`에 모아놴습니다. 이 최적화 기법들 각각에 관한 자세한 설명은 『밑바닥부터 시작하는 딥러닝』의 ‘6.1 매개변수 갱신’ 절을 참고하세요.

1.4 신경망으로 문제를 풀다

드디어 준비는 끝! 지금부터 간단한 데이터셋으로 신경망을 학습시켜 보겠습니다.

1.4.1 스파이럴 데이터셋

이 책에서는 데이터셋을 다루는 편의 클래스 몇 개를 `dataset` 디렉터리에 준비해놴습니다. 이번 절에서는 그중 `dataset/spiral.py` 파일을 이용합니다. 이 파일에는 ‘스파이럴^{spiral}; 나선형의 데이터’를 읽어 들이는 클래스가 구현되어 있으며, 다음과 같이 사용합니다(`ch01/show_spiral_dataset.py`).

```
import sys
sys.path.append('.') # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from dataset import spiral
```

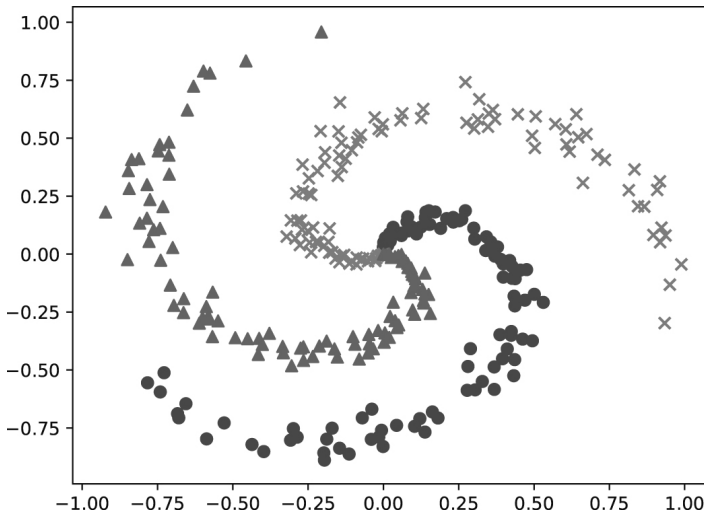
```
import matplotlib.pyplot as plt

x, t = spiral.load_data()
print('x', x.shape) # (300, 2)
print('t', t.shape) # (300, 3)
```

이 예에서는 ch01 디렉터리에서 dataset 디렉터리에 있는 spiral.py를 импортimport하여 이용합니다. 그래서 부모 디렉터리를 임포트의 검색 경로에 추가해야 하는데, 두 번째 줄의 sys.path.append('.')가 그 일을 수행합니다.

그리고 spiral.load_data()가 데이터를 읽어 옵니다. 이때 x가 입력 데이터이고, t가 정답 레이블입니다. x와 t의 형상을 출력해보면 각각 300개의 샘플 데이터를 담고 있으며, x는 2차원 데이터이고 t는 3차원 데이터임을 알 수 있습니다. 참고로 t는 원핫 벡터로, 정답에 해당하는 클래스에는 1이, 그 외에는 0이 레이블되어 있습니다. 그러면 이 데이터가 어떤 모습인지 그래프로 그려봅시다(그림 1-31).

그림 1-31 학습에 이용할 스파이럴 데이터셋(3개의 클래스 각각을 X, ▲, ●로 표기)



[그림 1-31]처럼 입력은 2차원 데이터이고, 분류할 클래스 수는 3개가 있습니다. 이 그래프를 보면 직선만으로는 클래스들을 분리할 수 없음을 알 수 있습니다. 따라서 비선형 분리를 학습해야 합니다. (비선형인 시그모이드 함수를 활성화 함수로 사용하는 은닉층이 있는) 우리의 신경망은 이 비선형 패턴을 올바르게 학습할 수 있을까요? 서둘러 실험해봅시다.

WARNING 실전에서는 데이터셋을 훈련용과 테스트용 (그리고 검증용) 데이터로 분리하여 학습과 평가를 수행하지만, 여기에서는 실험을 간단히 하기 위해 이 작업은 생략했습니다.

1.4.2 신경망 구현

그러면 신경망을 구현해보죠. 이번 절에서는 은닉층이 하나인 신경망을 구현합니다. 임포트 문과 초기화를 담당하는 `__init__` 메서드부터 보겠습니다(☞ `ch01/two_layer_net.py`).

```
import sys
sys.path.append('.')
import numpy as np
from common.layers import Affine, Sigmoid, SoftmaxWithLoss

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 가중치와 편향 초기화
        W1 = 0.01 * np.random.randn(I, H)
        b1 = np.zeros(H)
        W2 = 0.01 * np.random.randn(H, O)
        b2 = np.zeros(O)

        # 계층 생성
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]
        self.loss_layer = SoftmaxWithLoss()

        # 모든 가중치와 기울기를 리스트에 모은다.
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads
```

초기화 메서드는 3개의 인수를 받는데, 차례대로 input_size는 입력층의 뉴런 수, hidden_size는 은닉층의 뉴런 수, output_size는 출력층의 뉴런 수입니다. 메서드 안에서는 우선 편향을 영벡터(zero vector)로 초기화하고(np.zeros()), 가중치는 작은 무작위 값으로 초기화합니다(0.01 * np.random.randn()). 참고로 가중치를 작은 무작위 값으로 설정하면 학습이 잘 진행될 가능성이 커집니다. 계속해서 필요한 계층을 생성해 인스턴스 변수인 layers 리스트에 모아두고, 마지막으로 이 모델에서 사용하는 매개변수들과 기울기들을 각각 하나로 모읍니다.

WARNING Softmax with Loss 계층은 다른 계층과 다르게 취급하여, layers 리스트가 아닌 loss_layer 인스턴스 변수에 별도로 저장합니다.

이어서 TwoLayerNet에 3개의 메서드를 구현해 넣습니다. 추론을 수행하는 predict() 메서드, 순전파를 담당하는 forward() 메서드, 역전파를 담당하는 backward() 메서드입니다 (☞ ch01/two_layer_net.py).

```
def predict(self, x):
    for layer in self.layers:
        x = layer.forward(x)
    return x

def forward(self, x, t):
    score = self.predict(x)
    loss = self.loss_layer.forward(score, t)
    return loss

def backward(self, dout=1):
    dout = self.loss_layer.backward(dout)
    for layer in reversed(self.layers):
        dout = layer.backward(dout)
    return dout
```

보다시피 이번 구현은 이전보다 깔끔합니다! 신경망에서 사용하는 처리 블록들을 ‘계층’ 단위로 미리 구현해놨으므로, 여기에서는 그 계층들의 forward()와 backward()를 적절한 순서로 호출만 하면 되기 때문이죠.

1.4.3 학습용 코드

이어서 학습을 수행하는 코드를 보겠습니다. 여기에서는 학습 데이터를 읽어 들여 신경망(모델)과 옵티마이저(최적화기)를 생성합니다. 그리고 앞 절에서 본 학습의 네 단계의 절차대로 학습을 수행합니다. 참고로 머신러닝 분야에서는 문제를 풀기 위해서 설계한 기법(신경망이나 SVM 서포트 벡터 마신 등)을 가리켜 보통 ‘모델’이라고 부릅니다. 학습용 코드는 다음과 같습니다(☞ ch01/train_custom_loop.py).

```
import sys
sys.path.append('.')
import numpy as np
from common.optimizer import SGD
from dataset import spiral
import matplotlib.pyplot as plt
from two_layer_net import TwoLayerNet

# ❶ 하이퍼파라미터 설정
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

# ❷ 데이터 읽기, 모델과 옵티마이저 생성
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

# 학습에 사용하는 변수
data_size = len(x)
max_iters = data_size // batch_size
total_loss = 0
loss_count = 0
loss_list = []

for epoch in range(max_epoch):
    # ❸ 데이터 뒤섞기
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    for iters in range(max_iters):
```

```

batch_x = x[itters*batch_size:(itters+1)*batch_size]
batch_t = t[itters*batch_size:(itters+1)*batch_size]

# ④ 기울기를 구해 매개변수 갱신
loss = model.forward(batch_x, batch_t)
model.backward()
optimizer.update(model.params, model.grads)

total_loss += loss
loss_count += 1

# ⑤ 정기적으로 학습 경과 출력
if (itters+1) % 10 == 0:
    avg_loss = total_loss / loss_count
    print('| 에폭 %d | 반복 %d / %d | 손실 %.2f'
          % (epoch + 1, iters + 1, max_iters, avg_loss))
    loss_list.append(avg_loss)
    total_loss, loss_count = 0, 0

```

① 우선 하이퍼파라미터(hyperparameter)를 설정합니다. 구체적으로는 학습하는 에폭 수, 미니배치 크기, 은닉층의 뉴런 수, 학습률을 설정합니다. ② 계속해서 데이터를 읽어 들이고, 신경망(모델)과 옵티마이저를 생성합니다. 우리는 이미 2층 신경망을 TwoLayerNet 클래스로, 또 옵티마이저를 SGD 클래스로 구현해봤으니, 여기에서는 이 클래스들을 이용하겠습니다.

NOTE 에폭(epoch)은 학습 단위입니다. 1에폭은 학습 데이터를 모두 '살펴본' 시점(데이터셋을 1바퀴 돌아 본 시점)을 뜻하죠. 이 코드에서는 300에폭을 학습합니다.

학습은 미니배치 방식으로 진행되며 데이터를 무작위로 선택합니다. ③ 여기에서는 에폭 단위로 데이터를 뒤섞고, 뒤섞은 데이터 중 앞에서부터 순서대로 뽑아내는 방식을 사용했습니다. 데이터 뒤섞기(정확하게는 데이터의 '인덱스' 뒤섞기)에는 np.random.permutation() 메서드를 사용합니다. 이 메서드에 인수로 N 을 주면, 0에서 $N-1$ 까지의 무작위 순서를 생성해 반환합니다. 실제 사용 예는 다음과 같습니다.

```

>>> import numpy as np
>>> np.random.permutation(10)
array([7, 6, 8, 3, 5, 0, 4, 1, 9, 2])

```

```
>>> np.random.permutation(10)
array([1, 5, 7, 3, 9, 2, 8, 6, 0, 4])
```

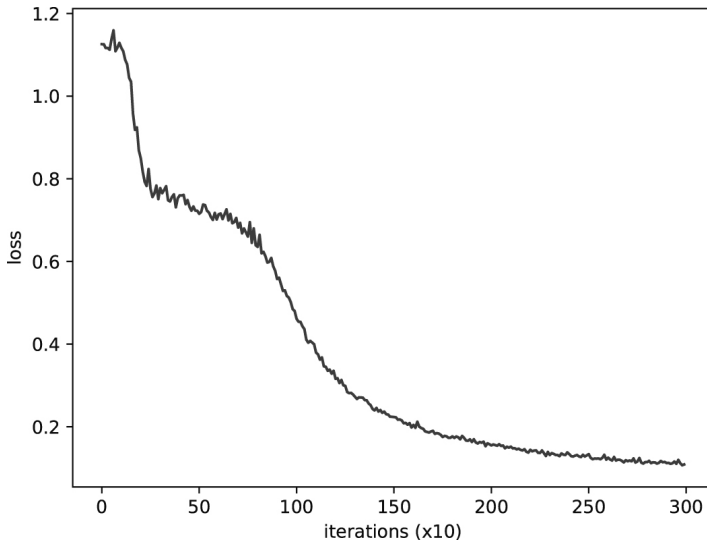
이처럼 `np.random.permutation()`을 호출하면 데이터 인덱스를 무작위로 뒤섞을 수 있습니다.

④ 계속해서 기울기를 구해 매개변수를 갱신합니다. ⑤ 마지막으로, 정기적으로 학습 결과를 출력합니다. 이 코드에서는 10번째 반복마다 손실의 평균을 구해 `loss_list` 변수에 추가했습니다. 이상으로 학습을 수행하는 코드를 살펴봤습니다.

WARNING_ 여기서 구현한 신경망의 학습 코드는 이 책의 다른 장소에서도 사용합니다. 그래서 이 코드를 `Trainer` 클래스로 만들어뒀습니다. 신경망 학습의 상세 내용을 이 클래스 안으로 밀어넣은 것이죠. 자세한 사용법은 '1.4.4. `Trainer` 클래스' 절에서 설명합니다.

이제 이 코드(`ch01/train_custom_loop.py`)를 실행해보세요. 그러면 터미널에 출력되는 손실 값이 순조롭게 낮아지는 것을 알 수 있습니다. 그리고 그 결과를 그래프로 그린 것이 바로 [그림 1-32]입니다.

