



Topic
Science

Subtopic
Engineering & Technology

Introduction to Machine Learning

Michael L. Littman, PhD
Brown University



Transcript Book



4840 Westfields Boulevard | Suite 500 | Chantilly, Virginia | 20151-2299
[PHONE] 1.800.832.2412 | [FAX] 703.378.3819 | [WEB] www.thegreatcourses.com

LEADERSHIP

PAUL SUIJK	President & CEO
BRUCE G. WILLIS	Chief Financial Officer
JOSEPH PECKL	SVP, Marketing
JASON SMIGEL	VP, Product Development
CALE PRITCHETT	VP, Marketing
MARK LEONARD	VP, Technology Services
DEBRA STORMS	VP, General Counsel
KEVIN MANZEL	Sr. Director, Content Development
ANDREAS BURGSTALLER	Sr. Director, Brand Marketing & Innovation
KEVIN BARNHILL	Director of Creative
GAIL GLEESON	Director, Business Operations & Planning

PRODUCTION TEAM

TRISH GOLDEN	Producer
JAY TATE	Content Developer
ABBY INGHAM LULL	Associate Producer
BRIAN SCHUMACHER	Graphic Artist
OWEN YOUNG	Managing Editor
COURTNEY WESTPHAL	Sr. Editor
CHRISTIAN MEEKS	Editor
CHARLES GRAHAM	Assistant Editor
EDDIE HARTNESS	Audio Engineer
JIM PETIT	Director & Camera Operator

PUBLICATIONS TEAM

FARHAD HOSSAIN	Publications Manager
BLAKELY SWAIN	Senior Copywriter
TIM OLABI	Graphic Designer
JESSICA MULLINS	Proofreader
ERIKA ROBERTS	Publications Assistant
JENNIFER ROSENBERG	Fact-Checker
WILLIAM DOMANSKI	Transcript Editor & Fact-Checker

Copyright © The Teaching Company, 2020

Printed in the United States of America

This book is in copyright. All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted, in any form, or by any means (electronic, mechanical, photocopying, recording, or otherwise), without the prior written permission of The Teaching Company.

Michael L. Littman, PhD

**Royce Family Professor
of Teaching Excellence
in Computer Science**

Brown University

Michael L. Littman is the Royce Family Professor of Teaching Excellence in Computer Science at Brown University. He earned his bachelor's and master's degrees in Computer Science from Yale University and his PhD in Computer Science from Brown University.



Professor Littman's teaching has received numerous awards, including the Robert B. Cox Award from Duke University, the Warren I. Susman Award for Excellence in Teaching from Rutgers University, and both the Philip J. Bray Award for Excellence in Teaching in the Physical Sciences and the Distinguished Research Achievement Award from Brown University. His research papers have been honored for their lasting impact, earning him the Association for the Advancement of Artificial Intelligence (AAAI) Classic Paper Award at the Twelfth National Conference on Artificial Intelligence and the International Foundation for Autonomous Agents and Multiagent Systems Influential Paper Award at the Eleventh International Conference on Machine Learning.

Professor Littman is the codirector of the Humanity Centered Robotics Initiative at Brown University. He served as program cochair for the 26th International Conference on Machine Learning, the 27th AAAI Conference on Artificial Intelligence, and the 4th Multidisciplinary Conference on Reinforcement Learning and Decision Making. He is a fellow of the AAAI, the Association for Computing Machinery, and the Leshner Leadership Institute for Public Engagement with Science.

Professor Littman gave two TEDx talks on artificial intelligence, and he appeared in the documentary *We Need to Talk about A.I.* He also hosts a popular YouTube channel with computer science research videos and educational music videos. ■

TABLE OF CONTENTS

INTRODUCTION

Professor Biography	i
Course Scope	1

LESSON GUIDES

Lesson 01	
Telling the Computer What We Want	4
Lesson 02	
Starting with Python Notebooks and Colab	28
Lesson 03	
Decision Trees for Logical Rules	44
Lesson 04	
Neural Networks for Perceptual Rules	59
Lesson 05	
Opening the Black Box of a Neural Network	76
Lesson 06	
Bayesian Models for Probability Prediction	92
Lesson 07	
Genetic Algorithms for Evolved Rules	110

TABLE OF CONTENTS

Lesson 08	Nearest Neighbors for Using Similarity	127
Lesson 09	The Fundamental Pitfall of Overfitting	144
Lesson 10	Pitfalls in Applying Machine Learning	161
Lesson 11	Clustering and Semi-Supervised Learning	180
Lesson 12	Recommendations with Three Types of Learning	200
Lesson 13	Games with Reinforcement Learning	219
Lesson 14	Deep Learning for Computer Vision	238
Lesson 15	Getting a Deep Learner Back on Track	253
Lesson 16	Text Categorization with Words as Vectors	271
Lesson 17	Deep Networks That Output Language	291
Lesson 18	Making Stylistic Images with Deep Networks	310
Lesson 19	Making Photorealistic Images with GANs	332

Lesson 20	
Deep Learning for Speech Recognition	350
Lesson 21	
Inverse Reinforcement Learning from People	370
Lesson 22	
Causal Inference Comes to Machine Learning	389
Lesson 23	
The Unexpected Power of Over-Parameterization	408
Lesson 24	
Protecting Privacy within Machine Learning	429
Lesson 25	
Mastering the Machine Learning Process	449

SUPPLEMENTARY MATERIALS

Bibliography	470
Answers	476
Glossary	488
Software Libraries	497
Image Credits	504

INTRODUCTION TO MACHINE LEARNING

COURSE SCOPE

Machine learning is becoming a household phrase, with references in popular media growing right along with its expanding influence in real-world applications. Machine learning algorithms are at the core of many efforts to deploy computer automation into society: Improvements in face recognition, voice command apps, and even the deployment of police patrols are all being driven by machine learning.

The goal of this course is to give you a broad introduction to the concepts and mechanics of creating machine learning systems, how they are being used in the real world, and how they can sometimes fall short of our aspirations. In addition, the video lesson will walk you through working examples of machine learning software so that you can see how systems are built and create your own working machine learning programs.

The first third of the course introduces you to fundamental approaches to machine learning: the problem of creating rules from data. You will get experience working with Python notebooks as a way to develop and run machine learning programs.

You'll first use this platform to explore decision trees, an approach to machine learning that produces logical rules from data. Then, you'll dive into neural networks—the leading approach to processing perceptual data—using them to recognize handwritten digits. Next, you'll see how Bayesian models work as they represent and reason about the probabilities of events and apply them to separate spam messages from normal messages.

Genetic algorithms leverage ideas from natural selection to construct rules given minimal background knowledge about the kind of rules that will be most successful, and you'll apply this idea to the search for high-accuracy rules. Perhaps the simplest machine learning algorithm to train is k -nearest neighbors, which you'll use to sniff out virus-infected programs. Confronting and unifying all of the main approaches to machine learning is the issue of overfitting, which you'll discover is fundamental to any use of machine learning in practice.

The middle of the course focuses on high-impact applications of machine learning methods, especially deep learning. And to understand successful applications, you'll first examine the context in which machine learning methods are deployed and how the context can make otherwise-accurate methods return misleading and sometimes harmful results. You will examine fundamental trade-offs in unsupervised clustering and use k -means clustering to improve classification performance when labels are scarce.

You'll look at recommendation—one of the biggest-impact applications of machine learning—and see how it can leverage unsupervised, supervised, and reinforcement feedback. Reinforcement learning is the key idea behind applications of machine learning to board games and video games, where some strategies are being created that are as good as, or better than, the best human players.

Neural network representations of text map words to vectors that capture the similarities in how words are used. You'll apply an analysis of a billion-word text collection to more efficiently categorize the topic of passages of text. Transformer networks can go further, outputting natural-sounding language, especially when they have hundreds of millions of trained parameters.

Next, you'll learn how networks can make pictures, first by mimicking stylistic features and then by synthesizing entire images from scratch via generative adversarial networks. The final stop on the tour of the most impactful applications of machine learning will be the problem of speech recognition: learning to recognize words from audio signals.

In the final third of the course, you will look beyond established algorithms and applications and contemplate possible future impacts of machine learning. You'll first take a look at what a computer can do to understand what you want using inverse reinforcement learning, which frames the problem as turning observations of behavior into predictions of preferences.

COURSE SCOPE

A drawback of the way machine learning has been described up to this point is that it has no explicit motivation to learn *why* models of the data it encounters. You'll discover how causal learning is different and what steps are being taken to make machine learners that can change their predictions in the face of changing data distributions.

The story of overfitting becomes considerably more nuanced in the context of deep neural networks. You'll see how neural networks can be over-parameterized yet resist overfitting and how trained networks can be pruned to work with fewer parameters. You will learn about keeping machine learning data private and secure using homomorphic encryption and differential privacy. And the final lesson will turn machine learning upon itself, using meta-learning to build machine learning systems automatically.

Throughout the course, you will see how representational spaces, loss functions, and optimizers underpin all approaches to machine learning, providing a lens that will help you make sense of even the most advanced and powerful algorithms being developed. ■

Try It Yourself

Each lesson of this course includes resources to follow along and try for yourself, available via

www.TheGreatCourses.com/MachineLearning

and more directly (as explained in **Lesson 02**) via Google Colab, from a code repository created for this course on GitHub by Professor Littman:

https://colab.research.google.com/github/mlittmans/great_courses_ml/blob/master/L02.ipynb

LESSON 01

TELLING THE COMPUTER WHAT WE WANT

Machine learning systems have begun doing all kinds of things that, until recently, were just science fiction. For example, machine learning can transcribe what you're saying while you say it, predict what word you might say next, translate what you're saying into other languages, and even use a simulation of your voice to speak on your behalf. There are a variety of inputs and outputs, but the glue that makes each work is machine learning.

What Exactly Is Machine Learning?

References to the term **machine learning** in *The New York Times* had been almost nonexistent until 2005 but then increased about 33% per year from 2014 to 2018. Yet many of the ideas that form the foundations of machine learning have been under development for decades.

The term *machine learning* first appeared in print in 1959. It described a computer program for playing checkers that could improve its ability over time.

Machine learning is a subarea of artificial intelligence, which itself is a subarea of computer science. The field of machine learning has grown up as a central topic within computer science from the beginning, going all the way back to the pioneering work of Alan Turing in 1950.

Machine learning is really just a different way of creating computer programs. In traditional programming, we tell the computer what to do. That's still part of machine learning. But at the heart of machine learning, we reverse the logic of traditional programming. Instead of giving the computer **rules**, or **algorithms**, telling it what to do with **data**, in machine learning we start by describing the kind of output we want, exemplified in a dataset, and the machine learner then creates rules—algorithms.

Turing envisioned learning as being a key tool for making machines that could pass his so-called imitation game, now known as the Turing test, and be viewed as intelligent.

Traditional Programming:

Rules + Data → Desired Output

Machine Learning:

Desired Output + Data → Rules

In essence, we give the computer a template about what it should look for, plus guidance about how to use the data as an example of what we want. What's so revolutionary is that a computer uses just those inputs to write its own program. Instead of humans giving a rule to the computer, we help the computer just enough so that it can find a rule on our behalf.

Elements of Machine Learning

Suppose a machine learning approach is being developed that uses heart sounds to identify potential heart problems. Classically, doctors use stethoscopes to listen to the sounds made by their patients' hearts. Subtle changes in the sounds can indicate a problem with the heart valves.

How could we automate this kind of heart exam? We would need a digital representation of the heart sounds—for example, we might have audio clips from a digital stethoscope. And we would need a machine learning program to process those recorded sounds to infer whether everything is functioning normally.

Because machine learning is such a general tool, the ways to use it are only limited by our creativity.

As you might imagine, it's really difficult for people to learn to identify the sound cues that suggest trouble. Doctors who do it well examine many patients and practice throughout their careers. But even for someone who's an expert, it's very hard to capture knowledge about heart sounds in the form of an explicit rule you can express in a traditional computer program.

With machine learning, we aim to avoid the difficulty of explicitly telling the computer what these rules are. Instead of telling the computer what to do, we tell the computer what we want—by deciding only on a basic format for the rule. For heart sounds, this format might be a particular kind of mathematical analysis of the waveforms that make up the sound.

The format can be thought of as a kind of space of possible rules—called the representational space. And as the designer of a machine learning system, you get to choose how to structure it.

Five Schools of Machine Learning

Central problem	Key algorithm(s)
Reasoning with symbols	Decision trees
Analyzing perceptual information	Neural networks, perceptron, deep networks
Managing uncertainty	Bayesian networks
Discovering new structures	Genetic programs
Exploiting similarities	Nearest neighbors

When making your choice, it's good for the representational space to be big and inclusive so that the learning system can handle a broad set of possibilities. But it's also good for the chosen space to be small and precise because that makes it possible to learn with much less time and data.

There are gazillions of variations to choose from, but there are five main perspectives, or schools of thought, in machine learning. In *The Master Algorithm*, Pedro Domingos calls these perspectives “the five tribes of machine learning” because they have been championed by different groups of researchers. Nonetheless, the five perspectives can and should work together. Each focuses on confronting a distinct central problem within machine learning and provides distinct algorithms for addressing it.

Perhaps the most straightforward of the main perspectives uses logical representations to capture rules. A very popular representational space that has this logical form is decision trees.

But for many problems, explicit rules are hard to come by, so a second perspective on machine learning is inspired by how individual brain cells work together to create a **neural network**. The neural networks of machine learning excel at processing perceptual information like sounds and images.

Another issue in machine learning, as emphasized by the Bayesian school of thought, is that “rules” often have some uncertainty, even when they do a good job. So a third approach structures the representational space as a probabilistic model—one that captures statistical dependencies in its data.* Unlike decision trees or neural networks, **Bayesian networks** assign probabilities to different outcomes and sort through combinations of symptoms to reach a probability-based diagnosis.

If we don’t know the structure of a problem well enough to choose a probabilistic model, we can use **genetic algorithms**, the fourth perspective. Inspired by how biological organisms evolve through natural selection, genetic algorithms learn program structures to capture our rules.

The fifth perspective uses similarities of cases without building up a more global view of how rules should be structured.[†]

The key point to keep in mind is that any representational space has some generality but also restrictions. The representational space is the “box” that the machine learning program is going to think inside of.

Once we choose a representational space, we next need a lot of data. For our heart sounds example, we’d need to have examples of digitized heart sounds. And we’d want experts to provide gold-standard labels indicating which audio clips correspond to normal heart sounds and which correspond to heart problems.

With this data in hand, we next need a way to evaluate potential rules to identify which of them are consistent with what the data is trying to tell us.

* In some types of medical diagnoses, such as with liver disorders, it’s important to integrate evidence across a wide variety of complicated symptoms.

[†] For example, a medical system can judge how similar your symptoms are to previous patients and use those judgments about similarity to provide a diagnosis or treatment.

For heart sounds, we could have the machine learner come up with its own candidate rules within a neural network representational space. Then, we want to score how well each rule is doing across an entire collection of example heart sounds.

In machine learning, the scoring function for evaluating potential rules is commonly called a **loss function**, because it defines an objective to be minimized with respect to the data.

One important component in many loss functions is a program that calculates how accurate the rule is on the examples. For each example in our dataset, the loss increases each time the rule predicts something different from what our dataset gives as the correct answer.

The point of defining a scoring function for rules—and this idea is central to machine learning—is that the scores can be used indirectly to *define* rules. In essence, the learner tries a rule, consults the score for feedback, revises the rule, and then tries again—over and over. Writing a program to calculate a scoring function is usually much simpler than writing the rules explicitly.

Another common component of loss functions is a penalty for rules that are larger or more complex. Without this encouragement, the learner might create supercomplicated rules that exactly mimic every tiny variation in the data—called **overfitting**—and fail to diagnose new patients correctly.

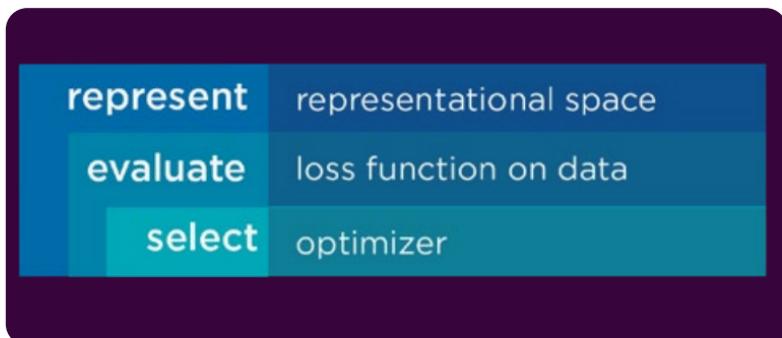
Once the loss function assigns scores to evaluate each candidate rule, an **optimizer** selects a rule with the best-available score from the space of candidate rules, given our dataset.

A straightforward approach would be to make an optimizer that runs through the set of all rules in the representational space, applies the loss function to each candidate rule, and simply reports the rule with the best score it finds.

In practice, however, representational spaces have billions upon billions of rules in them. We just do not have the time to sift through all of them.

What comes to the rescue is that machine learning optimizers can make educated guesses on how to improve the rules directly. These shortcuts allow optimizers to cleverly rifle through astronomically huge representational spaces and return an “approximately best” answer quickly.

What a machine learning program most often does, as a result, is approximately optimize loss over representational space.



The basic recipe for creating a machine learning program is to

- ◆ choose a representational space,
- ◆ design a loss function that uses data to score potential rules, and
- ◆ run an optimizer, which selects the rule in the representational space with the approximately best score.

The upshot of this recipe is that: Machine learning turns data into rules.

Imperative versus Declarative Programming

Traditional programming is sometimes called imperative because we issue specific commands that the computer has to follow. We tell it exactly what to do.

In machine learning, we do not tell the program what to do. We simply declare what the program should prefer by giving the computer code and examples that are used to assess how good its proposed rules are.

Instead of imperative, the approach to programming in machine learning is declarative: We declare that the program should make a good choice, and we give it input that defines what better and worse choices are, but we don't tell the computer what to do.

This distinction shouldn't be overstated, though. On a more fundamental level, we are always telling the computer what to do, so we're always writing imperative programs. But the thought process in machine learning has this backward aspect to it, and it can take some practice before it starts to feel natural.

Yet the more you are able to solve problems this way, the more you'll see that it's quite empowering. We can offload to the computer the search for rules that we can't express concretely.

Because of this backward structure, a machine learning program has to do a lot of work up front to find a way of meeting the demands of our declaration. Not surprisingly, the hardware needed to run a machine learning program can be much more than what is needed to run a traditional program.

Unsupervised versus Supervised Learning

In addition to the trio of elements a machine learning program uses to tell a computer what we want, the kind of data providing feedback to the loss function of the machine learning program is also important.

The heart sounds problem is an example of the archetype of machine learning feedback, known as **supervised learning**, in which expert humans provide data that includes both inputs and their corresponding outputs, or labels.

In supervised learning, the feedback is like premade flash cards. On one side of the flash card is the input—"What's another name for the scoring function in machine learning?"—and the other side is the target output—"the loss function." The machine learner has to learn to associate what's on the front of the card with what's on the back of it.

Ideally, the machine learner isn't just memorizing what's on the cards but is identifying a rule that makes it possible to give correct responses to cards it has never seen before.

The second class of machine learning is **unsupervised learning**, which is more like study hall—Independent reading and taking notes.

An unsupervised machine learner is not merely the logical opposite of a supervised learner. An unsupervised learner is one that tries to re-represent the data it is given in a more succinct way.

In computer science, the idea of representing the same information in less space is called compression. In machine learning, a more succinct output is achieved by clustering, or sometimes with dimension reduction.

However, getting enough data for fully supervised learning is hard, and unsupervised learning without any labeled examples is not directed to solve a particular problem. So there is also **semi-supervised learning**, where we leverage some examples to shed light on the rest.

There is another quasi-supervised class called **reinforcement learning**, which is partway between supervised and unsupervised learning. Reinforcement-learning algorithms expect an intermediate kind of labeling. They don't require complete answers like supervised learning, but they are also not as “hands-off” as unsupervised learning. Instead, the labels are evaluations, indicating how good a given answer is. They are like the evaluations captured by the overall loss function—but now with scoring extended down to the level of specific decisions, or sequences of decisions, as well.

You can think of reinforcement learning as a bit like having access to an Olympic judge or a writing coach. You are not being told how to perform or what to write. There is no longer an expert label telling you what to do. You are just getting feedback on how well you did it.

This kind of feedback is a great match for problems in which we can automate the process of assessing how well a system did but not what it could have done better. Historically, games have been a great application of reinforcement learning.

Reinforcement learning is a good candidate anytime the rule for determining who wins is easy to apply but the selection of the best strategy is hard.

Try It Yourself

Follow along with the video lesson via the Python code:

Auxiliary Code for Lesson:

[L01aux.ipynb](#)

Python Libraries Used:

keras.preprocessing.image: Prepares raw images for processing by a neural network.

numpy: Mathematical functions over general arrays.

Key Terms

algorithm: A process to be carried out for solving a specific problem by a computer. A learning algorithm (for example, ID3 for decision trees) is one that takes a dataset of labeled instances and produces a rule. An optimization algorithm (for example, backpropagation for gradient descent) is one that searches a space to find a solution with low loss.

Bayesian network: A graphical representation of a joint probability distribution over random variables.

data: The information a machine learning algorithm works with. It is often divided into training data, testing data, and sometimes also validation data.

genetic algorithm: An approach to optimizing a function inspired by Charles Darwin's principle of natural selection.

loss function: A measure of how "incorrect" a rule is. The loss function based on data can be used to guide the construction of better and better rules in the context of machine learning. Examples include mean squared error for regression data; cross entropy and Kullback-Leibler divergence for categorical data; and hinge loss function for -1/1 binary classification.

machine learning: The process of using data to construct rules. The main settings of machine learning are supervised learning, unsupervised learning, and reinforcement learning.

network: A collection of items with a directed set of connections between them. Examples include neural networks and Bayesian networks.

neural network: A representational space for rules consisting of activations propagating from input to output. One of five long-standing approaches to machine learning, which often leverage gradient descent for training.

optimizer: A program or algorithm for solving an optimization problem.

overfitting: A problem that arises when a rule is learned from a large rule space using too little data, resulting in poor performance on unseen examples.

reinforcement learning: The branch of machine learning concerned with generating behavior by interacting with an environment with the goal of maximizing reward given evaluative feedback.

rule: A function that takes an instance and produces an output, whether a Boolean, category, number, or vector. Used to refer to the output of a machine learning algorithm.

semi-supervised learning: A framework for machine learning that combines labeled and unlabeled data.

supervised learning: The problem of learning an input-to-output mapping, or rule, from examples.

unsupervised learning: The branch of machine learning concerned with learning the relationships between unlabeled input instances, often by clustering those instances by similarity or by finding a reduced dimensional representation of the instances.

READING

Domingos, *The Master Algorithm*, chaps. 1-2.

Mitchell, *Machine Learning*, chap. 1.

Russell and Norvig, *Artificial Intelligence*, secs. 19.1-19.2.

QUESTIONS

1. For what kinds of problems would you develop a rule using a machine learning approach instead of writing the rule by hand?
2. What are the three main kinds of feedback studied in machine learning, and how do they differ?
3. This lesson includes pseudocode for a rule in a representational space in which a pixel's color is compared to a prototype color and is considered "green" if the distance is less than a threshold.

```
if distance(c, studio_Image[x][y]) <= d:
```

Write a Python function that would compute this distance. Assume the input is two three-element lists representing the amount of red, green, and blue in each of two pixels. The distance between two points in three-dimensional space is the square root of the sum of the squares of the difference between their components.

Python Libraries:

math: Mathematical functions.

Answers on page 476

Telling the Computer What We Want

Lesson 1 Transcript

Machine-learning systems have begun doing all kinds of stuff that, until recently, was just science fiction. Here's some amazing machine-learning tricks. Machine learning can transcribe what I'm saying while I'm saying it; predict what word I might say next; make me look older or younger; write the background music that you're hearing; find a face in this image, even while it's moving around; label objects in this image; translate what I'm saying into other languages.

[Lyrebird:] Or machine learning can use a simulation of my voice to talk on my behalf.

Fancy, huh? There's a variety of different inputs and outputs, but the glue that makes each work is machine learning—specifically, a recent flavor referred to as deep learning.

A lot of the visible progress has been taking place since around 2015. That's when a variety of deep-learning applications began attracting broad attention. References to the term *machine learning* in *The New York Times* had been almost nonexistent until 2005, but then increased 33% per year from 2014 to 2018. And yet many of the ideas that form the foundations of machine learning have been under development for decades.

The term *machine learning* first appeared in print in 1959. It described a computer program for playing checkers that could improve its ability over time. Machine learning is a sub-area of artificial intelligence, which itself is a sub-area of computer science.

And the field of machine learning has grown up as a central topic within computer science from the very beginning, going all the way back to the pioneering work of Alan Turing in 1950. Turing envisioned learning as being a key tool for making machines that could pass his imitation game and be viewed as intelligent.

So, what is machine learning, really? It's a different way of creating computer programs. In traditional programming, we tell the computer what to do. That's still part of machine learning. But at the heart of machine learning, we reverse the logic of traditional programming. Instead of giving the computer rules or algorithms telling the computer what to do with data, in machine learning, we go in reverse.

LESSON 01 | TELLING THE COMPUTER WHAT WE WANT

TRANSCRIPT

We start by describing the kind of output we want, exemplified in a dataset, and the machine learner then creates rules or algorithms. Instead of Rules + Data goes to Desired Output, machine learning is Desired Output + Data goes to Rules. In essence, we give the computer a sort of template about what it should look for, plus guidance about how to use the data as an example of what we want. What's so revolutionary is that a computer uses just those inputs to write its own program. Instead of humans giving a rule to the computer, we help the computer just enough so that it can find a rule on our behalf.

Let's talk about an example from medical diagnosis. Suppose a machine-learning approach is being developed that uses heart sounds to identify potential heart problems. Classically, doctors use stethoscopes to listen to the sounds made by their patients' hearts. [heart beating] That's my heart, right there. Subtle changes in the sounds can indicate a problem with the heart valves.

Let's think about automating this kind of heart exam. We would need a digital representation of the heart sounds. We might have audio clips from a digital stethoscope, say. And we would need a machine-learning program to process those recorded sounds to infer whether everything is functioning normally.

As you might imagine, it's really difficult for people to learn to identify the sound cues that suggest trouble. Doctors who do it well examine a lot of patients and practice throughout their careers. But even for someone who's an expert, it's very hard to capture knowledge about heart sounds in the form of an explicit rule you can express in a traditional computer program. It's hard to tell a computer what to do.

With machine learning, we aim to avoid the difficulty of explicitly telling the computer what these rules are. Instead, we tell the computer what we want by deciding only on a basic format for the rule. For heart sounds, this format might be a particular kind of mathematical analysis of the waveforms that make up the sound.

The format can be thought of as a kind of space of possible rules. It could be called the model space or the hypothesis space, but we'll call it the representational space. And as designers of a machine-learning system, you get to choose how to structure it.

When making your choice, it's good for the representational space to be big and inclusive so the learning system can handle a broad set of possibilities. But it's also good for the chosen space to be small and precise because that makes it possible to learn with much less time and data.

There are gazillions of variations to choose from, but there are five main perspectives or schools of thought in machine learning. A 2015 book called *The Master Algorithm* by Pedro Domingos calls these perspectives “the five tribes of machine learning” because they've been championed by different groups of researchers. Nonetheless, the five can and should work together. As we will see, each perspective focuses on confronting a distinct central problem within machine learning and provides distinct algorithms for addressing it.

Perhaps the most straightforward of the five main perspectives uses logical representations to capture rules. A very popular representational space that has this logical form is decision trees. Decision trees are a natural fit for medical diagnosis because they correspond to the if-then branching process doctors are sometimes taught in medical school.

But for many problems, like heart sounds, explicit rules are hard to come by at all. So a second perspective on machine learning is inspired by how individual brain cells work together to create a neural network. The neural networks of machine learning excel at processing perceptual information like sounds and images.

Neural networks, in an earlier form called perceptrons, have been around since the 1950s. But, it was deep neural networks, made up of, say, more than three layers, that have been at the heart of the deep-learning revolution that has garnered so much attention since 2015.

Another issue in machine learning, as emphasized by the Bayesian school of thought, is that “rules” often have some uncertainty, even when they do a good job. So this approach structures the representational space as a probabilistic model—one that captures statistical dependencies in its data.

In some types of medical diagnosis, like with liver disorders, it's important to integrate evidence across a wide variety of complicated symptoms. Unlike decision trees or neural networks, Bayesian probabilistic models assign probabilities to different outcomes and sort through combinations of symptoms to reach a probability-based diagnosis.

LESSON 01 | TELLING THE COMPUTER WHAT WE WANT

TRANSCRIPT

And if we don't know the structure of a problem well enough to use a probabilistic model, we can use genetic algorithms. Inspired by how biological organisms evolve through natural selection, genetic algorithms learn program structures to capture our rules.

Lastly, there is a perspective that uses similarities of cases without building up a more global view of how rules should be structured. For example, a medical system can judge how similar your symptoms are to previous patients and use those judgments about similarity to provide a diagnosis or treatment.

Now, before machine learning came along, the rules used by computers were always generated by people. For example, doctors diagnosing heart failure use a rule called the Framingham Heart Failure Diagnostic Criteria. The representational space that includes the specific Framingham rule is one that assigns symptoms to either of two classes, A and B, and then lists symptom counts that cause the rule to report a "yes" answer. For example, maybe a patient needs at least two A symptoms, or a patient needs one A symptom and two or more Bs to be diagnosed with heart failure. That's the Framingham rule. But another possible rule would require five B symptoms *or* three A symptoms paired with one B symptom.

So lots of different specific rules can be written in this form. But notice all the other conceivable rules that are omitted. Sticking with this format, we can't say that symptoms come in three different categories or that the presence of a symptom in one category cancels out a symptom in another category. The key point to keep in mind is that any representational space has some generality, but also restrictions. The representational space is the "box" that the machine-learning program is going to think inside of.

After choosing a representational space, we'll next need lots of data. For our heart sounds example, we'd need to have examples of digitized heart sounds. And we'd want experts to provide gold-standard labels indicating which audio clips correspond to normal heart sounds and which correspond to heart problems.

With this data in hand, we next need a way to evaluate potential rules to identify which of the potential rules are consistent with what the data is trying to tell us. For heart sounds, we could have the machine learner come up with its own candidate rules within a deep-neural network representational space. Then we want to score how well each rule is doing, across an entire collection of example heart sounds.

In machine learning, the scoring function for evaluating potential rules is commonly called a loss function because it defines an objective to be minimized with respect to the data. Not surprisingly, one important component in many loss functions is a program that calculates how accurate the rule is on the examples. For each example in our dataset, the loss increases each time the rule predicts something different from what our dataset gives as the correct answer.

The point of defining a scoring function for rules—and this idea is central to machine learning—is that the scores can be used indirectly to define rules. In essence, the learner tries a rule, consults the score for feedback, revises the rule, and then tries again. Over and over. And the nice thing is that writing a program to calculate a scoring function is usually much simpler than writing the rules explicitly.

Another common component of loss functions is a penalty for rules that are larger or more complex. Without this encouragement, the learner might create super-complicated rules that exactly mimic every tiny variation in the data—something we call overfitting—and fail to diagnose new patients correctly.

OK, so the learner is generating lots and lots of candidate rules, and the loss function is assigning scores to evaluate each candidate rule. Now, we bring in something called an optimizer. It is this optimizer that selects a rule with the best available score from the space of candidate rules, given our dataset.

A straightforward approach would be to make an optimizer that runs through the set of all rules in the representational space, applies the loss function to each candidate rule, and simply reports the rule with the best score that it finds. But, in practice, representational spaces have billions upon billions upon billions of rules in them. We just do not have the time to sift through all of them.

What comes to the rescue is that machine-learning optimizers can make educated guesses on how to improve the rules directly. These shortcuts allow optimizers to cleverly rifle through astronomically huge representational spaces and return an approximately best answer quickly. So what a machine-learning program most often does is approximately optimize loss over representational space.

LESSON 01 | TELLING THE COMPUTER WHAT WE WANT

TRANSCRIPT

The basic recipe for creating a machine-learning program is: Choose a representational space, design a loss function that uses data to score potential rules, and run an optimizer that selects the rule in the representational space with the approximately best score. And here's the upshot of this recipe: Machine learning turns data into rules.

I think the generality of this basic approach to working with data is partly why machine learning has been so powerful. For example, the music at the opening of this lesson was generated by a machine-learning program created by a student at my university. Each time the program is run, it generates a new piece of music.

The loss function judged a candidate composer of new pieces to be good if it tended to produce the same kinds of notes as in a large collection of pieces written by Bach. The optimizer was a program using a technique we'll meet later in this course, known as gradient descent. As you can hear, the result is quite pleasant. Because machine learning is such a general tool, the ways to use it are only limited by our creativity.

OK, so the heart of a machine-learning program is telling computers what we want. The representational space defines a mini-universe of possible choices, the loss function evaluates and scores the possible choices, and an optimizer selects the approximately best rule. In brief: represent, evaluate, select. That's the fundamental recipe for machine learning.

To practice using these ideas, let's look at some informal examples. Suppose we have a program that suggests a driving route given a starting location and destination. Several well-known apps will tackle navigation for you. For this problem, each possible rule takes the form of a route—a sequence of intersections you need to drive through. So what's the representational space? The representational space is the space of all possible routes from the start to destination locations.

As for a loss function, we need something to minimize. The loss function might be something like the estimated time or distance it would take to follow this route from the start to the destination location. Or the loss function could be expressed in terms of cost: the total dollars paid for the driver, the cost of gas and tolls and such.

Alright, let's try a different example. Suppose we wanted a machine-learning system to play a game of poker. We need to represent possible ways to play, so the representational space would be permitted ways to play as defined by the game rules governing all possible decisions about how to bet and when

to fold. The loss function might be easy to imagine. It could be the expected dollars won for playing the strategy, but negated so more dollars won means less “loss” for the rule.

Let’s get even further into the details with a visual example: writing a program for processing green screen images. The cool thing about a green screen is that you can replace it with cool-looking backgrounds like this one. Or this one. Nice!

How does that work? Well, there is software that looks at each pixel in the image and determines whether or not it is green. If it is not green, the pixel from the image of me is shown. If it is green, then the software displays a pixel from the selected background image. It works really well, right? But check this out. If I take a green piece of paper and hold it in front of my chest, I’ve made a hole through my body. The software is making this simple decision, and it can lead to weird effects. I’m not ever going to wear that green scarf again.

One thing that makes green-screen software challenging to write is that the pixel colors are not simple color names like you’d find in a crayon box. Instead, there’s a set of roughly 224 distinct possible color values at each pixel position in the screen. We want a rule that decides which set of these 18 billion or so colors should count as “green.”

We’ll use the programming language Python in this example, and throughout the course. Here’s a snippet of Python code for a hypothetical video-editing system. The snippet is not intended to be a machine-learning program for you to run. It’s just to give you a bridge from what happens in traditional programming to what happens in machine learning.

OK. What this snippet would do is check each pixel in the video and then display either the studio or the background, depending on the color visible in the studio. For our discussion here, the important line in the code is where it asks if the studio_image pixel at location x, y is green.

It’s hard for humans to create a rule that decides if a color should count as green or not. So we’ll use a machine-learning approach to do the job. The representational space I picked for this problem has rules of the form: Take a prototype color c , then measure the difference between that color and the color at a position on the screen. If that difference is less than some value d , we declare that pixel to be green and will let the background show through.

Each rule in this representational space is a specific choice for the prototype color c and the distance d . The optimizer picks values for these variables c and d on our behalf. As is common, we represent colors as triples of numbers

LESSON 01 | TELLING THE COMPUTER WHAT WE WANT

TRANSCRIPT

reporting how much red, green, and blue are in the color of the pixel. Each number is an integer between 0 and 255. The distance d is some positive floating-point number between 0 and 441.7. That's the furthest two color triples can be from one another.

OK, to reflect the color/distance representational space we're using, our little snippet of code needs to be updated. The "equals green" statement becomes a distance comparison with the prototype color c . Otherwise, the snippet is unchanged. Cool! We have a representational space.

What's our loss function? It's a way of judging whether a given setting of color c and distance d are good. How can we define "good" for this problem? In machine learning, it's often "do as I do, not as I say." That is, I don't know how to say to the machine what's good about a rule, but I can show the machine what I would do in the same scenario.

I put together examples of green and non-green pixels by selecting part of the image that's from the background and then part of the image from the foreground. Since I want the background colors to be categorized as green, I label them as positive examples and place them in the Yes Set. The foreground pixels should not be considered green, so I label them as negative examples and place them in the No Set. We can now use the labeled pixels to define a loss function. Given a candidate rule, we can run each of these labeled pixels through the rule and see how often the rule disagrees with the labeled examples.

Here are seven lines of code that give us the loss for a given choice of color c and distance d . This example is what a loss function might look like in code, so let's go through it. In contrast with handwritten programming where we tell the computer what to do, I encourage you to think of this as telling the computer what to like.

So, notice how we define the loss function, called "computeLoss". It takes as input the choice of color c and the distance d , but also a Yes Set that lists a set of colors that the rule should return "yes" for and a No Set that lists a set of colors that the rule should return "no" for.

We set the initial loss to zero. We're just counting up the number of labeled pixels that are assigned the wrong class by our choice of color c and distance d . First, we run through all of the colors that we marked as "yes." In the code, we write "for $c1$ in YesSet". We add one to the loss by writing "loss += 1", and we execute that statement if the color is more than d away from our prototype color c .

Then we run through all of the colors that we marked “no,” and we add one to the loss if the color is less than d away from our prototype color c . So each time our choice of c and d causes the rule to make a decision on an example in the data that is different from how we labeled it, the rules gets a little penalty.

Now, for any choice of color c and distance d , we can find out how many of the labeled examples get misclassified. If I set c to be a pure blue of [0,0,255] and distance d to be a number like 100,000, I get a score of 48,321. If I set c to be pure green [0,255,0] and d to be a number like 10,000, I get a similar score of 48,638.

But if I bring the distance d down to 9,000, the score drops to 5,752. So, that’s not bad, is it? Actually, I don’t know. It seems like it’s going to take a long time to find the best choice.

That’s where machine learning brings in the optimizer. For now, let’s just assume the optimizer is a kind of magic box that can sift through all the values of color c and distance d to come up with a value that results in the lowest possible loss. In reality, the absolute best choice of color c and distance d is very difficult to find. So what we do instead is use a scheme that does pretty well at reducing loss for this combination of representational space and loss.

OK, the best c and d our optimizer is able to find has a loss of only 260. If we plug these values of color c and distance d into our rule, we can take a look to see what it does in the green screen example.

Look at that! We can see the background just like we wanted. Look how crisp the edges are! It’s not perfect. It missed the green card, but think about why: It’s a different shade of green, and I didn’t include any pixels from the green card in my Yes Set. But the problems could be fixed by labeling more examples or by using a richer representation space. Meanwhile, the basic machine-learning approach already does quite well.

Let’s contrast our machine-learning program with a more traditional approach to programming. Traditional programming is sometimes called imperative because you issue specific commands that the computer has to follow. We tell it exactly what to do—exactly what color comparison to make. We can classify pixels as “yes” or “no” using an imperative programming style, but that means it’s up to us to choose the values for our color c and distance d on our own. Until we provide a setting for each of those values, an imperative program cannot be run.