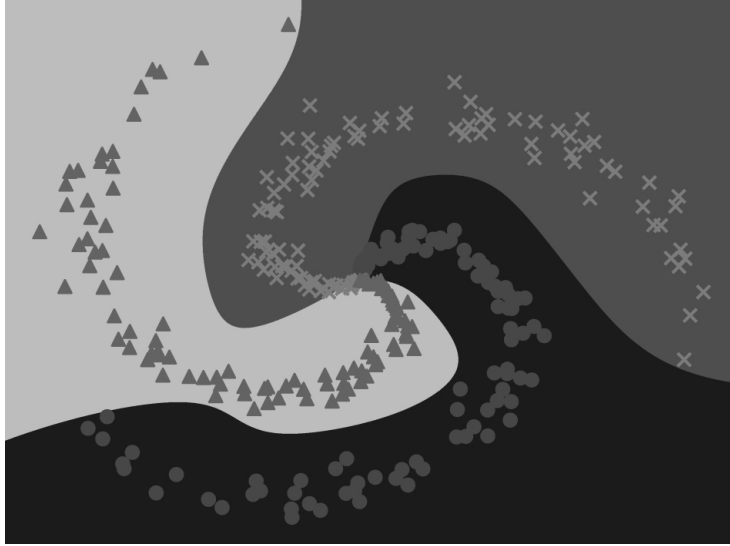
그림 1-32 손실 그래프: 가로축은 학습의 반복 수(눈금 값의 10배), 세로축은 학습 10번 반복당 손실 평균



[그림 1-32]에서 보듯, 학습을 진행함에 따라 손실이 줄어들고 있습니다. 우리 신경망이 올바른

방향으로 학습되고 있는 것이죠! 그럼, 학습 후 신경망이 영역을 어떻게 분리했는지 시각화해 봅시다(이를 **결정 경계**^{decision boundary}라고 합니다). 결과는 [그림 1-33]과 같습니다.

그림 1-33 학습 후 신경망의 결정 경계(신경망이 식별하는 클래스별 영역을 색으로 구분)



[그림 1-33]에서 보듯 학습된 신경망은 ‘나선형’ 패턴을 올바르게 파악했음을 알 수 있습니다. 즉, 비선형 분리 영역을 학습할 수 있었습니다! 이처럼 신경망에 은닉층을 추가하면 더 복잡한 표현이 가능해집니다. 층을 더 깊게 쌓으면 표현력 또한 더 풍부해지는 것이 딥러닝의 특징이죠.

1.4.4 Trainer 클래스

앞에서 언급했듯이, 이 책은 신경망 학습을 수행할 기회를 많이 드립니다. 다시 말해, 앞 절에서 본 학습 코드가 자주 필요한데, 매번 똑같은 코드를 다시 쓰고 있으면 아주 지루하겠죠. 그래서 이 책에서는 학습을 수행하는 역할을 Trainer라는 클래스로 제공합니다. 내용은 앞 절의 소스 코드와 거의 같습니다. 새로운 기능도 조금 추가했는데, 자세한 사용법은 필요할 때 설명하겠습니다.

Trainer 클래스는 common/trainer.py에 있습니다. 이 클래스의 초기화 메서드는 신경망(모델)과 옵티마이저를 인수로 받습니다. 구체적으로는 다음과 같이 사용합니다.

```

model = TwoLayerNet(...)
optimizer = SGD(lr=1.0)
trainer = Trainer(model, optimizer)

```

그리고 `fit()` 메서드를 호출해 학습을 시작합니다. 이 `fit()` 메서드가 받는 인수는 [표 1-1]에 정리했습니다.

표 1-1 Trainer 클래스의 `fit()` 메서드가 받는 인수: '(=XX)'는 기본값을 뜻함

인수	설명
<code>x</code>	입력 데이터
<code>t</code>	정답 레이블
<code>max_epoch (=10)</code>	학습을 수행하는 에폭 수
<code>batch_size (=32)</code>	미니배치 크기
<code>eval_interval (=20)</code>	결과(평균 손실 등)를 출력하는 간격 예컨대 <code>eval_interval=20</code> 으로 설정하면, 20번째 반복마다 손실의 평균을 구해 화면에 출력한다.
<code>max_grad (=None)</code>	가울기 최대 노름 ^{norm} 가울기 노름이 이 값을 넘어서면 가울기를 줄인다(이를 가울기 클리핑이라 하며, 자세한 설명은 '5장. 순환 신경망(RNN)' 참고).

Trainer 클래스는 `plot()` 메서드도 제공합니다. 이 메서드는 `fit()`에서 기록한 손실(정확하게는 `eval_interval` 시점에 평가된 평균 손실)을 그래프로 그려줍니다. 이제 Trainer 클래스를 사용해 학습을 수행하는 코드를 보시죠(☞ `ch01/train.py`).

```

import sys
sys.path.append('.')
from common.optimizer import SGD
from common.trainer import Trainer
from dataset import spiral
from two_layer_net import TwoLayerNet

max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

```

```
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

trainer = Trainer(model, optimizer)
trainer.fit(x, t, max_epoch, batch_size, eval_interval=10)
trainer.plot()
```

이 코드를 실행하면 이전 결과 같은 신경망 학습이 이뤄집니다. 앞에서 본 학습용 코드를 Trainer 클래스에 맡겼기 때문에 코드가 깔끔해졌죠. 이 책에서는 앞으로 학습을 해야 할 때면 항상 Trainer 클래스를 사용할 겁니다.

1.5 계산 가속화

신경망의 학습과 추론에 드는 연산량은 상당합니다. 그래서 신경망에서는 얼마나 빠르게 계산하느냐가 매우 중요한 주제죠. 그래서 이번 절에서는 신경망 가속화에 도움되는 ‘비트 정밀도’와 ‘GPU’에 관해 가볍게 설명해보겠습니다.

NOTE 이 책은 속도보다는 알기 쉽게 구현하기에 우선순위를 두었습니다. 다만, 가속화의 관점에서 앞으로는 데이터의 비트 정밀도를 의식해 구현합니다. 또한, 계산이 오래 걸리는 부분에서는 (선택적으로) GPU로 실행할 수 있도록 준비해뒀습니다.

1.5.1 비트 정밀도

넘파이의 부동소수점 수는 기본적으로 64비트 데이터 타입을 사용합니다(독자의 환경, 즉 OS나 파이썬/넘파이 버전 등에 따라 바뀔 수 있습니다). 실제로 64비트 부동소수점 수가 사용되지는 다음 코드로 확인할 수 있습니다.

```
>>> import numpy as np
>>> a = np.random.randn(3)
>>> a.dtype
dtype('float64')
```

이처럼 넘파이 배열의 인스턴스 변수 dtype을 출력해 데이터 타입을 알아볼 수 있습니다. 바로 앞 결과가 출력한 float64는 64비트 부동소수점 수라는 뜻입니다.

넘파이는 64비트 부동소수점 수를 표준으로 사용합니다. 그러나 신경망의 추론과 학습은 32비트 부동소수점 수로도 문제없이(인식률을 거의 떨어뜨리는 일 없이) 수행할 수 있다고 합니다. 32비트는 64비트의 절반이므로, 메모리 관점에서는 항상 32비트가 더 좋다고 말할 수 있습니다. 또, 신경망 계산 시 데이터를 전송하는 ‘버스 대역폭’⁶이 병목이 되는 경우가 왕왕 있습니다. 이런 경우에도 데이터 타입이 작은 게 유리하죠. 마지막으로 계산 속도 측면에서도 32비트 부동소수점 수가 일반적으로 더 빠릅니다(CPU나 GPU 아키텍처에 따라 다릅니다).

이런 이유로 이 책에서는 32비트 부동소수점 수를 우선으로 사용합니다. 넘파이에서 32비트 부동소수점 수를 사용하려면 다음과 같이 데이터 타입을 np.float32나 'f'로 지정합니다.

```
>>> b = np.random.randn(3).astype(np.float32)
>>> b.dtype
dtype('float32')

>>> c = np.random.randn(3).astype('f')
>>> c.dtype
dtype('float32')
```

또한 신경망 추론으로 한정하면, 16비트 부동소수점 수를 사용해도 인식률이 거의 떨어지지 않습니다.⁶ 그리고 넘파이에도 16비트 부동소수점 수가 준비되어 있죠. 다만, 일반적으로 CPU와 GPU는 연산 자체를 32비트로 수행합니다. 따라서 16비트 부동소수점 수로 변환하더라도 계산 자체는 32비트로 이뤄져서 처리 속도 측면에서는 혜택이 없을 수도 있습니다.

그러나 학습된 가중치를 (파일에) 저장할 때는 16비트 부동소수점 수가 여전히 유효합니다. 가중치 데이터를 16비트로 저장하면 32비트를 쓸 때보다 절반의 용량만 사용하니까요. 그래서 이 책에서는 학습된 가중치를 저장하는 경우에 한해 16비트 부동소수점 수로 변환하겠습니다.

NOTE_ 딥러닝이 주목받으면서, 최근 GPU들은 ‘저장’과 ‘연산’ 모두에 16비트 반정밀도 부동소수점 수를 지원하도록 진화했습니다. 구글에서 개발한 TPU 칩은 8비트 계산도 지원합니다.⁷*

* 옮긴이_ GPU들 역시 8비트 계산을 지원하기 시작했습니다.

1.5.2 GPU(쿠파이)

딥러닝의 계산은 대량의 곱하기 연산으로 구성됩니다. 이 대량의 곱하기 연산 대부분은 병렬로 계산할 수 있는데, 바로 이 점에서는 CPU보다 GPU가 유리합니다. 대부분의 딥러닝 프레임워크가 CPU뿐 아니라 GPU도 지원하는 이유가 바로 이것이죠.

이 책의 예제 중에는 쿠파이¹⁾라는 파이썬 라이브러리를 사용할 수 있는 게 있습니다. 쿠파이는 GPU를 이용해 병렬 계산을 수행해주는 라이브러리인데, 아쉽게도 엔비디아의 GPU에서만 동작합니다. 또한, CUDA라는 GPU 전용 범용 병렬 컴퓨팅·플랫폼을 설치해야 합니다. 자세한 설치 방법은 쿠파이 공식 설치 가이드²⁾를 참고하세요.

쿠파이를 사용하면 엔비디아 GPU를 사용해 간단하게 병렬 계산을 수행할 수 있습니다. 더욱 중요한 점은 쿠파이는 넘파이와 호환되는 API를 제공한다는 사실입니다. 여기 간단한 사용 예를 준비했습니다.

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([ 3., 12.], dtype=float32)
```

이와 같이 쿠파이의 사용법은 기본적으로 넘파이와 같습니다. 사용법은 같지만 뒤에서 열심히 GPU를 사용해 계산하는 것이죠. 다시 말해, 넘파이로 작성한 코드를 ‘GPU용’으로 변경하기가 아주 쉽다는 뜻입니다. 그저 (보통은) numpy를 cupy로 대체해주기만 하면 끝이죠!

WARNING_ 2018년 6월 현재, 쿠파이가 넘파이의 모든 메서드를 지원하는 건 아닙니다. 쿠파이는 넘파이와 100% 호환되지는 않지만, 공통된 API를 많이 제공합니다.

다시 이야기합니다만, 이 책에서는 이해하기 쉽게 구현하는 걸 우선하므로 기본적으로는 CPU에서 수행되는 코드로 작성합니다. 그러나 계산이 오래 걸리는 코드는 선택적으로 쿠파이를 사용한 구현도 제공합니다. 그리고 쿠파일을 사용하는 경우라도, 독자는 쿠파일을 사용한다는 사실을 의식하지 않아도 되도록 배려했습니다.

이 책에서 GPU를 지원하는 코드가 처음 등장하는 것은 4장의 ch04/train.py 파일입니다. 이 파일은 다음의 импорт 문으로 시작합니다.