LESSON 01 | TELLING THE COMPUTER WHAT WE WANT

TRANSCRIPT

In machine learning, we do not tell the program what to do. We simply declare what the program should like by giving the computer code and examples that are used to assess how good its proposed rules are. Instead of imperative, the approach to programming in machine learning is declarative. We declare that the program should make a good choice, and we give it input that defines what better and worse choices are, but we don't tell the computer what to do.

OK, I don't want to overstate this distinction. At a more fundamental level, we are always telling the computer what to do, so we're always writing imperative programs. But the thought process in machine learning has this backward aspect to it, and it can take some practice before it starts to feel natural. And yet, the more you are able to solve problems this way, the more you'll see that it's quite empowering. We can offload to the computer the search for rules that we can't express concretely.

Because of this backward structure, a machine-learning program has to do a lot of work up front to find a way of meeting the demands of our declaration. Not surprisingly, the hardware needed to run a machine-learning program can be much more than what is needed to run a traditional program.

We've been talking about the trio of elements a machine-learning program uses to tell a computer what we want. But the kind of data providing feedback to the loss function of the machine-learning program is also important.

The green-screen problem and the heart-sounds problem are examples of the archetypical type of machine-learning feedback, known as supervised learning. In supervised learning, expert humans provide data that includes both inputs and their corresponding outputs or labels.

In supervised learning, the feedback is like premade flashcards. On one side of the flashcard is the input: "What's another name for the scoring function in machine learning?" And the other side is the target output: "the loss function." The machine learner has to learn to associate what's on the front of the card with what's on the back of the card. Ideally, the machine learner isn't just memorizing what's on the cards, but it is identifying a rule that makes it possible to give correct responses to cards it has never seen before.

The second class of machine learning is unsupervised learning, which is like having no teacher at all. Recess! Actually, it's more like study hall: independent reading and taking notes. "Another common component in loss functions is a penalty for rules that are larger or more complex." Hmm, that's good to know!

An unsupervised machine learner is not merely the logical opposite of a supervised learner. What we call an unsupervised learner is one that tries to re-represent the data it is given in a more succinct way. In computer science, the idea of representing the same information in less space is called compression. In machine learning, a more succinct output is achieved by clustering, or sometimes with dimension reduction.

However, getting enough data for fully supervised learning is hard, and unsupervised learning without any labeled examples is not directed to solve a particular problem. So we also have semi-supervised learning, where we leverage some examples to shed light on the rest.

Last but not least is another quasi-supervised class called reinforcement learning, partway between supervised and unsupervised learning. It's also my personal favorite topic in machine learning. Reinforcement-learning algorithms expect an intermediate kind of labeling. They don't require complete answers like supervised learning. But they are also not as hands-off as unsupervised learning. Instead, the labels are evaluations, indicating how good a given answer is. They are like the evaluations captured by the overall loss function, but now with scoring extended down to the level of specific decisions or sequences of decisions as well.

You can think of reinforcement learning as a bit like having access to an Olympic judge or a writing coach. You are not being told how to perform or what to write. There is no longer an expert label telling you what to do. You are just getting feedback on how well you did it. "It's common for a loss function to include a penalty for overly complex rules." Ninety-six percent! I got an A!

This kind of feedback is a great match for problems in which we can automate the process of assessing how well a system did, but not what it could have done better. Historically, games have been a great application of reinforcement learning. Reinforcement learning is a good candidate anytime the rule for determining who wins is easy to apply but the selection of the best strategy is hard.

In many ways, the ideas of machine learning are revolutionary. My aim is to show you how to participate in this revolution, while also realizing that it's up to each of us to ensure that machine learning is applied in ways that benefit us all.

LESSON 01 | TELLING THE COMPUTER WHAT WE WANT

TRANSCRIPT

After all, machine learning can also go wrong. Bad actors can “poison” machine-learning datasets and exploit the resulting behavior for their own purposes. And even without malicious intent, machine-learning programs are complex, even opaque, and have many other ways that they can go wrong.

Given such challenges, the question that naturally arises is: Is machine learning good or bad? And my answer is: both. It’s kind of like natural language, in a sense. Are words powerful? Yes, of course. Are they dangerous? Well, yes. You can do amazing and scary things with language. And the same is true for computation, and for machine learning in particular.

Over the next several lectures, we’ll explore the main approaches to machine learning. We’ll start with decision trees in the next lecture. Then, we’ll have two lectures about neural networks and continue through each of the three remaining major perspectives on machine learning.

Each of these early lectures will focus on supervised learning, while later in the course, we’ll also encounter unsupervised, semi-supervised, and reinforcement learning. And throughout the course, we’ll see more and more of how to program backwards.

When we program in machine learning, we’re inputting a framework for what we want. What’s amazing are all the ways a computer can use programs about what we want to find rules that solve problems on our behalf.

LESSON 02

STARTING WITH PYTHON NOTEBOOKS AND COLAB

Programs in this course use the Python programming language. Python is the most widely used and supported language in many areas of computer science, including machine learning. There is roughly one Python program for you to try for each lesson of this course. Even if you've never written programs in Python—or any language—you can still run these programs to get a feel for what they do.

This Course's Approach

The approach that has been laid out for users of this course is

1. browser-based programs
2. using Python notebooks
3. hosted on GitHub that
4. run through Google Colaboratory (Colab).

Python notebooks offer a mode of interacting with the complexities of machine learning programs that is relatively easy for beginners and empowering for everyone.

Here's what you should know about each of the four components of this approach:

1. It's recommended that you run the Python programming examples from your web browser. Doing so allows you to avoid having to install the software on your computer. In addition, you get access to powerful server machines for running the code, which can be much faster than machine learning programs run on your own computer. Plus, making use of professionally maintained servers can save you some other headaches, such as needing to store and manage large data files or reinstalling libraries after software upgrades.
2. This course has been designed so that you are working with Python code by way of a special file format known as an interactive Python notebook, also called a Jupyter notebook.* The extension for an interactive Python notebook file format is `.ipynb`, and that's the main format for files provided for this course. The advantage of an interactive `.ipynb` file over a static `.py` Python file[†]—often used for Python programs—is that an interactive Python notebook integrates code, documentation, and the result of running the code all in one place.

* The IPython project, which began in 2001, adds interactive features to basic Python programming. Since 2014, interactive Python has also served as the kernel for Project Jupyter to extend the same interactive notebook approach across many other programming languages.

[†] Python notebook files are not directly usable as plain Python program files in the `.py` format. These notebooks cannot be usefully opened and edited as plaintext in a text editor like Notepad. The Python notebooks that have been provided for this course were not designed to run top to bottom as static `.py` files.

3. The Python notebooks are hosted and shared with you through a widely used code-sharing service called GitHub. You don't need to go to GitHub directly from a browser search engine. Instead, you can get there from this link, which takes you to a landing page that has been set up for this course:

www.TheGreatCourses.com/MachineLearning

4. The browser-based interpreter for Python that's recommended for these notebooks is a free service called Colaboratory, or Colab, provided by Google to anyone who has a Google account. The notebooks can also run from a browser on the Jupyter website, or even on your local computer if you download and install the Jupyter environment.

Starter Example

To work through a starter example in Colab, visit this link in your browser:

https://colab.research.google.com/github/mlittmans/great_courses_ml/blob/master/L02.ipynb

The link will take you to a file on GitHub, where you will see a “Hello, World!” program in Python.

Colab is optimized for only a few browsers—typically Firefox, Safari, and Chrome—so if the link fails or you have difficulty following any of the later steps, make sure you are using one of those browsers. You can also check the Great Courses landing page.

If you scroll through the file, you can see the code and its output from the last time it was run before the file was posted.

Anyone can see a file in Colab, but to run files in Colab, you'll need to be logged into a Google account. Once you are logged in, you can run code by clicking on the triangular play symbol on the boxes.

However, running the code out of order might result in errors if the code refers to variables or function names that have not been defined yet.

You can also edit the code and run it; you are not restricted to only running the provided code. Just click on the code and edit. Your changes won't be saved, but they will run nevertheless.

Finally, if you want to keep your changes, you can save your own personal copy of the file, preserving any edits that you made.

But before you can save, Colab insists that you make your own copy. By doing so there's no way anything you do in your own copy can ever affect the original hosted file. If you return to the hosted file, the original version will always be what appears.

The options for saving your own copy are under the File menu.

- ◆ You can download the file to your Google Drive by clicking “Save a copy in Drive”. Then, when you click on the file in Google Drive, it will start up a new Colab session for you.
- ◆ Or you can save your copy of the file to GitHub in a few different ways if you register for a free GitHub account: You can click “Save a copy as a GitHub Gist,” which is GitHub’s simpler format for saving single snippets of code; or you can click “Save a copy in GitHub,” which means you have established your own code repository, where your files can access more features.

After saving in Drive, GitHub Gist, or GitHub, you can use the plain Save command when you make updates.

When you run the code, the first block contains the definition of a function called `helloworld`. Click on this block to reveal the play symbol in the upper left. Click on that play symbol.

Google Colab will warn you that the file was “not authored by Google.” But that’s ok, because it was authored for this course.

If you read the fine print on the warning, it says the code “may request access to your data stored with Google, or read data and credentials from other sessions” and that you should “review the source code before executing this notebook.”

This sort of notice is pointing out—correctly—that software automates actions on your behalf, and you may not want those actions taken.

In the case of Colab, the code is not running on your computer and does not have access to the files on your computer. But it does have access to your Google account. However, even if you were to download and run a malicious notebook, that code would not automatically have access to your account. It would request access.

Before you allow any request for access, ask yourself these three questions:

1. **Is the code coming from someone you trust?** In this case, the code was specifically authored for your use in this course.
2. **If the code asks for access to resources, do those resources line up with what you expect the code to do?** If a program asks for access to all of your files just to show you a video, you should be suspicious. In this case, the code is asking for access to your account with Google because that's how Google validates, and places an upper limit on, your free use of their machines. If the code ever asks for deeper access, make sure that additional access lines up with what the code is supposed to do.
3. **Is this the safest alternative for you to get what you need?** If you have another way to get the job done without needing to trust a third party, that would be preferable. Also, consider whether you actually need what you thought you needed.

If you answer yes to all of these questions, you can feel comfortable clicking “Run Anyway” to run the code—even without attempting to “review the source code.”

You might also be asked about resetting runtimes. *Runtime* is Colab’s word for the connection between the notebook and one of their servers. Resetting a runtime will make it forget any calculations it has done on this notebook. You won’t lose the code, just the value of any variables that have been defined since you opened the notebook. You can just approve resetting all runtimes and then run the code again.

Now that you’ve clicked on the play symbol, the program code will execute. In this case, it doesn’t do anything obvious, except that the circle around the play symbol indicates activity. But behind the scenes, it is running the block that causes the `helloworld` program to be defined.

Now look at the second block, where it says `helloworld(4)`. Under that, you can see the output: `hello! hello! hello! hello!` (in other words, four hellos). That’s the result already obtained from running `helloworld(4)` when the notebook was set up.

Now run it just to check. Click on the triangular play symbol next to the command. The hellos briefly disappear and then reappear. That small change tells you that the code is live.

To confirm that you're not merely looking at a static web page, click on the `helloworld(4)` and change the four to some other number, such as 10. Now click play and you should see 10 hellos appear in response.

If you want to add your own coding block to the document, here are a few of the many options:

- ◆ You can click `+Code` in the top-left section of the screen.
- ◆ You can click on any empty block next to a pair of brackets: `[]`. Start typing in the block and you are editing a program or command of your own.
- ◆ You can add code to an existing block. For example, you can see where `4+5` was typed and it returned `9`. You can type in any Python command next to `4+5` and then hit the play symbol to execute it. Or you can hit Shift+Enter—a keyboard shortcut‡ for running the current block—to run the code without having to take your hands off the keyboard.

If there are errors in what you type, you'll see an error message appear in the box below. Otherwise, it'll run what you ask.

Python notebooks run only the blocks you specify, in whatever order you run them. Again, it's usually a good idea to run the blocks in order—from top to bottom. § If you had run the `helloworld(4)` block before running the `helloworld` definition block, it would give you an error pointing out that `helloworld` is not yet defined.

Python programs can produce text or graphics or sometimes basic user interactions. If the text output is sufficiently long, the output box will be scrollable. That lets you look at all the output or just scroll to the next block.

If you run a program that does not terminate, you might have to click the stop symbol—which appears in place of the play symbol, inside a spinning circle—while a block is running.

Just remember that if you want to save any of your changes, you have to make your own copy and save that copy elsewhere. Here are some options:

- ◆ Save to your computer's hard drive by clicking on the File menu in the upper left of the screen and then selecting “Download.ipynb”.
- ◆ Save to your personal GitHub account.
- ◆ Save to a personal account on Google Drive.

‡ Under the Tools menu, there's a list of other “Keyboard shortcuts”.

§ Running blocks out of order sometimes does come in handy—for example, if you're in the middle of developing a new program.

For beginners, the easiest option may be to save to your own account on Google Drive. When you save, a folder called something like “Colab Notebooks” should be automatically created to hold whatever you save from Colab to your Google Drive. Clicking on files there later will return you to your version. If necessary, manually select for it to be reopened with Colab and rerun the blocks to get back to where you left off.

If a run ever hangs, or gets too out of hand, you should be able to just kill the browser tab that Colab is running in and then open a new tab whenever you’re ready to try some more.

All the programs for this course are available at the Great Courses landing page. For some lessons, you’ll see that also included is a file with “aux” in the name, where you can run auxiliary programs that were used but not displayed in the video lesson.

Here are two work-arounds if you are unable to access files via the main link to the landing page address:

1. You could try searching the GitHub site for the string “great-courses-ml”. In that case, you’ll need to go through the search results GitHub presents at the time to find the correct repository. The page you get will have links to all of this course’s program files.
2. If the GitHub collection of files ever becomes unusable, a zipped collection of all instructional program files used in the course is available from the Great Courses landing page.[¶]

In Colab, the exclamation mark symbol (**!**) at the front of a command results in running an operating system command on the Colab computer. You will occasionally use this feature to fetch files and install libraries.

When running this kind of block, you’ll typically get some commentary from the download process in your output window, which you can skip over. But in this case just below the commentary, you’ll see a friendly ASCII picture message at the end.

These tools, and the experience you gain with them, will let you dive as deep as you want into the world of machine learning.

[¶] Due to intellectual property considerations, some of the data files you are permitted to use as an individual accessing from a site like GitHub are not hosted on The Great Courses.



Much of machine learning is less about hand-coding programs from scratch and more about making use of the variety of resources that have already been created. So this course was designed to strike a balance that exposes you to some programming details while always trying to keep a bigger picture in view.

Whenever you want to explore more details, there are plenty of online resources with further information about working with Python, its libraries, and Jupyter notebooks.

The main two libraries you'll be using in this course are Scikit-learn for classic machine learning programs and Keras for deep learning programs, with others brought in as needed. There's a vibrant and active user community associated with Python, Colab, and the various machine learning libraries.

Try It Yourself

Follow along with the video lesson via the Python code:

[L02.ipynb](#)

QUESTIONS

- Run `helloworld(20)`.
- Make and run a `goodbye(x,y)` program that writes `x` copies `hello` followed by `y` copies of `goodbye`.

Answers on page 476

Starting with Python Notebooks and Colab

Lesson 2 Transcript

Programs in this course use the Python programming language. Python is the most widely used and supported language in many areas of computer science, including machine learning. There is roughly one Python program for you to try for each lesson of the course. Even if you've never written programs in Python, or any language, you can still run these programs for yourself to get a feel for what they do.

The approach I have laid out for the users of this course is browser-based programs using Python notebooks, hosted on GitHub, that run through Google Colaboratory. I'll explain all that. But first, let's contrast with a more traditional approach, which I am not recommending or supporting for this course.

Traditionally, if you wanted to run Python from your own computer, you would want to download and install a Python interpreter on your own computer. The latest version of Python's basic interpreter, together with the Python Standard Library, could be downloaded and installed, free of charge, from python.org.

And you would also download and install to your own computer some of the many other powerful libraries from the Python Package Index, such as NumPy and Scikit-learn. These are libraries that expand Python's basic functionality and underpin many machine-learning applications we'll see in this course.

But for machine learning, you have to deal with a lot of additional complexity if you download everything to your own computer. And, for some runs, you would want more processing power than is available on your typical home computer. So I designed this course to make a different approach possible.

First, I recommend running the Python programming examples from your web browser. Running in the browser avoids the need to spend the time and effort installing the software on your own computer. And you get access to powerful server machines for running the code, which can be much faster than machine-learning programs run on your own computer. Plus, making use of professionally maintained servers can save you some other headaches, such as the need to store and manage large data files, or reinstalling libraries after software upgrades.

LESSON 02 | STARTING WITH PYTHON NOTEBOOKS AND COLAB TRANSCRIPT

Second, I have designed this course so that you are working with our Python code by way of a special file format known as an interactive Python notebook. I'll take a moment to explain what this format is and why we're using it.

The IPython project, which began in 2001, adds interactive features to basic Python programming. Since 2014, interactive Python has also served as the kernel for Project Jupyter to extend the same interactive notebook approach across many other programming languages. So sometimes Python notebooks are called Jupyter notebooks.

The extension for an interactive Python notebook file format is “dot ipynb”, and that's the main format for files I've provided. Just to be clear: Python notebook files are not directly usable as “plain” Python program files in the “dot py” format that's often used for Python programs. These notebooks cannot be usefully opened and edited as plain text in a text editor like notepad. But the advantage of interactive “ipynb” over a static “dot py” Python file is that an interactive Python notebook integrates code, documentation, and the results of running the code, all in one place.

So once you're inside a Python notebook, it's true there is a command for downloading into a plain “dot py” format, but I do not recommend that. The Python notebooks I've provided were not designed to run top to bottom as static “dot py” files, so the results won't always be directly usable.

Anyway, what's important is that Python notebooks offer a mode of interacting with the complexities of machine-learning programs that is easier for beginners and more empowering for everyone. In fact, even machine-learning veterans use Python notebooks to organize their experiments.

Third, we are hosting and sharing the Python notebooks with you through a widely used code-sharing service called GitHub, which was purchased by Microsoft in 2018. You don't need to go to GitHub directly from a browser search engine. Instead, you can get there from this link, which is also provided on a special course-support page from The Great Courses.

Fourth, and lastly, the browser-based interpreter Python I recommend you use for these notebooks is a free service called Colaboratory, provided by Google since 2014 to anyone who has a Google account. Colab, as it's known for short, is a convenient way of running Python notebooks. They can also run from a browser on the Jupyter website, or even on your local computer, if you download and install the Jupyter environment.

To work through a starter example in Colab with me, visit this lesson’s link in your browser. The link should take you to a file I’ve put on GitHub, where you will see a little “hello world” program in Python.

Colab is optimized for only a few browsers—typically Firefox, Safari, and Chrome—so if the link fails or you have difficulty following any of the later steps, make sure you are using one of those browsers. For more help, you can also check the course-support page we’ve set up at The Great Courses.

If you scroll through the file, you can see the code and its output from the last time I ran it before posting the file. Now, anyone can see a file in Colab. But to run files in Colab, you’ll need to be logged into a Google account. Once you are logged in, you can run code by clicking on the little triangular “play” symbol on the boxes. However, running the code out of order might result in errors if the code refers to variables or function names that have not been defined yet.

You can also edit the code and run it. You are not restricted to only running the code we provide. Just click on the code and edit. Your changes won’t be saved, but they will run nevertheless. Finally, if you want to keep your changes, you can save your own personal copy of the file, preserving any edits that you made.

But before you can save, Colab insists that you make your own copy. That way, there’s no way that anything you do in your own copy can ever affect the originally hosted file. If you return to the hosted file, the original version I’ve provided will always be what appears.

The options for saving your own copy are under the “file” menu. You can download the file to your Google Drive by “save a copy in Drive.” Then, when you click on the file in Google Drive, it will start up a new Colab session for you. Or you can save your copy of the file to GitHub, if you register for a free GitHub account.

“Save as a GitHub Gist” is GitHub’s simpler format for saving single snippets of code. “Save copy in GitHub” means you have established your own code repository, or repo for short, where your files can access more features. The choice is up to you. After saving in Drive, Github Gist, or Github, you can use the plain “save” command when you make updates.

LESSON 02 | STARTING WITH PYTHON NOTEBOOKS AND COLAB TRANSCRIPT

Let's run the code to see what happens. The first block contains the definition of a function called "helloworld". Click on this block to reveal the little "play" symbol in the upper left. Click on that "play" symbol. Google Colab will warn you that the file was "not authored by Google." But that's OK—it was authored by me, for you. Oh, you're welcome!

If you read the fine print on the warning, it says that the code "may request access to your data stored with Google, or read data and credentials from other sessions" and that you should "review the source code before executing this notebook." Now, an admonition that you "review the source code" is pretty intimidating. So there are a few things you should reflect on here.

This sort of notice is pointing out, correctly, that software automates actions on your behalf, and you may not want those actions taken. In the case of Colab, the code is not running on your computer and does not have access to the files on your computer, but it does have access to your Google account. However, even if you were to download and run a malicious notebook, that code would not automatically have access to your account. It would request access.

So before you say "yes" to any request for access, I recommend you ask yourself three questions:

1. Is the code coming from someone you trust? In this case, that would be me, working with The Great Courses. That's who is suggesting you run the code. Is running the code consistent with their stated goals? I definitely think so.
2. If the code asks for access to resources, do those resources line up with what you expect the code to do? If a program asks for access to all of your files just to show you a video, you should be suspicious. In this case, the code is asking for access to your account with Google because that's how Google validates and places an upper limit on your free use of their machines. If the code ever asks for deeper access, make sure that additional access lines up with what the code is supposed to do.
3. Is this the safest alternative for you to get what you need? If you have another way to get the job done without needing to trust a third party, that would be preferable. Also, consider whether you actually need what you thought you needed. You need to weigh the risks and benefits.

If you answer “yes” to all these questions, you can feel comfortable clicking “Run Anyway” to run the code, even without attempting to “review the source code.”

You might also be asked about resetting runtimes. *Runtime* is Colab’s word for the connection between the notebook and one of their servers. Resetting a runtime will make it forget any calculations it has done on this notebook. You won’t lose the code, just the value of any variables that you have defined since you opened the notebook. You can just say “yes” to resetting all runtimes, and then run the code again.

Now that you’ve clicked on the “play” button, the program code will execute. In this case, it doesn’t do anything obvious, except that the circle around the “play” button indicates activity. But behind the scenes, it is running the block that causes the “helloworld” program to be defined.

Now, look at the second block, where it says “helloworld(4)”. Under that, you can see the output: “hello!-hello!-hello!-hello!”. There are four *hellos*. And that’s the result already obtained from running “helloworld(4)” when I set it up.

Now let’s run it together, just to check. Click on the triangular “play” button next to the command. The *hellos* briefly disappear, then reappear. OK... But that small change tells us that the code is live, which is pretty exciting! To confirm that we’re not merely looking at a static webpage, click on the “helloworld(4)” and change the 4 to some other number like 10. Now click play, and you should see 10 *hellos* appear in response. It’s getting to be a very friendly program!

If you want to add your own coding block to the document, here are a couple of the many options. You can click the “+code” button near the top of the screen. You can also just click on any empty block next to a pair of brackets. Start typing in this block, and you are editing a program or command of your own. You can also see where I typed “4+5” and it returned “9”. You can run this block yourself, and shift-enter is an easy way to run the current block. That is, you can type in any Python command and then hit the “play” button to execute it. Or you can hit shift-enter to run the code without having to take your hands off the keyboard. Under the tools menu, there’s a list of other keyboard shortcuts, if you like that sort of thing.

LESSON 02 | STARTING WITH PYTHON NOTEBOOKS AND COLAB TRANSCRIPT

If there are errors in what you type, you'll see the error message appear in the box below. Otherwise, it'll run what you ask. Python notebooks run only the blocks you specify, in whatever order you run them. Again, it's usually a good idea to run the blocks in order, from top to bottom. If you had run the “helloworld(4)” block before running the “helloworld” definition block, it would give you an error pointing out that “helloworld” is not yet defined—yet. Running blocks out of order sometimes comes in handy—for example, if you're in the middle of developing a new program.

Python programs can produce text or graphics or sometimes little user interactions. If the text output is sufficiently long, the output box will be scrollable. That lets you look at all the output, or just scroll to the next block. If you run a program that does not terminate, you might have to click the “stop” button, which appears in place of the “play” button, inside a spinning circle, while a block is running.

Just remember: If you want to save any of your changes, you have to make your own copy and save that copy elsewhere. You can save to your computer's hard disk, using File menu, then download .ipynb. You can save to your personal GitHub account. Or you can save to a personal account on Google Drive.

Easiest for beginners may be to save to your own account on Google Drive. When you save, a folder called something like Colab Notebooks should be automatically created to hold whatever you save from Colab to your Google Drive. Clicking on files there later will return you to your version. If necessary, manually select for it to be re-opened with Colab, and rerun the blocks to get back to where you left off. If a run ever hangs, or gets too out of hand, don't worry: You should be able to just kill the browser tab Colaboratory is running in and then open a new tab whenever you're ready to try some more.

All the programs for this course are available from a landing page address shown here. The program we've just been running is accessible from that page, as are programs for each of the other lectures. For some lectures, you'll see that I have also included a second file with “aux” in the name, where you can run auxiliary programs I used but did not display in lecture.

Here are two workarounds if you are unable to access files via our main link to the landing page address:

1. You could try searching the GitHub.com site for the string “great dash courses dash ml.” In that case, you’ll need to go through the search results GitHub presents at the time to find the correct repository. The “great-courses-ml” repository is the one you want. The page you get will have links to all of our program files.
2. Or, if our GitHub collection of files ever becomes unusable, a zipped collection of all instructional program files used in the course are also available from a special course support page at The Great Courses. Note that due to intellectual property considerations, some of the data files you are permitted to use as an individual accessing from a site like GitHub are not hosted on The Great Courses set of files. We may also use this special course support page at The Great Courses to post updates to this quick start if there are any important changes at some point in the future.

Returning to the notebook, there is one more feature I want to show you. The exclamation mark symbol at the front of a command in Colab results in running an operating system command on the Colab computer. For example, we can use “exclamation mark wget” to fetch a text file from somewhere on the web and “exclamation mark cat” to output its contents. We will occasionally use this exclamation mark feature to fetch files and install libraries.

When running this kind of block, we typically get some commentary from the download process in our output window, which you can feel free to skip over. Then, just below the commentary, you’ll see a friendly ASCII picture message at the end.

Much of machine learning is less about hand-coding programs from scratch and more about making use of the variety of resources that have already been created. So I’ve designed this course to strike a balance that exposes you to programming details while always trying to keep a bigger picture in view. Whenever you want to explore more details, there are plenty of online resources with further information about working with Python, its libraries, and Jupyter notebooks.

LESSON 02 | STARTING WITH PYTHON NOTEBOOKS AND COLAB TRANSCRIPT

The main two libraries we'll be using are Scikit-Learn for classic machine-learning programs and Keras for deep-learning programs, with others brought in as needed. There's a vibrant and active user community associated with Python, Colaboratory, and the various machine-learning libraries. These tools, and the experience you gain with them, will let you dive as deep as you want into the world of machine learning.

LESSON 03

DECISION TREES FOR LOGICAL RULES

In machine learning, rules produced by a computer can be expressed logically in the form of if-then-else structures. These if-then splits can be diagrammed in a top-down tree structure. The splits represent yes-no, true-false decisions about which branch to follow, while the final nodes either represent categories at the end of all those decisions—in what are called categorical trees—or are numbers, in what are called regression trees. Taken together, these trees are called decision trees.

Decision Tree Algorithms

Decision tree algorithms first sprouted on the machine learning scene in the mid-1980s. One of the earliest examples was ID3 (Iterative Dichotomiser 3), published in 1986, which established the basic top-down structure for learning decision trees that has dominated the field.

One of the excellent properties of decision trees is that they can produce simple rules, especially if we limit the number of splits. In other branches of machine learning, it's uncommon that the learner ever produces rules simple enough to be directly and fully understandable by people.

But with decision trees, we can often understand why the learned classifier makes a particular prediction for a particular instance. Because we can understand what the decision tree is doing, we can refine the analysis if it makes a decision we regard as inappropriate to the problem at hand.

Because the procedure of decision tree learning builds trees top-down without revisiting earlier decisions about what branches to use, it's quite fast, even if the training set is large. But because it does not revisit its decisions, problems that require looking at subtle interactions of the features can be hard to learn with a decision tree.

Try It Yourself

Follow along with the video lesson via the Python code:

[L03.ipynb](#)

Auxiliary Code for Lesson:

[L03aux.ipynb](#)

Python Libraries Used:

graphviz: Graph visualization.

sklearn.tree: Scikit-learn's decision tree algorithms.

When following along with the video lesson in this course, keep in mind that many of the algorithms make use of randomness, which means that your results might not exactly match but should be very similar.

Key Terms

decision tree learning: Creating a hierarchically structured classifier from data.

regression: The problem of mapping instances to numbers.

READING

Domingos, *The Master Algorithm*, chap. 3.

Mitchell, *Machine Learning*, chap. 3.

Russell and Norvig, *Artificial Intelligence*, sec. 19.3.

QUESTIONS

1. If you arrange a training set into a matrix, what are its rows and columns?
2. Given a cap on the maximum number of leaves, is the decision tree learning algorithm guaranteed to find the tree with the highest accuracy? Why or why not?
3. Consider running the decision tree algorithm on the diabetes data with the goal of producing trees with more and more leaves. When the tree has 20 leaves, how helpful do you expect the rule that's learned to be? Is it easy to understand? Do you think it is likely to generalize well to new instances?

Python Libraries:

matplotlib.pyplot: Plots graphs.

random: Generates random numbers.

Answers on page 477

Decision Trees for Logical Rules

Lesson 3 Transcript

People summarize what they've learned in the form of pearls of wisdom. Feed a cold; starve a fever. Wash your hands for 20 seconds. Don't put anything in your ear smaller than your elbow. Machines can do something similar. A machine might summarize a large collection of routing traffic data by: "Internet congestion decreases on nights and weekends."

Some pearls of wisdom come in the form of scientific laws, like $E = mc^2$. It's kind of beautiful. It's powerful. It's very pithy. And it says something deep about the universe in a very short sequence of characters. Short rules like this are often the most transmittable, generalizable, and broadly applicable.

In machine learning, the rules produced by the computer may not be quite so pithy. But they can be expressed logically, in the form of if-then-else structures. We can diagram these if-then splits in a tree structure. The splits represent yes-no, true-false decisions about which branch to follow, while the final node represents categories at the end of all those decisions—categories like "vaccinated" or "cured." Those are categorical trees. Or the final nodes might be numbers, in which case they're called regression trees. Classification and regression trees, taken together, are sometimes known as CART, but we'll just call them decision trees.

Actually, they're kind of upside-down trees, aren't they? But remember, these are logical trees, with a top-down structure. So it's just easier to read the decisions as starting at the top.

Decision-tree algorithms first sprouted on the machine-learning scene in the mid-1980s. One of the earliest examples was ID3, the iterative dichotomiser 3, published in 1986. This algorithm established the basic top-down structure for learning decision trees, which has dominated the field.

As a running example for using decision trees, we'll look at a small-scale problem where we can draw directly on our intuitions for what makes a good set of rules: How can we tell if a word should be spelled I-E or E-I? The rule most widely used is the rhyme "I before E except after C." Let's take a look at whether we can use machine learning to create a better rule for deciding whether a word is spelled with an I-E or an E-I. Along the way, we'll learn some machine learning, and maybe even some spelling.

If we are going to find a better rule, we need to have some way to measure how good different rules are. How might we measure how good “I before E except after C” is? The fundamental recipe of machine learning says that we’re going to define a representational space of rules, and we’ll use data in some way to define a loss function for these rules. The bigger the loss of a rule, the worse the rule is. We want our optimizer to select the rule in the representational space that approximately minimizes this loss.

To quantify the loss of the “I before E” rule, we need to assign low loss to rules that guide us to spell things correctly. So a very natural first cut is to use a loss function defined as the number of instances the rule gets wrong. Here’s a few examples from a 20,000-word pronunciation dictionary—word on top, pronunciation below.

Pronunciations are encoded in a system called the ARPAbet, created in the 1970s to standardize some of the early research efforts in natural language processing. The ARPAbet system uses one character per phoneme, like lowercase *e* for the sound “AY” and capital O for the sound “OY.” I was surprised to find that, in this pronunciation dictionary, the letter patterns I-E or E-I can be pronounced 26 different ways!

The pronunciation dictionary we’re using is from an old machine-learning project known as NETtalk, and it’s available online. NETtalk itself was an interesting and highly influential project in the history of machine learning. Back in 1985, the original NETtalk project addressed the problem of going from a word’s spelling to its pronunciation. We’re going to use the same data to go the other way: from information about a word’s pronunciation to its spelling.

Using the NETtalk data, I pulled out all the words that were spelled with either an I-E or an E-I. Oh, *either* goes in the E-I list. That produced a list of 500 words. Of these words, 357 are I-E words, like *species* and *amplifier* and *dietary*, as well as words like *equestrienne* and *contrariety*.

And 143 are E-I words like *either* and *gneiss*! Nice. Notice that I gave five I-E words and only two examples of E-I words. That’s a visual cue that, in this data set, I-E words are 2.5 times more common. That’s going to turn out to be relevant when we assess possible rules.

TRANSCRIPT

Although we want a rule that minimizes loss, in this case, it's a little easier to talk about maximizing a goodness score. Our goodness score will be the accuracy of the rule on this collection of 500 words. Here's a good way to represent rules so we can compute their accuracy easily.

We draw a node representing all 500 words, broken down into the I-E words and the E-I words. Associated with the node is a test, which appears in the top of the node. So for the classic rule, the test is: Is the I-E or E-I right after a C? That splits up the collection of words into those where I-E or E-I come right after C and those that don't. The ones that do, marked "yes," are going to travel down the left branch. The ones that don't, marked "no," will travel down the right branch.

Following the "yes" branch for "comes after C," we can find that, of the 357 I-E words, 24 are C-I-E words, like *conscience* and *glacier*. Of the 143 E-I words, 11 are C-E-I words, like *misconceive*. So 24 plus 11 are "right after C," and those 35 words go down the left branch. The remaining 465 go down the right branch.

For the words on the left branch, we'll guess E-I because the rule says it's I before E except after C. For the words on the right branch, we'll guess I-E, the default.

Within this structure, we're in a good place to compute the accuracy of the rule. How many words will it get right? Well, it guesses E-I on the left branch and gets 11 words correct. It guesses I-E on the right branch and gets 333 words correct. So, that's 344 altogether. Out of the 500 words, that's 68.8% correct.

Is that good? Well, 68.8% on an exam doesn't seem all that great. What's something we can do to improve the rule? Well, on the left branch, it guesses E-I. That contributes 11 correct words. But if it had guessed I-E, it would have contributed 24 words, which is better!

This revised rule gets $24 + 333$, or 357 words, correct for an accuracy percentage of 71.4%. It's not a huge improvement, but it's better than the original rule! And what is the revised rule? The default is "I before E" because that's what it does on the right branch. The left branch is what it does in the "after C" condition. And in that condition, it also advocates "I before E." So, the rule becomes "I before E." Full stop.

Wow, is that all we need? It's shorter and more accurate than the original rule. And yet, something seems off. For this collection of 500 words, I-E is much more common—two and a half times more common. So, in this case, guessing what's more common all of the time is a lot better than just guessing randomly. That's an important point to be aware of when we're doing machine learning. If your data is not balanced—if it has lots more examples of one class than the other—be careful how you interpret accuracy rates. Even 99% accuracy is very bad if one class represents only 1% of the examples.

Anyway, these numbers are saying that the “except after C” thing is kind of a bust. “But,” you might say, “the rule does seem to be helpful, in spite of what your loss function is saying.” That's true. In fact, that's another important point: Take your loss function with a grain of salt unless you are certain that it is exactly what you want.

Our loss function here is about spelling words correctly from this entire list of 500 words. The “except after C” spelling rule is aimed at words that people tend to get confused about. For example, since the word *applied* is in our list, our rule is trained to spell it correctly: A-P-P-L-I-E-D. But would you ever wonder whether it's actually A-P-P-L-E-I-D? No. That's just not a word. Except maybe if you are a Mac user, you'll see “apple id” as an account name you use for accessing their cloud service. But even then, you would not get the two mixed up.

We might have a better loss function if we had a list of words with I-E or E-I that are commonly misspelled. But that brings up another point: You cannot do machine learning on data you do not have. You might be able to imagine data that is better suited to the problem you are solving. But gathering data can be quite costly. You have to ask yourself: Is it worth the cost? If not, can you still get a useful answer with the data you have?

Sticking with the data we have, there is more we can do. There's another part of the “I before E” rule that is perhaps less well known. The rule that I learned in fifth grade from Mrs. Wiener was “I before E, except after C, or when sounding like A, as in *neighbor* or *weigh*.” The poetic pronunciation exception is built right into the rule.

Let's take a look at the decision tree we get with this rule. I colored one of the new nodes orange because the majority class for that rule is E-I. That's the first time we saw a node that was reached more often by E-I words than by I-E words. We can calculate the accuracy of this extended rule.

LESSON 03 | DECISION TREES FOR LOGICAL RULES

TRANSCRIPT

It gets $24 + 44 + 332 = 400$, or 80% correct. Now we're getting somewhere! Mrs. Wiener's rule seems better. The improvement so far suggests that other trees might be even better.

It's a good time for us to take a step back and try to formulate a general approach to building decision trees from data. To start off, let's talk a little bit about what we mean by *data*. We want an algorithm that spits out a decision tree. The decision tree is a kind of rule that distinguishes positive and negative examples of some concept.

Drop a word like *neigh* in at the top of the tree, and the tree sorts it out. Does the E-I letter combination come right after a C? No, we go down the right branch. Does the E-I letter combination make the sound "AY"? Yes, so we slide down the left branch. We end at a leaf and make our prediction: N-E-I-G-H. Nailed it!

The concept we want our tree to represent is "a word that is spelled I-E." *Neigh* is a negative example of that concept because it is spelled E-I, not I-E. Our training data consists of 500 words that have either E-I or I-E in them. Each has an associated label indicating whether it is a positive example (an I-E word) or a negative example (an E-I word). As you can see in the data set, the word *heiress* is spelled E-I, so it's a negative example of the concept and is labeled "false." But a word like *coterie* is spelled I-E, so it's a positive example, and it's labeled "true." And so on.

In this problem, we want to separate inputs into one of two classes. In the context of machine learning, we call such problems binary classification tasks. Learning binary classification is the prototypical machine-learning problem. Lots of problems in machine learning have this form. We might want to learn whether a given face is me or my son Max, who looks kind of like me. Or we might want to learn to predict whether a paragraph of text is in English or in French. Any such problem is a binary classification task.

Before we can automate the construction of a decision tree on our labeled data, we need to be specific about what kinds of decision nodes our rule-learning algorithm is allowed to consider. The rhyme gives us the idea of asking whether I-E or E-I come right after some particular letter, like C. It also suggests we should consider the way the letter combination is pronounced.

In addition to pronunciations, I decided to have a total of five possibilities. There's right after C, or whatever the letter. I also included right before the letter, as well as whether the letter combination comes any time before each particular letter and any time after each letter. Lastly, a letter might appear not at all with I-E or E-I.

We can encode this information as a big table. Each row of this table is called an instance because it is a concrete example of what we want to apply our learned rule to. The columns of this table are called features. They are the information about the instances that we can refer to in our rule. This entire table, which we take to be the input to our decision-tree-creation algorithm, is called the training set.

For the I-E rule, the full training set has 500 rows, one for each of the instances. There are five different before/after possibilities for the 26 letters, so that's 130 features. Plus, there are 26 different pronunciations for I-E or E-I in the data set. That makes a total of 156 feature columns. These true-false values of the features are the only information the decision-tree-creation algorithm gets to see about the words. The decision-tree algorithm does not inspect the words directly.

An extremely natural and efficient way to build a decision tree for a training set is top down. That is, we first pick the decision node that will sit at the top of the tree. That node will separate all of our data into two sets: the set of instances that answer "true" to the decision node, and the set that answers "false." Both of these sets can be thought of as smaller training sets that we can proceed to turn into smaller decision trees. So by choosing the top decision node first, we can decompose the problem recursively, applying the same procedure repeatedly to simpler and simpler versions of the same problem.

To build our decision tree top down, we need to answer two big questions: Of all the possible decision nodes we could pick, which one do we pick first? And at what point do we just stop splitting our data and assign a label?

Let's tackle the question of what should be our first decision node. We can boil this question down to something very simple. Each feature is a candidate for the decision node. Thinking like machine-learning people, we need a way to score each of these candidates so the one we want is the one with the best score.

TRANSCRIPT

So what does a candidate decision node do? It takes all of the instances in our training set and separates them into two piles. The ideal result is for the labels on each side to be purely one label or the other. Here's a concrete example, separating just a few words into two piles.

Suppose our very first split is on the feature "right before N." So we would separate the word list into two piles: the trues and the falses. How good a split was this? All of the E-I words end up on the same side. That's good, right? Because we can classify the trues as "true," and we'll get them all right. But look at all the I-E words that are still mixed in with the E-I words. That's not good. There's still work to do to separate them out.

The ideal split would make each side consist purely of words that use the same spelling. So we want a way to measure purity. Then we could take the purity of the two sides and compute a weighted average to measure the overall goodness of the split. And that's what decision-tree learning algorithms do.

We'll measure impurity using a score popular in economics, as well as machine learning, known as the Gini index, named for an Italian statistician named Corrado Gini from the early 1900s. The formula measures the difference between the fraction of positive labels, x , and the maximum impure value, $1/2$. The difference is squared, then doubled, then subtracted from $1/2$. The resulting score has a maximum value of $1/2$ when labels are split evenly, and a minimum value of 0 when the labels are purely negative or purely positive.

With the Gini score in hand, we can look at a concrete example of the decision-tree search process in action. For our E-I data set, we start off with 500 instances. One of the features is "any time after A." This feature is true for a word like *impatient*, where the letter A is in the word, and it's before the E-I or I-E letter combination. This feature splits the training set into four categories defined by whether "after A" is true or false and whether the word is spelled I-E or E-I.

To determine if this proposed split is good or bad, we'll use the average Gini index of the two sides to measure the impurity. The left side has 102 out of 122 positive instances, or a fraction true of 0.836. The corresponding Gini index for this value is 0.274. The right side has 255 out of 378 positive instances, or a fraction true of 0.675. From our formula, that's a Gini index of 0.439.

Together, these two numbers tell us that the left side is somewhat pure, and the right side isn't very pure at all. To get an overall purity score for the whole split, we weight the two purity values by the probability that a training example falls on each of the respective sides.

The left side is reached in 122 out of 500 of the examples, or 24.4% of the time, while the right side is reached in 378 out of 500 of the examples, or 75.6% of the time. That gives us a weighted average impurity of 0.274 times 24.4% plus 0.439 times 75.6%, or 0.399. That doesn't seem very pure.

But what's important is how this score compares with the splits we get by choosing the other features. We'll try all of them to see which is best. Well, the computer will try them for us. And searching over all the available splits for our I-E example, we find that the very best score comes from—wait for it—splitting on “sounds like AY.” That was the rule that Mrs. Wiener’s rhyme suggested! The weighted average of 0.313 is better than the 0.399 we calculated for the “any time after A” split, and it’s the best we can get using just a single feature in our data set.

Splitting on “sounds like AY” at the root of the tree breaks up the 500 training examples into 45 on the left, where the E-I combo sounds like AY, and the 455 on the right, where the E-I combo does not sound like AY. We start the process of searching for a feature to split on for each of these two branches separately.

To build the best tree we can with a given number of nodes, we’ll split nodes in decreasing order of Gini score until a target number of nodes is reached. Stopping after seven splits, we get this tree. I took out some of the details to improve readability. But if you add up the correct labelings made in each of the leaves, it gets us an accuracy of 85%. That’s a solid B and arguably a reasonable tradeoff between tree size and accuracy. For comparison, the “I before E, except after C” rule had an accuracy of 69%.

I decided that this new rule would be catchier in rhyme—specifically anapestic tetrameter:

You use E before I when it makes the sound AY,
 In a word that's like *deign* or in *beige* or *inveigh*.
 It's the same if you hear the sounds “EE-IH” inside it.
 A theist or farseeing being would write it.
 The same if it's followed by G or by M,
 Or pronounced just like “EH”—when not right before N.

TRANSCRIPT

When we keep the number of splits low, a great property of decision trees is that we can interpret the rule directly. We can see if it agrees with our intuitions about how to solve the problem or see if it provides any new insights. In this case, we can see that the “AY” pronunciation is supported by the data and by the Mrs. Wiener rule.

Beyond that, our seven-split decision tree suggests that we spell the combination I-E if it’s pronounced “EH” right before the letter N. That appears to be the tree’s attempt to rule out F-R-E-I-N-D in various *friend* words, like *befriend* and *friendly*. Actually, that’s pretty reasonable.

How about “any time before G” or “any time before M”? Do either of those seem relevant for when to spell a word with E-I? They help with *seismographic*, but I don’t see any fundamental insight here. Sometimes machine-learning algorithms hit on rules that work, but not for any deep or lasting reason. I mean, the seven-split rule applies correctly to *einsteinium*—in two places, no less!—but it’s not adding any insight. Remember this *einsteinium* example before assuming a learner has stumbled across some new $E = mc^2$.

Now that we’ve been through an example of constructing and using a decision tree, let’s dive deeper and actually build a decision tree together in Python. You can sit back and watch me do it to learn more about how the algorithm works. Or you can try it yourself, following along with me and exploring how different settings produce different trees with different properties. To do so, just access the notebook file associated with this lecture. If you visit the page through the course landing page, it’ll load in Colab, and the program will be active.

If you are running things along with me, one thing to keep in mind is that many of the algorithms we’re looking at in the course make use of randomness. That means your results might not exactly match mine. But I’ve chosen examples where the behavior I display should be very similar to the behavior you are likely to see.

For data, we’re using a file called `diabetes.csv`. It’s a data set with information for each of 768 people, including whether they were diagnosed with diabetes within five years after the information was recorded. The data includes eight features, including the number of times the person has been pregnant, the glucose concentration in blood plasma, and a few others, such as body mass index. The label is 1 for people who had an onset of diabetes within five years and 0 otherwise.

Here's how we read this file into an array in Python. First, we'll open the file for reading, "r", and assign the file the name "f". With this value f, "f.readlines" produces the entire file as a single array, which we'll call "data". To organize the data, we remove the trailing newline character from the entry for the first line of the file, line 0. Then we split the top line up into components at the commas, which results in an array with the names of the features.

"print(feats)" outputs the features: Pregnancies, Glucose, and so on, up to Outcome, which is the column containing the labels. Since that last part is the label, not a feature, we'll trim it off by indexing into the list from the start at 0 to the length minus 1.

Now we'll read the file in, populating a list of all of the instances, "dat", and all of the labels, "labs". We process each line, then add the resulting vector into "dat". We convert the label to an integer and the other feature values to floating point numbers.

Now we've got our list of 768 instances. "dat" is a matrix with one row per instance and one column per feature. Let's take a look at one row—say, instance 15. We get this numeric vector out, and we see it's a list of the eight feature values for this instance. It represents a 32-year-old woman who has been pregnant seven times and has a body mass index of 30.0.

Now that we've got our training data ready to go, we're all set to build a decision tree. Python has a package called scikit-learn that includes many important learning algorithms. We'll run an algorithm to build a decision tree on our data.

We define "clf" to be a decision-tree classifier with at most three leaf nodes. How well does the resulting tree predict the data? We get a little over 77% correct. OK, not terrible. Especially for a tough problem like predicting whether someone will be diagnosed with a life-altering disease like diabetes.

What does the learned tree look like? I drew the decision trees earlier in this lesson in a way that would help us read these automatically generated trees more easily. Nevertheless, they demand a bit of explanation.

Let's take a look at the root of this tree—the very top node. It says there are 768 samples. That's the whole data set we're working with. The "value" line tells us that 500 of these samples have a 0 label—no diabetes—and 268 have a 1 label—the person gets diabetes within five years. The node includes "gini = 0.454".

The top line in the node represents how we're dividing the data for classification. It says "Glucose <= 127.5". That is, instances with a glucose value below 127.5 are going to go down the left branch, where it says "True". And the instances with a glucose value higher than 127.5 will go to the right.

It makes sense to start with a split on glucose for this data because a high glucose value is a strong indication that the person is not breaking down glucose well. That could be a sign that their insulin isn't doing its job, which puts the person at risk for future diabetes.

Take a look at what the tree does next. Down the left branch—the one with lower glucose levels—we get 485 people, 391 of whom do not end up suffering from diabetes in five years. That's a pretty good number, so the tree stops there and predicts "no diabetes" for these folks.

On the high-glucose branch, we have 283 people, 174 of whom do end up diagnosed with diabetes. That's more than half. Notice that the node is colored pale blue, indicating that the node includes mostly instances that are positive for diabetes.

The algorithm decides to branch on body mass index, BMI, for the high-glucose people. Larger values are taken to indicate that someone is overweight or obese. Splitting on BMI for people with high glucose turns out to make a lot of sense. The decision tree ends up classifying the low-BMI people as unlikely to be diagnosed with diabetes and the high-BMI people as likely to be diagnosed with diabetes. This example does a nice job of showing how the learned decision tree captures abstract intuitions about how to classify while also predicting using specific quantitative values.

Decision-tree learning is our first machine-learning algorithm, and it's worth reflecting on some properties of learning with decision trees. The procedure builds trees top down without going back and revisiting earlier decisions about what branches to use. As a result, it's quite fast, even if the training set is large. But because it does not revisit its decisions, problems that require looking at subtle interactions of the features can be hard to learn with a decision tree.

One of the excellent properties of decision trees is that they can produce simple rules, especially if we limit the number of splits. In other branches of machine learning, it's uncommon that the learner ever produces rules simple enough to be directly and fully understandable by people. But with decision trees, we can often understand why the learned classifier makes a particular prediction for a particular instance. And because we can understand what the decision tree is doing, that lets us refine the analysis, if it makes a decision we regard as more boneheaded than genius.

As we'll see next time, creating accurate rules becomes much harder when learning from images or auditory information. We need another approach to expand the kinds of problems we can successfully address using machine learning. Coming to the rescue: a world-changing revolution, based on neural networks.

LESSON 04

NEURAL NETWORKS FOR PERCEPTUAL RULES

Instead of having simple if-then logical rules like decision trees, neural networks have variables that can be set to any numeric value, allowing fine-scale control over their behavior. This approach to machine learning can build much more complicated systems of rules, using very low-level inputs, such as sounds or images. These more complicated systems of rules would be hard for humans to derive explicitly, or even fully understand. Humans don't think of pictures in terms of pixels or sounds in terms of sound waves. Yet focusing on these low-level features is what makes it possible for machines to solve perceptual problems for us.

Recognition Tasks

Neural networks allow machines to do the hard work of identifying higher-level patterns from lower-level inputs, such as words from acoustic signals, people from images of their faces, letters from handwritten stroke information, and breeds of dogs from photographs.

Deep learning—which has exploded in impact and popularity since 2015—is just an approach to building and training neural networks with many layers.

To see why neural networks are needed, let's consider the problem of recognizing human faces. It's one of many hard problems in AI that people are very good at, despite being unaware of what we're doing when we're solving these problems. Because we don't know how we do it, it's really hard to verbalize an explicit rule that a machine could use to solve the same task.

Returning to the fundamental recipe of machine learning, a machine learning approach requires a representational space, a loss function, and an optimization procedure for selecting low-loss rules.

When we create machine learning systems, some applications fit well with a simple rule-based approach. But short, human-understandable rules aren't a good fit for the problem of recognizing faces. Although it's conceivable that the right set of 33 features would be sufficient for uniquely identifying all 8 billion faces on Earth, no one has figured out which features those would need to be. There isn't a clean logical hierarchy.

To deal with low-level perceptual information like we find in facial images, we're going to need something more powerful, allowing many, many subtle distinctions to be made. Instead of a rule in any simple sense, we're going to want something more like an entire program.

To generate such a program capable of so many subtle distinctions, we need a way for machine learning to optimize over possible programs.

How could we optimize over programs? That's really hard. Even if we start off with a program that's pretty good, improving that program is very similar to the problem of debugging, which is hard even for experienced human programmers.

The fundamental insight of neural networks is that we can represent very big, very general programs capable of making many subtle distinctions, yet in a form that is simple and regular and therefore amenable to systematic optimization.

Architecture of Neural Networks

A neural network representation consists of

- ♦ units,
- ♦ weighted connections between the units, and
- ♦ an **activation function** in each unit.

Just like an architectural blueprint gives information about the rooms of a building, how they are connected, and information about their construction, neural networks also have an architecture.

A neural network's architecture specifies the number of units and how they are connected. It tells us whether any weights are reused in different parts of the network and what activation functions are used in each unit. Together, the components of the architecture define the representational space used for learning.

Often, the units are organized into layers, with the connections to units in one layer only coming from the units in the immediately previous layer.

Some popular activation functions are linear, sigmoid, and ReLU. All produce more activation given a larger weighted sum as its input.

The linear activation function is simple and easy to work with. These simple neural networks cannot be used to represent some important problems. The sigmoid activation function is powerful enough that using it in a sufficiently big neural network makes it possible to approximate any function. ReLUs are the most common activation function used in deep neural networks today.

The original metaphor for neural networks is that the units are like neurons, where each unit is getting activated to differing degrees. These activations determine what values will be propagated along the connections.

Try It Yourself

Follow along with the video lesson via the Python code:

[L04.ipynb](#)

Python Libraries Used:

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

random: Generates random numbers.

sklearn.datasets.fetch_openml: Provides access to OpenML datasets.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

sklearn.neural_network.MLPClassifier: Scikit-learn's neural network algorithm (multilayer perceptron)

sklearn.tree: Scikit-learn's decision tree algorithms.

sklearn.utils.check_random_state: Repeatable, random way to split training and testing data.

Key Terms

activation function: In a neural network, the function that translates a unit's incoming sum of weighted activations to its output activation. Examples include linear, step, sigmoid, and ReLU.

backpropagation: Efficient algorithm dating to 1986 for computing gradients (the high-dimension “slope”) of neural networks. Used for training neural networks by gradient descent.

deep neural network: Neural network with more than three or four layers.

gradient descent: The process of optimizing a function by iteratively moving its parameters in the direction that causes the function's value to decrease.

hyperparameter: Higher-level parameter, such as learning rate, that influences the process of setting other lower-level parameters.

natural language processing: The use of computer programs to solve problems in written or spoken language.

ReLU (rectified linear unit): In neural networks, an activation function that returns its incoming activation, thresholded at zero to prevent negative activations. A key innovation in the development of deep neural networks.

sigmoid: An S-shaped monotonically increasing activation function that returns a number near 1 if the input is positive and near zero if the input sums to a negative number, with a smooth transition between them around zero.

READING

Charniak, *Introduction to Deep Learning*, chaps. 1-2.

Domingos, *The Master Algorithm*, chap. 4.

Mitchell, *Machine Learning*, chap. 4.

Russell and Norvig, *Artificial Intelligence*, sec. 21.1.

QUESTIONS

1. In a neural network with 10 input units and two output units, what's the largest number of hidden units it can have?
2. For each of the following examples, do you think a neural network or a decision tree would be better?
 - (a) Deciding whether a given sound is a snap or a clap.
 - (b) Predicting someone's salary based on their training, years of experience, and past performance evaluations.
 - (c) Guessing the number of syllables in a word based on its spelling.
 - (d) Categorizing a galaxy as spiral, elliptical, or irregular from a telescope image.
3. What do you think will happen to the accuracy of training a neural network as the number of layers increases? Try training a neural network for the MNIST data using more layers to see what happens. Speculate as to why you are seeing what you are seeing.

Answers on page 477

Neural Networks for Perceptual Rules

Lesson 4 Transcript

When we create machine-learning systems, some applications fit really well with a simple rule-based approach. The machine-learning algorithm can discover the principles governing the task and express the results simply. That's what we focused on last time, and it can be a powerful approach. But for other problems, simple if-then logical rules just don't seem to be the right representation for capturing what's needed to solve a task. We need a different building block for the rules.

An analogy I like to use for understanding different approaches to building rules is LEGOs versus modeling clay. You can build a model out of either. Decision trees are like Lego models: They are made of interlocked pieces that come in a variety of types. Each piece is one LEGO brick, or one attribute split in a decision tree. By contrast, with modeling clay, you can build a model down to any size and smoothness you want.

In this lesson, we'll talk about neural networks, which are constructed out of parts that allow smoothness, more like modeling clay. Neural networks have variables that can be set to any possible numeric value, allowing fine-scale control over their behavior. In practice, this fine-scale control often works like the sliders on a mixing board that sound engineers use. It's not just on/off or the counting numbers 1, 2, 3, up to 10. It's also everything in between.

So thanks to this very fine-scale control, neural networks give us an approach to machine learning that can build much more complicated systems of rules using very low-level inputs, such as sound or images. These more complicated systems of rules would be hard for humans to derive explicitly, or even fully understand. Humans don't think of pictures in terms of pixels, or sounds in terms of sound waves. And yet, focusing on these low-level features is what makes it possible for machines to solve perceptual problems on our behalf.

Neural networks allow machines to do the hard work of identifying higher-level patterns from lower-level inputs. For example: words from acoustic signals, people from images of their faces, letters from handwritten stroke information, and breeds of dogs from photographs. Deep learning, which has exploded in impact and popularity since 2015, is just an approach to building and training neural networks with many layers.

TRANSCRIPT

To see why something like neural networks are needed, let's consider the problem of recognizing human faces. It's one of many hard problems in AI that people are very good at, despite being unaware of what we're doing when we're solving these problems. And because we don't know how we do it, it's really hard to verbalize an explicit rule that a machine could use to solve the same task.

If we try to get machines to recognize a face via simple rules, the rules might look like this:

If:

- brown eyes
- curly hair
- energetic, pleasant
- smart, but approachable
- larger than average nose

Then: It's Michael

This approach works in some cases. The person in this photo doesn't have brown eyes or curly hair. And the nose is small. It's not me. It turns out it's Emma Stone, star of many major motion pictures, but no machine-learning courses—yet. What does the rule do with this person? Well, it matches on all the features. So the rule would classify this person as me. But, no—it's not me.

Returning to the fundamental recipe of machine learning, a machine-learning approach requires a representational space, a loss function, and an optimization procedure for selecting low-loss rules. Short, human-understandable rules don't seem to be a good fit for the problem of recognizing faces. Although it's conceivable that the right set of 33 features would be sufficient for uniquely identifying all 8 billion faces on Earth, no one has figured out which features those would need to be. There's not a clean logical hierarchy.

So to deal with low-level perceptual information like we find in facial images, we're going to need something more powerful, allowing many, many subtle distinctions to be made. Instead of a "rule" in any simple sense, we're going to want something more like an entire program. To generate such a program capable of so many subtle distinctions, we need a way for machine learning to optimize over possible programs.

How could we optimize over programs? That's really hard. Even if we start off with a program that's pretty good, improving that program is very similar to the problem of debugging. And debugging is not a task we know how to automate, which is why it's really hard even for experienced human programmers.

The fundamental insight of neural networks is that we can represent very big, very general programs, capable of making many subtle distinctions, yet in a form that is simple and regular, and therefore amenable to systematic optimization. A neural network representation consists of units, weighted connections between the units, and an activation function in each unit. The original metaphor for neural networks is that the units are like neurons, where each unit is getting activated to differing degrees. These activations determine what values will be propagated along the connections.

In this diagram of a neural network, the hidden units are represented by yellow circles, the output units by green circles, and the connections between units, represented by light blue lines. In this picture, the connections are propagating information from the input on the left to the category of the input on the right.

This network is taking in images and distinguishing which images are cats and which are dogs. The input units correspond to pixels in the image being classified. The output units are the classes to be recognized. And the hidden units carry out useful intermediate calculations.

For instance, if we were designing one of these networks by hand to distinguish cats from dogs, we might assign a specific hidden unit to a specific quantity we'd want to compute, like the degree of pointiness of the ears. That's a hard thing to define. But because we can have many, many hidden units, we can divide up the recognition of pointiness into smaller computational pieces. We can recombine those pieces across the units later to complete the overall recognition at the output layer.

Here's how computation proceeds in this kind of network. Each unit holds a single real number. Following the nervous system metaphor, we call each unit's number its activation—how excited it is.

Some of these units hold the input the network is processing. For image recognition, there would be three input units for each pixel of the image: one for the amount of red in that pixel, one for the amount of blue in that pixel, and one for the amount of green in that pixel.

TRANSCRIPT

Some units hold the output the network produces. If the network is distinguishing pictures of cats from pictures of dogs, there might be two such units. One specifies the degree to which the network believes the input image is a cat, and the other specifies the degree to which the network believes the input image is a dog.

The rest of the units are called hidden units, and they carry out intermediate computations. For a problem like object recognition, there could be hundreds of thousands or even millions of hidden units.

Aside from the input units, whose values are set by the input, all of the other units perform very simple calculations. Each unit has connections from some subset of the other units. The connection patterns are acyclic—no unit's value can be based, even indirectly, on its own value. Each connection has an associated weight, which is just another numerical value.

The activation of a unit is obtained by taking the weighted sum of the activations of all the units that connect to it. This weighted sum is then transformed into an activation using, appropriately enough, the unit's activation function.

So, to process an input, the activation for each unit is computed based on the activations of the units that connect to it. The activations propagate through the network from input to output and then stop. Since there are no loops in the connectivity pattern, this propagation is guaranteed to end quickly. You can think of each unit as carrying out a simple rule-like calculation. By connecting the units into a network, the units can work together to produce the complicated computations we need for solving hard perceptual problems.

When the chain of propagation is relatively short, like two or three, we just call the network a neural network. When the chain is longer, like 20 or 60, we call it a deep neural network. There's no mathematical distinction that defines a sharp boundary. Even four layers seemed pretty deep early on. The adjective *deep* just reminds us that we're talking about the more recent approaches to neural networks that began to become popular around 2015.

Now that we have a pretty good idea of the overall flow of information in a neural network, let's zoom in to a single unit to see what it does. We can write $x_{\text{sub-}i}$ to stand for the activation of each upstream unit, and $w_{\text{sub-}ij}$ to be the weight on the connection from unit i to the unit j we're talking about. Then, the summation over i for $x_{\text{sub-}i}$ times $w_{\text{sub-}ij}$ gives us a weighted sum of those activations for this one unit. I'll call that sum s .

The weighted sum gets pushed through an activation function, written lowercase sigma. That activation gets sent downstream to the next layer of the units that it connects to.

Just like an architectural blueprint gives information about the rooms of a building, and how they are connected, and information about their construction, neural networks also have an architecture. A neural network's architecture specifies the number of units and how they are connected. It tells us whether any weights are reused in different parts of the network. And it tells us what activation functions are used in each unit. Together, the components of the architecture define the representational space used for learning. Often, the units are organized into layers with the connections to units in one layer only coming from units in the immediately previous layer.

Some popular activation functions are linear, sigmoid, and ReLU. Let's take a closer look at these functions. All produce more activation given a larger weighted sum as its input.

The linear activation function just returns the weighted sum coming into the unit as the activation for the unit. It has the form $\sigma(s) = s$. This activation function is simple and easy to work with. It's been shown that there are ways for you to train the weights of networks consisting solely of linear activation functions so that you are guaranteed to perfectly minimize loss. That's great!

But linear activation functions have the unfortunate property that using them in all the hidden units of a network renders those units irrelevant. That is, any neural network with hidden units that only use linear activation functions can be transformed into a neural network with no hidden units at all.

In 1969, Minsky and Papert published a book called *Perceptrons*, referring to a class of neural networks studied in the 1950s that were able to carry out simple recognition tasks. Minsky and Papert pointed out that these simple neural networks could not be used to represent some important problems. A simple example of such an unrepresentable function is one that maps temperature to comfort. As temperature increases, comfort rises. But once the temperature gets too high, comfort falls again.

This up-down pattern cannot be captured using purely linear activation functions. Clearly, we'll need some non-linearity for our network to be applicable to general problems.

TRANSCRIPT

The sigmoid activation function provides a very simple kind of non-linearity. Here, the sigmoid of the weighted sum s has the form 1 over (1 plus e to the negative s). The function moves smoothly from near zero activation to activation near 1 as the weighted sum goes from very negative to very positive.

The sigmoid activation function is powerful enough that using it in a sufficiently big neural network makes it possible to approximate any function, which is a neat trick. A drawback is that sigmoid can be a little computationally expensive to use. Each time activation flows through a unit with a sigmoid activation, the computer needs to calculate the transcendental number e raised to some power. If there are hundreds or thousands of units making up the network, and we need to process millions or billions of instances through that network, the cost of computing the e exponential really starts to add up.

One of the innovations that helped ramp up the deep-learning revolution is the introduction of another activation function in the year 2000 called ReLU. It's a fantastic compromise between the simplicity of the linear function and the non-linearity of the sigmoid function. The ReLU activation function has the form: sigma of the weighted sum s is equal to s if s is greater than 0, and 0 otherwise.

The name “ReLU” is a shortening of “rectified linear unit.” The “linear unit” part of the name comes from the fact that this activation function is just the linear activation function for positive inputs. But it is thresholded so the output value cannot go below zero—that’s the “rectifying” part.

ReLUs are the most common activation function used in deep neural networks today. They are blazingly fast on the graphics processing units, or GPUs, used for training big networks. And they have just enough non-linearity—the little bend at zero—to render hidden units relevant again, letting them capture complex computations.

OK. So a neural network’s architecture is the number of units, the connections between the units, and the activation functions those units use. Within this general framework, researchers have identified certain architectural patterns that seem to be very well-suited for solving specific problems. There are common architectures for problems in natural language processing. Others are common in speech recognition.

But the same architecture can carry out very different computations. A network that takes as input videos of faces could be trained to predict a person’s emotional state—happy, sad, angry. Or those same networks using those same video inputs could be trained to predict what sounds people are making with their mouths—mmmm, fffff, oooh. What differs in these two cases are the settings of the weights. Since the weights can take on many possible values, network weights are sometimes called parameters.

Training a neural network is the process of finding settings for these weight values that allow the network to compute the difference between cats and dogs, or whatever the problem at hand is. You could also think of setting the weights as programming the network.

We can take the view of a deep neural network as a kind of straight-through program, with no looping back, and no forking with if-then statements. Each unit can be thought of as a single assignment statement in the program where a variable’s value is computed and then made available for future computations.

So let’s look at a single assignment statement from such a program. When we are defining the value of the variable on line j , call it $x\text{-sub-}j$, we multiply the value computed before it on line i by the weight $w[i,j]$ between them. That is, we’re taking the value of a variable from a previous line of the program and multiplying it by $w[i,j]$. Then we sum things up for all of the values of i —all of the previously defined variables.

For conceptual simplicity, I’m allowing connections to all of the previous lines, although some of these would be removed from the sum if the corresponding units aren’t connected.

Next, we threshold the sum at 0, meaning we bump up the value so it’s not negative. That’s what the ReLU activation rule requires. The result is assigned to $x\text{-sub-}j$, and we can move on to the next variable.

We assume some set of values is given as the input, like $x\text{-sub-}0$ through $x\text{-sub-[}k\text{-}1]$ for some k . Those k values might be the numbers representing the brightness of each of the k pixels of an image. A 224×224 image, with three color values per pixel, means k equals over 150,000 values given to us in the input.

TRANSCRIPT

Then, the rest of the program consists of a sequence of nearly identical assignment statements. Each line assigns its own variable using its own weights. The number for the first line comes from the size of the input, which was 150,528 values. The number for the last line comes from adding the number of inputs plus the number of units in the network's architecture.

As you can see, the program itself is actually very boring. The only thing interesting about it is the weights—those w values. Setting those weights to different values gives us different programs—different computations on the input. And there are two powerful properties I want to point out about the structure of this boring program: universality and learnability. We'll look at each of these in turn.

First, the structure is universal. Even though this program structure is simple and regular, it can represent just about any computation. That's in spite of the fact that there are no "if" statements and there are no loops. The only mathematical functions are sum and max.

There's a formal logical argument that shows that we can convert almost any program, piece by piece, to a program that does the same computation but has this straight-through program form. To be clear, it's not quite any program. The lack of loops means we can only convert programs where there is a bound on the number of times a loop gets executed. So we can't convert a sorting program that handles lists of unbounded size. But, we can convert a program that sorts lists up to any given fixed size.

There are also some mathematical functions that we cannot compute precisely—for example, the smooth curves of trigonometric functions. The function mapping angles to their cosines cannot be written precisely as a combination of a finite number of linear pieces. But we can approximately represent cosine to any given degree of accuracy. More linear pieces means more precision. That's the essence of universality. Using this simple, regular program form, we can approximate any computation.

The other powerful property of our otherwise boring neural network structure is learnability. The form of the program—line after line after line of maxes and weighted sums—is fixed in advance. The program itself is captured in the values of the weights w . Change the weights, and we change what the program produces. If we randomly pick the values for the weights, the chance that the program does what we want seems pretty small. But we can look to see what it does, given those weights, and then go in and tweak them so the program does a better job. And that's what neural network training algorithms do.

From a machine-learning standpoint, the fixed architecture defines the representational space, while the setting of the network's weights defines a specific rule in that space. The loss function is then the sum over all of the examples in our training set D of the distance from the target output to the output produced by the program using our current weights.

There are a bunch of ways to express distance mathematically. Here's a particularly nice one. We sum over all the input-output pairs x, y in our training set D . For each one, we put the input through the network with the weights set a particular way. We take the output that results from that computation and compare it to the target output y . The squared difference is a convenient measure of how close the two are. If they are the same, the value is 0. If they are close, it's small. If it's far, in either direction, it's a big value.

OK? So we have a representation (the weights) and a loss function (the squared errors). Now we need a way to find weights that produce a good score, capturing the desired computation. That's the job of the optimizer.

A simple way to think about the optimization process is that we're going to iteratively shift each of the weights a little bit up or a little bit down, check the loss, and update the weights if the shift makes the loss improve. This approach is known as local search.

Even in a network with just two weights, which we can depict as a point in two dimensions, there are a lot of directions the weights could be shifted that have the potential to improve the loss. In a network with thousands or millions of weights, finding a productive shift seems like finding a needle in a very high-dimensional haystack.

Fortunately, there's a tool that answers the question: What weight shift most rapidly improves our loss score? It's precisely what differential calculus produces. The derivative of a one-dimensional function tells us the slope of that function, right? Well, the slope is telling us how much the function value increases if we take a small step to the right. This same concept applies to functions of many more dimensions, including loss functions. Because this quantity is a slope in higher dimensions, we call it the gradient. And moving down the gradient to minimize loss is gradient descent.

What's the strategy at this point? Write the skeleton program. That's the neural network architecture whose components define the representational space. Guess values for the weights, which we can just do by assigning them

TRANSCRIPT

small random values. Compute the gradient of the loss with respect to these weights. This high-dimensional slope gives us local information about how to decrease the loss.

If the gradient ever reaches near zero, we have no more information about how to improve, so we should stop. Otherwise, if the gradient is non-zero, we can use it to move the weights in a direction opposite the direction of the current gradient in an effort to shrink the next-round loss. We move the weights, then start the gradient computation process over again with this new set of weights.

An important step I left out is how we compute the gradient over such a vast space of parameters. There is a procedure that first began to be more widely appreciated in 1986, called backpropagation, which is an algorithm that efficiently calculates all of the needed components of the gradient.

Backpropagation works, as the name suggests, by running the program backwards and asking what sort of changes in the weights for each line would produce an output that the downstream lines could use to improve the score. We would have already run the program forward for all of the training examples. Then we essentially run it again backwards on those same examples. At that point, we have the information we need to change the weights.

Modern computer hardware and software libraries make this process fast and easy. To use a backpropagation approach to machine learning, you mainly need to choose the original program template, the data, and a few so-called hyperparameters for the search, like how far the weights should be shifted in each iteration. Then, the optimizer cranks away and finds the best weights it can. And that's neural networks in a nutshell.

Let's take a look at how this idea plays out with an example in Python. We'll train a network to solve a perceptual problem: recognizing handwritten digits.

We're using a famous dataset called MNIST. The name comes from the fact that the National Institute of Standards and Technology produced a modified version of an earlier digit database they had put together.

Python's Scikit-learn library includes a copy of MNIST. We can read it into an array of instances X and a vector of labels y . We'll randomly split 6,000 of the instances in this dataset into 5,000 digits for training and 1,000 other examples for testing what was learned. X_{train} and y_{train} are the instances and labels for training, and X_{test} and y_{test} are the instances for testing.

An instance in this case is a vector of 784 integers, each between zero and 255. Here's an example of one of these vectors and the corresponding label. I picked instance 417 arbitrarily. It says it belongs to class 7, though it's not yet obvious why.

We could try learning a decision-tree classifier based on this data as a baseline. I experimented with trees of various sizes and picked a big tree of 170 leaves. Smaller trees get significantly less good performance, and larger trees don't seem to get any better. Even so, 170 leaves is more than seems reasonable as an interpretable rule. Plus, the tests are also very difficult to interpret. And, as we'll see, an accuracy of 78% isn't great for this dataset.

Let's switch over to neural networks and see how that goes. First, it might be useful to see that these instance vectors of 783 integers can be analyzed visually. Here's the same instance vector from before. Now, it's easier to see why the instance vector we looked at has the label "7"!

OK, here's a purely linear neural network run on this same dataset. Training the neural network classifier in Scikit-learn is very similar to training the decision-tree classifier. The main difference is the use of "MLPClassifier" replacing "DecisionTreeClassifier" when defining the classifier object "clf". *MLPClassifier* means "multi-layer perceptron," or a neural network.

The "MLPClassifier" function takes parameters that control the architecture and how it will be trained. "Hidden_layer_sizes" is an empty list because we're building a network with no hidden units. Activation is "identity" because we're using linear activation, which leaves the accumulated sum "identical". "Max_iter" says we'll train for no more than 10,000 iterations.

Even with no hidden units, this simplest-possible neural network gets around 84 to 86% correct. Neural networks are very well suited to gathering low-level information from the entire input. The decision tree looked at only 170 of the pixels, but the neural network makes use of all 784 pixels.

Our network architecture has one input node for each pixel in the image. It has 10 output nodes, corresponding to the digits from zero through nine. Let's upgrade to a network architecture with a layer of 170 hidden units, using a ReLU activation function. I picked 170 because it looked like performance was peaking around 175, and I liked the idea of using the same constant that came out of the decision-tree run.

TRANSCRIPT

I trained it 10 times, and the results are interesting. It is doing solidly better than the training with no hidden units, peaking up around 91% accuracy. But there's a fair amount of variability from run to run. The non-linearity introduced by the ReLU activation makes the network more powerful, but also much more flakey.

I also did some ReLU runs with fewer hidden units, and sometimes the performance was no better than chance, with a problem with 10 classes, as it fell to 1 over 10, or 10%. Different runs give much more variable results with ReLU. You need to have enough hidden units in your neural network; otherwise, the improved results you were expecting may not materialize.

Neural networks are an approach to machine learning that allows for the combination of low-level features into higher and higher and higher level calculations. We can exploit almost the full power of computer programming, despite never even using a loop or an “if-then” statement in our network.

Moreover, these apparently boring programs are universal, and they are fully differentiable. That is, unlike classical computer programs, we can use calculus to tell us the dependence of the weights or parameters controlling the computation on the correctness of the program they specify. The gradient of these weights tells the optimizer how they can be altered to improve the quality of the program.

As we'll return to in later lessons, neural networks are being used to solve all sorts of cutting-edge problems, such as learning the meanings of words and recognizing 3D objects. But in the next lecture, I want to take you inside an especially simple neural network algorithm. That way, you can see and better understand for yourself how the machine-learning process unfolds.

LESSON 05

OPENING THE BLACK BOX OF A NEURAL NETWORK

This lesson takes you inside an especially simple neural network algorithm so that you can see and better understand for yourself how the machine learning process unfolds.

The Power of Libraries

The general availability of powerful and flexible libraries for specifying and using machine learning algorithms has been one of the reasons behind the explosion of interest in machine learning.

The programming language Python has been a very popular home for most of these libraries, due to Python's widespread use and extensibility.

One Python library that includes a wide variety of machine learning algorithms is called Scikit-learn. It was first released in 2007. The Scikit-learn library was used in the Try It Yourself exercises in the two previous lessons for training decision trees and simple neural networks. We'll continue using Scikit-learn extensively throughout the course.

When we get to deep learning systems, Scikit-learn lacks the flexibility and hardware support for building and running cutting-edge systems.

For deep learning, the main library we'll use in this course is Keras, developed in Python by an engineer at Google, where it was optimized for rapid prototyping of cutting-edge learning systems.

TensorFlow is another deep learning library developed at Google, and we'll be using Keras in a way that's layered on top of TensorFlow. Both were first released in 2015. As we use Keras, little glimmers of TensorFlow will show through in a few of the later lessons.

We'll also look at some advantages of PyTorch, a deep learning tool released by Facebook in 2017, in the final lesson of this course.

These are all freely available libraries that were written to take advantage of powerful hardware, and they incorporate the latest optimizers and network architectures.

One of the drivers of the deep learning revolution has been that everyone is welcome, and everyone can contribute.

However, these terrific libraries that make it so much easier to get started can also contribute to a feeling that the machine learning algorithms are inscrutable black boxes. That's why the video lesson takes you through a simple neural network algorithm in Python without using libraries.

Try It Yourself

Follow along with the video lesson via the Python code:

[L05.ipynb](#)

Python Libraries Used:

keras.preprocessing.image: Prepares raw images for processing by a neural network.

numpy: Mathematical functions over general arrays.

Because of the random choices in this code, when you run it, it'll come out somewhat differently from what you see in the video. But the overall pattern should be similar.

Key Terms

classifier: A rule that maps instances to discrete classes.

delta rule: An update rule for neural network learning that comes from calculating the derivative of a one-layer neural network with squared loss.

READING

McClelland and Rumelhart, *Explorations in Parallel Distributed Processing*.

Russell and Norvig, *Artificial Intelligence*, sec. 19.6.

QUESTIONS

1. Write a Python expression that counts the number of pixels in an image array that have more green than red in them. Assume the image is in a `numpy` array called `p`. Hint: It's easiest if you first turn the array into a list, `l`.
2. **Lesson 01** discussed a hypothesis class of this form:

Pixel `p` is `green` if `distance(c,p) <= d` for a selected color center `c` and distance `d`.

This lesson used the following:

Pixel `p` is `green` if `np.dot(w,p) <= c` for a selected weight vector `w` and constant `c`.

The two rules look very similar. Are they just two different ways of writing the same thing? Why or why not?

3. This lesson involved training a linear neural network to distinguish background green pixels from other foreground pixels. Do you think the network will converge to the same weights every time it is run? Try it and see. Are there interesting differences between the particular weights from one run to another?

Answers on page 478

Opening the Black Box of a Neural Network

Lesson 5 Transcript

I want to spend one lesson working through a simple neural network algorithm in Python without using libraries. It's worth noting that the general availability of powerful and flexible libraries for specifying and using machine-learning algorithms has been one of the reasons behind the explosion of interest in machine learning.