```
import sys
sys.path.append('.')
import numpy as np
from common import config
# GPU에서 실행하려면 아래 주석을 해제하세요(쿠파이 필요).
# =====
# config.GPU = True
# =====
...
```

이 코드를 CPU에서 실행하면 몇 시간이 걸리지만, GPU를 사용하면 수십 분 정도로 단축됩니다. 이 코드에서 “# config.GPU = True”의 주석을 풀기만 하면 넘파이 대신 쿠파이가 사용되고, 쿠파이가 계산을 GPU에 위임하여 학습이 빠르게 이뤄집니다. 호환되는 GPU를 가지고 계신 분은 꼭 이용해봅시다.

그리고 이 책의 예제 중 GPU를 지원하는 코드는 모두 앞의 코드처럼 단 1줄만 수정하면 GPU 모드로 실행할 수 있습니다.

NOTE_ 넘파이가 쿠파이로 바뀌는 구조는 매우 단순합니다. 관심 있는 분은 common/config.py나 common/hp.py, common/layers.py의 импорт 문을 참고하세요.

1.6 정리

이번 장에서는 신경망의 기본을 복습했습니다. 벡터와 행렬 같은 수학 개념부터 시작해서 파이썬(특히 넘파이)의 기본적인 사용법을 확인했습니다. 그런 다음 신경망의 구조를 살펴봤습니다. 특히 계산 그래프의 기본 부품(덧셈 노드와 곱셈 노드 등)을 몇 개 사용하여 순전파와 역전파를 설명했습니다.

또한 간단한 신경망도 구현해봤습니다. 모듈화를 고려해 신경망의 기본 구성요소를 계층으로 구현했습니다. 계층을 구현할 때는 모든 클래스가 forward()와 backward() 메서드를 제공

하고, params와 grads라는 인스턴스 변수를 갖는다는 ‘구현 규칙’을 따랐습니다. 이 규칙 덕분에 앞으로의 신경망 구현이 훨씬 쉬워집니다.

마지막으로 인공적인 ‘스파이럴 데이터셋’을 활용해 은닉층이 한 개인 신경망을 학습시키고, 그 모델이 올바르게 학습되었음을 확인했습니다.

이것으로 신경망 복습을 마쳤습니다. 이제부터 신경망이라는 믿음직한 무기를 손에 쥐고, 자연어 처리의 세계로 뛰어들겠습니다. 그럼 힘차게 전진합시다!

이번 장에서 배운 내용

- 신경망은 입력층, 은닉층(중간층), 출력층을 지닌다.
- 완전연결계층에 의해 선형 변환이 이뤄지고, 활성화 함수에 의해 비선형 변환이 이뤄진다.
- 완전연결계층이나 미니배치 처리는 행렬로 모아 한꺼번에 계산할 수 있다.
- 오차역전파법을 사용하여 신경망의 손실에 관한 기울기를 효율적으로 구할 수 있다.
- 신경망이 수행하는 처리는 계산 그래프로 시각화할 수 있으며, 순전파와 역전파를 이해하는 데 도움이 된다.
- 신경망의 구성요소들을 ‘계층’으로 모듈화해두면, 이를 조립하여 신경망을 쉽게 구성할 수 있다.
- 신경망 고속화에는 GPU를 이용한 병렬 계산과 데이터의 비트 정밀도가 중요하다.

자연어와 단어의 분산 표현

마티: “이건 심각한데요^{heavy}.”

박사: “미래에는 이 정도도 무겁단^{heavy} 말이야?”

— 영화 <백 투 더 퓨처>*

드디어 자연어 처리의 세계로 첫걸음을 내딛게 됩니다. 자연어 처리가 다루는 분야는 다양하지만, 그 본질적 문제는 컴퓨터가 우리의 말을 알아듣게(이해하게) 만드는 것입니다. 이번 장은 컴퓨터에 말을 이해시킨다는 것이 무슨 뜻인지, 그리고 어떤 방법들이 존재하는지를 중심으로 이야기를 풀어보겠습니다. 특히 고전적인 기법(딥러닝 등장 이전의 기법)들을 자세히 살펴볼려 합니다. 딥러닝(정확히는 신경망) 기반 기법들은 잠시 뒤, 다음 장에서 소개하겠습니다.

또한 이번 장에서는 파이썬으로 텍스트를 다루는 연습도 겸합니다. 텍스트를 단어로 분할하는 처리나 단어를 단어 ID로 변환하는 처리 등을 구현할 겁니다. 이번 장에서 구현하는 함수는 다음 장 이후에서도 이용합니다. 즉, 이번 장은 앞으로의 텍스트 처리를 위한 사전 준비도 겸하고 있습니다. 이쯤 이야기하고, 어서 자연어 처리의 세계로 뛰어들어봅시다!

* 옮긴이_ 같은 단어라도 세월이 흐르면서 새로운 의미가 덧씌워질 수 있음을 보여주는 예입니다. 영화 <백 투 더 퓨처>에서 마티는 박사보다 30년 미래에서 시간여행을 왔다는 설정입니다. 같은 예로 마티가 가게에서 ‘펍시 프리’를 주문하자 점원이 ‘펍시를 공짜로 달라고?’라며 황당해하는 장면도 나옵니다.

2.1 자연어 처리란

한국어와 영어 등 우리가 평소에 쓰는 말을 **자연어**(natural language)라고 합니다. 그러니 **자연어 처리**(Natural Language Processing (NLP))를 문자 그대로 해석하면 ‘자연어를 처리하는 분야’이고, 알기 쉽게 풀해보면 ‘우리의 말을 컴퓨터에게 이해시키기 위한 기술(분야)’입니다. 그래서 자연어 처리가 추구하는 목표는 사람의 말을 컴퓨터가 이해하도록 만들어서, 컴퓨터가 우리에게 도움이 되는 일을 수행하게 하는 것입니다.

그런데 컴퓨터가 이해할 수 있는 언어라고 하면 ‘프로그래밍 언어’나 ‘마크업 언어’와 같은 것이 떠오를 겁니다. 이러한 언어는 모든 코드의 의미를 고유하게 해석할 수 있도록 문법이 정의되어 있고, 컴퓨터는 이 정해진 규칙에 따라서 코드를 해석합니다.

여러분도 알다시피, 일반적인 프로그래밍 언어는 기계적이고 고정되어 있습니다. ‘딱딱한 언어’란 뜻이죠. 반면, 영어나 한국어 같은 자연어는 ‘부드러운 언어’입니다. ‘부드럽다’라는 것은 똑같은 의미의 문장도 여러 형태로 표현할 수 있다거나, 문장의 뜻이 애매할 수 있다거나, 그 의미나 형태가 유연하게 바뀐다는 뜻입니다. 세월이 흐르면서 새로운 말이나 새로운 의미가 생겨나거나 있던 것이 사라지기까지 하죠. 이 모두가 자연어가 부드럽기 때문입니다.

이처럼 자연어는 살아 있는 언어이며 그 안에는 ‘부드러움’이 있습니다. 따라서 머리가 굳은 컴퓨터에게 자연어를 이해시키기란 평범한 방법으로는 도달할 수 없는 어려운 도전입니다. 하지만 그 난제를 해결할 수 있다면, 즉 컴퓨터에게 자연어를 이해시킬 수 있다면 수많은 사람에게 도움되는 일을 컴퓨터에게 시킬 수 있습니다. 사실 그런 예는 흔히 볼 수 있습니다. 검색 엔진이나 기계 번역은 잘 알려진 예지요. 그 밖에도 질의응답 시스템, IME(입력기 전환), 문장 자동요약과 감정분석 등 우리 주변에는 이미 자연어 처리 기술이 널리 사용되고 있습니다.

NOTE_ 자연어 처리를 응용한 예로 ‘질의응답 시스템’이 있습니다. 그 대표주자인 IBM 왓슨(Watson)은 아주 유명하죠. 왓슨은 2011년 미국의 TV 퀴즈쇼 ‘제퍼디!(Jeopardy!)’를 통해서 세상에 널리 알려졌습니다. 이 퀴즈쇼에서 왓슨은 어떤 사람보다도 더 정확하게 대답하여 기존 챔피언을 물리치고 우승했습니다(왓슨에게는 문제가 텍스트로 주어진다는 이점이 있긴 했습니다). 이 ‘사건’은 세상의 이목을 집중시켰고, 인공지능에 대한 기대와 동시에 불안까지도 키우는 계기가 되었습니다. 또한 왓슨은 ‘의사결정 지원 시스템’으로 분야를 넓혀, 최근에는 과거의 방대한 의료 데이터를 활용해 난치병 환자에게 올바른 치료법을 제안해 목숨을 구한 사례가 보고되기도 했습니다.

2.1.1 단어의 의미

우리의 말은 ‘문자’로 구성되며, 말의 의미는 ‘단어’로 구성됩니다. 단어는 말하자면 의미의 최소 단위인 셈이죠. 그래서 자연어를 컴퓨터에게 이해시키는 데는 무엇보다 ‘단어의 의미’를 이해시키는 게 중요합니다.

이번 장의 주제는 컴퓨터에게 ‘단어의 의미’ 이해시키기입니다. 더 정확히 말하면 ‘단어의 의미’를 잘 파악하는 표현 방법에 관해 생각해봅시다. 구체적으로는 이번 장과 다음 장에서 다음의 세 가지 기법을 살펴보겠습니다.

- 시소러스를 활용한 기법 (이번 장)
- 통계 기반 기법 (이번 장)
- 추론 기반 기법(word2vec) (다음 장)

가장 먼저, 사람의 손으로 만든 시소러스^{thesaurus} (유의어 사전)를 이용하는 방법을 간단히 살펴봅시다. 그런 다음 통계 정보로부터 단어를 표현하는 ‘통계 기반 기법’을 설명합니다. 여기까지가 이번 장에서 배우는 내용입니다. 그 뒤를 이어 다음 장에서는 신경망을 활용한 ‘추론 기반’ 기법(구체적으로는 word2vec)을 다룹니다. 참고로 이번 장의 구성은 스탠퍼드 대학교의 ‘CS224d: Deep Learning for Natural Language Processing’ 수업^[1]을 참고했습니다.

2.2 시소러스

‘단어의 의미’를 나타내는 방법으로는 먼저 사람이 직접 단어의 의미를 정의하는 방식을 생각할 수 있습니다. 그중 한 방법으로 『표준국어대사전』처럼 각각의 단어에 그 의미를 설명해 넣을 수 있을 것입니다. 예컨대 『표준국어대사전』에서 “자동차”라는 단어를 찾으면 “원동기를 장치하여 그 동력으로 바퀴를 굴려서 철길이나 가설된 선에 의하지 아니하고 땅 위를 움직이도록 만든 차”라는 설명이 나옵니다. 이런 식으로 단어들을 정의해두면 컴퓨터도 단어의 의미를 이해할 수 있을지도 모릅니다.

자연어 처리의 역사를 되돌아보면 단어의 의미를 인력을 동원해 정의하려는 시도는 수없이 있어왔습니다. 단, 『표준국어대사전』 같이 사람이 이용하는 일반적인 사전이 아니라 **시소러스** 형태의 사전을 애용했죠. 시소러스란 (기본적으로는) 유의어 사전으로, ‘뜻이 같은 단어(동의어)’

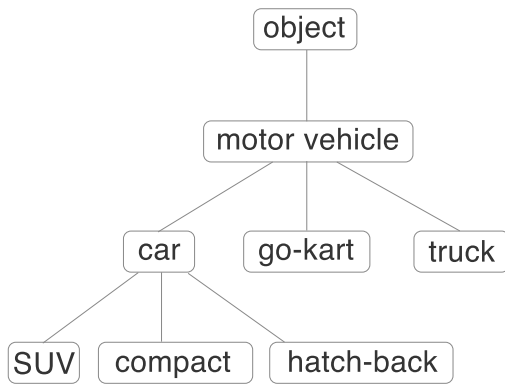
나 ‘뜻이 비슷한 단어(유의어)’가 한 그룹으로 분류되어 있습니다.

그림 2-1 동의어의 예: “car”, “auto”, “automobile” 등은 “자동차”를 뜻하는 동의어다.



또한 자연어 처리에 이용되는 시소러스에서는 단어 사이의 ‘상위와 하위’ 혹은 ‘전체와 부분’ 등, 더 세세한 관계까지 정의해둔 경우가 있습니다. [그림 2-2]의 예처럼 각 단어의 관계를 그래프 구조로 정의합니다.

그림 2-2 단어들을 의미의 상·하위 관계에 기초해 그래프로 표현한다(문헌 [14]를 참고하여 그림).



[그림 2-2]에서는 “car”의 상위 개념으로 “motor vehicle (동력차)”이라는 단어가 존재합니다. 한편 “car”의 하위 개념으로는 “SUV (스포츠 유틸리티 자동차)”, “compact (소형차)”, “hatch-back (해치백)” 등 더 구체적인 차종이 있음을 알려줍니다.

이처럼 모든 단어에 대한 유의어 집합을 만든 다음, 단어들의 관계를 그래프로 표현하여 단어 사이의 연결을 정의할 수 있습니다. 그러면 이 ‘단어 네트워크’를 이용하여 컴퓨터에게 단어 사이의 관계를 가르칠 수 있습니다. 이 정도면 컴퓨터에게 단어의 의미를 (간접적으로라도) 이해시켰다고 주장할 수 있을 것입니다. 그리고 그 지식을 이용하면 우리에게 유용한 일들을 컴퓨터가 수행하도록 할 수 있겠지요.

NOTE 시소러스를 어떻게 사용하는가는 자연어 처리 애플리케이션에 따라 다릅니다. 검색 엔진을 예로 생각해볼까요? “automobile”과 “car”가 유의어임을 알고 있으면 “car”의 검색 결과에 “automobile”의 검색 결과도 포함시켜주면 좋을 겁니다.

2.2.1 WordNet

자연어 처리 분야에서 가장 유명한 시소러스는 **WordNet**^[7]입니다. WordNet은 프린스턴 대학교에서 1985년부터 구축하기 시작한 전통 있는 시소러스로, 지금까지 많은 연구와 다양한 자연어 처리 애플리케이션에서 활용되고 있습니다.

WordNet을 사용하면 유의어를 얻거나 ‘단어 네트워크’를 이용할 수 있습니다. 또한 단어 네트워크를 사용해 단어 사이의 유사도를 구할 수도 있죠. 이 책에서는 WordNet을 자세히 설명하지는 않으니, WordNet을 사용한 파이썬 구현에 흥미가 있는 분은 ‘부록 B. WordNet 맛보기’를 참고하세요. 부록 B에서는 WordNet(정확하게는 NLTK 모듈)을 설치하고 몇 가지 간단한 실험을 해봅니다.

NOTE_ ‘부록 B. WordNet 맛보기’에서는 실제로 WordNet을 사용하여 단어의 유사도를 구하는 실험을 해봅니다. 사람이 직접 정의한 ‘단어 네트워크’를 기초로 단어 사이의 유사도를 구하는 사례를 살펴볼 텐데, 단어의 유사도를 (어느 정도 정확하게) 구할 수 있다면 ‘단어의 의미’를 이해하는 첫걸음을 내디뎠다고 말할 수 있을 것입니다.

2.2.2 시소러스의 문제점

WordNet과 같은 시소러스에는 수많은 단어에 대한 동의어와 계층 구조 등의 관계가 정의돼 있습니다. 그리고 이 지식을 이용하면 ‘단어의 의미’를 (간접적으로라도) 컴퓨터에 전달할 수 있습니다. 하지만 이처럼 사람이 수작업으로 레이블링하는 방식에는 크나큰 결점이 존재합니다. 다음은 시소러스 방식의 대표적인 문제점들입니다.

- **시대 변화에 대응하기 어렵다.**

우리가 사용하는 말은 살아 있습니다. 때때로 새로운 단어가 생겨나고, 구닥다리 옛말은 언젠가 잊혀집니다. 예컨대 “크라우드 펀딩(crowd funding)”은 최근에 등장한 신조어입니다.

또한 시대에 따라 언어의 의미가 변하기도 있습니다. 예컨대 영어 단어 “heavy”에는 ‘심각하다’라는 뜻도 있습니다(주로 비속어에서 이용됩니다만). 하지만 옛날에는 그런 뜻이 없었습니다. 영화 <백 투 더 퓨처>에서는 (영화 기준으로 미래인) 1985년에서 온 마티와 1955년에 사는 박사가 대화를 나누는 중 “heavy”의 의미가 통하지 않는 장면이 나옵니다. 이런 단어의 변화에 대응하려면 시소러스를 사람이 수작업으로 끊임없이 갱신해야 합니다.

• 사람을 쓰는 비용은 크다.

시소러스를 만드는 데는 엄청난 인적 비용이 발생합니다. 영어를 예로 들면, 현존하는 영어 단어의 수는 1,000만 개가 넘는다고 합니다. 따라서 이상적으로는 이 방대한 단어들 모두에 대해 단어 사이의 관계를 정의해줘야 합니다. 참고로 WordNet에 등록된 단어는 20만 개 이상입니다.

• 단어의 미묘한 차이를 표현할 수 없다.

시소러스에서는 뜻이 비슷한 단어들을 묶습니다. 그러나 실제로 비슷한 단어들이라도 미묘한 차이가 있는 법이죠. 예컨대 '빈티지'^[vintage, 낡고 오래된 것]와 '레트로'^[retro, 복고]는 의미가 같습지만, 용법은 다릅니다. 시소러스에서는 이러한 미묘한 차이를 표현할 수 없습니다(이 역시 수작업으로 표현하려 한다면 상당히 곤란한 일이 되겠지요).

이처럼 시소러스를 사용하는 기법(단어의 의미를 사람이 정의하는 기법)에는 커다란 문제가 있습니다. 이 문제를 피하기 위해, 곧이어 '통계 기반 기법'과 신경망을 사용한 '추론 기반 기법'을 알아볼 것입니다. 이 두 기법에서는 대량의 텍스트 데이터로부터 '단어의 의미'를 자동으로 추출합니다. 그 덕분에 사람은 순수 단어를 연결짓는 중노동에서 해방되는 것입니다!

NOTE _ 자연어 처리뿐 아니라, 이미지 인식에서도 특징^[feature]을 사람이 수동으로 설계하는 일이 오랜 세월 계속되었습니다. 그러다가 딥러닝이 실용화되면서 실생활 이미지로부터 원하는 결과를 곧바로 얻을 수 있게 되었습니다. 사람이 개입할 필요가 현격히 줄어든 것이죠. 자연어 처리에서도 똑같은 일이 벌어지고 있습니다. 즉, 사람이 수작업으로 시소러스나 관계(특징)를 설계하던 방식으로부터, 사람의 개입을 최소로 줄이고 텍스트 데이터만으로 원하는 결과를 얻어내는 방향으로 패러다임이 바뀌고 있습니다.

2.3 통계 기반 기법

이제부터 통계 기반 기법을 살펴보면서 우리는 **말뭉치**^[corpus]를 이용할 겁니다. 말뭉치란 간단히 말하면 대량의 텍스트 데이터입니다. 다만 맹목적으로 수집된 텍스트 데이터가 아닌 자연어 처리 연구나 애플리케이션을 염두에 두고 수집된 텍스트 데이터를 일반적으로 '말뭉치'라고 합니다.

결국 말뭉치란 텍스트 데이터에 지나지 않습니다만, 그 안에 담긴 문장들은 사람이 쓴 글입니다. 다른 시각에서 생각해보면, 말뭉치에는 자연어에 대한 사람의 '지식'이 충분히 담겨 있다고 볼 수 있습니다. 문장을 쓰는 방법, 단어를 선택하는 방법, 단어의 의미 등 사람이 알고 있는 자연어에 대한 지식이 포함되어 있는 것이죠. 통계 기반 기법의 목표는 이처럼 사람의 지식으로

근거 없이 추론하는 건 금물이야.

- 코넬 도일, <셜록 홈즈의 모험(보헤미아 왕국의 스캔들)>

앞 장에 이어 이번 장의 주제도 단어의 분산 표현입니다. 앞 장에서는 '통계 기반 기법'으로 단어의 분산 표현을 얻었는데, 이번 장에서는 더 강력한 기법인 '추론 기반 기법'을 살펴보겠습니다.

이름에서 알 수 있듯이 '추론 기반 기법'은 추론을 하는 기법입니다. 물론 이 추론 과정에 신경망을 이용하는데, 여기서 그 유명한 word2vec이 등장합니다. 이번 장에서는 word2vec의 구조를 차분히 들여다보고 구현해보며 확실하게 이해해보고자 합니다.

이번 장의 목표는 '단순한' word2vec 구현하기입니다. 처리 효율을 희생하는 대신 이해하기 쉽도록 구성해서 '단순한' word2vec입니다. 따라서 큰 데이터셋까지는 어렵겠지만, 작은 데이터셋이라면 문제없이 처리할 수 있습니다. 다음 장에서는 이번 장의 단순한 word2vec에 몇 가지 개선을 더해 '진짜' word2vec을 완성시킵니다. 자, 그럼 추론 기반 기법과 word2vec의 세계로 떠나봅시다!

3.1 추론 기반 기법과 신경망

단어를 벡터로 표현하는 방법은 지금까지 활발히 연구되었습니다. 그중에서도 성공적인 기법들을 크게 두 부류로 나눌 수 있는데, 바로 ‘통계 기반 기법’과 ‘추론 기반 기법’입니다. 단어의 의미를 얻는 방식은 서로 크게 다르지만, 그 배경에는 모두 분포 가설이 있습니다.

이번 절에서는 통계 기반 기법의 문제를 지적하고, 그 대안인 추론 기반 기법의 이점을 거시적 관점에서 설명합니다. 그런 다음 word2vec의 전처리를 위해 신경망으로 ‘단어’를 처리하는 예를 보여드리겠습니다.

3.1.1 통계 기반 기법의 문제점

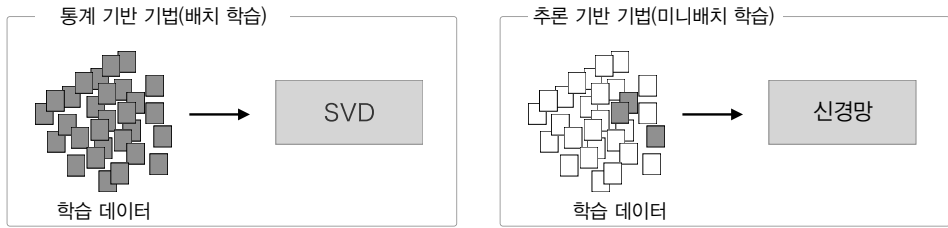
지금까지 본 것처럼 통계 기반 기법에서는 주변 단어의 빈도를 기초로 단어를 표현했습니다. 구체적으로는 단어의 동시발생 행렬을 만들고 그 행렬에 SVD를 적용하여 밀집벡터(단어의 분산 표현)를 얻었습니다. 그러나 이 방식은 대규모 말뭉치를 다룰 때 문제가 발생합니다.