The programming language Python has been a very popular home for most of these libraries due to Python's widespread use and extensibility. One Python library that includes a wide variety of machine-learning algorithms is called Scikit-learn. It was first released in 2007. We used the Scikit-learn library in the two previous lessons for training decision trees and simple neural networks. We'll continue using Scikit-learn extensively throughout this course. But when we get to deep learning systems, Scikit-learn lacks the flexibility and hardware support for building and running cutting-edge systems.

For deep learning, the main library we'll use in this course is Keras, developed in Python by an engineer at Google, where it was optimized for rapid prototyping of cutting-edge learning systems TensorFlow is another deep-learning library developed at Google, and we'll be using Keras in a way that's layered on top of TensorFlow. Both were first released in 2015. As we use Keras, little glimmers of TensorFlow will show through in a few of the later lessons. We'll look at some advantages of PyTorch, a deep-learning tool released by Facebook in 2017, in the final lesson of the course.

These are all freely available libraries, written to take advantage of powerful hardware, and they incorporate the latest optimizers and network architectures. It's not too much to say that one of the drivers of the deep-learning revolution has been that everyone is welcome and everyone can contribute.

However, these terrific libraries that make it so much easier to get started can also contribute to a feeling that the machine-learning algorithms are inscrutable black boxes. That's why I want to spend one lesson working through a simple neural network algorithm in Python without using libraries.

LESSON 05 | OPENING THE BLACK BOX OF A NEURAL NETWORK

TRANSCRIPT

Let's start by returning to the green screen application we saw in the first lecture of the course. The algorithm we'll develop is a simple neural network called a linear classifier. A linear classifier is a rule that predicts the class of a given instance by taking each feature in the instance, multiplying it by a feature-specific weight, summing up the weighted features, and then checking if the total is positive or negative. This green screen image is a still of me, talking about green screen images, in front of a green screen.

What we want is a classifier that can label each pixel of the image with whether it is part of the green screen or whether it is part of me in the foreground. For conceptual simplicity, we'll consider each pixel in isolation. More accurate classifiers might include information about the surrounding pixels.

Our classifier takes a pixel value as input—the amount of red, green, and blue in the pixel's color—and it should output “true” if the pixel is part of the green screen and “false” if the pixel is part of the foreground image.

One reason that this problem is hard is that what we see as green is actually thousands of different shades of green. Literally thousands! So our classifier needs to generalize across the many possible shades.

We'll write code to grab a copy of the example green screen image. That will give our program a file to refer to. Once again, I'm using a web-based python service hosted by Google called the Colaboratory, or Colab for short.

For doing numerical computation, we'll import a Python library called NumPy, which is used by many of the machine-learning libraries. “Import numpy as np” means we'll be able to use the shorthand “np” to refer to elements of the NumPy library in what follows.

Next, we'll run “from keras.preprocessing import image”. That means we're getting a piece of the “keras.preprocessing” library—specifically one called “image”. Yes, Keras is a machine-learning library. Yes, I said we would not be using machine-learning libraries for this lesson. The “image” sub-library has some very simple programs for dealing with images. I promise we won't cheat and use any of the machine-learning functionality of Keras in this lesson.

The command “image.load_img” lets us name an image file and get its data. The image file is the one we just copied. We're storing the data in the variable “img”. Running the python command “display” on “img” causes the image file to be displayed. Hey, I know that guy!

The image datatype is great for displaying but awkward to compute with. Fortunately, the image library lets us convert the image into an array of pixels, which is much more convenient. We can run “image.img_to_array” to convert the image into an array. We’ll call it “arr” for short.

Since the screen-shot image has a little bit from another window at the bottom, we should clip that off. The command “arr = arr[:697,:]” selects a sub-rectangle of the pixel array up to row 697. We assign it back to the variable “arr”.

To see what the resulting image looks like, we can use the image library to turn the array of pixels back into an image data structure for display. The flag “scale=False” means we want the pixels displayed as is. Without this flag, the conversion scales the image brightness so it uses the full range of values.

Notice that the weird little strip at the bottom is gone now. We’re going to apply supervised learning to create a classifier that assigns green screen pixels to the positive class and foreground pixels to the negative class. That way, the positive pixels in the image can be replaced by a background image.

For a program to learn this difference, we need a training set with both positive and negative examples in it. To construct this training set, we can identify a rectangle of pixels that is definitely green screen and a rectangle of pixels that is definitely foreground. Let’s talk through how to use code to build up a training set.

The left side of the image is all green, so let’s grab that to serve as a set of positive examples. The “reshape” command in NumPy allows us to take this two-dimensional image and reformat it as a list of red-green-blue triples. Reshaping doesn’t change the content of the array, but it does change the form in a way that’s worth discussing in more detail.

In Python, a number is a number. But you can also reshape it to be a list of length 1. Or a 1×1 array, which is a tiny two-dimensional array. Let’s take the number 3 and make it into a NumPy array. We can ask its shape, or dimensions. We get back an empty list. This particular 3 is just a dimensionless number.

Let’s reshape this number into a list. We tell the “reshape” command to take x and make its shape “open paren 1 comma close paren”. The comma here is just a signal to Python that we’re talking about the list that contains the number 1. If we leave the comma out, it thinks the parens are just there to group the expression. Anyway, the variable x_list now contains that same 3 as before, but it’s been reformatted as a list of length 1.

LESSON 05 | OPENING THE BLACK BOX OF A NEURAL NETWORK

TRANSCRIPT

Let's make a list of 3 numbers: 56, 28, and 0. The shape of this list is "left paren 3 comma right paren". That is, it has one dimension, and the size of that dimension is 3. That's a list of length 3.

This particular triple of numbers happens to be the red-blue-green values of the color brown. And not just any brown: It's the official brown used by sports teams at my university. It's Brown brown. Go 56-28-0 bears! We can think of this list as being a pixel—one "picture element" of an image. It gives the recipe for making a specific color.

So three numbers together is a list or a pixel. Its shape is 3. The shape of the $1,215 \times 697$ green screen image of me is (697, 1215, 3). Reading right to left, we can think of the image as being made up of pixels, which are groups of three numbers. The pixels are arranged in lists of 1,215 pixels, which make up a row. And 697 of these rows make up the entire image. So the shape is (697, 1215, 3).

Returning back to our sub-rectangle from the green screen image, we selected a part of the image that consists only of the color green. The shape of this temporary array "tmp" is (697, 360, 3). It's the full height of the image, but narrower.

"tmp" is a three-dimensional array, representing a two-dimensional array of pixels. What we want is for it to be a list of pixels. That's also a two-dimensional array, but one with one dimension for the length of the list and one dimension for the three values that make up each pixel.

The reshape command takes the array of pixels and turns it into a list of pixels. The "3" in the list means we want the shape of the resulting array to have a 3 in the second position. The "-1" in the shape gets filled in by the reshape command to be whatever number is needed to account for all the data. That is, it computes the length of the list on our behalf. The resulting list is 250,920 pixels.

Now that we've captured a bunch of background pixels and made a list out of them, we can again use an array access with bracket notation to collect a diverse rectangle of pixels in the foreground. We've got a "yesList" consisting of green pixels and a "noList" consisting of foreground pixels. Now we're in a good position to construct our labeled training set.

NumPy's concatenate command takes a list of arrays and glues them together into a bigger array. We'll form an "alldat" array that is all of the data, all in one list. To make sure we can keep track of the green screen "yes" pixels and the foreground "no" pixels in "alldat", we'll make a list of labels.

It consists of a one for each element in the “yesList” and a zero for each element in the “noList”. NumPy provides the commands “ones” and “zeros” that make an array of all ones or all zeros with the given shape. We concatenate them together to get the overall list of labels. Next, we’ll design a program to learn from this data.

A machine-learning program has three parts: a representational space, a loss function, and an optimizer. The data we collected already forms the cornerstone of the loss function. And because pixels are low-level sensory data, a good choice for the representational space is a simple neural network.

Our network takes in the pixel color as three numbers: r, g, and b. We’ll multiply each of the three numbers by corresponding scaling factors, or weights: w_r , w_g , and w_b . These weights can take on any values: a hundred times red, a tiny fraction of green.

Notice that the color pure white is (0,0,0), so any setting of the weights will cause pure white to have a weighted sum of zero. To make sure white can be classified flexibly, we include a bias weight that shifts the sum. Here, that means we’ll include a fourth number, a constant c, that will get added into the total.

Now we’re ready to add everything up. We’ll say the rule outputs “green screen” if the total is greater than or equal to zero. If it’s less than zero, then it’s me in the foreground.

The representational space, then, is the set of all possible values for w_r , w_g , w_b , and the constant c. Selecting specific values gives us a linear classifier rule for classifying pixels. What we’ve built here is a neural network consisting of a single output unit that takes in the pixel values and outputs a value used to make a classification decision.

It’s a pretty simple network, but it’s more expressive than you might think at first. For example, let’s say we want to classify any pixel as green screen if the value of the green component g is bigger than the sum of the other two. Setting the weight for red to -1, the weight for green to 1, the weight for blue to -1, and our constant c to 0 captures this rule exactly. Negative r plus g minus b is greater than or equal to zero, so the green component g is bigger than the sum of red and blue.

For a loss function, we want something that scores a rule as having low loss if that rule agrees with the examples in our labeled data set. We can break down the loss for the entire data set into the sum of the losses for the labeled instances. So that leaves us the problem of defining the loss for each labeled instance.

LESSON 05 | OPENING THE BLACK BOX OF A NEURAL NETWORK TRANSCRIPT

An instance is a pixel color—values for r, g, and b—along with a 0/1 label, indicating green screen or foreground. A rule is the weights w_r , w_g , and w_b , and the offset c . If the output of the rule on that instance matches the label, we want to have zero loss. Otherwise, we can record a loss of 1.

That's the sort of thing we want. Loss grows when the rule doesn't fit the data. But the inequality in the rule is kind of awkward. It would be nicer if we were working directly with numbers. So we'll restructure the loss function a bit. We can do that because the loss function is something we're choosing. We want to set things up so good rules get low loss, and we have freedom to decide exactly how to make that happen.

To restructure the loss function, we'll make use of a sigmoid function. We saw with the sigmoid function last time, and the sigmoid function takes any input from negative infinity to infinity and maps it to a number between zero and one. At negative infinity, it starts at zero, then as the input approaches zero, the output starts to get bigger. At zero, it crosses 1/2, then it approaches 1 as the input grows toward positive infinity.

The sigmoid has the nice mathematical property of being a function that returns something between 0 or 1, just like the label. And in this case, we're only using the sigmoid once, so the expense of the sigmoid is not very significant.

Now we can do one more trick. We want the sigmoid expression to match the label. Subtracting the label from the sigmoid expression gives us a value of 0 when they match, and then either -1 or 1 when they don't match.

To make the sigmoid expression minus the label look just like the loss we wanted, all we have to do is square it. That leaves the 0 alone and the 1 alone, and turns the minus 1 into a positive 1. The loss becomes the difference between the value that this sigmoid expression returns and the value from the label.

Now let's translate the loss function into code. That will give us everything for a complete machine-learning setup, except the optimizer: We're doing supervised learning with a labeled data set, where the pixels of green screen background are labeled 1, and anything foreground is labelled 0. We've got a neural net representational space—three weights and an offset, pushed through a sigmoid. And we've got squared difference on the data for the loss function.

To make the data a little easier to work with, we'll add a fourth column to it. The column consists of all ones. Setting it up this way means we do not have to handle the constant c in any kind of special way. Instead, we can just think of it as a weight, w_c , that gets multiplied by 1 before it is added to the total.

Now our data has four columns, and they correspond to the four parameters in our representational space—the weight for red, the weight for green, the weight for blue, and now the weight for the constant, 1. To compute the loss, we need to multiply the four weights by the four values in each of the data points.

Because of the way we arranged the data, we can use standard matrix multiplication to accomplish this calculation. It multiplies the weights by the data and sums exactly the way we need it to for our rule. The matrix multiply leaves us with a list with one value for each of the instances in our data.

In Python, the “numpy” function “`matmul`” does the matrix multiplication. We then transform the resulting list of values using the sigmoid. It just so happens that the sigmoid function is 1 over $1 + e^{-\text{input}}$, so that's how we'll transform the value.

To complete the loss calculation, we look at the difference between the output of the sigmoid, “`y`”, and the list of labels, “`labs`”. We square each difference. Then we sum up all of these squared differences. Our loss function can now give us a score for any set of weights we choose.

Let's try a couple of random weights to see what kind of values we get. We'll repeat things 10 times. Each time, we'll pick a random vector of four weights. The random function returns values between 0 and 1, so I doubled them and subtracted one so all the weights are now between -1 and 1 . We'll print each of these weight vectors along with the loss we compute.

Because of the random choices in this code, when you run it, it'll come out somewhat differently. But the overall pattern should be similar. We get a wide range of loss values. Here, the largest is over 280,000, and the smallest is around 37,000. That's a somewhat wide range.

Randomly picking weights and seeing which leads to the lowest loss is already a kind of optimizer. It's not a very good one, though. Even the weights that gave us our lowest loss of 37,000 can probably be improved upon. But how?

LESSON 05 | OPENING THE BLACK BOX OF A NEURAL NETWORK TRANSCRIPT

Because our loss function is expressed in terms of a set of continuous functions, we can optimize using an approach known as gradient descent. That is, we can use basic differential calculus to find the direction to move the weights that decreases the loss the fastest.

So relatively quickly, we should be able to arrive at a set of weights where the gradient is nearly zero because the loss function has gone flat. At that point, no further local improvements are possible, and we've reached a local minimum for loss.

You can think of this process as follows. Consider any of the four weights that make up our rule. If we set three of them to fixed values and let the other one vary, we have a one-dimensional function mapping the choice of value for that weight to the overall loss that our loss function outputs. The curve for varying the fourth weight in our example has a nice, smooth shape, with a minimum loss for a weight value around -0.9.

The idea of gradient descent is that the slope of the function at any point tells us which direction we need to move to find the minimum of the curve. For a weight value greater than -0.9, the slope is positive, so we need to decrease the weight value to decrease the loss. For a weight value lower than -0.9, the slope is negative, so we need to increase the weight value to decrease the loss. In general, we want to increase the weight value by the negative of the slope.

Gradient descent carries out these slope calculations—it computes the gradient—for all the weights simultaneously. This computation allows an exact update for all the weights to be calculated. The gradient for this particular example of a linear classifier with a sigmoid slope is known as the delta rule. The Greek letter delta is a very common symbol for indicating a change or difference. So the delta rule says that each weight should be updated by taking the difference—the delta—between what the rule outputs for an instance and the true label for the instance.

The difference between what the rule outputs for an instance and the true label for the instance is written $y-l$. That difference is then multiplied by the derivative of the sigmoid, g' of s . Then all of that is multiplied by the input value to the weight, x_i . The resulting value is then scaled down to be a small number by multiplying by a learning rate, alpha. These Δw differences are then summed up over all training examples to produce a total change for the weights. This total change is subtracted from the weight to move down the gradient.

To apply the delta rule to our problem, we'll define a subroutine called "fit". This subroutine will take as input an initial set of weights "w", the data instances "alldat", and the labels "labs". It will produce as output a new set of weights with low loss.

We'll be iteratively updating the weights by computing the gradient and then moving them a step in that direction. The critical line of the code is the "delta rule" itself. Let's read it inside out to see how it is expressed in code.

" $\text{labs} - \text{y}$ " is the difference between the true label and the output of the rule. That difference is multiplied by the gradient of the sigmoid, which we compute by taking e to the minus h using the numpy function "np.exp" and multiplying by y squared. Those values are reshaped into a matrix, which is multiplied by the input data "alldat". Finally, "np.add.reduce" sums up all of the recommended updates over all the training examples, resulting in a vector of accumulated updates with one component for each of the weights.

We're going to iteratively change the learning rate alpha to make it bigger, but we'll avoid taking steps that are too big and cause the loss to increase. The variable "done" starts off "false" and is set to "true" when we discover that any further step to decrease the loss is too small to be worth it, which I defined as a decrease in loss of less than 0.0001 for this problem.

We'll loop until done. In the loop, we'll flip a weighted coin. We'd like to monitor progress, so about one out of every hundred iterations, we print the weights, learning rate, and loss.

Next, we take the gradient of the loss function with respect to the four weights. The calculation is very similar to what we talked about when we were computing the loss. When these calculations are done, "delta_w" represents a direction to move the weights that decreases the loss the fastest.

We can take a step in this direction, but the size of the step is important. If we take a very small step, progress will be slow. If we take too big of a step, we might reach a set of weights where the loss is increased. Professional-grade optimizers work very hard to get this issue right. For the purposes of this example, we'll do something relatively simple.

First, we compute the "current_loss"—the loss we get with the weights w. We want to improve on this loss. Next, we double alpha with the hope that we'll be able to take a big step. Then we calculate the new weights, "neww". They're just the old weights plus an alpha-size step in the "delta_w" direction.

LESSON 05 | OPENING THE BLACK BOX OF A NEURAL NETWORK

TRANSCRIPT

Now, we check: Is the loss with the new weights any better than the loss with the old weights? If the new loss is better, we skip the while loop. Otherwise, we assume we must have taken too big of a step. Let's halve alpha and try again.

But there's no need to keep halving alpha forever. At whatever point alpha times the largest component of "delta_w" becomes very, very small, we'll declare ourselves done. We've reached something like a local min where the reasonable step size is vanishingly small and the learning rate is just about zero. Continuing longer, even for huge amounts of additional time, would scarcely improve the loss at all. Our solution is locally optimal.

We print alpha and "delta_w" just to see our solution. But until the new alpha we picked gets very tiny, we once again recompute our new weights and go back to the top of the loop. After the loop, if we're not done, we set w to be the new weights—the ones that gave lower loss—and we return to the outer loop to search for more improvements. Finally, when the loop is done, we return the weights we ended up with.

That's a relatively simple implementation of a gradient descent optimizer. It is specific to this particular representational space: weights for each component of the color. It is also specific to our chosen loss function: squared error of the sigmoid of the linear combination of the color and the weights. Machine-learning libraries provide more general implementations that let you choose other architectures and loss functions without having to reimplement the optimizer for each one.

Let's look at our optimizer program in action, starting with some random weights. We start with a loss at 23,008, and now you see the loss drop immediately to 492, and then gradually work its way, through a few hundred iterations, down to 404.7, when the procedure finally stops.

Also printed are the weights w, and we can see that they have changed a fair amount. As we might have guessed, the largest magnitude weight, 0.11, is on the second component of the weight vector—green. After all, the classifier is trying to recognize any of the thousands of shades of green that might be visible on the green screen.

The red component is strongly negative, almost -1, indicating that red very rarely has any part at all in green screen pixels. Conversely, blue has become only weakly negative, indicating that the presence of blue in a pixel is only an indirect indicator that a pixel is part of the background.

So it looks like our optimizer is doing pretty well at reducing our loss on the two rectangles we took from the image as our training set. Or is it? With neural networks, and most machine-learning approaches, it's difficult to interpret the loss in absolute terms. What really matters is whether the solution found using the training-set data also performs well on the original problem.

Let's write a routine that can take the weights learned from the training set and use them to classify all the pixels in the original green screen image. We'll make the green screen white and the foreground black. Ideally, it should pick out my silhouette and the card over my heart.

To apply the classifiers to the pixels in the image "arr", we need to first flatten out the two-dimensional array of pixels into a one-dimensional array—a list—of pixels. We concatenate a 1 onto each of them so that we can apply the classifier to each. A matrix multiply of the image pixels with our weights w produces a set of numbers. Our rule is that negative numbers are foreground and positive numbers are green screen, so we threshold at 0.

We reshape the resulting list of Boolean values so that we can concatenate the list with itself. Each entry of the list is now 0,0,0 or 1,1,1. We reshape this list to make it the same shape as the original image "arr". Finally, we display this image on the screen. We allow the conversion to scale the values so they become 0,0,0—black—and 255,255,255—white.

The classifier does a pretty nice job! You can see me, all in black. There's maybe a speckle or two on my belt and around my waist. These speckles will allow the green screen to show through, which is not perfect. There's also one small black streak across the bottom of the image that corresponds to the edge of the actual photo. To get these right, we could add more examples to our training set and trim the image more carefully.

But, overall, it works! We can put me into any scene we want now. The gradient descent optimizer we built is somewhat brittle. It can be slow to converge. And it occasionally encounters numerical instability when it tries to exponentiate numbers that are just too big. Most importantly, it was written only for the specific loss function we designed for this problem.

Modern deep-learning libraries provide much more robust mechanisms for optimization. An optimizer like Adam, which we'll use in future lessons, would handle not only the loss function we developed for this lesson, but a huge class of other loss functions just as well.

LESSON 05 | OPENING THE BLACK BOX OF A NEURAL NETWORK TRANSCRIPT

Going forward, we'll take advantage of the software engineering effort that has gone into making these machine-learning libraries much more flexible and dependable. These libraries almost make it possible to treat optimization like a black box, requiring no human attention. But in practice, we'll see that this experience of developing our own optimizer will help us help the optimizer do more of what we want.

LESSON 06

BAYESIAN MODELS FOR PROBABILITY PREDICTION

Many things in machine learning seem backward. Instead of writing a program that carries out a goal, we define a goal and let the program write itself. This lesson addresses another backward idea: classifying instances by learning to generate instances from their labels.

From Effects to Causes

In the other classifiers you've learned about so far, such as neural networks and decision trees, the learned classifier maps an instance to a label. But it's also possible, and sometimes very useful, to think of the label as generating the instance itself. This kind of representational space is called a generative model.

In a generative model, the problem of classification is turned into a problem of modeling a distribution. The key to this approach is a powerful mathematical idea for reasoning from effects to causes known as Bayes's rule.

In this lesson, we'll be combining two things: the backward logic of Bayes's rule, where we imagine classes generating instances probabilistically; and a particularly simple model of the generation process, where each feature is produced separately from the others. This combination of Bayesian reasoning and simple model is called **naive Bayes**.

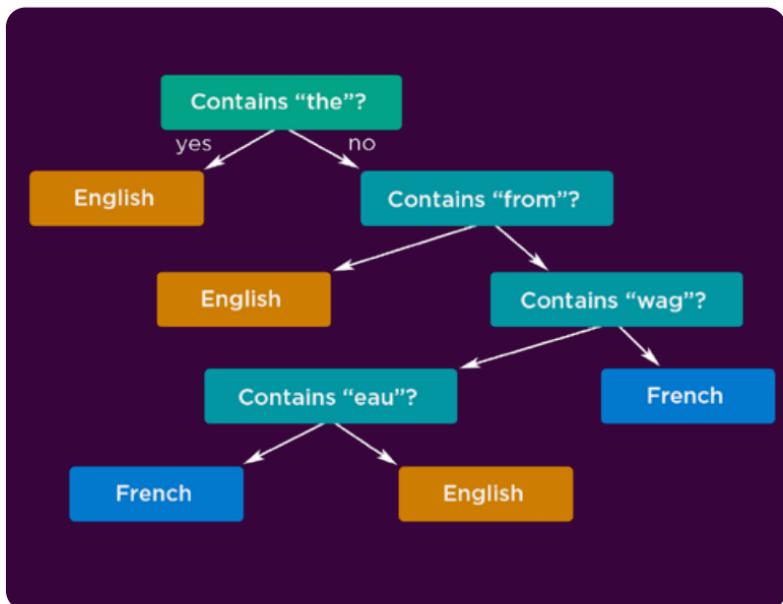
Text Classification

Consider a program designed to determine whether a short piece of text is in English or French. Using the decision tree approach, such a program could be written with a bunch of if-then rules. Each rule looks at the individual words in the text and cumulatively decides whether the presence of individual words in the document is enough to determine which language it was written in. For instance, if “Contains *the*?” is true, that might be enough to declare that the passage is in English.

So a decision tree would ask a series of questions about the text and use the answers to decide whether to predict “English” or “French.”

But it can be much simpler to think about it the other way: In particular, if we knew which language it was in, could we predict the kinds of words we would see?

Figuring out a way for machines to generate sentences the way humans do will be an important aspect of how researchers crack the problem of creating true intelligence.



Well, sure. If it's English, it would tend to use words like *Let's* and *go* and *children*. But a French snippet would use words like *Allons* and *enfant* and *de* and *la* and *patrie*.

If we could learn the kind of text generated by someone who speaks English and the kind of text generated by someone who speaks French, we could ask whether a given piece of text is more likely to have been generated by one or the other—and use those probabilities to classify the text.

Learning how text is generated could potentially be an easier problem because we have lots of text data out there to learn from.

Naive Bayes

The approach to machine learning known as naive Bayes exploits the Bayes's rule formula, which lets us flip around what we're assuming is given—our input—and what we are producing as an output. It is Bayes's rule that lets us go from effects to causes.

An amazing thing about naive Bayes models is how quickly they can run—that’s because learning just takes one pass through the training data. In contrast, a decision tree needs to consider all possible splits for each node, and a neural network requires many iterations of gradient descent.

Back in the early days of Google, naive Bayes was their tool of choice. They observed that, for the mountains of data they were processing, naive Bayes would learn extremely effective rules. And training naive Bayes on an enormous database is very, very inexpensive.

But Google changed their tune when deep learning became ascendant.

In 2016, Google declared itself a “machine learning first” company. While they had, from the very beginning, used machine learning approaches in their work, Google came to see that machine learning was one of their primary differentiators from their competitors.

Ever since, the company has embraced additional, cutting-edge approaches to machine learning, even doing some of the best research in the area. But it’s interesting to remember that even Google went through a period when naive Bayes was the best tool available for the problems they faced. It remains so for many problems today, especially when data for training large-scale deep networks is scarce.

Naive Bayes is especially good for working with language data.

Bayesian methods can be used to predict which way a tower of blocks might fall and model how people decide what to do in the face of partial information. They can translate between languages and figure out what pronouns are referring to, even in complicated sentences.

Bayesian methods get their power from letting the learning process focus on capturing the forward direction—the process by which examples are generated. Later, it reasons backward toward these causes, based on the observations available.

Why does the Bayesian approach of going backward work so well? The underlying reason is that real language is produced by intelligent agents who wish to communicate an idea to other intelligent agents.

Data that lacks this generative structure—or data for which the generative structure is too complex—are less suited to Bayesian methods.

The key takeaway here is a deep connection between language and probability:

The language of probability provides an excellent way to model the probability of language.

That's why even a very primitive probabilistic model can turn raw data into meaningful patterns. And that's why it's useful for a Bayesian probability model to reason from effects back to causes—just like humans do.

Try It Yourself

Follow along with the video lesson via the Python code:

[L06.ipynb](#)

Python Libraries Used:

graphviz: Graph visualization.

sklearn.metrics.confusion_matrix: Computes a confusion matrix.

sklearn.naive_bayes.MultinomialNB: Naive Bayes learner for binary features.

sklearn.tree: Scikit-learn's decision tree algorithms.

Key Terms

naive Bayes: A specific type of Bayesian network that models the observed features as being probabilistically related to the class but independent of one another.

READING

Domingos, *The Master Algorithm*, chap. 6.

Mitchell, *Machine Learning*, chap. 6.

Pearl, *Probabilistic Reasoning in Intelligent Systems*.

Russell and Norvig, *Artificial Intelligence*, chap. 12 and sec. 20.2.

QUESTIONS

- When would it make sense to use a generative Bayesian approach instead of a discriminative neural network or decision tree approach?
- In decision tree learning and neural network learning, the overall loss function can be thought of as accuracy on the training data. (It's a little more complicated for decision trees, since they are optimized top-down and not globally.) Naive Bayes learning looks very different. How would you describe the loss function used to choose the parameters for naive Bayes?
- How does the naive Bayes classifier compare to neural networks and decision trees on perceptual data? Try training all three on the MNIST data from [Lesson 04](#). Rank the three approaches in terms of accuracy and running time.

Python Libraries:

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

random: Generates random numbers.

sklearn.datasets.fetch_openml: Provides access to OpenML datasets.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

sklearn.neural_network.MLPClassifier: Scikit-learn's neural network algorithm (multilayer perceptron).

sklearn.utils.check_random_state: Repeatable, random way to split training and testing data.

Answers on page 478

Bayesian Models for Probability Prediction

Lesson 6 Transcript

A lot of things in machine learning seem backwards. Instead of writing a program that carries out a goal, we define a goal and let the program write itself. In this lesson, we'll talk about another backwards idea: classifying instances by learning to generate instances from their labels.

In the other classifiers we've talked about so far, like neural networks and decision trees, the learned classifier maps an instance to a label. It's pretty straightforward—emphasis on the *forward*. But as we'll see, it's also possible, and sometimes very useful, to go the other way, thinking of the label as generating the instance itself. We'll call this kind of representational space a generative model. The problem of classification is turned into a problem of modeling a distribution.

The key to this approach is a powerful mathematical idea for reasoning from effects to causes known as Bayes's Rule. We'll be combining two things: the backwards logic of Bayes's rule, where we imagine classes generating instances probabilistically; and a particularly simple model of the generation process, where each feature is produced separately from the others. This combination of Bayesian reasoning and simple model goes by the name naive Bayes.

For example, let's consider a program designed to determine whether a short piece of text is in English or in French. Using the decision-tree approach we've already seen, such a program could be written with a bunch of "if-then" rules. Each rule looks at the individual words in the text and cumulatively decides whether the presence of individual words in the document is enough to determine which language it was written in. For instance, if "Contains the word *the*?" is true, that might be enough to declare that the passage is in English.

That's what a decision tree would do. It would ask a series of questions about the text and use the answers to decide whether to predict "English" or "French." But it can be much simpler to think about it the other way. In particular, if we knew which language it was in, could we predict the kinds of words we would see?

Well, sure. If it's English, it would tend to use words like *let's* and *go* and *children*. But a French snippet would use words like *allons* and *enfant* and *de*, *la*, *patrie*. Oof. I'm sure native French speakers are cringing at my pronunciation. *Je suis désolé*. OK, that's making it worse, isn't it?

LESSON 06 | BAYESIAN MODELS FOR PROBABILITY PREDICTION

TRANSCRIPT

Anyway, if we could learn the kind of text generated by someone who speaks English and the kind of text generated by someone who actually speaks French, we could ask whether a given piece of text is more likely to have been generated by one or the other, and use those probabilities to classify the text. Learning how text is generated could potentially be an easier problem because we have lots of text data out there to learn from.

Let's stick with English and work through a real-world example involving spam detection. Spam is a significant problem in email and text messaging. People receive communications that are unsolicited and sometimes even fraudulent. Viewed as a machine-learning problem, we want a way to take a message as input and to classify it as either normal or spam.

In English, and many other languages, it's natural to use spaces to break the string of characters into its constituent words. A word is a pretty powerful chunk of meaning. It's short enough to allow for some overlap between messages, but long enough to convey semantic relationships. If I say the word *lungs*, you immediately get a vivid sense that we're talking about anatomy, not architecture.

Here's a message from a spam-detection data set consisting of text messages in the UK made available in 2012: "yo come over carlos will be here soon." The data set has 3,572 messages, 13% of which are spam. This message is not spam.

For comparison, here's a spam message, where "bt" is likely short for British Telecom: "congratulations! thanks to a good friend, u have won the 2000 dollar christmas prize. to claim is easy, just call 0871-872-6971, now only 10 pence per minute, bt national rate."

Breaking strings into words leads to a simple, powerful, and clever representation for text. We have one feature for each word in the English language. For a given passage of text, a feature has a zero value if the word does not appear in the text. If the word does appear in the text, we put a 1 in there.

Keeping track of the presence or absence of individual words in a text is sometimes called the bag of words representation. It's as if all the words are just thrown into a bag, and all we can tell is what's in the bag, or out of the bag—but nothing about the original ordering.

In a bag of words representation scheme, a string gets a yes-no vector representation for every word in a dictionary of words. Suppose we have just a tiny dictionary of 18 words. That means a string is represented by a vector with 18 components.

Then, suppose a message comes in, such as: “This message is not spam, it is real!” At that point, the positions in the dictionary corresponding to words that appear in the message are turned on. Note that the bag of words representation gives this message exactly the same representation as: “This message is not real, it is spam!”

The fact that scrambling the words doesn’t change the representation seems like a shortcoming of the representation scheme. Clearly, we are losing some information, and that information might be critical to the application. And yet more than half a century of research has shown that you can get remarkably far with this simple-minded way of looking at language.

We’re going to focus on bag of words representations in this lecture. But it’s worth mentioning an important variant of the idea of using a bag of words. Consider the message: “I love my puppy! It’s the best puppy in the world!” and “I’m in love. Not just puppy love—the real kind!”

Both of these messages include *puppy* and *love*. But the first one is about puppies, and the second one is about love. One strong indicator is the number of repetitions of the words. The first contains *puppy* twice and *love* once, and the second is the opposite. The repetitions carry valuable information. It can make sense to not just distinguish between the presence and absence of words but also their frequency in the message. That could mean filling in the vector with counts of the words that appear in the message.

But the word *the* appears twice in the first message, making it seem like it’s carrying as much information as *puppy*. But it’s not. Words like *the* are common in all messages. It might be a good idea to downweight those kinds of words, perhaps by the inverse of the number of messages they appear in. That is, if a word appears t times in a message and has a frequency f across all of the training examples, we use a weight of t divided by f in the vector.

One concrete way to do so is to represent a message or document by a giant vector. The vector has one component for each dictionary word of that word’s count in the document, downweighted by the number of documents that word appears in.

This idea is an old one. It dates back to the pioneering work of computer scientist Gerry Salton and his colleagues at Cornell in the 1960s. The weighting, used in a vector space model, is known as term frequency, inverse document frequency, or TFIDF. Updated versions of frequency-based

LESSON 06 | BAYESIAN MODELS FOR PROBABILITY PREDICTION

TRANSCRIPT

weighting are important in assessments of document relevance for sites like Google and Bing. With the bag of words representation in our pockets, along with counts, we can approach spam filtering as a machine-learning problem.

I prepared a file of the words that will appear in the training data. Each line of the file consists of a word preceded by a count of the number of times the word appears. When a rare word appears and we haven't made a feature for it, we'll just replace the word with a marker saying the word was unknown. Such unknown words are sometimes abbreviated u-n-k and called "unk"s.

We're only going to make a feature for words that appear at least twice in the training data. A singleton word sometimes goes by the fancy term hapax legomenon. Of course, I will only mention this term once. Anyway, for our analysis, we're going to skip these singleton words because we do not have enough examples to be able to generalize from them. All other words are added to the dictionary, along with a count of how many times the word appears.

Let's load our file of words into Colab. Tokenize is a routine that turns a string into a vector. The vector is initially all zeros. Then, we bust the string up into words using spaces as the delimiter. We count up the number of appearances of each word and return the resulting vector. The "getdat" routine takes a file as input, reads in each line, turns the sequence of words into a vector, and returns the set. Both the instance vectors and the labels are returned.

We're next going to call the "getdat" function to assign the data to variables in Python. Since "getdat" returns two items—the instances and the labels—we can assign one variable to the instances and one variable to the labels. "Traindat" will get the instance vectors, and "trainlabs" will get the labels vector. We use this "getdat" routine to read in the training data, "spam-train.csv", and the testing data, "spam-test.csv".

To have a baseline for comparison, let's first build a decision tree. As before, we set up the classifier, train it, and see what it predicts on the testing data. We get about 94% accuracy, which seems quite solid.

The resulting decision tree implements the rule: If the message has "call" in it, but not "i", it's spam. If it doesn't have "call", but it has either "txt" or "www", it's spam. Those are the blue nodes. Otherwise, it's classed as non-spam—a normal message. Here, that's orange.

The rule makes a lot of sense. The spam messages include exhortations to contact them back. In "claim ur award call free," *call* indicates spam. But in "can i call now," the *i* partly cancels it out.

We can beat 94% by turning things around. Remember that these vectors aren't just randomly created. They come about from actual people writing actual messages.

Here's how we've been talking about classifiers up to this point. There's some hidden rule that takes vectors as input and produces labels as outputs. We need to find a rule that approximates that hidden rule as accurately as possible. Such algorithms are sometimes described as discriminative. It's not because they discriminate against people, although, as we'll see in a later lecture, they can be misused that way.

Discriminative here just means that we are judging algorithms solely on their ability to discriminate—to tell the difference—between vectors that belong to the different classes. But in the spam-filtering world, there's no reason to believe that such a rule exists. In actuality, when there's a message being written, there's two kinds of people who might be doing the writing: spammers and normal people just trying to communicate.

Spammers write spam messages. We use suspicious words like *call* and *free* and *claim*, but also innocent words like *the* and *you* and *to*. Normal people write normal messages. We use a lot of personal words like *me* and *my* and logical connectives like *so* and *not* and *can*.

When it comes to the creation of spam, it's not like the rule comes first and then messages get tagged with the labels by the rule. First, we can make a random decision of whether spam or non-spam is being written. That's the label of the instance. The probability that the chosen label is y is given by probability of y . In the data we have, it's more likely that a message is normal, so the probability that the label y equals spam is less than a half. In fact, it's around 13%.

Then we generate an instance vector x conditioned on the label y . We call this quantity a conditional probability and read it as “the probability of x given y .” That is, I produce a message at random, and the probability of the specific message depends on whether I am acting as a spammer or as a normal person.

That's our generative process—the way we are imagining the strings in our data set are created. Now we need to think about how to reason backwards from this generative process to assigning a label to a newly observed string.

In the forward direction, we have a normal person and a spammer sending out messages. For conceptual simplicity, I'm using colors to stand in for specific messages. In real life, of course, there are more than just a handful of possible messages.

LESSON 06 | BAYESIAN MODELS FOR PROBABILITY PREDICTION

TRANSCRIPT

OK, I just got a message. I haven't looked at it yet. Assuming I just get one of the sent messages at random, what's the chance it came from a spammer? In this diagram, there are 20 messages from a normal person and 9 from a spammer, so the chance a random message came from a spammer is 9/29, or about 31%. We write that quantity $\text{Probability}(\text{spammer})$. That's our prior—the probability of a message coming from a spammer before we look at the message itself.

Now we look at the message. It's purple. Now what's the probability it came from a spammer? Looking at the diagram, there are four purple messages. Three came from a spammer and one came from a normal person. So amongst the purple messages, there's a 3/4 or 75% chance it came from the spammer. That's our posterior—the probability of the message coming from a spammer after seeing the message, or $\text{Probability of spammer given purple}$.

That's a straightforward way of doing the computation, but it is not accurate when the message counts are small. We can reorganize the computation in a way that generalizes better to sparse data.

First, let's reflect on how we calculated the posterior. I expressed it as counts, but we can also think of it in terms of probabilities over the space of all 29 messages. The probability of the message coming from a spammer given that it's purple comes from the probability that a message is from a spammer and is purple. That probability is 3/29, because 3 of the 29 messages fit the description of being from a spammer and being purple. That quantity is then divided by the probability of a purple message overall. Since 4 of the 29 messages are purple, that probability is 4/29.

How did the messages come about that are from a spammer and are purple? First, they came from a spammer, then the spammer generated the purple message. The probability of a message coming from a spammer is $\text{Probability}(\text{spammer})$, which we already calculated as 9/29. The probability of a spammer generating a purple message is 3/9. So the probability of a message being a purple message generated by a spammer is the product of these probabilities, again 3/29.

Note that we can also compute the probability of a purple message. It's the probability that a purple message is generated by a spammer plus the probability that a purple message is generated by a normal person. Notice that the probability of a normal person generating a purple message can be calculated using the same process we used to figure out the probability that a spammer generates a purple message.