현업에서 다루는 말뭉치의 어휘 수는 어마어마합니다. 예컨대 영어의 어휘 수는 100만을 훌쩍 넘는다고 합니다. 어휘가 100만 개라면, 통계 기반 기법에서는 ‘100만×100만’이라는 거대한 행렬을 만들게 됩니다. 이런 거대 행렬에 SVD를 적용하는 일은 현실적이지 않겠죠.

NOTE SVD를 $n \times n$ 행렬에 적용하는 비용은 $O(n^3)$ 입니다. $O(n^3)$ 이란 계산 시간이 n 의 3 제곱에 비례한다는 뜻입니다. 슈퍼컴퓨터를 동원해도 처리할 수 없는 수준이죠. 실제로는 근사적인 기법과 희소행렬의 성질 등을 이용해 속도를 개선할 수 있습니다만, 그렇다고 해도 여전히 상당한 컴퓨팅 자원을 들여 장시간 계산해야 합니다.

통계 기반 기법은 말뭉치 전체의 통계(동시발생 행렬과 PPMI 등)를 이용해 단 1회의 처리(SVD 등)만에 단어의 분산 표현을 얻습니다. 한편, 추론 기반 기법에서는, 예컨대 신경망을 이용하는 경우는 미니배치로 학습하는 것이 일반적입니다. 미니배치 학습에서는 신경망이 한 번에 소량(미니배치)의 학습 샘플씩 반복해서 학습하며 가중치를 갱신해갑니다. [그림 3-1]은 이 두 기법의 큰 차이를 보여줍니다.

그림 3-1 통계 기반 기법과 추론 기반 기법 비교



[그림 3-1]처럼 통계 기반 기법은 학습 데이터를 한꺼번에 처리합니다(배치 학습). 이에 반해 추론 기반 기법은 학습 데이터의 일부를 사용하여 순차적으로 학습합니다(미니배치 학습). 이 말은 말뭉치의 어휘 수가 많아 SVD 등 계산량이 큰 작업을 처리하기 어려운 경우에도 신경망을 학습시킬 수 있다는 뜻입니다. 데이터를 작게 나눠 학습하기 때문이죠. 게다가 여러 머신과 여러 GPU를 이용한 병렬 계산도 가능해져서 학습 속도를 높일 수도 있습니다. 추론 기반 기법이 큰 힘을 발휘하는 영역이죠.

추론 기반 기법이 통계 기반 기법보다 매력적인 점은 이 외에도 더 있습니다. 이 매력에 관해서는, 먼저 추론 기반 기법(특히 word2vec)을 자세히 알아본 후 ‘3.5.3 통계 기반 vs. 추론 기반’ 절에서 다시 한번 논의하겠습니다.

3.1.2 추론 기반 기법 개요

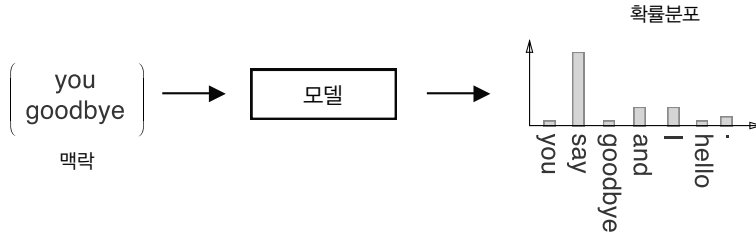
추론 기반 기법에서는 당연히 ‘추론’이 주된 작업입니다. 추론이란 [그림 3-2]처럼 주변 단어(맥락)가 주어졌을 때 “?”에 무슨 단어가 들어가는지를 추측하는 작업입니다.

그림 3-2 주변 단어들을 맥락으로 사용해 “?”에 들어갈 단어를 추측한다.

you ? goodbye and I say hello.

[그림 3-2]처럼 추론 문제를 풀고 학습하는 것이 ‘추론 기반 기법’이 다루는 문제입니다. 이러한 추론 문제를 반복해서 풀면서 단어의 출현 패턴을 학습하는 것이죠. ‘모델 관점’에서 보면, 이 추론 문제는 [그림 3-3]처럼 보입니다.

그림 3-3 추론 기반 기법: 맥락을 입력하면 모델은 각 단어의 출현 확률을 출력한다.



[그림 3-3]처럼 추론 기반 기법에는 어떠한 모델이 등장합니다. 우리는 이 모델로 신경망을 사용합니다. 모델은 맥락 정보를 입력받아 (출현할 수 있는) 각 단어의 출현 확률을 출력합니다. 이러한 틀 안에서 말뭉치를 사용해 모델이 올바른 추측을 내놓도록 학습시킵니다. 그리고 그 학습의 결과로 단어의 분산 표현을 얻는 것이 추론 기반 기법의 전체 그림입니다.

NOTE_ 추론 기반 기법도 통계 기반 기법처럼 분포 가설에 기초합니다. 분포 가설이란 '단어의 의미는 주변 단어에 의해 형성된다'는 가설로, 추론 기반 기법에서는 이를 앞에서와 같은 추측 문제로 귀결시켰습니다. 이처럼 두 기법 모두 분포 가설에 근거하는 '단어의 동시발생 가능성'을 얼마나 잘 모델링하는가가 중요한 연구 주제입니다.

3.1.3 신경망에서의 단어 처리

지금부터 신경망을 이용해 '단어'를 처리합니다. 그런데 신경망은 “you”와 “say” 등의 단어를 있는 그대로 처리할 수 없으니 단어를 ‘고정 길이의 벡터’로 변환해야 합니다. 이때 사용하는 대표적인 방법이 단어를 **원핫^{one-hot} 표현**(또는 **원핫 벡터**)으로 변환하는 것입니다. 원핫 표현이란 벡터의 원소 중 하나만 1이고 나머지는 모두 0인 벡터를 말합니다.

원핫 표현에 대해 구체적으로 살펴봅시다. 여기에서는 앞 장과 같이 “You say goodbye and I say hello.”라는 한 문장짜리 말뭉치를 예로 설명하겠습니다. 이 말뭉치에는 어휘가 총 7개 등장합니다(“you”, “say”, “goodbye”, “and”, “i”, “hello”, “.”). 이중 두 단어의 원핫 표현을 [그림 3-4]에 나타내었습니다.

word2vec 속도 개선

모든 것을 알고 애쓰지 마라.
그러다 보면 아무것도 기억할 수 없다.
- 데모크리토스(고대 그리스 철학자)

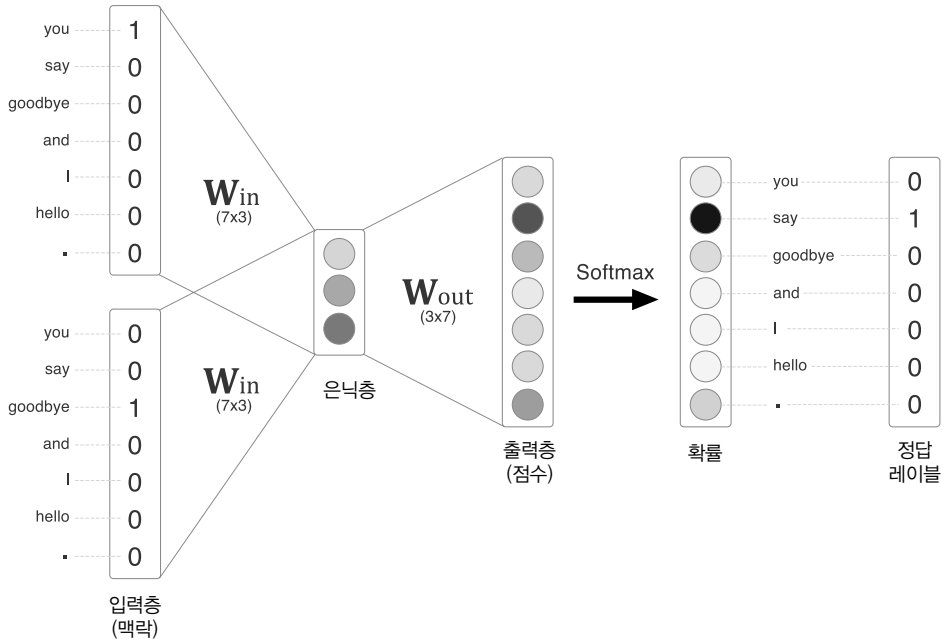
앞의 3장에서 word2vec의 구조를 배우고 CBOW 모델을 구현했습니다. CBOW 모델은 단순한 2층 신경망이라서 간단하게 구현할 수 있었습니다. 그러나 그 구현에는 몇 가지 문제가 있었죠. 가장 큰 문제는 말뭉치에 포함된 어휘 수가 많아지면 계산량도 커진다는 점입니다. 실제로 어휘 수가 어느 정도를 넘어서면 앞 장의 CBOW 모델은 계산 시간이 너무 오래 걸립니다.

그래서 이번 장의 목표는 word2vec의 속도 개선으로 잡아봤습니다. 구체적으로는 앞 장의 단순한 word2vec에 두 가지 개선을 추가할 겁니다. 첫 번째 개선으로는 Embedding이라는 새로운 계층을 도입합니다. 그리고 두 번째 개선으로 네거티브 샘플링이라는 새로운 손실 함수를 도입합니다. 이 두 가지 개선으로 우리는 ‘진짜’ word2vec을 완성할 수 있습니다. 진짜 word2vec이 완성되면 PTB 데이터셋(실용적인 크기의 말뭉치)을 가지고 학습을 수행할 겁니다. 그리고 그 결과로 얻은 단어의 분산 표현의 장점을 실제로 평가해볼 것입니다.

4.1 word2vec 개선 ①

그럼 앞 장의 복습부터 시작합니다. 앞 장에서는 [그림 4-1]과 같은 CBOW 모델을 구현했습니다.

그림 4-1 앞 장에서 구현한 CBOW 모델



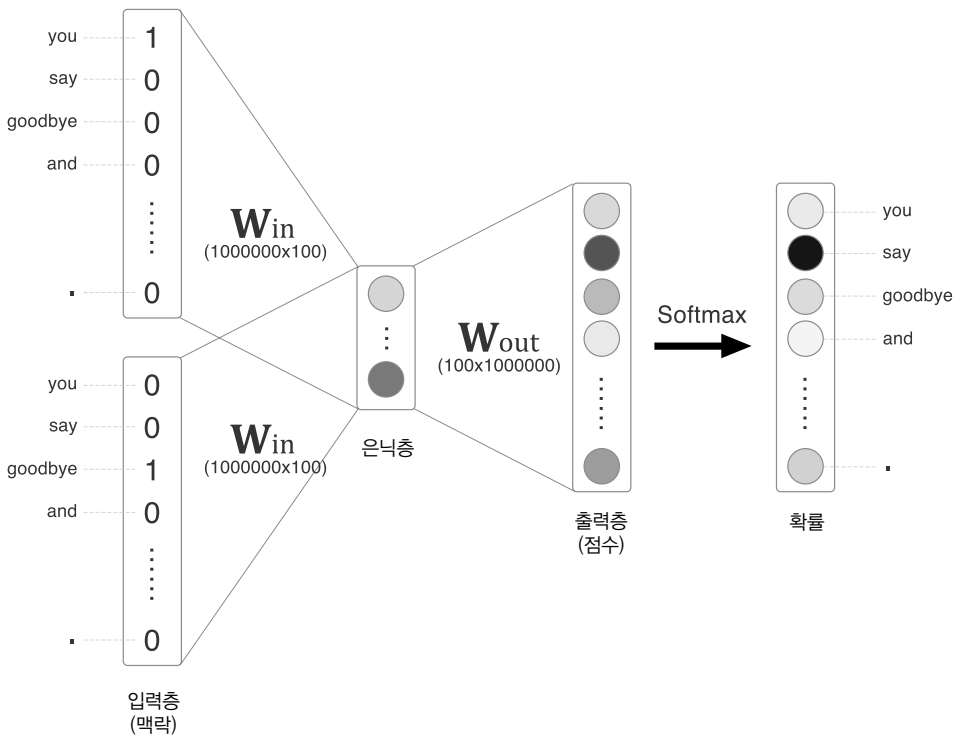
[그림 4-1]과 같이 앞 장의 CBOW 모델은 단어 2개를 맥락으로 사용해, 이를 바탕으로 하나의 단어(타깃)를 추측합니다. 이때 입력 측 가중치(W_{in})와의 행렬 곱으로 은닉층이 계산되고, 다시 출력 측 가중치(W_{out})와의 행렬 곱으로 각 단어의 점수를 구합니다. 그리고 이 점수에 소프트맥스 함수를 적용해 각 단어의 출현 확률을 얻고, 이 확률을 정답 레이블과 비교하여(정확히는 교차 엔트로피 오차를 적용하여) 손실을 구합니다.

WARNING_ 앞 장에서는 맥락의 윈도우 크기를 1로 한정했습니다. 다시 말해 타깃 앞뒤 한 단어씩만 사용한 것입니다. 이번 장에서는 나중에 어떤 크기의 맥락도 다룰 수 있도록 기능을 추가할 것입니다.

[그림 4-1]의 CBOW 모델도 작은 말뭉치를 다룰 때는 특별히 문제될 게 없습니다. 실제 [그

림 4-1]에서 다루는 어휘 수는 모두 7개인데, 이 정도는 전혀 문제없이 처리할 수 있죠. 그러나 거대한 말뭉치를 다루게 되면 몇 가지 문제가 발생합니다. 그 문제점을 보여드리고자 어휘가 100만 개, 은닉층의 뉴런이 100개인 CBOW 모델을 생각해보겠습니다. 그러면 word2vec은 [그림 4-2]처럼 됩니다.

그림 4-2 어휘가 100만 개일 때를 가정한 CBOW 모델



[그림 4-2]에서 보듯, 입력층과 출력층에는 각 100만 개의 뉴런이 존재합니다. 이 수많은 뉴런 때문에 중간 계산에 많은 시간이 소요되죠. 정확히는 다음의 두 계산이 병목이 됩니다.

- 입력층의 원핫 표현과 가중치 행렬 W_{in} 의 곱 계산(4.1절에서 해결)
- 은닉층과 가중치 행렬 W_{out} 의 곱 및 Softmax 계층의 계산(4.2절에서 해결)

첫 번째는 입력층의 원핫 표현과 관련한 문제입니다. 단어를 원핫 표현으로 다루기 때문에 어휘 수가 많아지면 원핫 표현의 벡터 크기도 커지는 것이죠. 예컨대 어휘가 100만 개라면 그 원핫 표현 하나만 해도 원소 수가 100만 개인 벡터가 됩니다. 상당한 메모리를 차지하겠죠. 게다가

가 이 원핫 벡터와 가중치 행렬 \mathbf{W}_{in} 을 곱해야 하는데, 이것만으로 계산 자원을 상당히 사용하게 됩니다. 이 문제는 이번 4.1절에서 Embedding 계층을 도입하는 것으로 해결합니다.

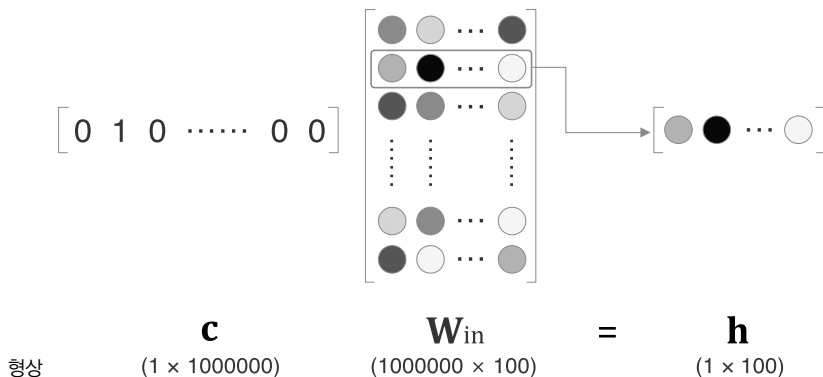
두 번째 문제는 은닉층 이후의 계산입니다. 우선 은닉층과 가중치 행렬 \mathbf{W}_{out} 의 곱만 해도 계산량이 상당합니다. 그리고 Softmax 계층에서도 다루는 어휘가 많아짐에 따라 계산량이 증가하는 문제가 있습니다. 이 문제는 4.2절에서 네거티브 샘플링이라는 새로운 손실 함수를 도입해 해결합니다. 그러면 두 병목을 해소할 수 있도록 각각의 개선을 적용해봅시다.

NOTE 개선 전의 word2vec 파일(앞 장의 word2vec 구현)은 ch03 디렉터리의 simple_cbow.py(혹은 simple_skip_gram.py), 개선 후의 파일은 ch04 디렉터리의 cbow.py(혹은 skip_gram.py)에 있습니다.

4.1.1 Embedding 계층

앞 장의 word2vec 구현에서는 단어를 원핫 표현으로 바꿨습니다. 그리고 그것을 MatMul 계층에 입력하고, MatMul 계층에서 가중치 행렬을 곱했습니다. 그럼 여기서 어휘 수가 100만 개인 경우를 상상해봅시다. 이때 은닉층 뉴런이 100개라면, MatMul 계층의 행렬 곱은 [그림 4-3]처럼 됩니다.

그림 4-3 맥락(원핫 표현)과 MatMul 계층의 가중치를 곱한다.



[그림 4-3]처럼 만약 100만 개의 어휘를 담은 말뭉치가 있다면, 단어의 원핫 표현도 100만

순환 신경망(RNN)

아무튼 어두컴컴하고 축축한 데서
야옹야옹 울고 있었던 것만은 분명히 기억한다.
- 나쓰메 소세키, 『나는 고양이로소이다』

지금까지 살펴본 신경망은 **피드포워드**(feed forward, 앞먹임)라는 유형의 신경망입니다. 피드포워드란 흐름이 단방향인 신경망을 말합니다. 다시 말해, 입력 신호가 다음 층(중간층)으로 전달되고, 그 신호를 받은 층은 그다음 층으로 전달하고, 다시 다음 층으로... 식으로 한 방향으로만 신호가 전달됩니다.

피드포워드 신경망은 구성이 단순하여 구조를 이해하기 쉽고, 그래서 많은 문제에 응용할 수 있습니다. 그러나 커다란 단점이 하나 있으니, 바로 시계열 데이터를 잘 다루지 못한다는 것입니다. 더 정확하게 말하면, 단순한 피드포워드 신경망에서는 시계열 데이터의 성질(패턴)을 충분히 학습할 수 없습니다. 그래서 **순환 신경망**(Recurrent Neural Network (**RNN**))이 등장하게 됩니다.

이번 장에서는 피드포워드 신경망의 문제점을 지적하고, RNN이 그 문제를 훌륭하게 해결할 수 있음을 설명합니다. 또한 RNN의 구조를 차분히 시간을 들여 설명하면서, 파이썬으로 구현도 해볼 것입니다.

5.1 확률과 언어 모델

이번 절에서는 RNN 이야기를 시작하기 위한 준비 과정으로, 먼저 앞 장의 word2vec을 복습해보겠습니다. 그런 다음 자연어에 관한 현상을 ‘확률’을 사용해 기술하고, 마지막에는 언어를 확률로 다루는 ‘언어 모델’에 대해 설명합니다.

5.1.1 word2vec을 확률 관점에서 바라보다


그럼 word2vec의 CBOW 모델부터 복습해보죠. 여기에서는 w_1, w_2, \dots, w_t 라는 단어열로 표현되는 말뭉치를 생각해보겠습니다. 그리고 t 번째 단어를 ‘타깃’으로, 그 전후 단어($t-1$ 번째와 $t+1$ 번째)를 ‘맥락’으로 취급해보죠.

WARNING_ 이 책에서 타깃은 ‘중앙 단어’를, 맥락은 타깃의 ‘주변 단어’를 가리킵니다.

이때 CBOW 모델은 [그림 5-1]처럼 맥락 w_{t-1} 과 w_{t+1} 로부터 타깃 w_t 를 추측하는 일을 수행합니다.

그림 5-1 word2vec의 CBOW 모델: 맥락의 단어로부터 타깃 단어를 추측한다.

$w_1 \ w_2 \ \dots \ w_{t-1} \ \boxed{w_t} \ w_{t+1} \ \dots \ w_{T-1} \ w_T$



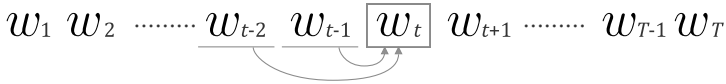
그럼 w_{t-1} 과 w_{t+1} 이 주어졌을 때 타깃이 w_t 가 될 확률을 수식으로 나타내봅시다.

$$P(w_t | w_{t-1}, w_{t+1}) \quad \text{[식 5.1]}$$

CBOW 모델은 [식 5.1]의 사후 확률을 모델링합니다. 이 사후 확률은 ‘ w_{t-1} 과 w_{t+1} 이 주어졌을 때 w_t 가 일어날 확률’을 뜻합니다. 이것이 윈도우 크기가 1일 때의 CBOW 모델입니다.

그런데 지금까지는 맥락을 항상 좌우 대칭으로 생각해왔습니다. 이번에는 맥락을 왼쪽 윈도우만으로 한정해보겠습니다. 예컨대 [그림 5-2]와 같은 경우를 생각해보죠.

그림 5-2 왼쪽 윈도우만 맥락으로 고려한다.



[그림 5-2]처럼 왼쪽 두 단어만을 맥락으로 생각하겠습니다. 그러면 CBOW 모델이 출력할 확률은 [식 5.2]처럼 됩니다.

$$P(w_t | w_{t-2}, w_{t-1}) \quad \text{[식 5.2]}$$

NOTE_ word2vec에서 맥락의 윈도우 크기는 하이퍼파라미터입니다. 임의의 값으로 설정할 수 있다는 뜻이죠. 이번 예에서는 윈도우 크기를 '왼쪽 2 단어, 오른쪽 0 단어'라는 좌우 비대칭으로 설정했습니다. 이렇게 설정한 이유는 나중에 설명하는 '언어 모델'에서 이야기하겠습니다.

그런데 [식 5.2]의 표기를 사용하면, CBOW 모델이 다루는 손실 함수를 [식 5.3]처럼 쓸 수 있습니다. 즉, [식 5.3]은 교차 엔트로피 오차에 의해 유도한 결과입니다(자세한 내용은 '1.3.1 손실 함수' 절 참고).

$$L = -\log P(w_t | w_{t-2}, w_{t-1}) \quad \text{[식 5.3]}$$

CBOW 모델의 학습으로 수행하는 일은 [식 5.3]의 손실 함수(정확히는 말뭉치 전체의 손실 함수의 총합)를 최소화하는 가중치 매개변수를 찾는 것입니다. 이러한 가중치 매개변수가 발견되면 CBOW 모델은 맥락으로부터 타깃을 더 정확하게 추측할 수 있게 되죠.

이처럼 CBOW 모델을 학습시키는 본래 목적은 맥락으로부터 타깃을 정확하게 추측하는 것입니다. 이 목적을 위해 학습을 진행하면, (그 부산물로) 단어의 의미가 인코딩된 '단어의 분산 표현'을 얻을 수 있습니다.

그럼 CBOW 모델의 본래 목적인 '맥락으로부터 타깃을 추측하는 것'은 어디에 이용할 수 있을까요? [식 5.2]의 확률 $P(w_t | w_{t-2}, w_{t-1})$ 은 실용적인 쓰임이 있을까요? 여기서 '언어 모델'이 등장합니다.

5.1.2 언어 모델

언어 모델 Language Model은 단어 나열에 확률을 부여합니다. 특정한 단어의 시퀀스에 대해서, 그 시퀀스가 일어날 가능성이 어느 정도인지(얼마나 자연스러운 단어 순서인지)를 확률로 평가하는 것이죠. 예컨대 “you say goodbye”라는 단어 시퀀스에는 높은 확률(예: 0.092)을 출력하고, “you say good die”에는 낮은 확률(예: 0.00000000000032)을 출력하는 것이 일종의 언어 모델입니다.

이 언어 모델은 다양하게 응용할 수 있습니다. 기계 번역과 음성 인식이 대표적인 예입니다. 예를 들어 음성 인식 시스템의 경우, 사람의 음성으로부터 몇 개의 문장을 후보로 생성할 것입니다. 그런 다음 언어 모델을 사용하여 후보 문장이 ‘문장으로써 자연스러운지’를 기준으로 순서를 매길 수 있습니다.

또한 언어 모델은 새로운 문장을 생성하는 용도로도 이용할 수 있습니다. 왜냐하면 언어 모델은 단어 순서의 자연스러움을 확률적으로 평가할 수 있으므로, 그 확률분포에 따라 다음으로 적합한 단어를 ‘자아낼’(샘플링) 수 있기 때문이죠. 참고로 언어 모델을 사용한 문장 생성은 7장. RNN을 사용한 문장 생성’에서 설명합니다.

그러면 언어 모델을 수식으로 설명해보겠습니다. w_1, \dots, w_m 이라는 m 개 단어로 된 문장을 생각해보죠. 이때 단어가 w_1, \dots, w_m 이라는 순서로 출현할 확률을 $P(w_1, \dots, w_m)$ 으로 나타냅니다. 이 확률은 여러 사건이 동시에 일어날 확률이므로 동시 확률이라고 합니다.

이 동시 확률 $P(w_1, \dots, w_m)$ 은 사후 확률을 사용하여 다음과 같이 분해하여 쓸 수 있습니다.

$$\begin{aligned} P(w_1, \dots, w_m) &= P(w_m | w_1, \dots, w_{m-1}) P(w_{m-1} | w_1, \dots, w_{m-2}) \\ &\quad \dots P(w_3 | w_1, w_2) P(w_2 | w_1) P(w_1) \\ &= \prod_{i=1}^m P(w_i | w_1, \dots, w_{i-1}) \end{aligned} \quad \text{[식 5.4]*}$$

[식 5.4]의 \prod 파이 기호는 모든 원소를 곱하는 ‘총곱’을 뜻합니다(총합을 뜻하는 \sum 시그마와 구분해주세요). [식 5.4]에서 알 수 있듯, 동시 확률은 사후 확률의 총곱으로 나타낼 수 있습니다.

* 수식을 간소화하기 위해 여기에서는 $P(w_i | w_0)$ 를 $P(w_i)$ 로 처리합니다.

게이트가 추가된 RNN

망각은 더 나은 전진을 낳는다.*

- 니체

5장에서 본 RNN은 순환 경로를 포함하며 과거의 정보를 기억할 수 있었습니다. 구조가 단순하여 구현도 쉽게 할 수 있었죠. 하지만 안타깝게도 성능이 좋지 못합니다. 그 원인은 (많은 경우) 시계열 데이터에서 시간적으로 멀리 떨어진, 장기^{long term} 의존 관계를 잘 학습할 수 없다는 데 있습니다.

요즘에는 앞 장의 단순한 RNN 대신 LSTM이나 GRU라는 계층이 주로 쓰입니다. 실제로 그냥 ‘RNN’이라고 하면 앞 장의 RNN이 아니라 LSTM을 가리키는 경우도 흔합니다. 오히려 앞 장의 RNN을 명시적으로 가리킬 때 ‘기본(적인) RNN’이라 하기도 하죠.