Putting all of these pieces together, we can calculate the probability of the cause—that the purple message came from a spammer—just using the probabilities of the effects: how likely a spammer is to produce a purple message.

Alright, let's bring this insight to bear on the machine-learning problem. We've learned the generative process by estimating the probability of the classes y and the probability of a feature vector x given the classes y . Now we need to go from a new vector x -prime to the probability of its label y -prime. That will tell us, for example, the likelihood a message is spam or normal. Finally, we can proclaim the most likely class as our guess. We'll say a message is spam if probability of spam given message is greater than a half.

To make a concrete algorithm, let's look more closely at each of these expressions in turn. The first one is just a single probability, and you can think of it like flipping a weighted coin. For this data set, I've already said that the probability of spam is about 13%.

The estimation procedure is pretty intuitive. You just estimate the probability of the event happening as the ratio of the number of times it did happen to the number of times it could have happened. The technical term for this approach is maximum likelihood estimation. That just means the estimate maximizes the likelihood that the observed data would be generated. And it is provably true that the way to maximize the likelihood of the observed data is to set the probability parameters of our model using the frequency observed in the data.

Next, how can we define the probability of an instance given the labels spam and normal? Here, we'll use the insight that the bag of words representation is simpler than considering sequences of words, but still retains a lot of the meaning of a message.

And since we're dealing with probabilities for a bag of words model, we're going to imagine that a spammer creates messages just by drawing words randomly, one at a time. The spammer has a specific probability of choosing any given word. Depending on the word, that probability might be quite different from the chance that a normal person would use that word.

You can think of this process as taking all of the words that make up spam messages, writing them on ping pong balls, and throwing them into a big air-mixer scrambling machine. To make a message, we pull out a ball, write down the word that appears on the ball, and then toss

LESSON 06 | BAYESIAN MODELS FOR PROBABILITY PREDICTION TRANSCRIPT

the ball back into the bin. We're throwing them back—sampling with replacement, as they say in statistics—to allow a word to be chosen multiple times. We repeat the process of drawing words until we've filled out the whole message.

Anyway, we're composing messages, one word at a time, by pulling ball after ball out of the mixer. This model of the writing process is grounded in terms of probabilities. And probabilities are important because they provide a way of scoring each message by how likely it is.

The bag of words assumption is that each word is selected independently of the others. So that translates into being able to calculate the overall probability of a message from its words by simple multiplication. We can estimate our conditional probabilities by counting words in the training data.

At classification time, we need to compute the probability of the possible labels for a new message x -prime. So we want probability of y given x -prime. And since x -prime is a new message, that's not one of the quantities we have already estimated.

We can use Bayes's rule to write the probability of y given x -prime as the probability of y and x -prime divided by the probability of x -prime. Multiplying by the probability of y over the probability of y doesn't change anything, since the probability of y over the probability of y is 1.

But it lets us reorganize the calculation so the probability of x -prime given y pops out. That's a quantity we figured out in step 2. The probability of y we figured out in step 1. And the probability of x -prime is just a normalization factor, which we can compute—or simply ignore it, because it doesn't depend on y and therefore doesn't influence which class we pick as the label for x -prime.

What we have here is an expression that is computable, using quantities we can extract very easily from our training data. Just take a new message, plug in the various quantities, and see whether it was more likely to be produced by a spammer or a normal person. Whichever is more likely is our best guess.

This approach to machine learning is known as naive Bayes. It exploits the Bayes's rule formula, which lets us flip around what we're assuming is given—our input—and what we are producing as an output. It is Bayes's rule that lets us go from effects to causes, from observations to facts, from symptoms to diseases.

Returning to the example to finish it up, if we learn a naive Bayes classifier for the data, we get 96% correct. That beats the 94% correct that we got with the small decision tree. It's only slightly more accurate.

But, the naive Bayes classifier also tells us, for each possible word that can appear in a message, whether its presence is evidence that a message is spam or if its presence is evidence that the message is not spam. It precisely quantifies the degree to which each piece of evidence contributes to the overall assessment.

An amazing thing about naive Bayes models is how quickly they can run. That's because learning just takes one pass through the training data. In contrast, a decision tree needs to consider all possible splits for each node, and a neural network requires many iterations of gradient descent.

Back in the early days of Google, naive Bayes was their tool of choice. They observed that for the mountains of data they were processing, naive Bayes would learn extremely effective rules. And training naive Bayes on an enormous database is very, very inexpensive.

But Google changed their tune when deep learning became ascendant. In 2016, Google declared itself a “machine-learning first” company. While they had, from the very beginning, used machine-learning approaches in their work, Google came to see that machine learning was one of their primary differentiators from their competitors.

Ever since, the company has embraced additional cutting-edge approaches to machine learning, even doing some of the best research in the area. But it's interesting to remember that even Google went through a period when naive Bayes was the best tool available for the problems they faced. It remains so for many problems today, especially when data for training large-scale deep networks is scarce.

We've seen that naive Bayes is especially good for working with language data as the bag of words assumption does a good job of capturing the topics of a passage of text. But the Bayesian idea of reasoning from effects to causes can be used more generally, in settings where the naive bag of word model is replaced with a model that preserves more information.

The broader, non-naive version of the Bayesian idea allows probabilities of words that are tied to each other. Words are no longer assumed to be independent. And sequences can be modeled too. General Bayesian models can be even more powerful and flexible. But the speed advantage of naive Bayes is sacrificed. Other Bayesian models can be much more computationally demanding.

LESSON 06 | BAYESIAN MODELS FOR PROBABILITY PREDICTION TRANSCRIPT

Bayesian methods can be used to predict which way a tower of blocks might fall. Bayesian methods can model how people decide what to do in the face of partial information. Bayesian methods can translate between languages and figure out what pronouns are referring to, even in complicated sentences.

Bayesian methods get their power from letting the learning process focus on capturing the forward direction—the process by which examples are generated. Later, it reasons backwards towards these causes based on the observations available.

Here's an example of a more sophisticated Bayesian model used in language processing. Consider the sentence: Zoey selected her words very (blank). What would you fill in there? I was thinking *carefully*, but *quickly* also works. Maybe *randomly*? The context strongly suggests an adverb.

By contrast, naive Bayes has no sense of sentence structure. It would assign the highest probability to *the*, since that's the most common English word. I counted up the words used up to this point in the lesson. The word *the* occurs 300 times, and *randomly* appears 6 times. So naive Bayes assigns *the* a probability 300 over 6, or 50 times higher than that of *randomly*.

But *the* almost never appears at the end of a sentence. It's improbable. We need a probability model that is more sophisticated than the bag of words and makes use of grammatical probabilities about sentence structure. Grammars are a way of generating sequences. And one very simple grammatical model is known as a probabilistic context-free grammar.

Context-free means each expansion is nested and therefore isolated from the context of every other expansion. In the sentence "Zoey selected her words very carefully," *Zoey* and *words* are playing the role of nouns. *Very* and *carefully* are adverbs. *Selected* is a verb. And *her* is a possessive pronoun.

These parts of speech can be grouped together into phrases. *Zoey* is a noun phrase. *Her words* is another noun phrase. *Very carefully* is an adverbial phrase. Going to a higher level, *selected her words very carefully* form a verb phrase. And with *Zoey* as our noun phrase stuck to the front, we have a well-formed sentence.

Viewing things the other way, a sentence is expanded into a noun phrase followed by a verb phrase. A noun phrase might just be a single noun, but it can also be a noun modified by an adjectival phrase, and so on. So this kind of probabilistic context-free grammar is a generative model that assigns concrete probability values to the likelihood that each box can be expanded.

For example, the “noun phrase” box could expand to any of several possibilities. Selecting probabilities for all possible expansions results in a probability distribution over all possible sentences.

Natural languages like English are not perfectly captured by context-free grammars. Expansions do not always nest perfectly, because long-range sharing of contextual information sometimes matters. But a context-free grammar captures a lot more of the structure of English than a naive Bayes model can, while retaining some of its computational efficiency. As a result, a context-free grammar is a considerably better probability model than naive Bayes.

Scholars at the University of Pennsylvania released a treebank of diagrammed sentences beginning in 1992. Data in the treebank is like our Zoey sentence. Thanks to the painstaking efforts to create data files with many grammatically labelled sentences, it became possible for a probabilistic grammar to be learned in a supervised fashion. The training process is much like the one in naive Bayes: count up how often each of the transformations takes place in the data set.

A significant advantage of naive Bayes models, though, is that no special tagging is needed. The probabilities can be computed directly from an unadorned collection of text. Thus, probabilistic grammars are better when diagrammed data is available and greater precision is needed. But naive Bayes provides a greater ease of training that is sufficient for a surprising variety of cases.

Why does the Bayesian approach of going backwards work so surprisingly well? The underlying reason is that real language is produced by intelligent agents who wish to communicate an idea to other intelligent agents. Data that lacks this generative structure, or data for which the generative structure is too complex, are less suited to Bayesian methods.

That said, more recent Bayesian methods have combined with deep learning to capture richer context available in patterns of the other words in the sentence. These probability models go beyond the rigidly nested, context-free structure we’ve seen. And they’ve gotten very good. A method trained in early 2019 called GPT-2 was already composing eerily reasonable-looking messages by using richer context-sensitive probabilities to generate each word of the message.

LESSON 06 | BAYESIAN MODELS FOR PROBABILITY PREDICTION TRANSCRIPT

Still, even these more recent methods are not capturing the true process by which ideas are turned into sentences. Figuring out a way for machines to generate sentences the way humans do will be an important aspect of how researchers crack the problem of creating true intelligence.

We'll return to these ideas in a later lesson on networks that output language. But the key takeaway here is a deep connection between language and probability: The language of probability provides an excellent way to model the probability of language. That is why even a very primitive probabilistic model can turn raw data into meaningful patterns. And that's why it's useful for a Bayesian probability model to reason from effects back to causes, just like humans do.

LESSON 07

GENETIC ALGORITHMS FOR EVOLVED RULES

Some representational spaces suggest clear approaches for doing optimization. When you have a continuous problem and you understand the shape of the problem very well, then gradient descent and neural networks are a great fit. When you have a discrete problem with a clear set of features to branch on, decision trees are a terrific way to go. When you have a clear generative process, Bayesian networks are there to help. But there are many times—especially when you are just starting out in a new problem area—when you don't yet know what will work well. Enter genetic algorithms.

The Rise of Genetic Algorithms

At the dawn of computing, in the late 1940s, Alan Turing was grappling with the question of how we might make machines smart. It was far from obvious how to write a program that would replicate the subtle decision-making and learning skills that humans possess.

But in a 1950 paper that introduces the “imitation game,” now known as the Turing test, he proposed that hand-programming adult-level intelligence was neither practical nor necessary. Instead, we just needed to create a “child program” that would learn from experience, much the way a human child learns. Basically, he was hinting at machine learning of the sort you’ve been studying in this course.

But where does the child program itself come from? Turing speculated that machine learning could be complemented by a procedure that searches for good child programs using a random generate-and-test technique. He likened this kind of mechanism to the process of evolution.

Why can’t we just turn to neural networks as a way to solve the problem? A typical neural network using gradient descent is demanding: It uses derivatives from calculus to compute the rate of change of the loss function as a function of changes in the network’s parameters.

But many problems lack a clear path from the loss function back to the parameters being learned. There’s no concrete function for calculus to do its work on.

For example, if a problem has a real-world component, such as predicting human reactions to a decision or optimizing the movement for a complex physical robot, we do not have an explicit mathematical function that defines the problem we are solving. Without such a function, we cannot take derivatives or do gradient descent.

Natural evolution faces a similar problem, as it optimizes organisms for their ecological niches without detailed mathematical representations of how changes to the organisms will impact their reproductive fitness.

This insight suggests bringing the principles of evolution to bear in machine learning—in the form of what are now called genetic algorithms.

Genetic algorithms caught on in the 1970s and 1980s through the work of John Holland and his students, who formalized evolutionary optimization as a computational problem-solving technique.

By 2010, genetic algorithms had already produced a variety of results competitive with, yet also different from, solutions created by humans in fields ranging from analog circuits, to assembly-code creation, to software repair.

One of the benefits of adopting an evolutionary approach to optimization is that it works well on essentially any problem. That also means we can try it on hard problems that we don't know how to address sufficiently to use other machine learning approaches.

Genetic algorithms harness nature's best optimization strategy—evolution—to make smarter programs.

Evolutionary Approaches

Evolutionary approaches act as an optimizer: They find high-scoring rules using the principles of natural selection.

If we intend to follow an evolutionary process, then it's worth being clear about the main components that Charles Darwin advanced in his theory of natural selection:

1. Individual organisms vary in their traits.
2. Organisms compete for resources, with some reproducing and some not.
3. Some trait variants will give organisms with those variants an advantage over others, giving those organisms a better chance at reproduction.
4. An organism's traits are passed along, often with some changes, to its offspring.

Darwin argued that these principles could, over many generations, result in new species forming.

When we interpret each of these components of natural selection from a computational point of view, the result is the approach known as a genetic algorithm.

The “individuals” under selection in a genetic algorithm can be all sorts of things—bit strings, numeric vectors, even entire neural networks. Each “individual” can also be an entire executable computer program. This variant is known as **genetic programming**.

Try It Yourself

Follow along with the video lesson via the Python code:

[L07.ipynb](#)

Auxiliary Code for Lesson:

[L07aux.ipynb](#)

Python Libraries Used:

gplearn.genetic.SymbolicRegressor: Genetic programming approach to learning a symbolic expression.

math: Mathematical functions.

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

random: Generates random numbers.

statistics: Algorithms for statistics like computing the median.

Key Terms

genetic programming: The application of genetic algorithms to optimize programs.

symbolic regression: Regression where the output rule is a mathematical expression.

READING

Domingos, *The Master Algorithm*, chap. 5.

Mitchell, M., *An Introduction to Genetic Algorithms*.

Mitchell, T., *Machine Learning*, chap. 9.

QUESTIONS

1. In the four-part recipe for natural selection—and genetic algorithms—the third step is as follows: Some trait variants will give organisms with those variants an advantage over others, giving those organisms a better chance at reproduction. What will happen if you leave out this step?
2. Think about a representational space consisting of n -bit strings. Such a representational space might be a Boolean feature vector like you saw in the *I-E* example in [Lesson 03](#) or an unweighted bag-of-words vector like you learned about in [Lesson 06](#). Pose a crossover operator that makes sense for n -bit strings.
3. The symbolic regressor did a great job with the target function that was used in the lesson: a cubic. In principle, a powerful aspect of genetic approaches is that they can be used when the functions they are optimizing are not particularly smooth. Consider the function $|x-4|+4$. It's a simple function, but it has a kink in it. If you fit the function with the symbolic regressor using just `add` and `mul`, what do you expect to happen? What if you add `abs` (absolute value) to the set of functions the algorithm can use in its symbolic expression? Modify the code to find out.

Answers on page 479

Genetic Algorithms for Evolved Rules

Lesson 7 Transcript

At the dawn of computing, in the late 1940s, Alan Turing was grappling with the question of how we might make machines smart. It was far from obvious how to write a program that would replicate the subtle decision-making and learning skills that humans possess.

But in a 1950 paper that introduces the imitation game, now known as the Turing test, he proposed that hand-programming adult-level intelligence was neither practical nor necessary. Instead, we just needed to create a “child program” that would learn from experience, much the way a human child learns. Basically, he was hinting at machine learning of the sort we’ve been studying in this course.

But where does the child program itself come from? Turing speculated that machine learning could be complemented by a procedure that searches for good child programs using random generate-and-test. He likened this kind of random generate-and-test mechanism to the process of evolution.

Why can’t we just turn to neural networks as a way to solve the problem? A typical neural network using gradient descent is demanding. It uses derivatives from calculus to compute the rate of change of the loss function as a function of the changes in the network’s parameters.

Great work, when you can get it. But many problems lack a clear path from the loss function back to the parameters being learned. There’s no concrete function for calculus to do its work on.

For example, if a problem has a real-world component, like predicting human reactions to a decision or optimizing the movement for a complex physical robot, we do not have an explicit mathematical function that defines the problem we are solving. Without such a function, we cannot take derivatives and we cannot do gradient descent. One can’t backprop through reality; we just don’t know the function it’s using.

Natural evolution faces a similar problem as it optimizes organisms for their ecological niches without detailed mathematical representations of how changes to the organism will impact their reproductive fitness. This insight suggests bringing the principles of evolution to bear in machine learning, in the form of what we now call genetic algorithms.

Genetic algorithms caught on through the work of John Holland and his students in the 1970s and '80s. Their work formalized evolutionary optimization as a computational problem-solving technique. By 2010, genetic algorithms had already produced a variety of results competitive with, and yet also different from, solutions created by humans, in fields ranging from analog circuits, to assembly code creation, to software repair.

One of the benefits of adopting an evolutionary approach to optimization is that it works well on essentially any problem. That also means we can try it on hard problems that we don't know how to address sufficiently to use other machine-learning approaches.

As we've seen, some representational spaces suggest clear approaches for doing optimization. When you have a continuous problem and you understand the shape of the problem very well, then gradient descent and neural networks are a great fit. When you have a discrete problem with a clear set of features to branch on, decision trees are a terrific way to go. When you have a clear generative process, Bayesian networks are there to help. But there are many times, especially when you are just starting out in a new problem area, when you don't yet know what will work well.

Researchers have said that genetic algorithms are the second-best solution to any problem. This phrase has two important messages in it. One is not very complementary toward genetic algorithms. After all, why would you ever want the second best when you can have the best? But there's also a strong vote of confidence in there. Whenever you don't know what will be the best way to solve a problem, the second best is a darn good place to start.

Suppose you're trying to devise a strategy for hitting a target, but you don't know in advance the air resistance or the other forces that are relevant to computing the ideal trajectory. Or suppose you're planning to manufacture a chemical product and you want a process that takes the shortest amount of time. Or what if you want a process that has the fewest waste products?

Or suppose you're developing a new video game. Human reactions and strategies are hard to predict, but you'd want to make sure that there aren't loopholes that undermine the way that the game rules are constructed. Each of these problems can be, and has been, solved with an evolutionary optimization approach to machine learning using genetic algorithms.

I learned about genetic algorithms in college at the very first research talk I ever attended in the '80s. I saw that John Holland was coming to campus, and a fellow student asked me if I was going to attend his talk. "Wait, I can do that?" I sat down in the audience, and I was very self-conscious about being in a lecture, sitting with the faculty and the graduate students.

I worried that someone would notice me and kick me out for pretending to be a researcher. But I was worried for no reason. And once Holland started talking about genetic algorithms, I was hooked. It was such a cool idea! Harnessing nature's best optimization strategy for making smarter programs? Sign me up!

Evolutionary approaches act as an optimizer. They find high-scoring rules using the principles of natural selection. If we intend to follow an evolutionary process, then it's worth being clear about the main components that Charles Darwin advanced in his theory of natural selection:

- Individual organisms vary in their traits.
- Organisms compete for resources, with some reproducing and some not.
- Some trait variants will give organisms with those variants an advantage over others, giving those organisms a better chance at reproduction.
- An organism's traits are passed along, often with some changes, to its offspring.

Darwin argued that these principles could, over many generations, result in new species forming.

When we interpret each of these components of natural selection from a computational point of view, the result is the approach known as a genetic algorithm. Let's talk through an evolutionary approach to an example with a physical interpretation.

We've got a launcher, and we're trying to figure out how to aim the launcher to hit a target below. The launch velocity is fixed. There's a wall left of the launcher where the ball bounces off. What angle should we set the launcher to to hit the target below?

To get a feel for the problem, here's a set of six different angles that all lead to the same distance from the target: 8. Two angles land 8 units to the right of the target—one by shooting slightly downwards at 331 degrees, and one by shooting slightly upwards at 49 degrees.

Four angles land 8 units to the left of the target. Two of the four get there without using the wall—one by shooting almost straight up at 86 degrees, and one by shooting almost straight down at 276 degrees. Finally, two other angles land 8 units to the left of the target by ricocheting off of the wall—one by shooting slightly upward at 148 degrees, and one by shooting slightly downwards at 188 degrees.

Looking at a plot of all the angles versus the distance the ball lands from the target, we can see that it's a pretty interesting shape. At a zero angle, the ball ends up 13 units from the target—pretty bad. Setting the angle to 50 degrees gets a lot closer. The distance the ball lands from the target is around 7 units.

We can hit the target—a distance of zero units—at two angles. The one that's close to 300 is shooting directly toward the target. The one that's close to 70 is shooting up into the air and to the right so that it lands on the target. There's also a local minimum at 170 or so. That's aiming a little bit upwards at the wall so the ball bounces off the wall towards the target. You can't hit the target that way, but it comes closer than any of the nearby options—6.4 units away.

If we had an algorithm that could see this whole graph, the problem becomes pretty easy. Just look to see where any global minima are on the plot, and set the angle accordingly. But there is a problem with this approach. To make this plot, I effectively launched 3,600 balls. I fired one at 0.0 degrees, one at 0.1 degrees, one at 0.2 degrees, all the way up to 359.9 degrees.

If we were actually collecting the data needed for that experiment, it would be pretty wasteful to try every setting like that. But notice that changing the angle a small amount typically changes the distance to the target by a small amount. That means that there is some hope that we could check a small set of angles and have a pretty good idea of what's happening at the angles we did not check.

On the other hand, the exact shape of the function is a little unpredictable. This mapping from rule to evaluation is sometimes called a fitness landscape because it looks like it's made out of mountains and valleys.

TRANSCRIPT

A complex function like this one requires a flexible and adaptive strategy to find the optimum angle. That's where natural selection comes in. From a Darwinian evolutionary perspective, we are no longer merely fine-tuning the weights on neurons inside an individual brain. Instead, we need a population of individuals to select from. In our simple physics-based example, we can take an individual to be a single angle—any value, including fractions. A population is a set of angles of various values.

If we go back to the four defining properties of natural selection, we can see that calling the angles “individuals” allows us to satisfy the first property. Individuals vary in their traits.

But what about “individuals compete for resources, with some reproducing and some not”? In biology, individuals compete because resources are finite. We can use the same idea if we limit the population of angles. The resource here becomes the limited number of spots on the population roster. Let’s say we’ll cap the population at 10. That means our environment can only include 10 individuals—10 angles—at a time.

What about “some trait variants will give individuals with those variants an advantage over others, giving them a better chance at reproduction”? Now is when the optimization perspective starts to kick in. We’re going to start with, let’s say, a random population of 10 individual angles. Then we’re going to let them reproduce. Which angles should have an advantage at reproduction time? Since we’re trying to find the angle that gets the ball closest to the target, distance from the target seems like the differentiator we care about.

We want reproduction to favor the angles that get the ball closest to the target. That means we take all 10 of those angles, use them to launch balls, and then see how close the balls get to the target. The close ones get to influence the next generation. The others? Well, they’re going to have to go.

How do we make that happen? We could allow reproduction only by individuals that are within a specified distance to the target. Or we could evaluate pairs of individuals in tournament-style death matches, eliminating losers to get down to a smaller size population.

But let’s adopt a simple scheme, where the top five angles get to reproduce and the bottom five don’t. Harsh, I know. Nature can be pretty cruel, even when everybody’s working their best angle.

We've taken our set of 10 individuals, scored them based on their outcomes, and selected the top half for reproduction. If this were a non-evolutionary algorithm, we could be done. Instead, now we're in position to interpret the last principle of natural selection: An individual's traits are passed along, often with some changes, to its offspring.

So we need to use the five selected individuals to repopulate the world. How do we do that? Well, we could clone the five winners. But let's go with something almost as simple that will give us more variation and a more powerful algorithm.

Instead of exact cloning, suppose each of the five reproducing individuals will make a noisy copy of themselves to replace the five individuals we got rid of. We have a population of angles, so a noisy copy would be an angle with a small random number added to it.

I'm going to pick a uniform random angle change for each individual reproducing in each generation. The angle change is somewhere between plus or minus 5 degrees. That way, the kids stay pretty close to the parents, but also have enough space around them to be a bit different. Kids will be kids.

Concretely, what might an evolutionary optimization of this function look like? We start off with 10 individuals representing 10 random angles. Each individual is represented by an orange dot. We measure the loss of each of these individuals in generation 0, where *loss* means the function we are trying to minimize. In this example, it is the distance the ball ends up from the target when fired at the given angle.

For this random population, the minimum-loss individual has a loss of 6.40. It's an individual that ricochets the ball off the wall. That's close to a local minimum—ultimately, a losing strategy in this game.

Next, we get rid of the individual with the five worst losses. We find those all have a loss above 12.5. So everyone worse than 12.5 is removed. The remaining individuals are then allowed to reproduce, making copies of themselves nearby in the space. None of the kids fare any better than the individuals from the first generation, though. The minimum loss is still 6.40 in generation 1.

LESSON 07 | GENETIC ALGORITHMS FOR EVOLVED RULES

TRANSCRIPT

A few generations later, we have two clear subpopulations. The dot on the left that was just below our original cutoff got outcompeted and has no surviving descendants. There are still some in the ricochet category, around 160 degrees. Then others are in the more promising 325-degree group. By generation 4, the 325-degree group is starting to get a little edge, and the minimum loss is 5.35.

A few generations later, and the 160-degree group has died out. Specifically, they all had loss worse than the median of the population. The individuals in the 325-degree group have outcompeted all of the individuals in the 160-degree group by achieving lower loss. The minimum loss is now at 3.53.

Over the next six or seven generations, progress is rapid as some of the new kids randomly spread toward slightly smaller angles with slightly smaller loss. By generation 13, the population is clustered around a global minimum near 300 degrees. The minimum loss has reached 0.04.

Additional progress is possible, but it's slow. In particular, there's really no direction to the search anymore. The difference between individuals in the population is now well below the minus-to-plus-five range of noise used in reproduction. So any further progress is undirected and happens only if a new kid happens to be born closer to the minimum.

There are a few interesting things to notice here. One is that kids tend to be near their parents and are preferentially reproduced as they go downhill. That is, we're seeing a kind of gradient descent, though it's highly inefficient due to all the randomness in the evolutionary procedure. With gradient descent, we use an explicit representation of a mathematical function to compute a change that is guaranteed to result in a reduction in loss.

Here, we're just generating random individuals and checking whether any happen to be resulting in lower loss. Downhill steps toward an optimum are a matter of luck instead of design. But since we lack an explicit representation of the function we're optimizing, it's not clear how we could do any better.

Because low-loss individuals reproduce and create other low-loss individuals, a subpopulation can squeeze out all other individuals in just a few generations of doubling. That's great, because it means good answers spread through the population quickly. But it sometimes results in a bandwagon effect, where the population exclusively focuses on a good but not great part of the fitness landscape.

The problem can be mitigated somewhat by increasing the population size to get greater coverage of potential winning solutions. It can also help to change the rules of reproduction to encourage more diversity. For example, we could have given a better chance of reproduction for individuals that are far from the rest of the population to increase diversity in the angles considered. But complex representational spaces are complex, so there's no one magic bullet. We again have to content ourselves with approximate optimization.

Another thing to notice is that our choice of uniform random noise for reproduction here leads to only gradual progress. No big leaps across the space are taken from parent to offspring. But some other genetic algorithms make big leaps deliberately by including a few completely new individuals in each generation, not just reproducing near-copies of existing individuals. That can help make bigger leaps and explore more of the fitness landscape, though at the cost of jumping some individuals away from the currently most promising parts of the space.

But maybe the most important thing to notice is that our simple evolutionary algorithm worked! Whereas the original scheme of trying every 1/10 of an angle required testing 3,600 individuals to get an answer, the genetic approach landed on a good solution in 13 generations, each of which required evaluating 5 new individuals. So the total number of individuals we needed to test was only 70. That's the initial 10 individuals, plus 5 from each of the succeeding 12 generations. That's a much smaller and smarter search. And this example is in a space with only one dimension.

Such benefits typically increase rapidly when we move to higher dimensional spaces, where individuals are something more complex than just a single angle. Genetic algorithms are fine handling whole bundles of traits. In fact, we can apply genetic algorithms to supervised learning, where the individuals can be some representation of a classifier, like any we've seen in this course.

The most important element a designer needs to consider when applying a genetic algorithm to a new representation is how variants will be created. We need to wonder what the kids will look like. It's important that any individual can be reached by some sequence of reproductions. If not, those individuals cannot be considered by the algorithm and will never be discovered, no matter how long it runs. Notice that every individual can be reached in our launcher problem.

TRANSCRIPT

Remember that each generation can produce individuals up to 5 degrees larger or smaller than the previous. After as few as 72 generations, it's possible for a 0-degree individual to have a great-great-great-great-etc. grandchild that is a 360-degree individual. Indeed, given that angles wrap around, any angle can be reached within 180 degrees, or 36 increments of 5 degrees. So it's possible for any individual to be reached in as few as 36 generations.

Offspring of individuals are likely to be close to their parents in a way that suggests that their loss will be similar. It's OK if low-score individuals can generate high-score kids. I'm sure they'd be very proud. But it's not great for a genetic algorithm if high-score individuals can only be reached from low-score individuals.

That would happen in our example problem if the minimum loss individual was surrounded—on 5 degrees on either side—by only individuals with much worse loss. If your fitness landscape is rife with these sorts of weird discontinuities, you'd be better off finding another scheme for reproduction that doesn't have this property.

But a genetic algorithm is fine as long as there is some sense that incremental progress is possible. In the physics-based example earlier, incremental progress is possible because the fitness landscape is relatively smooth.

The individuals under selection in a genetic algorithm can be all sorts of things: bit strings, numeric vectors, even entire neural networks. A scheme called neuroevolution of augmenting topologies, or NEAT, is a genetic approach to learning neural networks.

Another particularly interesting representation is when each individual is an entire executable computer program. One convenient representation of individual computer programs is something called a syntax tree, which shows the abstract structure of a program. They're analogous to the context-free grammatical structure we discussed in the naive Bayes lesson.

Genetic algorithms can be applied to computer programs in the form of a population of syntax trees. This variant of genetic algorithms goes by the name genetic programming. We can use genetic programming to optimize over computer programs if we want to produce individuals that can map input to output. That's more than the earlier example, where individuals are just a number—an angle.

Consider a machine-learning classifier. It maps instances to labels. We could represent a classifier as a simple program that tests attributes of an instance and then uses the results to decide what to output. So that's a candidate for optimization as a genetic program.

Here's an example where we try to discover the relation of an x variable to a y variable. A statistician would call this problem regression. You might be familiar with linear regression or polynomial regression. These problems are concerned with finding the coefficients of fixed equations that cause numerical inputs x to map to numerical outputs y .

Here, we're going to use genetic programming to solve the regression problem, returning a symbolic representation of the answer. That is, our program will be finding and returning an entire mathematical expression, not just the coefficients of a given expression.

Let's define a target function to learn, aptly named "fitme". Given an input x , our target function will be a simple cubic function: $1/10 x$ cubed plus x squared. We'll make a training set "X_train" for this function consisting of 50 uniform random Xs between -10 and 10. We run these points through our target "fitme" function to get the corresponding targets, "y_train". Using Matplotlib, we can plot these points to see the shape of the overall curve.

Next, we'll learn a classifier for this data using genetic programming. The "gplearn" package, first released in 2015, adopts the approach of the Scikit-learn library and extends it to genetic programming. We install it and import it into our online environment on Colab.

Our genetic programming-based estimator, "est_gp", is a symbolic regressor. That is to say, we're solving a regression problem by finding a symbolic expression—a little piece of a program.

The program it finds will use only two operators: "add" and "mul". "Add" for addition and "mul" for multiplication. A program can use as many of these operators as necessary. However, there's a tradeoff in the search between accurately matching the training data and being parsimonious—using a small expression. The "parsimony_coefficient" tells the genetic programming search how much weight to put on parsimony compared to accuracy. The parsimony coefficient can be any non-negative value, where smaller numbers for parsimony tend to result in much bigger programs created.

TRANSCRIPT

The data is ready for fitting. How does it proceed? The algorithm represents individuals as syntax trees. The internal nodes of such a tree are the various operators—just “mul” and “add” in this case. The leaves of the tree are either variables or constants, like our input X_0 and a number like 0.742.

For example, an expression tree with “add” at the top and 0.742 and X_0 as its children represents the mathematical expression X_0 plus 0.742.

We can embed this expression tree as a subtree of a larger tree. For example, the previous tree could become a subtree under a larger tree that has “mul” at its top node and X_0 as another child to the new top node.

The mathematical expression represented by this tree is X_0 times the sum of X_0 and 0.742. In other words, it’s 0.742 X_0 plus X_0 squared.

For a genetic algorithm, we need to define an operation to produce new individuals. And instead of noisy clones, the “gplearn” package uses an operation called crossover that takes elements of two members of the population to create a new individual. It’s kind of like each individual has two parents. We hope the baby gets the best of its parents. But the combinations are random, so we never know until we compute their loss at the next generation.

So let’s consider the crossover operation used in “gplearn”. Crossover between two individual trees randomly picks two nodes and their subtrees—one from each of the two reproducing trees. I’ve highlighted the selected node and subtree from the first tree in red. This particular subtree is just a single node. For the second tree, I’ve used green to highlight the selected node and subtree.

The crossover creates a new individual that is an exact copy of the first individual, except the red subtree is replaced by the green subtree from the second individual. That little bit of genetic blending produces a new expression. In this case, it’s 0.045 plus 0.742 plus X_0 , which simplifies to 0.787 plus X_0 .

“gplearn” also includes some simple random mutation operators, somewhat like the random noise we added in the physics-based example. Using crossover and mutation, “gplearn” applies the genetic algorithm idea to an initial population of random individuals to search for one individual program that fits the training data well, without being so big that it incurs a large parsimony penalty.

We use the “fit” method to ask the genetic programming algorithm to fit the data. To see what the learned function looks like, let’s take 250 random inputs as a test set, sort them, and then ask the learned model to predict the outputs.

Plotting the results, we see a very good fit to the training points. The expression it learned has eight operators in it. It can be simplified, though. We see it’s predicting $0.100125 X$ cubed plus X squared. That’s amazingly close to the real function, which was $0.1 x$ cubed plus x squared.

I’ve run the same data through several other regression algorithms, such as decision trees, and genetic programming was the best. Decision trees did well, but not quite as well. I was a bit surprised, because the function is pretty simple and smooth, which is usually a great case for other regression approaches.

It seems like the “gplearn” library is well-tuned for solving this kind of problem. And since the libraries continue to improve, don’t blindly assume that genetic programming is always second best. For fitting smooth functions like this cubic, this example of genetic programming already produces accurate and symbolically generalized expressions that are best in class, or nearly so.

Genetic algorithms can be used to evolve programs. And genetic algorithms may also be used to evolve an overall structure of other machine-learning algorithms, such as neural nets. Of course, knowledge about the problem being solved is great, when you have it! Then you can take a more directed approach to searching for good solutions.

But it’s hard to beat a genetic algorithm if you don’t know much about what problem you are solving. After all, figuring out the kind of problem is often where the process of understanding begins.

LESSON 08

NEAREST NEIGHBORS FOR USING SIMILARITY

Each of the approaches to machine learning you've encountered up to this point leverages some kind of similarity between the inputs. But the focus has been on how the algorithms process data; similarity has only been used implicitly by the algorithms. In this lesson, similarity will be measured explicitly. In particular, this lesson focuses on nearest neighbor algorithms that work by diffusing labels from training instances to other nearby instances. Being explicit about similarity allows us to use a very simple algorithm to solve tough machine learning problems.

K-Nearest Neighbors

Instead of learning a decision tree or other classifier from a set of data, we can use the nearness of training instances as a way to make predictions.

Using the rule known as the 1-nearest neighbor classifier, you look through the labeled training data and guess that the most similar training point is the best predictor for the new testing point. Instances with high similarity tend to belong to the same labeled class.

The 1-nearest neighbor rule can get confused if there's a single mislabeled point in a sea of correctly labeled points. Any points that are close to the erroneous point will inherit its mistaken label.

We can make the rule a bit more robust by finding the k closest training points for k larger than 1 and having them vote with their labels. This variant is called the **k -nearest neighbor** algorithm.

Nearest neighbors is sometimes called an instance-based* learning approach because it directly uses the training instances. It's also called lazy learning because it puts off the work of learning a global model until later. Instead of taking the training data and working hard to process it into an easily applied rule, it just holds onto all the training data and then—when the system knows what predictions are needed—finds the most similar examples and uses them to make the prediction. It's like a web search: Once the query is made, the search engine can find relevant examples to respond to the query.

Comparing K-Nearest Neighbors to Decision Trees

Decision Tree	K-Nearest Neighbor
Extract a global rule from the training data.	Don't extract a global rule from the training data.
Training data can be thrown away after the tree is learned.	Training data needs to be kept around forever.
Pre-processing somewhat expensive.	Pre-processing free.
Prediction is cheap---just follow a path from the root of the tree to a leaf.	Prediction requires looking at all the training data.
Rule complexity grows with the number of leaves.	Effective rule complexity declines with k .

* Another synonym for *nearest neighbors* is *memory-based learning*.

K-nearest neighbors is kind of the opposite of learning a decision tree. For a decision tree, we look at the data and try to find global trends and construct a tree that captures those overall trends. Once we have the tree, we don't even need the training data anymore; we can just apply the tree to any new example.

With *k*-nearest neighbors, no global information is gleaned. The algorithm focuses only on examples that are close to the new example we want a label for. Thus, we can't get rid of the training data.

There's no cost to learning in *k*-nearest neighbors. However, at prediction time, all of the training data needs to be examined. Prediction time grows linearly with the amount of training data. For a decision tree, prediction is cheaper since the time depends only on the depth of the decision tree that remains. The work of applying a decision tree rule does not grow with more training data.

Decision trees get more complex if the number of leaves is big compared to the amount of available training data.

Paradoxically, *k*-nearest neighbors has kind of the opposite trend. If we make *k* as big as the whole training set, then we have a rule that always predicts the majority class. Note that that's the same behavior we get with a decision tree with only 1 leaf. As we decrease *k*, nearest neighbors makes more and more fine-grained distinctions. Once it gets to *k* = 1, it gets perfect performance on the training set. And that's just like a decision tree with a distinct leaf for every instance in the training set.

If things go so well with *k*-nearest neighbors, should we just use them all the time over alternatives like decision trees?

There's a good argument to be made for *k*-nearest neighbors in low-dimensional problems. But in high-dimensional spaces, *k*-nearest neighbors says we should look for a neighbor among the labeled training data—but chances are that the closest neighbor is going to be far away. As a result, we'd need a lot of examples before finding information about a nearby neighbor.

K-nearest neighbors with *k* = 1 is the simplest and fastest training algorithm ever created. Testing with *k*-nearest neighbors is still costly, but it's very conceptually simple.

Try It Yourself

Follow along with the video lesson via the Python code :

[L08.ipynb](#)

Auxiliary Code for Lesson:

[L08aux.ipynb](#)

Python Libraries Used:

graphviz: Graph visualization.

math: Mathematical functions.

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

pandas: Library for organizing datasets.

random: Generates random numbers.

seaborn: Data visualization.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

sklearn.neighbors: K -nearest neighbor algorithms.

sklearn.tree: Scikit-learn's decision tree algorithms.

Key Terms

***k*-nearest neighbors:** Also known as nearest neighbors, instance-based learning, memory-based learning, or lazy learning. An approach to supervised machine learning that uses “nearness” as a stand-in for “likely to share labels.”

READING

Domingos, *The Master Algorithm*, chap. 7.

Mitchell, *Machine Learning*, chap. 8.

Russell and Norvig, *Artificial Intelligence*, sec. 19.7.