LSTM이나 GRU에는 ‘게이트^{gate}’라는 구조가 더해져 있는데, 이 게이트 덕분에 시계열 데이터의 장기 의존 관계를 학습할 수 있습니다. 이번 장에서는 앞 장에서 설명한 기본 RNN의 문제점을 알아보고, 이를 대신하는 계층으로써 LSTM과 GRU와 같은 ‘게이트가 추가된 RNN’을 소개합니다. 특히 LSTM의 구조를 시간을 들여 차분히 살펴보고, 이 구조가 ‘장기 기억’을 가능하게 하는 메커니즘을 이해해봅니다. 그리고 LSTM을 사용해 언어 모델을 만들어, 실제로 데이터를 잘 학습하는 모습도 보여드리겠습니다.

* 옮긴이_ 영어 원문 “Blessed are the forgetful : for they get the better even of their blunders”를 일본에서는 이렇게 번역합니다. 우리는 “망각하는 자는 복이 있나니, 자신의 실수조차 잊기 때문이다”로 번역하는데, 일본은 조금 다른 의미로 해석하나 봅니다. 어느 쪽이 니체의 뜻에 가깝냐와는 별개로, 이번 장의 내용은 일본에서 쓰는 표현과 어울리기 때문에 그 표현 그대로를 우리말로 옮겼습니다.

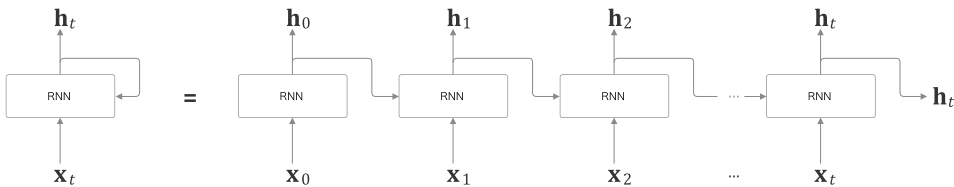
6.1 RNN의 문제점

앞 장에서 설명한 RNN은 시계열 데이터의 장기 의존 관계를 학습하기 어렵습니다. 그 원인은 BPTT에서 기울기 소실 혹은 기울기 폭발이 일어나기 때문입니다.* 이번 절에서는 앞 장에서 배운 RNN 계층을 복습하고, 뒤이어 RNN 계층이 장기 기억을 제대로 처리하지 못하는 이유를 사례를 들어 설명할 것입니다.

6.1.1 RNN 복습

RNN 계층은 순환 경로를 갖고 있습니다. 그리고 그 순환을 펼치면 다음 그림과 같이 옆으로 길게 뻗은 신경망이 만들어집니다.

그림 6-1 RNN 계층: 순환을 펼치기 전과 후

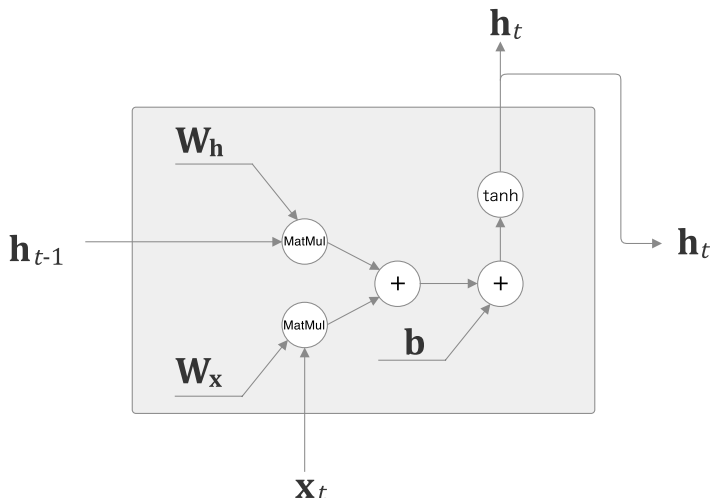


[그림 6-1]에서 보듯, RNN 계층은 시계열 데이터인 \mathbf{x}_t 를 입력하면 \mathbf{h}_t 를 출력합니다. 이 \mathbf{h}_t 는 RNN 계층의 **은닉 상태**라고 하여, 과거 정보를 저장합니다.

RNN의 특징은 바로 이전 시각의 은닉 상태를 이용한다는 점입니다. 이렇게 해서 과거 정보를 계승할 수 있게 되죠. 이때 RNN 계층이 수행하는 처리를 계산 그래프로 나타내면 [그림 6-2]처럼 됩니다.

* 울림이는 기울기 소실은 역전파의 기울기 값이 점점 작아지다가 사라지는 현상을 말하며, 기울기 폭발은 반대로 매우 큰 수가 되는 현상입니다. 두 경우 모두 학습이 제대로 이뤄지지 않게 됩니다.

그림 6-2 RNN 계층의 계산 그래프(MatMul 노드는 행렬 곱을 나타냄)



[그림 6-2]처럼 RNN 계층의 순전파에서 수행하는 계산은 행렬의 곱과 합, 그리고 활성화 함수인 \tanh 함수에 의한 변환으로 구성됩니다. 이상이 앞 장에서 본 RNN 계층입니다. 이어서 이 RNN 계층이 안고 있는 문제, 즉 장기 기억에 취약하다는 문제를 살펴보겠습니다.

6.1.2 기울기 소실 또는 기울기 폭발

언어 모델은 주어진 단어들을 기초로 다음에 출현할 단어를 예측하는 일을 합니다. 앞 장에서는 RNN을 사용해 언어 모델을 구현했습니다(RNNLM이라 불렀죠). 이번 절에서는 RNNLM의 단점을 확인하는 차원에서 [그림 6-3]의 문제를 다시 한번 생각해보겠습니다.

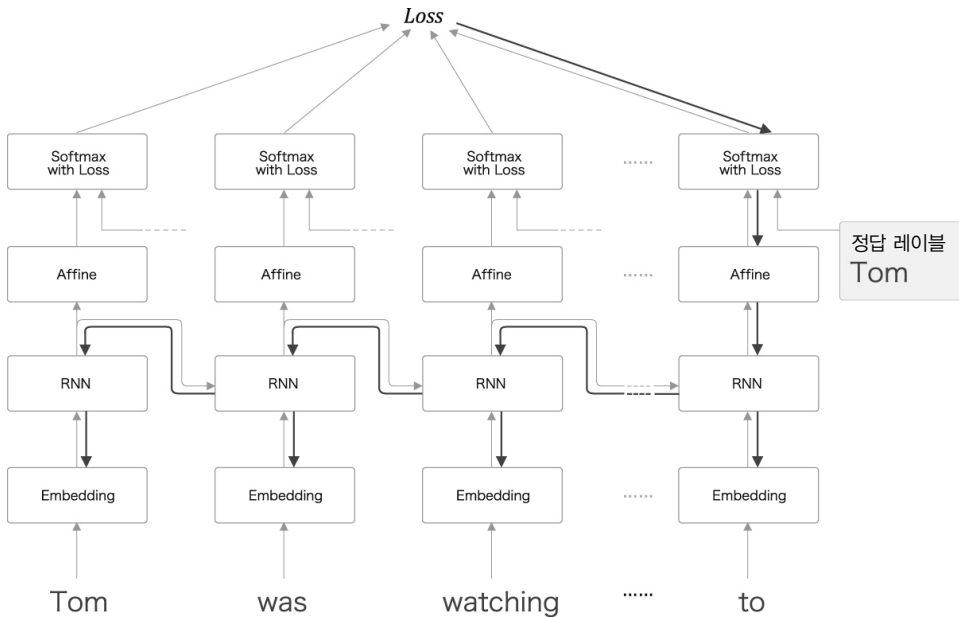
그림 6-3 “?”에 들어갈 단어는?: (어느 정도의) 장기 기억이 필요한 문제의 예

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

앞에서도 나왔듯이 “?”에 들어가는 단어는 “Tom”입니다. RNNLM이 이 문제에 올바르게 답하려면, 현재 맥락에서 “Tom이 방에서 TV를 보고 있음”과 “그 방에 Mary가 들어옴”이란 정보를 기억해둬야 합니다. 다시 말해 이런 정보를 RNN 계층의 은닉 상태에 인코딩해 보관해둬야 합니다.

그럼 이 예를 RNNLM 학습의 관점에서 생각해봅시다. 여기에서는 정답 레이블로 “Tom”이라는 단어가 주어졌을 때, RNNLM에서 기울기가 어떻게 전파되는지를 살펴보겠습니다. 물론, 학습은 BPTT로 수행합니다. 따라서 정답 레이블이 “Tom”이라고 주어진 시점으로부터 과거 방향으로 기울기를 전달하게 됩니다. 그림으로는 [그림 6-4]처럼 됩니다.

그림 6-4 정답 레이블이 “Tom”임을 학습할 때의 기울기 흐름



[그림 6-4]에서와 같이 정답 레이블이 “Tom”임을 학습할 때 중요한 것이 바로 RNN 계층의 존재입니다. RNN 계층이 과거 방향으로 ‘의미 있는 기울기’를 전달함으로써 시간 방향의 의존 관계를 학습할 수 있는 것이죠. 이때 기울기는 (원래대로라면) 학습해야 할 의미가 있는 정보가 들어 있고, 그것을 과거로 전달함으로써 장기 의존 관계를 학습합니다. 하지만 만약 이 기울기가 중간에 사그라들면(거의 아무런 정보도 남지 않게 되면) 가중치 매개변수는 전혀 갱신되지 않게 됩니다. 즉, 장기 의존 관계를 학습할 수 없게 됩니다. 안타깝지만, 현재의 단순한 RNN 계층에서는 시간을 거슬러 올라갈수록 기울기가 작아지거나(기울기 소실) 혹은 커질 수 있으며(기울기 폭발), 대부분 둘 중 하나의 운명을 건게 됩니다.

RNN을 사용한 문장 생성

세상에 완벽한 문장은 없어,

완벽한 절망이 없듯이

— 무라카미 하루키, 『바람의 노래를 들어라』

5장과 6장에서는 RNN과 LSTM의 구조와 구현을 자세하게 살펴봤습니다. 바야흐로 우리는 이 개념들을 구현 수준에서 이해하게 된 것이죠. 이번 장에서는 지금까지의 성과(RNN과 LSTM)가 꽃을 피웁니다. LSTM을 이용해 재미있는 애플리케이션을 구현해볼 것이기 때문이죠.

이번 장에서는 언어 모델을 사용해 ‘문장 생성’을 수행합니다. 구체적으로는 우선 말뭉치를 사용해 학습한 언어 모델을 이용하여 새로운 문장을 만들어냅니다. 그런 다음 개선된 언어 모델을 이용하여 더 자연스러운 문장을 생성하는 모습을 선보이겠습니다. 여기까지 해보면 ‘AI로 글을 쓰게 한다’라는 개념을 (간단하게라도) 실감할 수 있을 겁니다.

여기서 멈추지 않고 seq2seq라는 새로운 구조의 신경망도 다룹니다. seq2seq란 “(from) sequence to sequence (시계열에서 시계열로)”를 뜻하는 말로, 한 시계열 데이터를 다른 시계열 데이터로 변환하는 걸 말합니다. 이번 장에서는 RNN 두 개를 연결하는 아주 간단한 방법으로 seq2seq를 구현해볼 겁니다. 이 seq2seq는 기계 번역, 챗봇, 메일의 자동 답신 등 다양하게 응용될 수 있습니다. 이 간단하면서 영리하고 강력한 seq2seq를 이해하고 나면 딥러닝의 가능성이 더욱 크게 느껴질 것입니다!

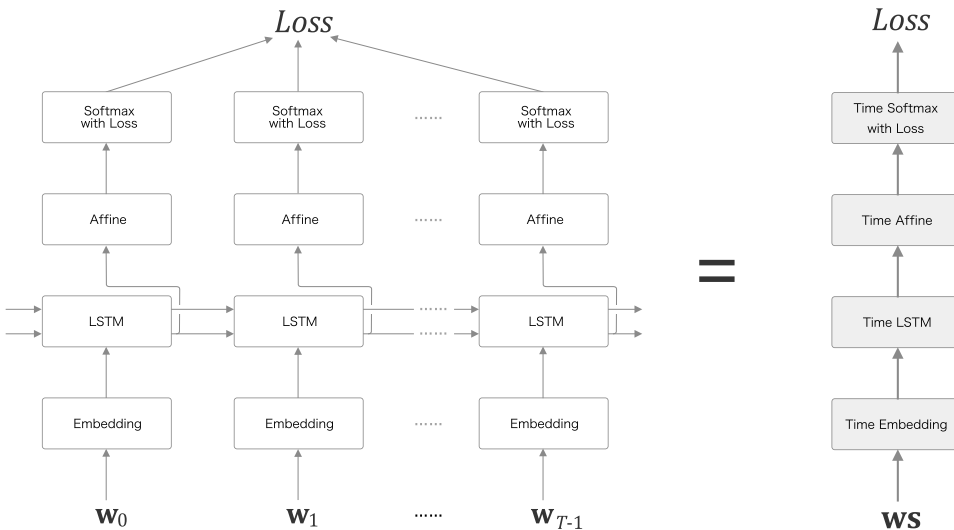
7.1 언어 모델을 사용한 문장 생성

지금까지 여러 장에 걸쳐서 언어 모델을 다뤄왔습니다. 다시 말하지만, 언어 모델은 다양한 애플리케이션에서 활용할 수 있습니다. 대표적인 예로는 기계 번역, 음성 인식, 문장 생성 등이 있죠. 이번 절에서는 언어 모델로 문장을 생성해보려 합니다.

7.1.1 RNN을 사용한 문장 생성의 순서

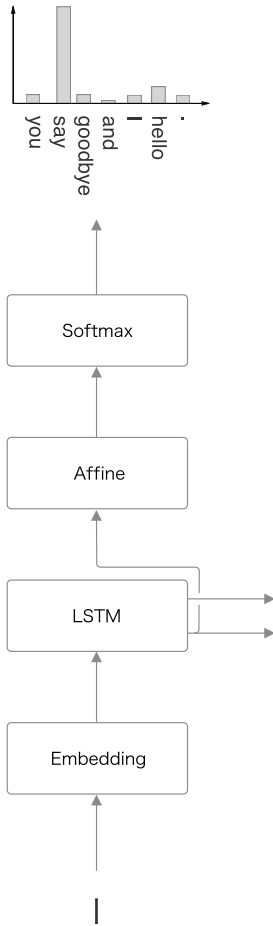
앞 장에서는 LSTM 계층을 이용하여 언어 모델을 구현했는데, 그 모델의 신경망 구성은 [그림 7-1]처럼 생겼었습니다. 그리고 시계열 데이터를 (T 개분 만큼) 모아 처리하는 Time LSTM과 Time Affine 계층 등을 만들었습니다.

그림 7-1 앞 장에서 구현한 언어 모델: 오른쪽은 시계열 데이터를 한꺼번에 처리하는 Time 계층을 사용했고, 왼쪽은 같은 구성을 펼친 모습



이제 언어 모델에게 문장을 생성시키는 순서를 설명해보지요. 이번에도 친숙한 “you say goodbye and I say hello.”라는 말뭉치로 학습한 언어 모델을 예로 생각하겠습니다. 이 학습된 언어 모델에 “I”라는 단어를 입력으로 주면 어떻게 될까요? 그러면 이 언어 모델은 [그림 7-2]와 같은 확률분포를 출력한다고 합니다.

그림 7-2 언어 모델은 다음에 출현할 단어의 확률분포를 출력한다.

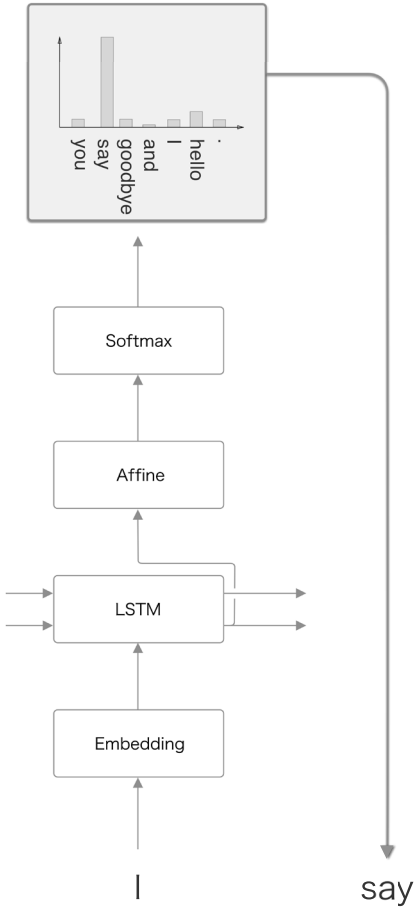


언어 모델은 지금까지 주어진 단어들에서 다음에 출현하는 단어의 확률분포를 출력합니다. [그림 7-2]의 예는 “I”라는 단어를 주었을 때 출력한 확률분포를 보여줍니다. 이 결과를 기초로 다음 단어를 새로 생성하려면 어떻게 해야 할까요?

첫 번째로, 확률이 가장 높은 단어를 선택하는 방법을 떠올릴 수 있을 것입니다. 확률이 가장 높은 단어를 선택할 뿐이므로 결과가 일정하게 정해지는 ‘결정적’인 방법입니다. 또한, ‘확률적’으로 선택하는 방법도 생각할 수 있겠죠. 각 후보 단어의 확률에 맞게 선택하는 것으로, 확률이 높은 단어는 선택되기 쉽고, 확률이 낮은 단어는 선택되기 어려워집니다. 이 방식에서는 선택되는 단어(샘플링 단어)가 매번 다를 수 있습니다.

저는 매번 다른 문장을 생성하도록 하겠습니다. 그 편이 생성되는 문장이 다양해져서 재미있을 겁니다. 그래서 후자의 방법(확률적으로 선택하는 방법)으로 단어를 선택하도록 합니다. 우리의 예로 돌아와서, [그림 7-3]과 같이 “say”라는 단어가 (확률적으로) 선택되었다고 합시다.

그림 7-3 확률분포대로 단어를 하나 샘플링한다.



[그림 7-3]은 확률분포로부터 샘플링을 수행한 결과로 “say”가 선택된 경우를 보여줍니다. 실제로 [그림 7-3]의 확률분포에서는 “say”의 확률이 가장 높기 때문에 “say”가 샘플링될 확률이 가장 높기도 하죠. 다만, 필연적이지는 않고(‘결정적’이 아니고) ‘확률적’으로 결정된다는 점에 주의합시다. 다른 단어들도 해당 단어의 출현 확률에 따라 정해진 비율만큼 샘플링될 가능성이 있다는 뜻입니다.

당신에게 필요한 건 주목^{attention}뿐
- 바즈와니 등이 저술한 논문 제목^[52]

앞 장에서는 RNN을 사용해 문장을 생성해봤습니다. 그리고 2개의 RNN을 연결하여 하나의 시계열 데이터를 다른 시계열 데이터로 변환도 해봤습니다. 우리는 이를 seq2seq라고 하며, 덧셈 같은 간단한 문제를 푸는 데 성공했습니다. 마지막으로, seq2seq에 몇 가지 개선을 적용한 결과, 간단한 덧셈이라면 거의 완벽하게 풀 수 있음을 보았습니다.

이번 장에서는 seq2seq의 가능성 그리고 RNN의 가능성을 한결음 더 깊이 탐험할 겁니다. 어텐션^{Attention}이라는 강력하고 아름다운 기술이 등장할 차례가 온 것이죠. 어텐션은 최근의 딥러닝 분야에서 틀림없이 중요한 기술 중 하나입니다. 우리가 이번 장에서 목표로 하는 것은 어텐션의 구조를 코드 수준에서 이해하는 것, 그리고 실제 문제에 적용하여 그 놀라운 효과를 경험하는 것입니다. 이제 드디어 마지막 장의 문을 엽니다!

8.1 어텐션의 구조

앞 장에서 살펴본 것처럼 seq2seq는 매우 강력한 시스템으로, 다양하게 응용할 수 있습니다. 이번 장에서는 지금까지 배운 seq2seq를 한층 더 강력하게 하는 **어텐션 메커니즘**이라는 아이디어

어를 소개합니다. 이 어텐션이라는 메커니즘 덕분에 seq2seq는 (우리 인간처럼) 필요한 정보에만 ‘주목’할 수 있게 됩니다. 게다가 지금까지의 seq2seq가 안고 있던 문제도 해결할 수 있습니다.

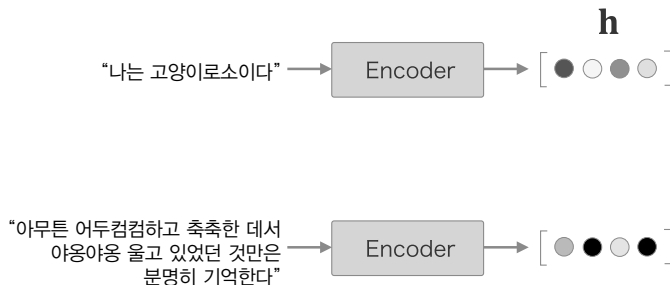
이번 절에서는 현재의 seq2seq가 안고 있는 문제를 먼저 살펴봅니다. 그 후에 어텐션의 구조를 설명하면서, 동시에 구현까지 해보기로 하죠. 그럼 다시 한번, seq2seq가 어떤 일을 수행하는지 떠올려봅시다.

NOTE 앞 장에서도 수행한 seq2seq 개선은 ‘작은 개선’이었습니다. 이와 달리, 지금부터 설명하는 어텐션 기술은 지금까지의 seq2seq가 안고 있던 근본적인 문제를 해결하는 ‘큰 개선’입니다.

8.1.1 seq2seq의 문제점

seq2seq에서는 Encoder가 시계열 데이터를 인코딩합니다. 그리고 인코딩된 정보를 Decoder로 전달하죠. 이때 Encoder의 출력은 ‘고정 길이의 벡터’였습니다. 그런데 실은 이 ‘고정 길이’라는 데에 큰 문제가 잠재해 있습니다. 고정 길이 벡터라 함은 입력 문장의 길이에 관계없이(아무리 길어도), 항상 같은 길이의 벡터로 변환한다는 뜻입니다. 앞 장의 번역 예로 설명하면, 아무리 긴 문장이 입력되더라도 항상 똑같은 길이의 벡터에 밀어 넣어야 합니다(그림 8-1).

그림 8-1 입력 문장의 길이에 관계없이, Encoder는 정보를 고정 길이의 벡터로 밀어 넣는다.



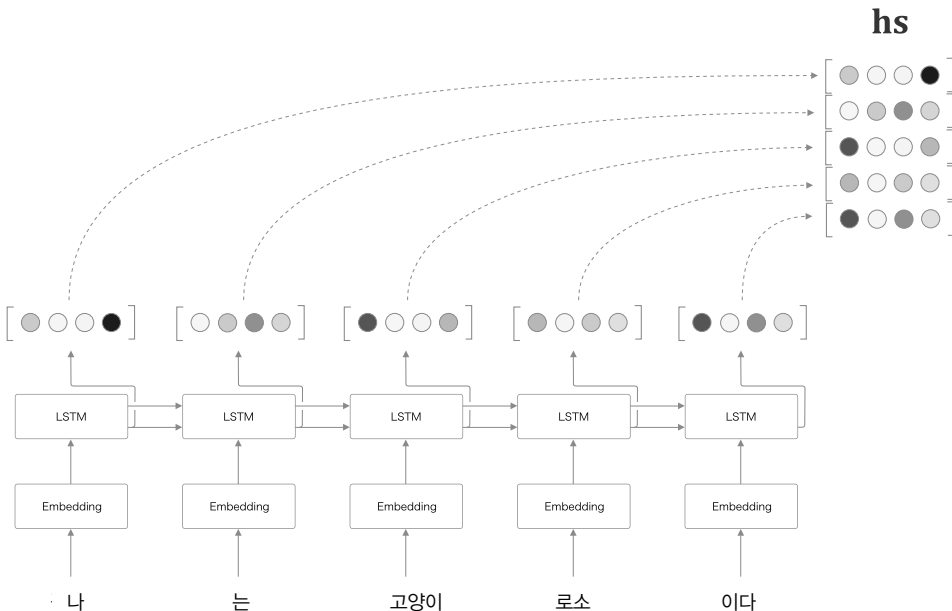
현재의 Encoder는 아무리 긴 문장이라도 고정 길이의 벡터로 변환합니다. 마치 많은 옷가지를 옷장에 옥여넣듯이 억지로 고정 길이의 벡터로 밀어 넣는 것입니다. 하지만 이렇게 하면 곤

한계가 찾아올 것이 분명하겠죠. 결국에는 옷이 옷장에서 빠져나오듯, 필요한 정보가 벡터에 다 담기지 못하게 됩니다. 그러니 seq2seq 개선에 힘써봅시다. 우선은 Encoder를 개선하고, 이어서 Decoder도 개선하겠습니다.

8.1.2 Encoder 개선

지금까지 우리는 LSTM 계층의 마지막 은닉 상태를 Decoder에 전달했습니다. 그러나 Encoder 출력의 길이는 입력 문장의 길이에 따라 바뀌주는 게 좋습니다. 이 점이 Encoder의 개선 포인트입니다. 구체적으로는, [그림 8-2]처럼 시각별 LSTM 계층의 은닉 상태 벡터를 모두 이용하는 것입니다.

그림 8-2 Encoder의 시각별(단어별) LSTM 계층의 은닉 상태를 모두 이용(hs로 표기)

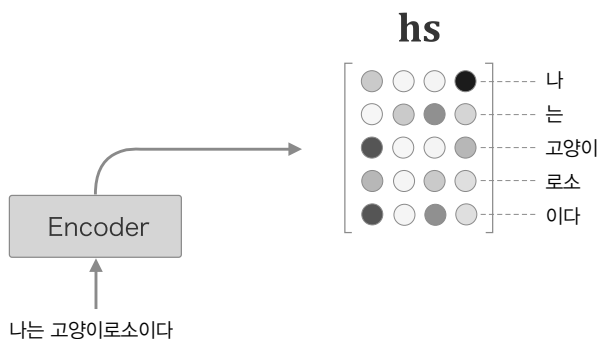


[그림 8-2]처럼 각 시각(각 단어)의 은닉 상태 벡터를 모두 이용하면 입력된 단어와 같은 수의 벡터를 얻을 수 있습니다. [그림 8-2]의 예에서는 5개의 단어가 입력되었고, 이때 Encoder는 5개의 벡터를 출력합니다. 이것으로 Encoder는 '하나의 고정 길이 벡터'라는 제약으로부터 해방됩니다.

NOTE 많은 딥러닝 프레임워크에서는 RNN 계층(혹은 LSTM 계층과 GRU 계층 등)을 초기화할 때, '모든 시각의 은닉 상태 벡터 반환'과 '마지막 은닉 상태 벡터만 반환' 중 선택할 수 있습니다. 케라스를 예로 들면 RNN 계층의 초기화 인수로 `return_sequences=True`로 설정하면 모든 시각의 은닉 상태 벡터를 반환합니다.

그런데 [그림 8-2]에서 주목할 것은 LSTM 계층의 은닉 상태의 '내용'입니다. 시각별 LSTM 계층의 은닉 상태에는 어떠한 정보가 담겨 있을까요? 한 가지 말할 수 있는 것은 각 시각의 은닉 상태에는 직전에 입력된 단어에 대한 정보가 많이 포함되어 있다는 사실입니다. [그림 8-2]에서라면, 예컨대 “고양이” 단어를 입력했을 때의 LSTM 계층의 출력(은닉 상태)은 직전에 입력한 “고양이”라는 단어의 영향을 가장 크게 받습니다. 따라서 이 은닉 상태 벡터는 “고양이”의 ‘성분’이 많이 들어간 벡터라고 생각할 수 있습니다. 이렇게 생각하면, Encoder가 출력하는 **hs** 행렬은 각 단어에 해당하는 벡터들의 집합이라고 볼 수 있겠지요(그림 8-3).

그림 8-3 Encoder의 출력 **hs**는 단어 수만큼의 벡터를 포함하며, 각각의 벡터는 해당 단어에 대한 정보를 많이 포함한다.



WARNING Encoder는 왼쪽에서 오른쪽으로 처리하므로, 방금 전의 “고양이” 벡터에는 정확히 총 3개 단어(“나”, “는”, “고양이”)의 정보가 담겨 있습니다. 그런데 전체의 균형을 생각하여 “고양이” 단어의 ‘ 주변’ 정보를 균형 있게 담아야 할 때도 있을 텐데, 그런 경우엔 시계열 데이터를 양방향으로 처리하는 **양방향 RNN**(혹은 **양방향 LSTM**)이 효과적입니다. 양방향 RNN에 관해서는 나중에 설명하기로 하고, 여기에서는 지금까지처럼 단방향 LSTM을 이용하겠습니다.

직접 구현하면서 배우는 본격 딥러닝 입문서

Deep Learning from Scratch 2

이 책은 『밑바닥부터 시작하는 딥러닝』에서 다루지 못했던 순환 신경망(RNN)을 자연어 처리와 시계열 데이터 처리에 사용하는 딥러닝 기술에 초점을 맞춰 살펴봅니다. 8장 구성으로 전체를 하나의 이야기처럼 순서대로 읽도록 꾸몄습니다. 전편에서 배운 내용을 요약한 신경망 복습을 첫 장에 배치하여 신경망과 파이썬 지식을 어느 정도 갖춘 분이라면 전편을 읽지 않아도 무리 없이 따라올 수 있도록 배려했습니다.

1장 신경망 복습

2장 자연어와 단어의 분산 표현

3장 word2vec

4장 word2vec 속도 개선

5장 순환 신경망(RNN)

6장 게이트가 추가된 RNN

7장 RNN을 사용한 문장 생성

8장 어텐션

이 책은 널리 사용되는 딥러닝 모델을 직접 구현하면서 기본 원리를 체득하는 체험형 입문서입니다. 전편이 딥러닝의 기본 이론을 밑바닥부터 구현하도록 안내했다면 이번에는 자신만의 딥러닝 프레임워크를 구축하는 토대를 만들어줍니다. 파이썬과 넘파이 활용까지 책임지는 훌륭한 지침서입니다. 딥러닝 프레임워크를 연구하는 모든 분께 이 책을 추천합니다.

윤영선, 한남대학교 정보통신공학과 교수

이 책은 RNN 기반 신경망에 대한 개념을 쉽게 설명해주고, 복잡한 응용 모델들을 그림과 예제로 알려줍니다. 시계열을 공부하는 분, RNN 공부를 시작하는 분께 큰 도움이 될 것입니다.

김동성, 삼성 리서치(Samsung Research) 연구원

관련 도서

밑바닥부터
시작하는
딥러닝

텐서플로를
활용한
머신러닝

한 권으로
끝내는
딥러닝
텐서플로

예제 소스 <https://github.com/WegraLee/deep-learning-from-scratch-2>

O'REILLY®

한빛미디어
Hanbit Media, Inc.

인공지능 / 딥러닝



ISBN 979-11-6224-174-5

정가 29,000원