QUESTIONS

1. In a high-dimensional space, are there more nearby neighbors or more non-neighbors?
2. Do you think a nearest neighbor classifier would work well for the green-screen example from [Lesson 05](#)? Why or why not?
3. In four or more dimensional spaces, most things are far away, so k -nearest neighbors is likely to have a difficult time. But 1-nearest neighbor performs quite well in the 486-dimensional space created for detecting malware. How could that be? Write a program for checking the distances between testing instances and each one's nearest neighbor in the training set. Compare these distances to what results from a dataset with the same attribute values scrambled between instances—a dataset that represents a space with the same dimensionality and same distribution over attribute values but where associations between attributes are random. Plot histograms of the distances to see how nearest neighbor distances in the real data compare to the randomized data. What do you notice? What impact would you expect the differences to have on the accuracy of 1-nearest neighbor in this dataset?

Python Libraries:

numpy: Mathematical functions over general arrays.

seaborn: Data visualization.

Answers on page 479

Nearest Neighbors for Using Similarity

Lesson 8 Transcript

Let's try a little experiment. We have a labeled training set for a binary classification problem. We've got Carlo, Georges, and four others in Group A; and Bill and Phil and four others in Group B. Given these examples, where would you put Niels? Since you don't have any information about what the groups mean, you are stuck having to find relationships—similarities—between the names that are there.

Debbie and Michela are traditionally female names, and they are both in Group B. César and Carlo might be of Latin origin, and both are in Group A. Niels kind of sounds like Bill and Phil, so maybe it should go into Group B?

Actually, I don't want you to think too hard about it, because it's a trick question. Group A consists of the first names of 1984's Nobel Prize winners. Group B consists of the first names of 1984's Winter Olympics Alpine Skiing Gold Medalists. Niels is actually another Nobel prize winner—he worked with Georges and César—so he's in Group A.

How'd you do? The question was totally unfair, right? Without more information about how we can think of these instances as being related to each other, there's really no way to get a foothold on a problem like this. For induction to be possible, to generalize to new inputs, there needs to be some notion of similarity between the inputs.

Arguably, each of the approaches to machine learning we've seen so far leverages some kind of similarity between the inputs. For a decision tree to work, instances with similar feature values should tend to have the same labels. Neural networks process feature values mathematically, resulting in nearby instances getting mapped to the same output. A Bayesian probability model is expressed in terms of the relationships between the features. Even evolutionary search, where we don't know the type or structure of the problem when we start, assumes that instances with a family resemblance should get similar scores.

But although similarity is very important in each of the algorithms we have discussed up to this point, our focus has been on how the algorithms process the data. Similarity has only been used implicitly by the algorithms. In this lesson, we'll measure similarity explicitly. In particular, we'll look at nearest neighbor algorithms that work by diffusing labels from training instances to other nearby training instances.

LESSON 08 | NEAREST NEIGHBORS FOR USING SIMILARITY

TRANSCRIPT

Think of it like painting with a broad brush. Labels in the training data spill over to possible instances they are close to. It's guilt or innocence by association. Being explicit about similarity allows us to use a very simple algorithm to solve tough machine-learning problems.

One of the few running disagreements I've had with my spouse is about how to set the temperature in the house. My comfort zone is something like 68 to 72 degrees Fahrenheit. Within that range, I prefer that the house temperature be set as close as possible to the outdoor temperature—something like 68 in the winter and 72 in the summer.

But that doesn't work so well for my spouse. Here's the part that has always confused me: She wants it to be 72 degrees in the winter and 68 degrees in the summer. That is, the temperature that she prefers in the winter is too hot for her to be comfortable in the summer. Umm?

Well, it turns out that scientists in Denmark are acknowledged masters on the topic of indoor comfort. I got some data from them and cleaned it up a bit. It has temperatures and relative humidity, plus color coding for judgments about what's comfortable. There are 500 comfort judgments. Each point represents a measurement of the temperature and humidity of a room, along with a consensus judgment as to whether the room is comfortable.

An important thing to notice here is that the range of comfortable temperatures varies depending on the humidity. Here, at a high relative humidity, like 70%, comfortable temperatures range from about 68 degrees to 71 degrees. That's the sort of thing you might find in the summer.

At a low relative humidity, like 30%, 70 to 75 degrees is more comfortable for people. Mystery solved! My spouse isn't inconsistent and wasteful. She's just sensitive to humidity, like normal people!

OK, with this information in hand, let's think about what we'd need to do to automate climate control in the house. We'd need to be able to answer the question, "For a given temperature and humidity, is that comfortable?" Then, we can design a control system that moves the temperature and/or humidity to a comfortable value.

The first part—predicting if a given temperature/humidity combination is comfortable—is exactly the problem of learning a classifier. Given the 500 comfort judgments, we can train a decision-tree learner to predict comfort from temperature and humidity.

I trained a decision tree with seven leaf nodes on this data, and here's what the learned decision-tree rule does. I'm plotting blue where the learned rule says it's uncomfortable for people, and it really is. Orange shows where the learned rule says it's comfortable for people, and it really is. Thus, blue and orange are where the learned rule matches the data.

On the other hand, red is where the rule says it's uncomfortable, but it's actually comfortable for people. Green is where the rule says it's comfortable, but it's not for people. So if you focus on the green/orange region that's shaped kind of like Massachusetts, that's where the learned decision tree thinks the comfort region is.

For this kind of data, it seems like we could do much better. Consider what we might do to make a prediction for a temperature of 68 degrees and a humidity of 30%. That's one of the places where the decision tree is making a mistake. That combination is comfortable, but the decision tree says it's not.

If we look at the training data, there's a point at 69 degrees, 29% that is labeled "comfortable." So wouldn't you think that 68 degrees and 30% would be, too? Well, it is, according to the comfort control specialists. But the decision tree rule doesn't guess that.

Instead of learning a decision tree from the data, maybe we can use the nearness of training instances as a way to make predictions. Here's a really simple way to do that. If you want to make a prediction for a given temperature-humidity combination, look through the labeled training data to find something that's close to it. If there's something really close, we assume that the new point is going to share its label.

But even if there's nothing really close, and it's hard to know what's going on, we might as well use the label of whatever labeled data point comes closest. This rule is known as the 1-nearest neighbor classifier. Guess that the most similar training point is the best predictor for the new testing point.

If you are trying to decide whether spelling should be "I before E" for a given word, find the most similar word and assume this word follows the same pattern. If you want to know if a given temperature/humidity combination is comfortable, find the most similar known example and assume it's going to be the same. Instances with high similarity tend to belong to the same labeled class. You know, birds of a feather flock together.

LESSON 08 | NEAREST NEIGHBORS FOR USING SIMILARITY

TRANSCRIPT

The 1-nearest neighbor rule can get confused if there's a single mislabeled point in a sea of correctly labeled points. Any points that are close to the erroneous point will inherit its mistaken label. We can make the rule a bit more robust by finding the k closest training points for k larger than one and having them vote with their labels. This variant is called the k -nearest neighbor algorithm.

Let's apply 5-nearest neighbors to our training data. It looks a little messy, but the amount of mistakes, in green and red, is much smaller than what we saw in the decision-tree example. And with $k = 1$ nearest neighbors, it actually gets a little bit better, rising from 93.2% of the random combinations correct to 94.1%. That's because this example didn't have any erroneous labels, so the advantage from smoothing that $k = 5$ offers doesn't really help.

Nearest neighbors is sometimes called an instance-based learning approach because it directly uses the training instances. Another synonym is *memory-based*. It's also called lazy learning because it puts off the work of learning a global model until later. Instead of taking the training data and working hard to process it into an easily applied rule, it takes an "acquire now and pay later" approach.

Just hold on to all the training data. Then later, when the system knows what predictions are needed, it finds the most similar examples and brings them to bear to make the prediction. It's a little like web search—once the query is made, the search engine can find relevant examples to respond to the query.

Let's compare k -nearest neighbors to decision-tree learning. k -nearest neighbor is kind of the opposite of learning a decision tree. For a decision tree, we look at the data and try to find global trends and construct a tree that captures those overall trends. Once we have the tree, we don't need the training data anymore. We can just apply the tree to any new example. In contrast to the old saying about not seeing the forest for the trees, a single decision tree lets us scope out the whole forest.

k -nearest neighbors is focused only on the closest items in the forest. No global information is gleaned—the algorithm focuses only on examples that are close to the new example we want a label for. Thus, we can't get rid of the training data.

There's no cost at all to learning in k-nearest neighbor. However, at prediction time, all of the training data needs to be examined. Prediction time grows linearly with the amount of training data. For a decision tree, prediction is cheaper because we've incorporated the training data—the time depends only on the depth of the decision tree that remains. The work of applying a decision-tree rule does not grow with more training data.

Decision trees can get more complex if the number of leaves is big compared to the amount of available training data. Paradoxically, k-nearest neighbors has kind of the opposite trend. If we make k as big as the whole training set, then we have a rule that always predicts the majority class. Note that that's the same behavior we get with a decision tree with only one leaf. As we decrease k , nearest neighbors makes more and more fine-grained distinctions. Once it gets to $k = 1$, it gets perfect performance on the training set. And that's just like a decision tree with a distinct leaf for every training-set example.

In addition to the k in k-nearest neighbor, there's another parameter to define. It is the similarity computation itself. Generally, training instances are numeric vectors. What are different ways we can judge how similar two vectors are? Let's say d is the dimensionality of our vectors, and the two vectors are a and b . Maybe a is one of our training instances and b is the vector we want to make a prediction for.

Probably the most obvious and common method is to compute Euclidean distance. In math, it's the square root of the sum across all of the dimensions of the difference between the components of the vectors squared. We might also use the square of the distance, since the square root isn't needed here. After all, we're just going to be comparing relative distances, without needing to look at the exact distance values themselves. The square root does not change the relative ordering of distances—it's a monotonic function.

Or we could also substitute other values for the 2 and the square root. Using 2 and square root gives a particular distance measure called the L2 norm— L is for mathematician Henri Lebesgue. Different powers other than the 2 in the distance computation give other norms. The choice of power has an impact on how close two vectors are judged to be, depending on the closeness of individual components. As the power increases—L3 or 4 or 10—the farthest components dominate more and more in determining the nearness of two vectors. Conversely, as the norm decreases—L1.5, L1, L0.5—it's the nearest component that matters the most.

TRANSCRIPT

The L1 metric has another name because it's used very often. It's called Manhattan distance. It sums up the absolute value of the differences in each component. It's pretty similar to the Euclidean norm except there's an absolute value in there and no square root.

It's called Manhattan norm because it captures distances in a city street grid in which you have to walk along the streets and avenues and cannot cut through the buildings. Summing the differences of the components is like seeing how many streets apart the two locations are, and then seeing how many avenues they are apart, and then summing those differences.

Another common vector operation we can use for measuring distances is the dot product. It's the sum of the product of the components. You can implement it easily in Python. But you can also use a built-in version once you import the numpy package. The built-in version is likely to be faster.

Euclidean and Manhattan metrics are distance metrics. In a distance metric, zero is the closest possible value, and it means the points are maximally similar. Dot product is a similarity metric. The bigger the value, the more similar the vectors are. A value of zero means the vectors are unrelated. Similarity metrics can also give negative results, indicating that the vectors are more different than they are the same—they're inversely related..

A variant of dot product is cosine similarity—it's the dot product of vectors normalized to have a Euclidean length of 1. Just divide each vector by the dot product with itself before taking the dot product of the two vectors.

Normalized vectors have a nice property that connects several of the metrics we've been talking about. If we have two vectors, A and B, and normalize them to length one, then both vectors sit on a circle of radius one centered at the origin. In higher dimensions, they sit on a radius one hypersphere.

The distance, d , between the points varies monotonically with the angle between their vectors, theta. Bigger distances result in bigger angles. The smallest possible distance and the smallest possible angle are both zero. The largest possible distance is 2, which corresponds to the largest possible angle of 180 degrees.

The cosine similarity of the vectors is precisely the trigonometric cosine function of the angle theta. It also varies monotonically with the distance, but in the opposite direction. The smallest distance corresponds to the largest cosine: 1. The biggest distance corresponds to the smallest cosine: -1. When the vectors A and B form a right angle, the cosine is zero.

So there are four main ways we judge the relationship between vectors: Euclidean or Manhattan distance, or dot product or cosine similarity. These are the go-to methods in k-nearest neighbors for ranking the vectors that are closest or most similar to some query vector.

Rescaling of dimensions can also be done before computing similarities or distances. That's probably a good idea for our temperature-humidity example. You might note the weird vertical smudges on the 1-nearest neighbor plot from earlier. They come about because the relative humidity scale goes from 20 to 90 (70 points), while the temperature scale goes from 66 to 76 (only 10 points).

So very small relative differences in temperature count a lot more than small relative differences in humidity. If we scale up the temperatures by a factor of seven when we do our distance calculation, we get a plot that looks a lot better. And performance goes up to 96.7%. It was 94.1% using unweighted distances.

Those are the main distance and similarity metrics used for k-nearest neighbors on numeric vector data. K-nearest neighbors can also be used on strings of characters. There, we might use string edit distance to relate sequences to other sequences. String edit distance asks: How many insertions or deletions does it take to change one string into the other?

k-nearest neighbors can also be used on discrete probability distributions or probability densities over continuous spaces. There, there are functions like Kullback-Leibler divergence and earth-mover's distance that are used. My point isn't to get into all the variations. I just want to call attention to the fact that nearest neighbors is a very versatile algorithm because it can be used with so many different notions of similarity.

If things are going so well with k-nearest neighbors, should we just use them all the time over alternatives like decision trees? I think there's a pretty good argument to be made for k-nearest neighbors in low-dimensional problems like the one we've been talking about. In two dimensions, nearness is a very good indicator of label similarity when we're dealing with relatively smooth functions.

Let's think through what happens to distances as we increase the dimensionality d . Let's start with $d = 1$, a one-dimensional space. Imagine we both live in the same apartment building and that the apartment building has just one hallway. For simplicity, let's imagine that there

TRANSCRIPT

are 10 apartments on the hallway. If I'm in Apartment 3 and you are in Apartment 4, we're near each other, and it is reasonable to expect that things that affect me will also affect you. Like, if there's some kind of weird noise in the hall and it bothers me, it's going to bother you, too. In machine-learning terms, we'd expect our labels to be similar. In this layout, we have two apartments that are my neighbors, which is 20% of the building.

Now, let's go up one dimension to $d = 2$. Our building has a bunch of hallways now. Again, let's say there are 10 of them. We can be near each other by being close on the same hallway, or we could have apartments that are the same number, but on adjacent hallways. I now have four neighbors, two on my hallway and two that are the same number as me but on an adjacent hallway. That's double the number of neighbors I had before! But our building has 100 total apartments, so my neighbors only represent 4% of the building. I have more neighbors, but also a lot more non-neighbors.

If we go up to a dimension like $d = 3$, that's like adding floors to our apartment building. Again, we'll make it a 10-story building. If you are in the same hallway and the same number on that hallway, but up one floor, then you can hear me when I bang on my ceiling. I share a wall or ceiling or floor with six neighbors now. Which seems like a lot, until you realize that our building has 1,000 units. My personal neighbors only make up 0.6% of the total apartments in the building.

Going up to $d = 4$ dimensions is like imagining we have a whole street of buildings, 10 buildings in a row. In five dimensions, our complex extends to parallel streets. In six dimensions, we need a stack streets, one on top of the other. Maybe that makes sense in a space station. In six dimensions, I've got 12 neighbors, but I have 1 million minus 12 non-neighbors. There goes the neighborhood! It has almost entirely vanished.

High-dimensional spaces can be hard to understand, but notice the important idea here. In high dimensional spaces, almost everything is far away. k-nearest neighbors says we should look for a neighbor amongst the labeled training data. But chances are that the closest neighbor is going to be in a different apartment, in a different hallway, on a different floor, in a different building, on a different street. It's getting increasingly hard to believe that my neighbor's concerns have any bearing on my own. Even my nearest neighbor's label tells me almost nothing about my own label. We'd need a lot of examples before finding information about someone living close by.

Now, let's apply the nearest neighbor idea to the problem of malware detection, which is of prime importance for computer security. Malware is a portmanteau word for *malicious software*. It's nasty code that, if it finds its way into your computer system, it could end up giving you a system that's compromised or just plain broken. A malware-detection program looks at newly downloaded code and predicts whether it is malware or safe software. If it believes a piece of software is malware, it alerts the user and avoids installing it.

The machine-learning competition site, Kaggle, has a data set for learning to recognize malware. The data set consists of Windows PE programs. *PE* means “portable executable”—that is, actual code that can be run on a Windows computer. About half are infected with malware and are potentially dangerous. The other half are normal, safe programs.

This problem sounds similar to the spam classification problem, where we used a Bayesian classifier based on probabilities. Before we can feed information about whether programs are safe or infected to a nearest neighbor classifier, we need the programs to be represented as fixed-length numeric vectors. We could do that by selecting a fixed length equal to the size of the biggest program we intend to process, and then designate one feature for each byte of the software. But even very similar programs—for example, where one program is part of a larger program—would look quite different when viewed this way.

Instead, we're going to convert the program into a higher-level vector representation where similar programs get similar vectors. The process of vectorizing objects for the benefit of a machine-learning algorithm is called feature engineering.

In my experience, getting the right features is more important than getting the right learning algorithm, and that is especially true for similarity algorithms such as nearest neighbors. Meanwhile, a lot of the research has been focused on how to improve learning algorithms, and very little on helping people improve their choice of features. It strikes me as a missed opportunity.

Let's read the relevant files, `malware-train.csv` and `malware-test.csv`, into the Colab space. We'll be doing our distance calculations for nearest neighbors in 486-dimensional space, which is kind of trippy. Nearest neighbors, like any machine-learning approach, can get hopelessly lost with just a few hundred feature dimensions. Let's see what happens!

LESSON 08 | NEAREST NEIGHBORS FOR USING SIMILARITY

TRANSCRIPT

The features that make up the dimensions for the malware examples include 11 Boolean features that encode the answers to questions like “Is the code digitally signed?” Then, there are 64 features that, together, capture the raw code of the program’s entry point function. It is common for malware to share entry-point code, so including information about it in the feature representation gives the learning algorithm a chance to match code across examples.

There are 256 features that represent a histogram of the byte codes that make up the program—how many of each byte codes the program has. There are 150 features that correspond to whether the program makes use of any of 150 preselected external libraries, which may correlate with safe versus dangerous code. Along with a few other features computable from the code, there are a total of 486 features.

The data set we’re using has about 200,000 training examples split evenly between positive and negative examples. I randomly generated a subset of size 6,000 and split them into files with 4,000 for training and 2,000 for testing.

For starters, I wrote a little file-reading program that goes through the lines of the given file one by one, splitting each one into components at the commas. The data file uses “pe-malicious” to label the positive instances of malware. So, we use “equals-equals pe-malicious” to turn those labels into False—0—for safe and True—1—for malware. The function returns the data and the labels for the file.

Since the data is read from the file as a string, each component needs to be converted to a floating point number. We run the function `getdat` on the training data file, `malware-train.csv`, and on the testing data file, `malware-test.csv`. For nearest neighbors calculations, we’ll use Scikit-learn, which has a library we’ll import called “neighbors”.

The `testscore` function is given a data set in the form of the instances and labels, and it tests out the most recently trained model using this data. It returns the number of instances in the data set where the model predicts the label correctly. We set $m = 4,000$ for a training set size of 4,000.

We’re going to plot the training and testing accuracy for four different values of k for k-nearest neighbors—specifically, 1, 5, 7, and 9. That will give us a sense of how performance varies with this parameter that controls the neighborhood size.

For each k , we set up the classifier by calling “KNeighborsClassifier”. I’d say we train it, but nearest neighbors doesn’t really do any training. However, this command is where we tell nearest neighbors what similarity measure to use. I selected cosine because it is insensitive to vector length, which could be important in this problem. Other options include the L_p metrics mentioned earlier, specifically Euclidean, which is L_2 ; Manhattan, which is L_1 ; Chebyshev, which is L_∞ ; and Minkowski, which lets you pick your own exponent. The default is Euclidean, because it tends to capture a generic sense of what it means for vectors to be near each other.

We call `testscore` to see how well the model predicts the training data and the testing data, gathering up everything into `data` to be plotted. We plot the accuracy on the training data in blue and the testing data in red as a function of the number of neighbors used. When we run it, it gives us a high-level summary of this experiment. What do we see?

For one thing, the 1-nearest neighbor predictor on the training data is perfect. It gets an accuracy of 1, meaning 100%. Why is that? Well, we’re asking an instance to predict its own label. After all, for any instance in the training data, the closest instance in the training to it is, well, itself. That fits quite well, but it isn’t really all that useful in practice since we are rarely asked to just repeat back a label for something we’ve already seen.

An important property of the graph is that accuracy degrades with increasing neighbor size, even for the training data. That suggests that labels change somewhat rapidly with increasing distance. For this data, it’s a bad idea to construct too big of a consensus view. You just end up asking Hawaiians about the weather in Kansas, or vice versa.

Of course, the predictor does better on the training data than the testing data. After all, training data is where the label predictions are coming from. But performance on the test data is only a bit worse, just under 90%. That seems pretty solid.

At the end of the day, our high-dimensional-similarity-based classifier is able to identify 90% of code snippets correctly as malware versus not. Still, that means 10% of bad programs might slip through, and we might occasionally think an innocent program is dangerous. But we’re identifying most of the programs well.

LESSON 08 | NEAREST NEIGHBORS FOR USING SIMILARITY

TRANSCRIPT

The big picture here is that we can do quite well at doing a complex, intelligent-ish classification task simply by noting when new instances look like instances that have previously been tagged as safe or dangerous. Malware is often based on earlier malware, so it seems plausible that a classifier based on similarity might do a good job.

Nearest neighbor algorithms give us a pretty simple and yet quite successful way to make tricky decisions. It's pretty simple in that we just need to pick a neighborhood size, and, of course, we need to say what we mean for two things to be similar. Defining similarity seems like an extra step, one that's not needed in other machine-learning approaches, but it's also a nice opportunity to introduce valuable information into the classification process. It forces us, as people employing machine-learning technology, to reflect on what the useful signal is in the data that we're analyzing.

Part of the reason leveraging more neighbors did not allow us to do better is that we're making our comparisons in a very high-dimensional space. More dimensions means a lot more space. And when we look at a distance that's big enough to include nine neighbors, we're actually blurring together an awful lot of space that can include very different examples.

In my experience, keeping the number of nearest neighbors small has worked better. In any case, k -nearest neighbors with $k = 1$ is the simplest and fastest training algorithm ever created. Testing with k -nearest neighbors is still costly, but it's very conceptually simple.

So what's key is to use the right kind of similarity and the right size of neighborhood that matches well with the way the data is labeled. After all, what we're really doing is using instance similarity as a stand in for label similarity.

LESSON 9

THE FUNDAMENTAL PITFALL OF OVERFITTING

Finding real signals among spurious associations is at the heart of a machine's ability to learn rules that generalize to new situations from limited data. There is one pitfall that encapsulates the problem of spurious associations across all approaches to machine learning: overfitting—the idea that a rule can look good on the data you use to pick it but not be good in the broader set of situations in which the rule will be applied.

The Ideal Learning Law

There is a three-way relationship between the main quantities in machine learning:

- ◆ the dimensionality, r , of the representational space that the learning algorithm considers;
- ◆ the size of the dataset, s , used in the training to select a rule; and
- ◆ how good a rule it will find, as measured by its error, e , on new data.

The precise relationship can be expressed in equation form as the ideal learning law:

$$se^2 = Kr,$$

where K is a constant.

According to this equation, if you hold the amount of data used for training constant, then increasing the size of the representational space increases the error to keep the equation balanced—they are on opposite sides of the equation.

Through the lens of the ideal learning law, overfitting is the problem of having a big representational space compared to the size of the training set, resulting in high error.

We can think of training a machine learning algorithm as finding the rule in our representational space that does the best on our training data. This procedure will likely produce a good rule—one with low error—as long as there's enough training data.

But in practice, the amount of data is not under our control. Instead, we have a given set of training data, and where we have choice is in picking the size of the representational space.

Cross-Validation and Regularization

A rule that is underfit can easily be improved by considering a bigger space of more rules. In the case of a rule that is overfit, the algorithm is attached to the particulars of the training data, and the rule is tailored to things that simply don't generalize.

A famous real-world example of overfitting was the Google Flu Trends project in 2013. Because of overfitting, its learned search query that seemed miraculously accurate at predicting flu in the years it was trained on produced nonsense predictions in practice.

What can we do in practice to try to hit the sweet spot that is neither underfit nor overfit? If we had access to our testing data as part of the training process, that would do the trick. We could just keep increasing our representational space until performance on the testing data degrades.

But using our testing data as part of the training process is cheating. After all, just bringing all of the testing data in and making it part of the training process will lead to improved training performance, but we won't be able to tell anything about generalization.

We can get almost the same effect by hiding data from ourselves. This idea is called **cross-validation**.

Under this plan, we now have a three-way split for our data:

- ◆ training data, or the data we use to construct our rule;
- ◆ testing data, or the data we use to evaluate our final rule; and
- ◆ validation data, or the data that helps us decide on a representational space size.

The validation data is sort of in between training and testing. It acts like testing data for use during the training process. You can think of it like training data because it's used to create the rule. But you can also think of it as testing data because it's only used to evaluate rules, not construct them.

Cross-validation is an essential part of machine learning because the models are very, very complex. If we don't reign them in—somehow—overfitting is bound to happen.

Another amazing trick that machine learning algorithms use to fight overfitting is **regularization**, which is about trying to make the rules more general and consistent.

The trick to doing regularization is to add a component to the loss function that penalizes model complexity. The new loss function makes it the optimizer's job to strike a good balance between accuracy and rule complexity. The loss function is telling the optimizer that the rule can be made more complicated, but only if doing so has a big benefit to accuracy.

For example, in decision trees, we can do regularization by penalizing trees that have a larger number of leaf nodes. That will encourage the trees to stay small.

Overall, cross-validation and regularization are the most important tools for keeping machine learning algorithms from misleading us. Regularization encourages models to stay simple. Cross-validation helps make sure models remain accurate. They give you a model with enough complexity for the amount of data you have but stop the learning algorithm from latching onto spurious correlations in the data that result from not having enough data. Together, they help the learning process strike a balance that's just right.

Try It Yourself

Follow along with the video lesson via the Python code:

[L09.ipynb](#)

Auxiliary Code for Lesson:

[L09aux.ipynb](#)

Python Libraries Used:

math: Mathematical functions.

matplotlib.pyplot: Plots graphs.

random: Generates random numbers.

sklearn.svm: Scikit-learn's support vector machine.

sklearn.tree: Scikit-learn's decision tree algorithms.

Key Terms

cross-validation: Technique to combat overfitting by setting aside some of the training data to help assess generalization and avoid using a representational space that is too big.

optimization problem: The computational challenge of finding objects that result in high score or low loss. A key step in producing rules via machine learning.

perceptron: A representational space for rules equivalent to a one-layer neural network with step-function activation. The earliest neural network that was studied.

regularization: Technique to fight overfitting by modifying the loss function to penalize both error and representational space size.

support vector machine: An approach to supervised learning that leverages similarity between instances to maximize the distance between the decision boundary and the nearest labeled example.

READING

Lazer and Kennedy, “What Can We Learn from the Epic Failure of Google Trends?”

Mitchell, *Machine Learning*, chap. 7.

Russell and Norvig, *Artificial Intelligence*, sec. 19.5.

QUESTIONS

1. What are the two main algorithmic tools for fighting overfitting?
2. According to the ideal learning law, what do you need to do to keep error at a fixed level if you suddenly discover that you have half as much data for training?
3. How do support vector machines compare to nearest neighbors in terms of generalization accuracy and training time? Run a support vector machine on the malware data from [Lesson 08](#) to see. What kernel (linear, polynomial, sigmoid, RBF) appears to be the best for this dataset?

Python Libraries:

`sklearn.neighbors`: *K*-nearest neighbor algorithms.

Answers on page 480

The Fundamental Pitfall of Overfitting

Lesson 9 Transcript

If there's one thing everyone should know about the art and practice of machine learning, it's in this lesson. How do we tell the difference between real signals and spurious associations? Finding real signals is at the heart of a machine's ability to learn rules that generalize to new situations from limited data.

And there is one pitfall that encapsulates the problem of spurious associations across all approaches to machine learning: overfitting. Overfitting is the idea that a rule can look good on the data that you use to pick it, but not be good in the broader set of situations in which the rule will be applied.

A famous real-world example of overfitting was the Google Flu Trends project in 2013. The project's goal was to identify flu outbreaks by spikes in the use of search terms. The system considered a vast number of possible search terms for their rule to try to find even subtle signals of the flu. At the same time, Google only had a small amount of flu data to inform their choice of rule. Because of overfitting, their learned search query that seemed miraculously accurate at predicting flu in the years it was trained on produced nonsense predictions in practice.

To understand what happened in this example, let's look at the three-way relationship between the main quantities in machine learning: the dimensionality r of the representational space that the learning algorithm considers, the size of the data set s used in the training to select a rule, and how good a rule it will find, as measured by its error e on new data. The precise relationship is that the size of training set times error on new data squared is equal to the dimensionality of the representational space times a constant.

We can read off several useful facts from the equation. For example, if you hold the representational space fixed, increasing the size of the training set drives down the square of the error. So the error drops with the square root of the size of the training set. If you hold the amount of data used for training constant, then increasing the size of the representational space increases the error—they are on opposite sides of the equation. In the flu example, Google used a very large representational space and a very small training set. The equation tells us that the error will increase—maybe a lot—to keep the equation balanced.

We'll call this equation the ideal learning law, because it reminds me of the ideal gas law from chemistry that relates pressure, temperature, and several other quantities. Through the lens of the ideal learning law, overfitting is the problem of having a big representational space compared to the size of the training set, resulting in high error.

The issue of overfitting is so fundamental that it's worth diving back into the diabetes example we looked at in the decision-tree discussion of lecture 3. At the time, I split the available data into training data and testing data, but I didn't really explain why.

Remember what we were doing: We were choosing a loss function—in that case, we were measuring classification error. And then we were asking our optimizer to minimize classification error on the training data. But what we really want is a rule that generalizes well—one that will give good answers on all sorts of new examples.

But how could we know how well the rule would perform on examples that we don't even have? And that is why we split data into training and testing sets. We can learn a rule given our training data. And then, we can statistically simulate evaluating its performance on all possible new examples by bringing in the examples that we previously held in reserve. That's our testing data.

We know our solution generalizes really well if the loss it gets on the training data and the loss it gets on the testing data approximately match each other. That means that there are two separate quantities we're worrying about now. We want low error on the training data. But we also want high generalization of this result to our testing data. With this idea in mind, let's see how training and testing error are impacted by increasing the number of nodes in a learned decision tree.

I ran the program for building a decision tree from the diabetes training data multiple times. Each time, I varied the maximum number of leaf nodes, ranging from 2 to 30. For each of these trees, I measured the error on the training data and plotted the prediction error measured against the maximum number of leaf nodes.

The plot shows the more leaves, the lower the error. If we let the tree continue to grow up to 135 leaf nodes, this blue line would go all the way to 0% error. Good news, right?

LESSON 9 | THE FUNDAMENTAL PITFALL OF OVERFITTING

TRANSCRIPT

But is it? The big decision trees you'd get would have all sorts of questionable splits—they seem to be there just to let the decision tree make correct decisions for specific instances in the training set. In a manner of speaking, they memorize the data. And just like a student who memorizes the textbook without really understanding it, you have to wonder whether its answers on new questions would be any good.

And you can do more than just wonder. Our test data is new—it was not used to build the trees—so checking the rule on our test data is a way to address exactly the concern about how the rule will do with new data. Our graph will get a new orange line that reports the error on the testing data on the trees we learned with the training data.

For small trees up to five nodes, the two curves follow each other reasonably well, and that's great. Error on the training set and error on the testing set are decreasing, in tandem. Error on the training data is a little lower, which is pretty common—after all, the training data is what was used to pick the trees.

And because the values are similar, we can say that small decision trees that were learned are generalizing well. The error we see on the training data carries over well to the testing data. For trees with five nodes, prediction error on the test data is about 24%, while prediction error on the training data is about 22%.

But then something changes. Adding more leaf nodes causes error on the training set to decrease. But the result for test data no longer moves in tandem; in fact, prediction error on the testing set increases for a while. And even as we look at bigger and bigger decision trees, the error on the testing set never gets below 25% again.

The pattern in this kind of plot is universal to all machine-learning algorithms. It's not specific to this data set. It's not specific to this algorithm. Looking only at performance on the training data will always give an illusion of perfect mastery. We need results from the testing data to draw a line between what's predictive and what's illusory. The test data are what allow us to find the difference between results that fit the data versus results that overfit the data.

Returning to the ideal learning law, we see that holding data constant and increasing the size of the representational space increases the error. In this example, representational space size is measured by the number of leaves in the decision trees being produced. The number of possible trees

the algorithm can output grows rapidly with the number of leaves the tree can have. And other machine-learning approaches have their own measures of representational space size.

Let's look at slicing things up another way. We'll hold the representational space fixed as the set of decision trees with five leaves. We'll vary the amount of training data available, from 50 instances to over 500. Looking at the ideal learning law, testing set error should decrease with increasing training set size.

Again, let's look at the error on the training data first. Why does the error increase with increasing data set size? Because when there's a small amount of data compared to the number of possible trees, there's a good chance of finding a tree with low error.

But as the amount of training data increases, the best tree has to satisfy lots of different examples—something that's much harder to do. It's kind of like the old adage: A person with one watch knows exactly what time it is. A person with 500 watches knows it's hard to know exactly what time it is.

From a training perspective, more data seems bad—it's making the error go up. But once again, the training data alone cannot tell you what's actually going on—not really, not in a way that tells us about new data. We need to look at error on the test set to get a more complete story.

The error on the testing data starts off very different and has a much more encouraging trend. The trees that were learned from more training data indeed have less error than the trees that were learned from a small amount of data. In addition, these lines are getting closer and closer together. That's because the bigger the data set for training, the more we can be very confident that the learned rule generalizes—it exhibits similar behavior on the training data as on new examples it has never seen before. As you can see, 500 training instances is not quite enough data to reach the point at which training and testing error match. But it's getting close.

We can think of training a machine-learning algorithm as finding the rule in our representational space that does the best on our training data. This procedure will likely produce a good rule, one with low error, as long as there's enough training data. But in practice, the amount of data is not under our control. Instead, we have a given set of training data, and where we have choice is in picking the size of the representational space.

LESSON 9 | THE FUNDAMENTAL PITFALL OF OVERFITTING

TRANSCRIPT

Let's go back to our graph of the number of leaves on the decision trees versus accuracy for the training and testing data. A rule selected from the early part of the graph is called underfit—it can easily be improved by considering a bigger space of more rules. A rule from the later part of the graph is called overfit—the algorithm is attached to the particulars of the training data, and it tailored itself to things that simply don't generalize.

What can we do in practice to try to hit the sweet spot that is neither underfit nor overfit? If we had access to our testing data as part of the training process, that would do the trick. We can move to the right on the graph until the two lines start to diverge. But using our testing data as part of the training process is cheating. After all, just bringing all of the testing data in and making it part of the training process will lead to improved training performance, but we won't be able to tell anything about generalization.

We can get almost the same effect by hiding data from ourselves. This idea is called cross-validation. Under this plan, we now have a three-way split for our data: training data, the data we use to construct our rule; and testing data, the data we use to evaluate our final rule. But we can set aside another batch of data: validation data, the data that helps us decide on a representational space size.

The validation data is sort of in between training and testing. It acts like testing data for use during the training process. You can think of it like training data because it's used to create the rule. But you can also think of it as testing data because it's only used to evaluate rules, not construct them.

The idea of cross validation comes from the field of statistics, but it's an essential part of machine learning because the models are very, very complex. If we don't reign them in somehow, overfitting is bound to happen.

There's another amazing machine-learning trick algorithms use to fight overfitting. It's called regularization. You can get a sense of what regularization is about by thinking about regular and irregular verbs in English. Transforming an irregular verb to the past tense requires all kinds of special cases, whereas the regular verbs follow a general rule. Regularization would be making more verbs become regular—for example, using the *-ed* ending for the past tense for more of the verbs. This process happens naturally in language, and some irregular verbs already have a regular form, like *sneaked* instead of *snuck*, or *dived* instead of *dove*.

Anyway, regularization in machine learning is about trying to make the rules more consistent and general. The trick to doing regularization is to add a component to the loss function that penalizes model complexity. The new loss function makes it the optimizer's job to strike a good balance. The loss function is telling the optimizer that the rule can be made more complicated, but only if doing so has a big benefit to accuracy.

In decision trees, we can do regularization by penalizing trees that have a larger number of leaf nodes. That will encourage the trees to stay small.

In neural networks with perhaps millions of weights, a common trick is to add a penalty to weight values. The most common penalty is one that is proportional to the sum of the squares of all the weights in the network. Any penalty that grows with the weight magnitude encourages the network to set weights to zero wherever possible. And setting weights to zero decreases the effective complexity of the network.

In naive Bayes, initializing the counts of how often words have been seen to a large number has a regularizing effect by forcing the algorithm to only make strong predictions when the available data overwhelms the initialization.

In nearest neighbors, increasing the number of neighbors included in the vote acts as a regularizer.

Overall, cross-validation and regularization are the most important tools for keeping machine-learning algorithms from misleading us. Regularization encourages models to stay simple. Cross-validation helps make sure models remain accurate. They give you a model with enough complexity for the amount of data you have, but stop the learning algorithm from latching onto spurious correlations in the data that result from not having enough data.

Together, they help the learning process strike a balance that's just right. However, there can be other problems, caused by the poor quality of your data. That is, another way things can go wrong is if there are spurious correlations in the training distribution itself.

In practical applications, it can cause trouble when the process used to collect the training data is inconsistent with how the rule will actually be deployed. For example, I bought a learning thermostat one summer, and it learned the following rule: Michael never turns on the heat. That rule was perfect all summer. That's when the rule was trained and then locked in. But when fall arrived and temperatures dropped, the thermostat froze my whole family. It hadn't learned the right rule. But it wasn't merely because it didn't have enough data. It was because the distribution of the data changed.

LESSON 9 | THE FUNDAMENTAL PITFALL OF OVERFITTING

TRANSCRIPT

It's important to keep in mind that standard machine-learning methods learn correlations. My thermostat learned correlations that held all summer. That rule scored well on summer data. But just because a learned rule scores well doesn't mean it's actually "good" for the data the rule will encounter. To keep error low, machine-learning algorithms have to balance the size of the representational space and the amount of data.

One algorithm that provides very fine-grained control over this tradeoff is the support vector machine. Support vector machines were first created in 1963 by some of the pioneers of statistical learning theory, including Vladimir Vapnik, then working in the Soviet Union. But it wasn't until 1995 that Corinna Cortes and Vapnik, by that time at Bell Labs, published a practical approach for learning with support vector machines. By the early 2000s, and before that the late '90s, support vector machines had become the dominant approach to machine learning. They remain a very popular approach, especially for problems that have less data than can be used to effectively train a deep network.

Support vector machines can be very easily applied to new problems, and they can work quite well without a tremendous amount of fiddling. To understand support vector machines, we need to talk a little bit about perceptrons. Perceptrons are perhaps the oldest approach to machine learning, dating back to 1958. They were introduced by a psychologist named Frank Rosenblatt, who created a physical instantiation of the perceptron as a learning machine. While specialized perceptron machines are no longer built, the training algorithm he devised remains influential in the field of machine learning.

Given a binary classification problem, the perceptron tries to find a rule that perfectly classifies the training data. It can be seen as a type of neural network with no hidden units. The perceptron training procedure searches for settings for a collection of weights that constitute a linear classifier.

If we visualize the training data as a set of points in space, a linear classifier slices a line through the space so that the positive examples are on one side and negative examples are on the other. If the input dimensionality is greater than two, it's not really a line. It's a hyperplane. But that's not important right now.

Here's a set of training examples. You might think of each point as representing a person's height and jumping ability as the two dimensions. The labels on the points are green for people who are successful volleyball players and red for people who aren't. This particular data set is made up, so don't try to take anything specific away for forming your next team.

Anyway, a machine-learning algorithm is one that takes data like this and learns a rule that generalizes to other points it may encounter in the future. A 1-nearest neighbor classifier might end up learning to break up the space like this. I've colored the space red where a point would be classified as negative and green where a point would be classified as positive. The green region consists of any position that's closer to a green training point than to a red one.

Here's what a decision tree with three leaf nodes might do. Each decision-tree split introduces a horizontal or vertical line, so the space is broken up into nested boxes. A linear classifier, like what a perceptron learns, is a single line in this space at an arbitrary angle. Here's one of many possible linear classifiers for this data set.

What's important is that the perceptron will find a way to separate the positives and the negatives if it can be done—if the data is linearly separable. That it will find such a split, if it exists, is a guarantee. Getting a guarantee like that is an exciting and rare opportunity in machine learning.

But there are two big problems with the perceptron approach. First, what does it do if the data is not linearly separable? I'm sure if we plotted real volleyball success data, we won't find any linear separator that perfectly divides the good players from the bad ones—some shorter players will be very successful. Second, even if there is at least one line that divides positives from negatives, are all such lines equally good?

In our artificial volleyball data, I've now drawn a perfectly valid linear classifier that separates positives and negatives. But from a nearest-neighbor perspective, there are points that are really close to a negative example that would be classed as positive and points really close to a positive that would be classed negative by this linear classifier. That seems bad. Vapnik showed that it is indeed bad. He proved that, to generalize most effectively, we should choose a line that is as far as possible from the training data.

To do that, let's think about inflating each of those points and drawing our linear separator so that it misses all of the puffy points. In fact, let's keep inflating until we can't draw such a splitting line. What does that last linear classifier—or linear separator—look like?

LESSON 9 | THE FUNDAMENTAL PITFALL OF OVERFITTING

TRANSCRIPT

It should look something like this. If we make the points any bigger, we will no longer be able to thread a linear classifier through there. You can see that the line bumps up against two puffy red points and one puffy green point between them. If we inflate the points any more than that, the red points will necessarily push the line past the point of contact with the green point. So, we're done.

Let's think about the line we're left with. In a way, the line makes the least possible commitment—it continues to assign nearby neighbors the same labels as long as possible. This kind of linear classifier is known as the maximum margin classifier. The margin is sort of a demilitarized zone on either side of the separating line.

Equivalently, we can imagine that the line itself is being inflated instead of the points. We can keep jiggling the angle to make more and more room until, finally, it's wedged in there. The maximum margin classifier is the one with the best chance of generalizing to new examples.

Finding a large margin classifier that separates your data is more constraining than finding a small margin classifier that separates your data. That means we can use margin size like a measure of representational space size. If our optimizer simultaneously searches for a linear classifier with low error and large margin, it's carrying out a form of regularization and therefore making the best use of available data.

In the optimization problem, we want margin size to be maximized and the amount of misclassification error to be minimized. These two elements of the loss function are typically combined by summing them together with a tunable parameter. The parameter tells the support vector machine how much to care about misclassified points and how much to care about making a large margin.

In practice, it can take some monkeying around to get this parameter set correctly. If it's too large in one direction, the optimization solver will only care about making the margin big, even if it ends up misclassifying most of the data as a result. If it's too large in the other direction, it ends up classifying the data correctly, even if that means shrinking the margin down to zero.

All of this is very useful, but there's an attractive feature of support vector machines I haven't even touched on yet. The optimization problem solved to learn a support vector machine depends on the dimensionality of the data.

In the visual examples I was giving, that was two dimensions. But recall the malware example we solved using nearest neighbors in the previous lesson. That had hundreds of dimensions.

The amazing thing about support vector machines is that the only place data dimension matters is in the cost of taking a dot product between pairs of data points. That makes it possible to replace the dot product with nearly any kind of vector similarity we think makes sense for the problem we're addressing. Like the nearest neighbor algorithm, we have a free hand to choose a way to measure similarity that lets us bring in our background knowledge about the problem we are solving.

Similarity-based approaches like nearest neighbor and support vector machines let us bring in such knowledge explicitly. And that can help greatly reduce the amount of data the algorithm needs to learn an accurate classifier.

Python's Scikit-learn has routines for learning support vector machines. We can load the diabetes data. And we can create alldat and alllabs as the entire labeled data set, just as we did in the decision-tree lesson.

As described at the beginning of the lesson, we can separate our data into training and testing. I'll randomly assign a third of the data to be testing data and rest is the training data by picking a random number in the set 0, 1, or 2 for each training instance. The zeros will be our testing data.

In the jargon of support vector machines, the similarity function is known as a kernel. The package comes with several built-in kernels, plus options for users to write and use their own. There is a built-in linear kernel, which learns a linear classifier like the one we have been discussing. But there is also a polynomial kernel that allows for higher-order transformations of the features.

The RBF kernel does something more akin to nearest neighbors, using more local information-based parameter fitting. RBF stands for "radial basis function," which is a kind of local Gaussian "bump" around the data points. Finally, a sigmoid kernel is also available, which is a neural net-like similarity function.

Here, we are training our support vector machines with the polynomial kernel, varying the degrees of the polynomial using "degree=degree". That is, the degree parameter of the function is set to the value of the degree variable in our loop.

LESSON 9 | THE FUNDAMENTAL PITFALL OF OVERFITTING

TRANSCRIPT

The Scikit-learn function we call is “`svm.SVC`”. `SVC` stands for “support vector classification.” Other support vector algorithms can be used for the problem of regression, meaning using data to predict continuous quantities, not just discrete classes. We measure performance in terms of accuracy on the training and testing data by summing up the number of data points where the SVM’s prediction disagrees with the labels in our data set.

We can plot the results using Python’s Matplotlib library. What we observe is a plot very similar to what we saw earlier when we talked about overfitting in decision trees. We see training data error performance in blue falling slowly as we increase the degree of the polynomial used to compare vectors. We see testing data error in orange improving up to the second degree, but then rebounding toward larger error as the algorithm ends up overfitting slightly at higher-degree polynomials.

The rebound in this case is small because the support vector machine finds large margins that automatically regularize, even if we select a representational space that is too big. That returns us to the main theme of this lesson: overfitting. All machine-learning algorithms face a fundamental tension between the size of their representational space and the amount of data needed to train them. That is, on the one side is the size or complexity of the representational space used in learning. On the other side is the amount of data needed to identify a rule that generalizes well from the training data to new examples.

With a fixed representational space, generalization performance improves with increasing data. With fixed data, there is a kind of U-shaped curve that can be seen. Increasing the size of the representational space improves accuracy but degrades generalization. With fixed data, models with more complexity get better and better on the data we have, but worse on any other data.

The shape of the curve means there is a sweet spot where the representational space is a maximally beneficial size with respect to the available data. We use two approaches to fight overfitting and stay in the sweet spot. In cross-validation, we set aside some of our precious training data to help assess generalization so we can avoid using a representational space that is too big. In regularization, we modify the loss function to penalize both error and representational space size.

This lesson concludes the section of the course focused on the fundamentals of machine learning. Now you know the five basic approaches to machine learning, but also how the issue of overfitting spans all five.

I think overfitting is the most important issue in machine learning to understand before you set out to train machine-learning models yourself. Do not be seduced by excellent predictions on the data that you used for training! It's what the rule does on new data that really matters. A lack of understanding of overfitting can lead to creating and deploying rules that will misbehave in practice.

But even if you build a sound model that avoids overfitting, there are some more practical considerations we need to keep in mind when we apply machine learning to the real world. When we train a machine-learning algorithm to solve our problem, how do we know it's doing the right thing?

Next time, we'll see how to avoid some common pitfalls that plague machine-learning applications in the real world, starting with how algorithms can end up discriminating against minority populations.

LESSON 10

PITFALLS IN APPLYING MACHINE LEARNING

This lesson addresses some of the pitfalls you can encounter in the broader context of the real world when trying to bring machine learning algorithms to bear on real-life problems. You'll learn about a few ways that you can intervene to mitigate these pitfalls.

Overvaluing Trends in Data

Throughout the field of data science, some of the most surprising and insidious pitfalls reside in data collection. But with machine learning, it's also more than that: If there is a tendency in the data, your machine learner may not only notice the tendency but also accentuate it beyond what actually existed in the training data.

In 2019, the organization OpenAI released one of the most powerful language models ever trained. Formally, a language model is an allocation of probabilities to sequences of words. In practice, that means we can query a language model with a partial sentence and ask how likely various completions of that sentence are.

A language model can be used to help speech recognition or optical character recognition by bringing to bear top-down constraints on what things are more likely to be said than others. It can also be used for autocompletion or even text generation.

The Allen Institute for AI created a web interface for the OpenAI language model. It can be accessed on their website at

<https://demo.allennlp.org/next-token-lm?text=You%20are%20studying%20an%20introduction%20to%20machine>

When prompted with two different beginnings of a sentence—“When the child was finished playing with the truck” and “When the child was finished playing with the doll”—the system predicted that it was twice as likely for a male pronoun to follow *truck*: 15.25% versus 7.6%. It was also slightly more likely for a female pronoun to follow *doll*: 14% versus 13%.

In a sense, the machine learning system has internalized some gender stereotypes. It makes some sense, given that it is trained on real-world text, and the real world often has some pretty strong gender stereotypes.

There are well-documented cases of facial recognition systems working considerably less well for women of color, likely in part because they appeared too infrequently in the training data. Improving performance in cases like this can turn into a matter of social justice. Often, it is possible to remedy such issues by paying closer attention to the data used to train the system.

Training on data with unwanted stereotypes is always problematic. But it becomes more worrisome when we think about the influence of stereotypes on critical decisions like hiring.

The OpenAI system is much more likely to use a male pronoun for *doctor* and a female pronoun for *nurse*. When asked to complete the sentence “I will only hire a doctor who knows how to use _____”, it assigns a probability of 7.4% to *his* and less than 2% to *her*.

Substituting *nurse* for *doctor* in the sentence brings the probability of *her* up to 12.1% and *his* down to less than 2.1%.

Consider how we might use a language model as a way of assessing job applications. We might ask, “What probability would you assign to this application?” as a stand-in for whether it seems like the applicant would be a good fit for the opening.

To the extent that the model hasn’t been exposed to certain groups in certain jobs, it’s going to assign applicants from those groups a lower probability. They will look like bad fits for the job simply because the system hasn’t been exposed to such combinations.

In this case, it could be argued that we’re just using the wrong tool for the wrong job. Language models aren’t well formulated for assessing résumés.

But in October 2018, Reuters reported that Amazon had explicitly tried to make a résumé assessment tool based on machine learning. And try as they might, they couldn’t get it to give women a fair shake for technical jobs. Their trained system rated applicants as less appropriate if their résumé showed that they had attended specific all-women’s schools, for example.

The issue here is that the data that was collected wasn’t suited to the task it was to be used for. Given past data of résumés and hires, a machine learning system is trying to match the decisions made in the past. If past decisions have been biased away from women, then the system will learn to bias its decisions away from women as well. It’s what it’s trained to do.

The designers of the system had wanted to interpret the output as indicating who they *should* hire. But what it was actually trained to guess was who they *would* hire. In this instance, the team was unable to find a way to de-bias the output and gave up on the project.

Many similar failures have been reported, and certainly there are many we’ll never hear about. It’s a hard problem to jump from past data to advice about how to decide in the future.

Using Historical Data to Change Future Behavior

Another subtle pitfall that is particularly difficult to remove comes up when we are trying to use historical data to change how we will behave in the future. Historical data has baked into it the way we have behaved in the past, and that can send misleading signals.

Machine learning researcher Rich Caruana described a compelling case in which hospital information was used to predict patient outcomes. He wanted to use a machine learning approach to assess patients coming into a hospital to see which were at risk for pneumonia. If a patient is predicted to be at high risk for pneumonia, the doctors can allocate more resources to these patients to help make sure they survive their stay.

The team creating the system did everything by the book. They collected data on patients as they entered the hospital, recording age, medical history, vital signs, etc. Then, when the patient was discharged, the team recorded whether the patient had suffered from pneumonia.

The team used this data to train a highly accurate neural network classifier. And indeed, it did an excellent job with its predictions. But the doctors and machine learning researchers on the project balked at putting the system into practice.

It turned out that their concerns were well founded. They weren't sure what the neural network was doing, so they also trained a method called **logistic regression** on the same data. Logistic regression wasn't performing quite as well as the neural network, but the team could more easily tell what logistic regression was doing.

What the more transparent method of logistic regression was doing was odd. When assessing a new patient, it would rate someone with asthma as being at low risk for pneumonia. When the doctors saw that, they were very perplexed. They know that people suffering from asthma are very prone to pneumonia. Why would the machine learner believe the opposite?

Whenever we use machine learning algorithms to solve real-world problems, knowing how to cast a skeptical eye on the results we're getting can help prevent their misuse.

Again, it's precisely because doctors know that people suffering from asthma are very prone to pneumonia. The doctors treat patients with asthma very carefully and are tuned in to early warning signs that indicate that pneumonia might be taking hold.

So the machine learners were not finding that people with asthma are in general less likely to have trouble with pneumonia; they were finding that people with asthma are less likely to have trouble with pneumonia when treated by doctors who *know* they have asthma.

And of course, the learner was correct—in this narrower sense. But the output of the learner should not be used to change hospital policy. Taken at face value, the results suggest not treating asthma patients with particular care.

Using Proxy Measures

The biggest factor that lies behind failures of machine learning is the use of an inappropriate proxy measure.

A proxy refers to a substitute. We often train our models to predict one quantity when we really want to measure another. The proxy measure is the one we have. As long as our proxy measure is well aligned with what we really want, the results can have validity.

Consider zip code and income level. These two features are not the same. But zip code is a proxy for other information that correlates extremely well with income level: People tend to live near other people with similar incomes. So zip code can serve as an effective proxy measure of income—and race, for that matter.

A well-studied example of a bad proxy measure comes up in predictive policing. If we could train a classifier to tell us where crime is likely to occur, we could deploy security officers to those areas in a more targeted way, reducing the overall cost of protecting the community. But we don't have a way to detect when crime happens, so we can't train a classifier using this information.

But we could use arrests as a proxy for crime. That's something for which there is data—and surely there's some connection between arrests and crime, right? Yes, but the two are better aligned in some communities than others. Moreover, if the police force is already deployed in a low-income community, it is more likely to make arrests in that community independent of the actual distribution of crime.

If we train a model to predict where arrests occur, it will end up predicting that arrests are more likely where there are more police officers. If we then take arrests as a proxy for crime, these predictions suggest that we assign more police officers to wherever they were already patrolling.

Initial imbalance leads to even worse imbalances. It's a positive feedback cycle, but acceleration of this positive feedback cycle is not evidence that real progress is being made in preventing crime.

Try It Yourself

Follow along with the video lesson via the Python code:

[L10.ipynb](#)

Python Libraries Used:

csv: Parses data from comma-separated-value files.

random: Generates random numbers.

sklearn.linear_model.LogisticRegression: Scikit-learn's logistic regression algorithms.

sklearn.neural_network.MLPClassifier: Scikit-learn's neural network algorithm (multilayer perceptron).

Key Terms

logistic regression: An approach to classification that's equivalent to constructing a one-layer neural network with a sigmoid activation function. Similar in overall structure to a perceptron, naive Bayes, or a linear support vector machine.

word embedding: A mapping from words to vectors, usually selected so that words that appear in similar contexts are given similar vectors.

READING

Angwin, Larson, Mattu, and Kirchner, "Machine Bias."

O'Neil, *Weapons of Math Destruction*.

Russell and Norvig, *Artificial Intelligence*, sec. 19.9.

QUESTIONS

1. In real-world deployment of machine learning systems, there is often a feedback loop where new data is collected for training after an initial learning system has been fielded. What is a benefit of collecting new data? What is a risk of collecting new data? When might the benefit outweigh the risk?
2. A hypothetical machine learning system uses customer complaint data to assess communities where a company's product is likely to fail. The system predicts that a particular affluent community near the ocean has the most product failures, and the engineering team is considering redesigning the product to be more robust to salty air. Why might this course of action be inappropriate?
3. Naive Bayes, a linear classifier (a neural network with no hidden units and linear activation), and logistic regression are similar in that they all learn a one-weight-per-feature model. How do they compare on the MNIST digit data from [Lesson 04](#) on neural networks? If you run all three algorithms, which would you expect to do the best job of combining evidence from across the image to make a classification?

Python Libraries:

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

sklearn.datasets.fetch_openml: Provides access to OpenML datasets.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

sklearn.naive_bayes.MultinomialNB: Naive Bayes learner for binary features.

sklearn.tree: Scikit-learn's decision tree algorithms.

sklearn.utils.check_random_state: Repeatable, random way to split training and testing data.

Answers on page 480

Pitfalls in Applying Machine Learning

Lesson 10 Transcript

Throughout computer science, the maxim “garbage in, garbage out” reminds us that we need to pay attention to the quality of the inputs when assessing the quality of the outputs. For machine learning, the quality of predictions output by our classifiers depends on inputting good training data.

Our focus so far has been on specific algorithms that do a good job of turning data into a classifying rule. But any recipe for turning data into a rule is only part of any real-world machine-learning problem.

We could employ machine learning to help decide how to allocate security resources, or where to build a new factory, but we need to keep the broader context in mind. After all, data about past security incidents might not have been collected uniformly, impacting the algorithm’s perception of where security resources are needed. The way we define our loss function to trade off pollution concerns versus cost of transporting raw materials would affect where the learner recommends we put our factory.

In this lesson, we’re going to situate our use of a machine-learning algorithm into this broader context. We’re going to look at some of the pitfalls that you can encounter in that broader context when trying to bring machine-learning algorithms to bear on real-life problems. We’ll talk about a few ways that you can intervene to mitigate these pitfalls.

We’ll also see that some of the problems just don’t have good remedies. In these cases, being able to recognize the pitfalls will help you steer clear of situations where things will go wrong. In the worst cases, machine learning might do more harm than good, and you might want to just pull the plug. Not every problem with machine learning can be solved with more machine learning.

To be able to talk about what can go wrong in an application of machine learning, even if the machine-learning algorithm itself is fine, we need to widen our view a bit. We need to see the whole pipeline. So far, we’ve scoped the machine-learning problem as: Labeled data goes in, a machine-learning algorithm of your choice processes that data, and a rule pops out.

But the labeled instances come from someplace. There’s a data-collection process by which things that are in the world come to be captured in a particular data set used for training. And once the rule is learned, the goal in an application is to have that rule deployed in some context that impacts the world in some way.

LESSON 10 | PITFALLS IN APPLYING MACHINE LEARNING TRANSCRIPT

Finally, in many cases, we continue to collect data from the world. But now, it's a world that has been changed in some way by the deployment of our previous rule. Each of these additional arrows provides additional opportunities for things to go wrong.

Imagine you are a company that is building a machine learning-based system for responding to customer questions about transportation. For example, someone might ask, “What bus should I take to go from Warwick to Quincy?”

You might record the incoming calls and hire people to translate the audio into text. Once you've collected enough text data correctly matched with audio, you could train a machine-learning system to create a rule that recognizes names of cities based on the audio signal. The data is raw speech. So you might choose a neural network as your classifier, because they have proven particularly adept at interpreting low-level sensory information.

So you run your raw speech data through your neural network, and you train it up well. Then you'd install the recognizer in your phone-based system, and it will be automated. Success! After it's up and running, you continue to record calls so that you can identify problems and improve your system. That's very responsible of you!

What could go wrong? Well, it turns out that people in Quincy call their town “Kwinzy.” When they say Warwick, if they say it at all, they say “Waw-wick.” However, people in Warwick call their town “Wah-ik,” and they call the other town “Kwitsy.”

Now, Quincy has a bigger population than Warwick, and it's close to a major population center—Boston, Massachusetts. So there are likely to be a lot more examples of the Quincy accent than the Warwick accent in the initial data.

That's not bad in and of itself. But machine-learning systems, being imperfect, have to make tradeoffs. Your learning system will work hard to get the common cases right so as to reduce the overall error on the training set. That's what your loss told it you wanted. If you're lucky, the learner will get all of these cases right. But if it has to make a tradeoff, it will do so in favor of the Quincy people. That's true even if it means the Warwick people's speech is recognized less well.

If you deploy a system that is very reliable for people from Quincy but error-prone for people from Warwick, who do you suppose will use your system regularly? Probably the Quincy people. If the system is frustrating enough, the Warwick people will gravitate to alternative ways of getting information.

And that means when you re-collect data, it will be biased even more heavily in favor of the Quincy people. The next time you train your machine-learning system, it will work even harder to get the Quincy people right, and it will even try less strenuously to get the dwindling number of Warwick people right. Over time, your system will essentially be discriminating against the unfortunate folks from Warwick. They will be getting systematically poorer performance than the Quinzy-ites.

This particular example is somewhat fanciful. But this kind of discrimination is no joke. Recognition accuracy can end up being distributed unfairly across subgroups. For example, there are well-documented cases of facial recognition systems working considerably less well for women of color, likely in part because they appeared too little in the training data. Improving performance in these cases can turn into a matter of social justice.

Often, it is possible to remedy these issues by paying closer attention to the data used to train the system. In the Warwick-Quincy scenario, you might make sure to include a feature for where the caller is based. Quincy and Warwick have different area codes, so you could use the caller's area code to identify the two groups. Then, you could train separate classifiers for the two regions based on this specific input feature. That way, the representational resources for the two groups wouldn't be in direct conflict. It's no longer the case that the system is getting the Warwick people wrong because it is trying so hard to get the Quincy people right.

Throughout the field of data science, some of the most surprising and insidious pitfalls reside in the data-collection part of the pipeline. But with machine learning, it's also more than that. If there is a tendency in the data, your machine learner may not only notice the tendency; it may also accentuate the tendency, beyond what actually existed in the training data.

I'll use a powerful machine-learning system to demonstrate how easy it is for a machine learner to overvalue a trend it finds in the data. In 2019, an organization called Open AI released one of the most powerful language models ever trained. Formally, a language model is an allocation of probabilities to sequences of words. In practice, that means we can query a language model with a partial sentence and ask how likely various completions of that sentence are.

A language model can be used to help speech recognition or optical character recognition by bringing to bear top-down constraints on what things are more likely to be said than others. It can also be used for autocompletion or even text generation.

LESSON 10 | PITFALLS IN APPLYING MACHINE LEARNING TRANSCRIPT

The Allen Institute for AI created a web interface for the Open AI language model. I accessed it on their website, and I prompted it with two different sentences: “When the child was finished playing with the truck, ...” and “When the child was finished playing with the doll, ...”.

The system predicted that it was twice as likely for a male pronoun to follow *truck*—15.25% versus 7.6%. It was also slightly more likely for a female pronoun to follow *doll*—14% versus 13%. In a sense, the machine-learning system has internalized some gender stereotypes. It makes some sense, given that it was trained on real-world text, and the real world often has some pretty strong gender stereotypes.

Training on data with unwanted stereotypes is always potentially problematic. But it gets more worrisome when we think about the influence of stereotypes on critical decisions, such as hiring. The Open AI system is much more likely to use a male pronoun for *doctor* and a female pronoun for *nurse*. When I asked it to complete “I will only hire a doctor who knows how to use ...”, it assigns a probability of 7.4% to *his* and less than 2% to *her*. Substituting *nurse* for *doctor* in the sentence brings the probability of *her* up to 12.1%, and *his* drops below 2.1%.

Consider how we might use a language model as a way of assessing job applications. We might ask “What probability would you assign to this application?” as a stand-in for whether it seems like the applicant would be a good fit for the opening.

To the extent that the model hasn’t been exposed to certain groups in certain jobs, it’s going to assign applicants from those groups lower probability. They will look like bad fits for the job simply because the system hasn’t been exposed to those combinations. In this case, you could argue that we’re just using the wrong tool for the wrong job. Language models aren’t well formulated for assessing resumes.

But in October 2018, Reuters reported that Amazon had explicitly tried to make a resume-assessment tool based on machine learning. And try as they might, they couldn’t get it to give women a fair shake for technical jobs. Their trained system rated applicants as less appropriate if their resumes showed that they had attended specific all-women’s schools, for example.

The issue here is that the data that was collected wasn’t suited to the task it was to be used for. Given past data of resumes and hires, a machine-learning system is trying to match the decisions made in the past. And if past decisions have been biased away from women, then the system will learn to bias its decision away from women as well. It’s what it was trained to do.

Reuters reported that the software didn't "like" women. But that's unfair. The designers had wanted to interpret the output of the system as indicating whether they *should* hire. But what it was actually trained to guess was who they *would* hire. In this instance, the team was unable to find a way to de-bias the output and gave up the project.

Many similar failures have been reported, and I'm sure there are many we'll never hear about. It's a hard problem to jump from past data to advice about how to decide in the future. In a related example, a team at Microsoft Research looked at word embeddings. These are systems that represent the meanings of words by locating them in high-dimensional vector spaces. Each word is assigned a point in space so that words that are used similarly are assigned similar locations in the space.

Typically, these systems use hundreds of dimensions, which is very difficult to visualize. But conceptually, we have something like this two-dimensional rendition. People have observed an amazing fact: Systems trained this way tended to learn certain kinds of concepts as directions in the high-dimensional space. For example, the direction from the position of *actor* to the position of *actress* is similar to the one from *him* to *her* or from *brother* to *sister*. You can think of this direction as being the male-to-female direction.

It's pretty cool that the system discovers the idea of gender, especially since maleness and femaleness are not explicit features in the training data. The learning system is just trying to predict patterns of word usage. We can make use of these implicitly discovered concepts to solve an analogy problem like "king is to queen as aviator is to what?" The embedding has a very good chance of proposing *aviatrix* as the point that completes the parallelogram.

But the machine-learning algorithm can't quite tell the difference between definitional male-to-female shifts, like bachelor-to-bachelorette, and statistical male-to-female shifts, like *truck* to *doll* or *doctor* to *nurse*. For example, if we move in the male-female direction from *carpentry*, we get to *sewing*. If we move in the male-to-female direction from *chuckle*, we get to *giggle*. If we move in the male-female direction from *brilliant*, we get to *lovely*. These examples don't reflect the formal meanings of the words. They capture stereotyped patterns of usage.

The Microsoft team showed that they could reduce this bias without doing too much damage to the overall embedding. They proposed the following steps.

LESSON 10 | PITFALLS IN APPLYING MACHINE LEARNING TRANSCRIPT

Figure out the high-dimensional direction that encodes gender. To do so, they identified 10 gender pair difference vectors. They identified the main direction of difference using a machine-learning and statistical technique called principal component analysis, or PCA. PCA is an unsupervised method that makes use of similarities between instances. It leverages computational techniques from linear algebra to re-represent its input data in terms of a lower-dimensional set of underlying directions in the data.

The smaller set of dimensions are known as the principal components, which is where PCA gets its name. The principal components themselves are sorted by importance. So, dimension one is the best single dimensional reconstruction of the data. In the case of pairs of gendered words, that single best direction ends up being gender.

Then, for any pair of words that should not have a difference, they did a high-dimensional projection of those words so that they no longer differed in the gender direction while leaving all other dimensions unchanged. In this low-dimensional visualization, that would mean collapsing *carpentry* and *sewing* to a single point and collapsing *brilliant* and *lovely* to a single point.

But keep in mind that these points are embedded in hundreds of dimensions. It's like we're looking down on two points that are at different depths. If we rotate our view to another perspective, we can see that they remain distinct in many other ways. They just have no distinction in a particular direction that corresponds to gender.

Geometry is where the problem showed up, and geometry is the tool that was used to solve it. Of the handful of different pitfalls that have been identified, this is the one that has the most comprehensive solution.

Another subtle pitfall that is particularly difficult to remove comes up when we are trying to use historical data to change how we will behave in the future. Historical data has baked into it the way we have behaved in the past, and that can send misleading signals.

Machine-learning researcher Rich Caruana described a compelling case in which hospital information was used to predict patient outcomes. He wanted to use a machine-learning approach to assess patients coming into a hospital to see which were at risk for pneumonia. If a patient is predicted to be at high risk for pneumonia, the doctors can allocate more resources to these patients to help make sure they survive their stay.

All good, right? The team creating the system did everything by the book. They collected data on patients as they entered the hospital. They recorded age, medical history, vital signs, etc. Then, when the patient was discharged, the team recorded whether the patient had suffered from pneumonia.

The team used this data to train a highly accurate neural network classifier. And indeed, it did an excellent job with its predictions. But the doctors and machine-learning researchers on the project balked at putting the system into practice.

It turned out that their concerns were well founded. They weren't sure what the neural network was doing, so they also trained a method called logistic regression on the same data. Logistic regression wasn't performing quite as well as the neural network, but the team could more easily tell what logistic regression was doing.

And what the more transparent method of logistic regression was doing was odd. When assessing a new patient, it would rate someone with asthma as being low risk of pneumonia. When the doctors saw that, they were very perplexed. They know that people suffering from asthma are very prone to pneumonia. Why would the machine learner believe the opposite?

Again, it's precisely because the doctors know that people suffering from asthma are very prone to pneumonia. The doctors treat patients with asthma very carefully and are tuned in to early warning signs that indicate that pneumonia might be taking hold. So the machine learners were not finding that people with asthma are, in general, less likely to have trouble with pneumonia. They were finding that people with asthma are less likely to have trouble with pneumonia when treated by doctors who know they have asthma.

And of course, the learner was correct, in this more narrow sense. But the output of the learner should not be used to change hospital policy. Taken at face value, the results suggest not treating asthma patients with particular care. The hero in this story is logistic regression. Well, the hero is Rich Caruana, who figured out what was going on. But the machine-learning tool he used to do it was logistic regression.

It's worth taking a closer look at this algorithm. I wish I had the asthma-pneumonia data, but I did find another data set that has some interesting parallels. It's a standard data set from a tragic maritime disaster—a shipwreck—and we can look at the data to see what lessons it contains.

LESSON 10 | PITFALLS IN APPLYING MACHINE LEARNING TRANSCRIPT

Let's pretend we are safety engineers and we want to put measures into place to help prevent fatalities. It would be useful for us to be able to predict who is at risk of dying in the event of a shipboard accident. If we can figure out who is at risk, we could perhaps train the at-risk people on safety measures or place them on the ship close to trained personnel—something to give them a better chance of surviving.

So we take historical data, run a machine learner on it, and use the learned rule to figure out who is at risk. Let's see what we get. We read in the training data, ship.csv. We'll convert the file one line at a time into a numerical Python array. The resulting features are:

- Pclass: the passenger's class. It can be 1, 2, 3, for first class, second class, or third class.
- Sex: We code male as 0 and female as 1.
- Age
- SibSp: the number of siblings and spouses the passenger has on board
- Parch: the number of parents and children the passenger has on board
- Fare: how much the passenger paid for the ticket
- Embarked S: Did the passenger get on the ship at Port S?
- Embarked C: Did the passenger get on the ship at Port C?
- Embarked Q: Did the passenger get on the ship at Port Q?

We randomly split the labeled passengers into training and testing sets. We train a neural network to predict who will survive. It gets a respectable error rate of 17%.

But something is not quite right. I went through all the data and asked the model to predict the probability of survival for each passenger. On average, passengers are 44% more likely to survive if they are female. In fact, for 100% of the passengers, their predicted chance of survival is higher as a female than as a male.

From a policy standpoint, it sounds like the right thing to do is invest in protecting the men. The women seem to be fending for themselves quite well. But that's almost certainly a misleading result. We can't tell from this data whether women or men are better at surviving a shipwreck, all things being equal, because we know that things were not equal.

The name of the ship involved in the accident we're studying is the *Titanic*, which struck an iceberg and sank in the chilly waters off the coast of Canada in 1912. The staff of the *Titanic* were told to put women and children first onto the lifeboats. So what we're seeing here is not useful for creating a new policy, because what we're seeing are the results of a prior policy used when the data was generated.

The pattern of male-female results we're seeing is analogous to what researchers saw in the asthma-pneumonia example. We can see these effects more clearly by applying logistic regression, just like Rich Caruana did in the asthma-pneumonia data.

I ran logistic regression, which is available as part of Scikit-learn. It gets about an 18% error rate on the *Titanic* survivors data, which is only a bit worse than what we saw from the neural network. More importantly, the way logistic regression works makes it a lot easier to interpret the machine-learning result.

Logistic regression learns one weight for each feature in the data. So, in this sense, it is very similar to the perceptron or a linear support vector machine. Even more so, logistic regression can be described as a special type of neural network. Logistic regression is kind of like a one-layer neural network with a sigmoid activation function. Actually, that's exactly what it is. It has one weight per input feature, and output activations are computed by taking the weighted sum and transforming it with the sigmoid function. Because of this specialized structure, the weights in logistic regression can be found somewhat more efficiently than training the equivalent neural network.

I once attended a talk by a prominent statistician addressing a workshop of machine-learning researchers. He said that he had to give credit to machine-learning people. Specifically, he thought we were much better than statisticians at naming things. He gave a bunch of examples and pointed out that *neural network* is just a much more exciting name than *logistic regression*.

Of course, machine-learning folks would not agree that machine learning is just statistics with better names. I mean, for one thing, *neural network* refers to a far richer class of models than just the logistic regression model we're talking about now. And since it's better to call a shovel a shovel, we'll refer to this model by its more specific name: logistic regression.

LESSON 10 | PITFALLS IN APPLYING MACHINE LEARNING TRANSCRIPT

So in what sense is logistic regression better than other neural network models in this situation? That's because of the way that the logistic regression model works. The model takes each feature in the input and weights it. Then it sums the weighted features and applies the sigmoid to the result.

The sigmoid transforms the total into something in the range zero to one—like a probability. The higher the weight that's on a feature, the more that feature contributes to pushing the probability toward one. That means we can directly interpret the feature weights. Positive weights are evidence for the positive class, and negative weights are evidence against the positive class.

If we output the learned feature values in the *Titanic* data, what we see is that the feature with the most impact on survival is sex, and it's a big boost toward survival when the feature value is one—female, as it turns out.

There's also a significant negative impact of passenger class. The first class passengers have the highest chance of survival, then things fell rapidly for people in second and third class. Here, we have a signal that should be important for the design of future safety systems. Third-class passengers had to travel a long, complicated route to reach the lifeboats, and many lifeboats had already launched by the time the passengers arrived. In addition, it's believed that the third-class passengers, unlike first class, were traveling with all of their worldly possessions and were more reluctant to abandon ship.

There's a somewhat strong influence of number of siblings plus spouse in the negative direction. And there's an even stronger positive influence of people departing from Queenstown. A *Titanic* expert might disagree, but my guess is that Queenstown Q was more affluent than Southampton S, and what we're really seeing is a proxy for passenger class.

Regardless, the takeaway here is that some machine-learning methods, like logistic regression, provide a much better window into how the learner makes decisions. Other machine-learning methods, like more general neural networks, are much more opaque. An advantage of transparent methods is that there's a better chance we can tell that they have problems before they are deployed.

It's not entirely clear what to do when the data lacks the necessary perspective to be able to tell us how to intervene to help save more people. In a later lesson, we will talk about causal inference in machine learning.

The causal perspective may provide just the right tools for drawing conclusions from this kind of confounded data that mixes data-collection policies with differences in outcomes.

Arguably, what we're seeing here is an example of the number-one biggest factor that lies behind failures of machine learning. It is due to the use of an inappropriate proxy measure. A *proxy* refers to a substitute. We often train our models to predict one quantity when we really want to measure another. The proxy measure is the one we have. As long as our proxy measure is well aligned with what we really want, the results can have validity.

Consider zip code and income level. These two features are not the same. But zip code is a proxy for other information that correlates extremely well with income level—people tend to live near other people with similar incomes. So zip code can serve as an effective proxy measure of income. And race, for that matter.

A well-studied example of a poor proxy measure comes up in predictive policing. If we could train a classifier to tell us where crime is likely to occur, we could deploy security officers to those areas in a more targeted way, reducing the overall cost of protecting the community. But we don't have a way to detect when crime happens, so we can't train a classifier using this information.

But we could use arrest as a proxy for crime. That's something for which there is data, and surely there's some connection between arrests and crime, right? Sure, but the two are better aligned in some communities than others. Moreover, if the police force is already deployed in a low-income community, it is more likely to make arrests in that community independent of the actual distribution of crimes.

If we train a model to predict where arrests occur, it will end up predicting that arrests are more likely where there are more police officers. If we then take arrests as a proxy for crime, these predictions suggest that we assign more police officers to wherever they were already patrolling.

Just like the Warwick-Quincy case, initial imbalance leads to even worse imbalances. It's a positive-feedback cycle, but acceleration of this positive-feedback cycle is not evidence that real progress is being made in preventing crime.

LESSON 10 | PITFALLS IN APPLYING MACHINE LEARNING TRANSCRIPT

To sum up, the technical part of machine learning is concerned with turning labeled instances into prediction rules. But when we apply machine learning to real world problems, we have to pay attention to the broader context in which data is collected and used, including how new data comes in after the system has been deployed.

Like in the predictive policing setting, we need to make sure that the quantity being predicted—arrests—is a good proxy for the quantity we want to reduce—crime. If it's not, we should seek out better data that measures the quantity of interest more directly. We need to make sure that policies employed when data is collected don't obscure how predictions will change if a new policy is adopted.

To avoid the problem of asthma patients looking like they are at very low risk for pneumonia, it can help to collect data in a more controlled setting. For instance, we could record precisely how doctors are treating the different patients and try to treat some of the non-asthma patients in similar ways to the asthma patients so direct comparisons can be made.

We need to make sure that the loss function we use doesn't encourage the machine-learning approach to ignore minority subgroups, such as the accent of people from Warwick or the faces of African American women. When that happens, we can consider alternate loss functions that more evenly weight errors across the classes. We can also try to proactively collect more balanced data.

In some cases—like Amazon's attempt at creating an automated resume screening program—the machine learner is hopelessly biased. In these cases, we should give up. Machine learning makes it all too easy to deploy a plausible but damaging system, and harms are especially acute for smaller and more vulnerable populations.

Whenever we use machine-learning algorithms to solve real-world problems, knowing how to cast a skeptical eye on the results we're getting can help prevent their misuse. Machine learning is a powerful tool. But it's up to all of us to demand that machine learning is deployed in ways that minimize possible harm, even while society reaps the benefits.

LESSON 11

CLUSTERING AND SEMI-SUPERVISED LEARNING

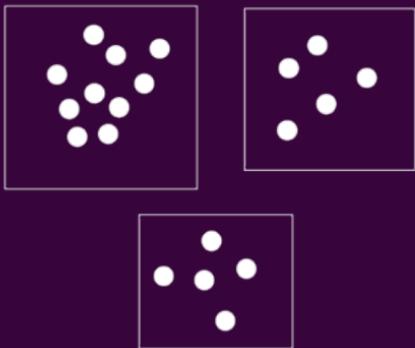
The early and canonical approaches to machine learning all involved providing expert labels on the data—supervised learning. Most of what you've been seeing so far has been supervised learning. But wouldn't it be great if a machine learner could figure out the labels for itself and somehow be made to work with less supervision? The similarity-based algorithms you've seen have already offered ideas about providing less supervision. This lesson will teach you how to create semi-supervised learning algorithms that benefit from a combination of supervised and unsupervised examples.

Working with Unlabeled Data

Imagine we have a set of data points in two dimensions. The data is unsupervised; there are no labels.

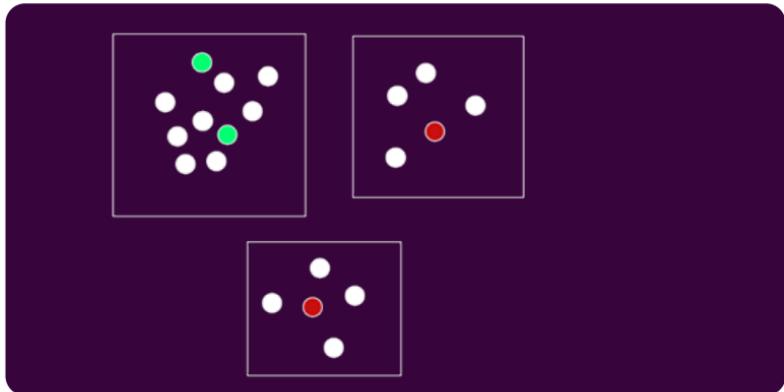


But using what we know about the similarity, or nearness, of the points in space, we might think of the points as falling into three clusters.

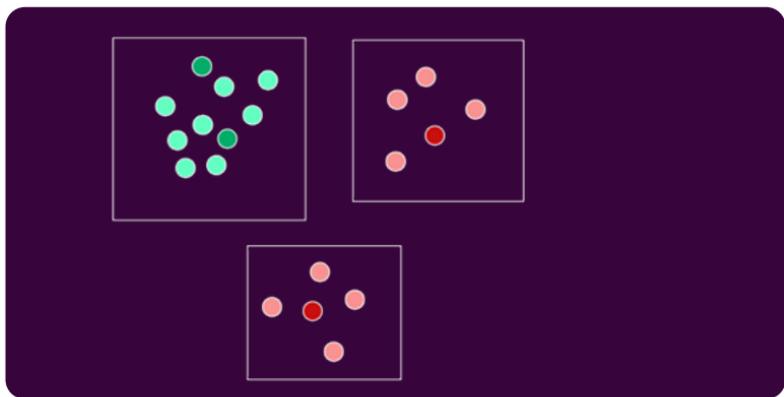


We don't know what labels the points have, but we might guess that, within each of these clusters, points share the same label. That's reminiscent of what happens in a nearest neighbor classifier or a support vector machine: Unlabeled instances get their labels from nearby labeled instances.

Now imagine that we get a very small number of labels—just two per class on average. The labels go to whichever cluster they happen to fall into.



The analysis that we did on the unsupervised points suggests that we can label the remaining points consistently within each cluster.



This approach—unsupervised clustering, followed by diffusing any known labels to the other points within the cluster—is an example of a semi-supervised learning algorithm.

The unsupervised clustering step depends on being able to assess “nearness.” In this two-dimensional case, straight-line distance, also known as Euclidean distance, is a very natural choice.

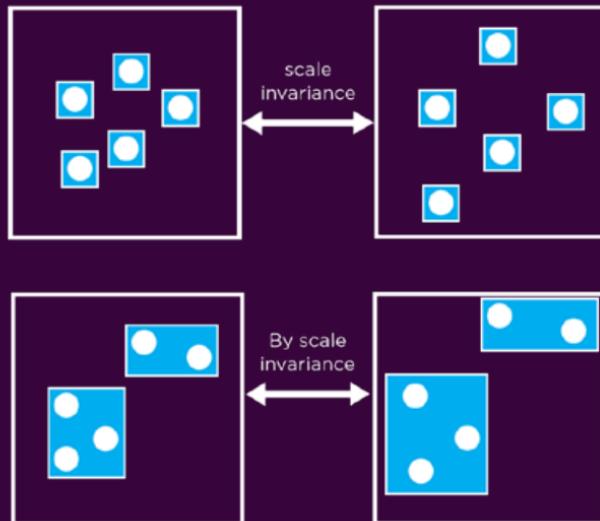
In this approach to semi-supervised learning, the supervised part is a very easy computation: To apply the supervision, the approach just diffuses labels to other instances in the same cluster. The clustering part is where the machine learning action is.

The Impossibility Theorem

Up until the 21st century, there had been a lot of vaguely competing ideas about what a clustering algorithm should do but little work on formulating a precise problem definition.

In 2002, Jon Kleinberg, a theoretical computer scientist at Cornell, analyzed clustering, and his perspective helped introduce some clarity about what clustering might mean and how to set realistic expectations about what it can do.

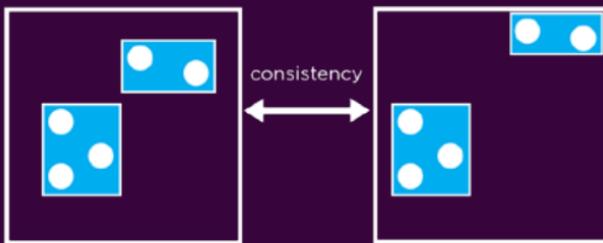
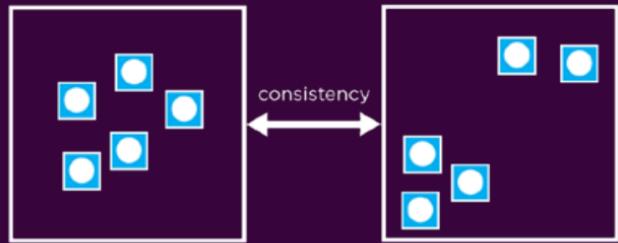
Returning to our simple example, the clusters that we see in a picture remain the same even if we zoom in or out. An algorithm that violates this rule would, for example, merge clusters as we zoom out.



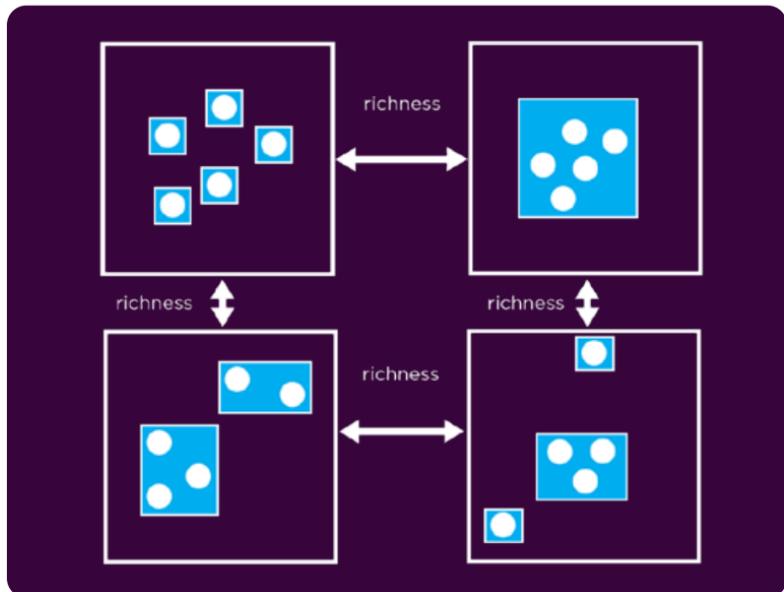
This property of clusters remaining the same before and after a zoom operation is called scale invariance, according to Kleinberg. It's the first of three properties that make for a "good" clustering algorithm.

A second and related property is that a clustering algorithm should remain consistent if the individual clusters are compressed or if the distance between clusters is stretched. If the clustering algorithm clusters a set of data in a particular way, it should continue to do so if the within-cluster distances decrease and the between-cluster distances increase. In other words, if the clustering algorithm likes a clustering, it should also like it if the set of clusters becomes more spread out and each individual cluster becomes more compact. Kleinberg called this consistency.

Whenever it's cheap to gather large amounts of unlabeled data but obtaining labels is expensive, semi-supervised learning applies.



A third desirable property in Kleinberg's analysis is richness, where no conceivable way of clustering is entirely forbidden. In other words, the algorithm's choice of clustering should range from each item being in its own cluster, to every point clumped together into a big single cluster, to everything in between. Nothing is ruled out beforehand.



In its simplest form, richness has two aspects: There must be some arrangement of the points that makes the clustering algorithm put every point in its own cluster, and there must be some other arrangement of the points that makes the clustering algorithm group some points together.

As it turns out, in 2002, Kleinberg showed that no clustering algorithm can simultaneously provide all three properties.

Semi-supervised methods are essential tools that help people apply machine learning in all sorts of cases where a lack of labeled data could otherwise make machine learning impossible.

His so-called impossibility theorem shows that the three properties, in fact, are not fully compatible. An algorithm can have any two of the properties, but not all three at the same time.

The impossibility theorem suggests that there is never going to be a single “best” algorithm for all clustering problems. The theorem also licenses us to use algorithms that relax one or more of the three desirable properties.

Try It Yourself

Follow along with the video lesson via the Python code:

[L11.ipynb](#)

Python Libraries Used:

functools.reduce: Summarizes a vector or list by a single value.

keras.preprocessing.image.array_to_img: Converts an array into an image.

math: Mathematical functions.

numpy: Mathematical functions over general arrays.

random: Generates random numbers.

scipy.stats: Computes vector statistics like the mode of a list.

sklearn.datasets.fetch_openml: Provides access to OpenML datasets.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

Key Terms

***k*-means:** An approach to unsupervised clustering that iteratively defines a set of centers and assigns vectors to their closest centers.

transfer learning: Any technique that leverages statistical insights gained from doing supervised learning on related problems.

READING

Charniak, *Introduction to Deep Learning*, chap. 7.

Domingos, *The Master Algorithm*, chap. 8.

Kleinberg, “An Impossibility Theorem for Clustering.”

Russell and Norvig, *Artificial Intelligence*, sec. 21.7.

QUESTIONS

1. What are the three conditions for clustering that Jon Kleinberg showed were mutually incompatible?
2. K -means is guaranteed to terminate after a finite number of iterations because each iteration improves the loss if possible. Each iteration of gradient descent also improves the loss if possible, but it is not guaranteed to terminate after a finite number of iterations. What's the reason for this difference?
3. K -means produces more compact clusters than agglomerative clustering. Which do you think is better for active semi-supervised learning? You can compare Scikit-learn's KMeans and AgglomerativeClustering (50 clusters) on the MNIST dataset to see what they do. Which clustering approach do you expect to perform better?

Python Libraries:

sklearn.cluster.AgglomerativeClustering: Scikit-learn's tree-based hierarchical clustering algorithms.

sklearn.cluster.KMeans: Scikit-learn's k -means clustering algorithms.

sklearn.neighbors.KNeighborsClassifier: K -nearest neighbor classification algorithm.

Answers on page 481

Clustering and Semi-Supervised Learning

Lesson 11 Transcript

The early and canonical approaches to machine learning all involved providing expert labels on the data. We call that “supervised learning.” Most of what we’ve been doing so far has been supervised learning. But, wouldn’t it be great if a machine learner could figure out the data for itself and somehow be made to work with less supervision?

The similarity-based algorithms we’ve seen have already given us ideas about providing less supervision. Let’s think of it this way. Let’s say we have two data points that are very close, but only one is labeled. The algorithm could go out on a limb and label the second instance. In this lesson, we’ll talk about how we can create semi-supervised learning algorithms that benefit from a combination of supervised and unsupervised examples.

Let’s do a thought experiment. Imagine we have a set of data points in two dimensions. The data is unsupervised; we have no labels. But, using what we know about the similarity or nearness of the points in space, we might think of the points as falling into three clusters. We don’t know what labels the points have, but we might guess that, within each of these clusters, points share the same label. That’s reminiscent of what happens in a nearest-neighbor classifier or a support vector machine; unlabeled instances get their labels from nearby labeled instances.

Now, imagine that we get a very small number of labels, just two per class on average. The labels go to whichever cluster they happen to fall into. The analysis that we did on the unsupervised points suggests that we can label the remaining points consistently within each cluster. This approach—unsupervised clustering, followed by diffusing any known labels to the other points within the cluster—is an example of a semi-supervised learning algorithm.

The unsupervised clustering step depends on being able to assess nearness. In this two-dimensional case, straight line distance, also known as Euclidean distance, is a very natural choice. In this approach to semi-supervised learning, the supervised part is a very easy computation. To apply the supervision, the approach just diffuses labels to other instances in the same

LESSON 11 | CLUSTERING AND SEMI-SUPERVISED LEARNING TRANSCRIPT

cluster. The clustering part is where the machine-learning action is. Let's take a closer look at clustering to understand what is possible and how we can find good clusters automatically.

First of all, what kind of behavior do we want from a good clustering algorithm? Up until the 21st century, there had been a lot of vaguely competing ideas about what a clustering algorithm should do, but little work on formulating a precise problem definition. In 2002, Jon Kleinberg, a theoretical computer scientist at Cornell, analyzed clustering, and his perspective helped introduce some clarity about what clustering might mean and how to set realistic expectations about what it can do.

Returning to our simple example, the clusters that we see in a picture remain the same, even if we zoom in or out. An algorithm that violates this rule would, for example, merge clusters as we zoom out. Jon Kleinberg called the property of clusterings remaining the same before and after a zoom operation scale invariance. This property is the first of three properties that make for a good clustering algorithm.

A second and related property is that a clustering algorithm should remain consistent if the individual clusters are compressed, or the distance between clusters is stretched. If the clustering algorithm clusters a set of data in a particular way, it should continue to do so if the within-cluster distances decrease and the between-cluster distances increase. That is, if the clustering algorithm likes a clustering, it should also like it if the set of clusters becomes more spread out and each individual cluster becomes more compact. Kleinberg called this property consistency.

A third desirable property in Kleinberg's analysis is richness, where no conceivable way of clustering is entirely forbidden. That is, the algorithm's choice of clustering should range from each item being in its own cluster to every point clumped together into a big single cluster, to everything in between. Nothing is ruled out beforehand. In its simplest form, richness has two aspects: There must be some arrangement of the points that makes the clustering algorithm put every point in its own cluster and there must be some other arrangement of the points that makes the clustering algorithm group some points together.

With these three properties in mind, let's look at a few candidate clustering algorithms. First, what happens if we have a clustering algorithm with a fixed distance threshold, let's say 0.1? This choice puts points in the same cluster whenever they are closer than that fixed threshold. This algorithm provides consistency. If the within-cluster distances are already below the distance threshold, shrinking those distances won't make them rise above the threshold. And, if the between-cluster distances are above the distance threshold, expanding those distances won't make them smaller than the threshold.

The algorithm also provides richness. If we want the clustering algorithm to cluster the points all separately, we just have to place the points far enough apart. If we want the clustering algorithm to cluster some points together, we just need to put some points closer than the fixed threshold. But the distance threshold approach does not provide scale invariance. If we zoom in enough by increasing all the distances by a fixed ratio, points break off to form new clusters. The problem, of course, is that a fixed distance threshold of 0.1 didn't scale up with the spacing between the points.

Let's consider an algorithm that chooses a threshold dynamically by scaling it to be a fixed percentage of the largest distance between points. As the distances are scaled up or down, the clustering will remain the same! This approach does provide scale invariance. This algorithm has the property of richness, too, as we can space the points out so that they form clusters or not. But, this ratio-threshold algorithm lacks the consistency property. As we spread out the clusters, the threshold grows and can end up merging clusters together.

Alright, one more try. Let's tweak the algorithm to get consistency back. Instead of a fixed threshold, or a ratio-distance threshold, we can recompute the threshold to be whatever value is needed to ensure that there are k clusters for some selected k . This approach has scale invariance. That is, as we zoom in or out of the points, we're always making the comparison based on the distances needed to ensure there are k clusters. And, it will always be the same k clusters, when we scale things uniformly.

LESSON 11 | CLUSTERING AND SEMI-SUPERVISED LEARNING TRANSCRIPT

This algorithm also exhibits consistency. Making the points within a cluster closer will not change which k clusters are found. Neither will making the clusters further apart change which k clusters are found. So, the same clustering will result from contracting points in a cluster and expanding the clusters themselves.

However, this approach does not exhibit richness because it will always return k clusters. For richness to hold, an algorithm has to be able to produce different numbers of clusters. Trying to get all three desirable properties at once feels a little like whack-a-mole. We looked at three sensible algorithms. All of them are variations of a standard hierarchical clustering method called single linkage clustering, which dates back to the 1950s. They vary only in the rule they use to stop merging clusters.

Scikit-Learn provides an implementation of single-linkage clustering as part of a broader clustering package called `sklearn.cluster`.

`AgglomerativeClustering`. Unfortunately, none of the single-linkage clustering variations could simultaneously capture all three desirable clustering properties. Each time we modified the stopping rule to cover a property the previous version lacked; it lost some other property.

As it turns out, this problem is not just caused by a lack of imagination. In 2002, Kleinberg showed that no clustering algorithm can simultaneously provide all three properties. His impossibility theorem shows that the three properties, in fact, are not fully compatible. An algorithm can have any two of the properties, but not all three at the same time.

Alex Williams provided a lovely visual proof of Kleinberg's theorem. The property of richness means we can arrange five points in two different ways: in one case, each point is in its own cluster, while in the other case, some other clustering is created. Now, the property of scale invariance says we can expand the second arrangement of points without changing the clustering. So, let's expand the distances until the closest two points are as far apart as the two closest points in the first arrangement.

The property of consistency says we can shift around the first arrangement to match the second arrangement without changing the clustering. That's because all the between-cluster distances are increasing. But we've reached a conflict! Following the constraints dictated by the three properties has led

us to an impossible requirement that the same arrangement of points must be assigned two different clusterings. It's a proof by contradiction. And, the consequence is that no clustering algorithm can obey all three properties.

The impossibility theorem suggests that no one clustering algorithm can be all things to all people. There is never going to be a single best algorithm for all clustering problems. And, the impossibility theorem also licenses us to use algorithms that relax one or more desirable properties.

In particular, let's look at an algorithm that lacks richness; we have to tell it in advance how many clusters to produce. The setting is that our algorithm will be given a set of points along with the ability to compute distances between any pair of points. We choose k , the desired number of clusters. Let's pick $k=2$ in this case. Next, the algorithm searches for a set of k centers and an assignment of data points to these centers. Its goal is to minimize the total squared distance between data points and their respective centers. The assignment to centers defines the clustering.

We have two of the three elements necessary to define a machine-learning algorithm. We have a representational space, which consists of k centers and an assignment of the data to centers. We have a loss function, which is the total squared distances from the points to their respective centers. The last piece we need to define is the optimizer. Unfortunately, optimizing this loss perfectly in all cases is unlikely to be tractable. The problem is in a class known as NP-complete, members of which have resisted efficient exact solutions since even before the class was first explicitly identified in the 1970s.

Fortunately, there's a simple and elegant algorithm, called k -means, that is guaranteed to find a local optimum of the loss function in finite time. K -means is spiritually similar to k -clusters in that they both partition the points into k categories. One difference between the algorithms is that k -means is stochastic, so it is not guaranteed to find precisely the same answers each time it is run. K -means also favors compact groups. Remember that the mean is a measure of central tendency, whereas k clusters can produce long chains as it merges nearby points together.

LESSON 11 | CLUSTERING AND SEMI-SUPERVISED LEARNING TRANSCRIPT

Here's how k -means goes. The optimizer starts off with center locations and assignments of data to the centers. It then alternates between two steps. Both steps are simple, and both are guaranteed to decrease the loss, when they produce any change at all. The first step is to reassign the data points to centers. Specifically, each data point is assigned to its closest center.

Either the data point is already assigned to its closest center, in which case we're at a local minimum of the loss function. Or, some data point is moved to a center closer to it than it is to the one it is currently assigned to. That reassignment must decrease the loss, which is the sum of the squares of the distances from data points to their centers. Minimizing the distance also minimizes the squared distance because squaring two numbers does not change their relative ordering. In fact, of all the reassessments of data points to centers, assigning each data point to its closest center decreases the loss the most.

The second step is to recompute the centers for each cluster. Specifically, each center is moved to the mean of the data points it is assigned to. That's why the algorithm is called k -means. Either the center is already assigned to the mean of its associated data points, in which case we're at a local minimum of the loss function, or some center is moved. Such a move decreases the loss the most over all center relocations.

Let's simplify to just one dimension and consider how you'd minimize the squared distance between a center and a set of data points in that one dimension. Anyplace we put the center on the number line creates a set of squared distances to the data points. That means that the amount of loss due to each data point increases quadratically with the distance between the center and the data point. The sum of those squared distances moves around smoothly and also quadratically. To find the center that minimizes this quantity, we can use a little differential calculus. Thanks again, calculus!

For any choice of center c , we're summing up the squared differences between c and each data point to get the loss for c . Calculus tells us that the minimum of this function is found when the derivative is zero. That's the value of c where the loss is in balance; it's made worse by moving c in either direction.

The derivative, L prime, is found by taking the derivative of each of the terms in the sum. Setting this quantity to zero and solving for c gives us: the sum of the data points divided by the number of data points. That is, it is the average or mean!

The mean is what we get in one dimension. But, the choice of value for the center in each dimension is independent of the choice for the other dimensions. So, we can minimize loss by finding the mean value of the data points in each dimension separately. No other selection of center improves the loss more than that.

Okay, each time we reassign the data points, the centers might need to move. Each time we move the centers, the data points might need to be reassigned. But, since the loss improves each time, we take either of these steps and since the number of ways of assigning data points to centers is finite, we can only repeat these steps a finite number of times before no further improvements are possible.

To solidify this concept, let's build up an implementation of k -means. The algorithm is available in the scikit learn library, but the steps are simple enough that we can build our own to see what it looks like. The first subroutine we need is one that takes the dataset and a set of k centers and assigns each item in the dataset to its closest center. We set n , the number of datapoints, d , the dimensionality of the datapoints, and k , the number of centers.

Next, we need to compute the squared distance between each center and each data point. By reshaping the data to be 1 by n by d , and the centers to be k by 1 by d , that signals to NumPy that when it subtracts these two arrays, it creates an array of shape k by n by d . That is, it computes all combinations of the k centers and the n datapoints for each of the d dimensions. We assign those differences to `res`. Squaring each of the differences, then summing along dimension 2—that's the components of the vectors—produces the sum of squared distances, which is the squared distance between the centers and the datapoints. The resulting array is of shape k by n .

Now, we want to know which center has the smallest squared distance for each data point. `Argmin` produces the index of an array with the smallest value along the given dimension. Here, we're using dimension 0, which varies over the k centers. `Centerids` is now an array with one integer for

LESSON 11 | CLUSTERING AND SEMI-SUPERVISED LEARNING TRANSCRIPT

each datapoint that indicates which of the centers is closest. In addition to returning these centerids, we also calculate and return the current loss, which is the min squared distance summed over all data points.

The meat of the subroutine is three assignment statements, which could be chained together in a single line. That's one of the beautiful things about NumPy. It lets you execute powerful operations over entire arrays without writing loops. In exchange, NumPy expects us to be very careful to keep track of what the different dimensions represent in our array.

The second subroutine we need for implementing k means is one that takes the data and the centerids and computes the centers by averaging all of the datapoints with the same id. After extracting the number of datapoints and dimension, we initialize the array of center locations to a k by d array of all zeros.

This time, I didn't see a way to avoid a loop. So, for each of the cluster id values from 0 to k , we do the following operations. First, form a smaller array consisting of all the datapoints with the current center id. Call that array cols.

To be robust, let's make sure cols isn't empty; its length isn't zero. That can happen if there's a center that has been elbowed out of the running by the other centers being closer to all of the data points. If it is, that means our center is out of the action and we should probably pick a different location for it. We simply choose one of the data points at random to be this new location. As a quick aside, notice that reassigning the center location in that way cannot increase the loss, so our earlier analysis of the algorithm's convergence is still valid.

Otherwise, there are data points close to the center and we want to move that center to the mean of the closest points. NumPy's mean method computes the average of an array along any given dimension. Here, we choose dimension 0, which corresponds to the different data points. So, mean produces a component-wise average of all the data points with cluster id equal to i .

After completing the loop, we return the newly computed centers. With these two subroutines in hand, k -means can be described very simply. We initialize the k centers by selecting random data points. This choice is quite

sensible, because it ensures that centers are in the right neighborhood. Choosing random d -dimensional points is another option, but it runs the risk of having all of the random centers starting quite far from the data.

We now loop until the loss stops changing. If the oldloss is different from the new loss, we use assign data to assign each datapoint to its closest center. Then, we use compute means to move the centers to the means of the points assigned to them. And that's it. Repeat until the loss stops changing.

Here's what we get when we run k -means with 10 clusters on some image data. Here, the data are a collection of grayscale images of handwritten digits known as MNIST. The centers are averages of these images, so we can plot them as images. K -means does a very good job of categorizing the digits just based on their appearance. K -means isn't using the official labels, just the raw data. But we can see that k -means identifies a zero-looking cluster, a 9-looking cluster, a 1-looking cluster, a... another 9-looking cluster? Another 9-looking cluster? 8, 3, 2, 6, and another 1. Okay no 5, no 4, although one of the 9s looks pretty 4 like.

Okay, so it does not perfectly assign images to digit categories. In part, it's discovering that 5s and 8s are more similar to each other than slanty 1s are from up and down 1s. Even though there are 10 digits, there are more than 10 clusters in the data.

Nevertheless, we can use this clustering to label the training data following the semi-supervised approach described earlier. For each cluster, discovered from analyzing the training data, we pick one of the data points in that cluster at random and request its label. The labels we get for these clusters are: 0, 7, 1, 9, 4, 5, 3, 2, 6, 1. Next, for each testing image, we find its closest cluster and predict its label as the label of the associated cluster. The resulting accuracy is 50%. Not great, but way better than chance. Since there are ten categories, chance performance would be 10%. By way of comparison, nearest neighbor classification on this data using 10 labeled examples results in a lower accuracy of 45%.

We can improve things significantly by letting k -means identify more clusters and labeling each cluster. I found that k equals 50 clusters allowed semi-supervised k -means to achieve an accuracy of 80%. K nearest neighbors with 50 labeled examples also improves to 60% accuracy; it's not improving as much as k -means does.

LESSON 11 | CLUSTERING AND SEMI-SUPERVISED LEARNING TRANSCRIPT

The specific flavor of semi-supervised learning I just described uses the clustering to request labels for specific points. Instead of passively accepting labels on arbitrary points that might be in the same cluster, it actively requests labels that will be as informative as possible based on the unsupervised analysis. Algorithms that can request their own labels are called active learning algorithms. So, our k -means approach is an active semi-supervised learning algorithm.

Semi-supervised learning, especially when enhanced by active requests for informative labels, gives us an important way to make the most of a small set of labeled examples. For the most part, active learning algorithms are a special case of semi-supervised learning algorithms because active learning can select among the unlabeled examples for examples whose labels would accelerate the semi-supervised learning process. K -means is used standalone in unsupervised settings, too.

In business analytics, it is sometimes helpful to organize a business's customer base into segments that might have different interests, preferences, or needs. K -means can provide an analysis of customer descriptions that groups similar customers together. A streaming service might discover that some customers are most interested in romantic comedies and others in murder mysteries and tailor the service differently for the two groups. K -means can be useful in image analysis to break up the pixels in an image into regions, automatically splitting foreground objects from background objects.

Back in the day when computer displays could only show a small number of colors at any one time, a version of k -means was used called vector quantization. The *quanta* in this case were the 16 or 32 or 256 representative colors that were used to replace all the millions of colors that might be in the original image. For an ocean scene, it might use the 256 colors to capture subtle shading distinctions amongst blues and greens. For a crowd scene, it might allocate a much broader palette. This idea is still used as a way of compressing an image so that it can be represented more compactly without losing the most important visual details.

Semi-supervised learning strives to obtain good performance on a dataset while reducing the need for labeled data.

Another well-known approach to this problem, called transfer learning, is to leverage statistical insights gained from doing supervised learning on related problems. This approach is used by experimental psychologists to quickly train image-recognition neural networks to analyze the behavior of mice in their experiments. They start with a pre-trained network known as VGG-16 and can quickly teach it to recognize specific mouse behaviors for automatically gathering statistics on the reactions of the mice to different stimuli. Transfer learning is now a tool for automating data collection for scientists, providing the same kind of amplification of their capabilities that statistical methods like t-tests provide for analyzing statistics.

Of course, there is no free lunch. We still needed millions of labeled examples to train the VGG-16 network. But transfer learning allows the cost of obtaining those labeled examples to be amortized over a set of visual-recognition tasks. To some degree, it's a matter of luck whether transfer learning or semi-supervised learning via unsupervised clustering work in any given problem. That is, the representation learned for one task or the clustering produced might be a good match for other tasks. But, nothing in the learning process is specifically trying to make the representations relevant to other problems.

Another approach to learning works by training on multiple tasks at the same time, so there's an opportunity for each solution to be explicitly tailored to help the others. This general category of approach is called multitask learning. One instantiation of multitask learning is a neural network with a single input and different outputs for each task, where the branches of the network splitting into the different outputs are known as heads. Kind of like a hydra.

The multi-headed learning architecture makes the most sense when all of the tasks can be applied to the same inputs. For example, if the input is an image, there might be one task head identifying the category of the objects in the image and a separate task head specifying its position in the image. Two tasks. Same inputs. Some possible sharing of internal processing that benefits both.

How to make effective use of data with only limited labels is a core problem in machine learning. Even though massive amounts of labeled data are easier to come by than ever before, there is always a long tail of new problems for

LESSON 11 | CLUSTERING AND SEMI-SUPERVISED LEARNING
TRANSCRIPT

which labeling of data is impractical. As such, many learning variants have been created to address different situations in which other knowledge can be leveraged to reduce the need for labels.

Whenever it's cheap to gather large amounts of unlabeled data, but obtaining labels is expensive, semi-supervised learning applies. Semi-supervised learning can be augmented with active learning if it can be made possible to request labels for specific informative examples. Or, the large amounts of data that have been used to train related problems can be leveraged to perform transfer learning. And, for related problems that share the same input space, multitask learning can be used to magnify existing labeled data, when it is available.

All of these approaches amplify traditional supervised learning methods. Semi-supervised methods are essential tools that are helping people apply machine learning in all sorts of cases where lack of labeled data could otherwise make machine learning impossible.

LESSON 12

RECOMMENDATIONS WITH THREE TYPES OF LEARNING

Recommendation has become one of the most important applications of machine learning technology on the web. It helps people navigate the explosion of information that's available at their fingertips. But it also gives machines a tremendous responsibility: deciding whether something is worthy of our attention.

Recommendation Systems

Some of the earlier recommendation systems to appear were developed by machine learning researchers to support machine learning research.

In 2010, Richard Zemel was program cochair for a highly influential machine learning conference called Neural Information Processing Systems (NeurIPS). His team at the University of Toronto wanted a better way to solve the problem of assigning appropriate reviewers to submitted papers. So they created the Toronto Paper Matching System to help automate the process. It was a hit. By 2013, the service had been picked up by many of the major artificial intelligence, machine learning, and computer vision conferences.

And this happened just in time, as the number of submissions to the top machine learning conferences was exploding. The Association for the Advancement of Artificial Intelligence conference received more than 1,000 submissions for the first time in 2014. For the 2020 conference, there were just shy of 10,000 submissions!

Reviewers for the 2020 conference had to flag a set of about 50 of those papers that they thought would be well suited to review so that the chairs of the conference could assign each reviewer six to eight papers and each paper three reviewers.

In other words, each reviewer had to pick 50 papers out of 10,000. Even focusing on the most relevant keywords, reviewers found this task quite daunting.

If only there were a way to get a computer to help identify the papers that were likely to be a good match for a particular reviewer, that might save a lot of effort. It's much more manageable if the computer can present a reviewer with maybe 100 titles and the reviewer marks half of those as topics he or she is well informed about.

Identifying papers to review is one example of the more generic recommendation problem. We want to make programs that can sift through many, many items on behalf of a group of people to find items that each person should look at more closely. The items could be web pages or movies or apps or news articles or restaurants.

And these recommendation systems are more than just labor-saving devices. They select what items users see most easily. As a result, recommendation systems color our perception of reality.

The General Recommendation Problem

In general, the recommendation problem looks something like this. We have a person and an item and we want a numerical assessment of how well that person goes with that item—that is, their compatibility. It could be 0/1, indicating that they go together. Or it could be a number indicating how much the person would enjoy the item. Or it could represent how likely the person would be to buy the item or how much time the person would spend interacting with the item.

If the computer can predict the compatibility of a person and an item—however assessed—it can rank-order all of the items for a person and present the most compatible ones for consideration.

Different recommendation problems differ based on what kind of information there is about the person and what kind of information there is about the item.

In one version of the problem, each person and each item are associated with a feature vector. The recommendation system can then predict the degree to which the two are compatible by the similarity of those vectors.

This version of the problem is unsupervised in that the recommender has no explicitly labeled data about compatibility. There's just an assumption that the more similar the features are between the person and item, the more compatible they are.

Some news-story recommendation systems work this way based purely on similarities. Each article is tagged by editors with keywords like *sports* or *local politics* that describe what they are about. And the users select descriptors for themselves from this same set. Articles where the keywords match a user's descriptors are judged as compatible.

By contrast, in the supervised version, there are explicit compatibility judgments between people and items. That's the structure that's used in systems like online shops or video services that ask you to rate what you just bought or watched.

The recommendation system then needs to generalize from these examples to identify additional compatible pairs of people and items, even though no compatibility judgment is available for these other pairs. This version of the problem is supervised in that a sample of compatibility judgments is available for training the system.

A common variant of supervised recommendation is a recommendation system that can actively solicit compatibility judgments to gain new information for prediction. For example, it can recommend an item and then wait to find out how compatible the person and item actually are. This new information could be based on the person's direct judgment or an indirect measure, such as whether the person takes the recommendation.

One difference between these active and passive versions of supervised learning is that an active system might seek information by recommending an item that is not its best guess for the most compatible item. Instead, it might calculate that knowing a particular person-item compatibility value will better situate the algorithm for making future judgments more accurately. As a result, it can get up to speed more quickly for new items or new people. The strategic, sequential nature of this decision makes the active version closely related to reinforcement learning.

All three of these approaches to recommendation—unsupervised, supervised, and reinforcement learning—are behind popular user-facing information technologies and remain an important and active area of research.

From 2006 to 2009, Netflix held a competition known as the Netflix Prize to help improve their existing recommendation system, Cinematch. They offered \$1 million for the first team that could produce a 10% improvement over Cinematch's recommendations. In the end, Netflix cherry-picked just two algorithms that together accounted for most of the improvement.

Justification-Based Recommendation

Learning-based recommendation systems have been a lifesaver to people who would otherwise be swamped by information.

The case of recommending papers to reviewers is an example where the incentives are mostly aligned: Authors want their papers to be read by reviewers who are knowledgeable and interested, while reviewers want to read interesting papers.

But machine learning systems are deployed to recommend all sorts of items to people, and sometimes providers face strong incentives to game the system to get their items recommended.

On an ad-supported video delivery platform, the situation is more like the Wild West. Video producers get paid for views, even if the viewers are unhappy, or even if they are harmed in the long term.

Big tech companies like Google would prefer to limit incentives for unwanted behavior that threatens the value or reputation of their platforms. But it's an arms race.

It's not easy to make systems that learn and improve but at the same time avoid latching on to misleading inputs from unscrupulous participants.

One suggestion is the idea of moving away from implicit measures of user interest, such as clicks or viewing time, toward systems that let users explain why they like or do not like particular content—so the systems can tune their behavior accordingly.

For example, some existing sites let people write reviews. What if the recommender systems read the reviews to understand the justifications for a user's rating and then recommended items based directly on justifications?

Justification-based recommendation could distinguish between providers who are gaming the system and those who are producing items that people actually want. And that could make the internet healthier for all of us.

Try It Yourself

Follow along with the video lesson via the Python code:

[L12.ipynb](#)

Python Libraries Used:

matplotlib.pyplot: Plots graphs.

random: Generates random numbers.

sklearn.naive_bayes.MultinomialNB: Naive Bayes learner for binary features.

Key Terms

k-armed bandit: A decision problem where an agent must decide which of k initially unevaluated actions to choose to maximize payoff. For example, A/B testing is a two-armed bandit. A contextual bandit can make its judgments based on vectors that describe the current context.

linear regression: Regression where the output rule is the coefficients of a line.

singular value decomposition: A matrix decomposition approach that analyzes data and finds a lower-dimensional representation for it. Used in latent semantic analysis.

READING

Koren, Bell, and Volinsky, "Matrix Factorization Techniques for Recommender Systems."

Russell and Norvig, *Artificial Intelligence*, chap. 17 and sec. 19.8.

QUESTIONS

1. When learning to predict compatible items, what is the downside of overexploring? What is the downside of overexploiting?
2. For each of the approaches to recommendation that follows, how could content providers cheat to get their items recommended to users?
 - (a) Recommend items that have the most reviews.
 - (b) Ask content providers to associate an interest vector (all 0s and 1s) with each item saying which of various topics the item relates to. Recommend an item to a user if the dot product of the user's interest vector with the item's interest vector is large.
 - (c) Recommend items that users click on the most based on their descriptions.
3. In this lesson, you saw a chooser for the incremental recommendation simulation that selected the article with the highest probability of being interesting. A popular alternative to this “greedy” approach is known as Thompson sampling, in which an article is chosen probabilistically based on its probability of being interesting. Create a chooser that uses the naive Bayes model to estimate the probability of each of the offered articles being interesting, flips a weighted coin for each based on its probability to determine whether we will treat it as interesting, and then randomly chooses any of the interesting articles. What setting for alpha leads to the best average performance? Does it perform as well as the greedy approach?

Answers on page 481

Recommendations with Three Types of Learning

Lesson 12 Transcript

Recommendation has become one of the most important applications of machine-learning technology on the web. It helps people navigate the explosion of information available at their fingertips. But it also gives machines a tremendous responsibility. That's because we are delegating to the machines the job of deciding whether or not something is worthy of our attention. Some of the earlier recommendation systems to appear were developed by machine-learning researchers to support machine-learning research.

By 2010, Rich Zemel was program co-chair for a highly influential machine-learning conference called Neural Information Processing Systems, known as NeurIPS. His team at the University of Toronto wanted a better way to solve the problem of assigning appropriate reviewers to submitted papers. So, they created the Toronto Paper Matching System to help automate the process. It was a hit. By 2013, the service had been picked up by many of the major artificial intelligence, machine-learning, and computer-vision conferences. And, just in time. The number of submissions to the top machine-learning conferences was exploding. The Association for the Advancement of Artificial Intelligence conference got more than 1000 submissions for the first time in 2014. For the 2020 conference, there were just shy of 10,000 submissions!

I was one of the reviewers for the 2020 conference. Like all the other reviewers, that meant I had to flag a set of 50 or so of those papers that I'd be well suited to review, so the chairs of the conference could assign each of us six to eight papers, and each paper three reviewers. I had to pick 50 papers out of 10,000. Even focusing in on the most relevant keywords, it was quite daunting. If only there were some way to get a computer to help identify the papers that were likely to be a good match for me, that might save a lot of effort. It's much more manageable if the computer can present me with 100 or so titles, and I mark half of those as topics I'm well informed about.

Identifying papers to review is one example of the more generic recommendation problem. We want to make programs that can sift through many, many items on behalf of a group of people to find items that each person should look at more closely. The items could be web pages or movies

or apps or news articles or restaurants. And, these recommendation systems are more than just labor-saving devices. They select what items users see most easily. As a result, recommendation systems color our perception of reality.

In general, the recommendation problem looks something like this. We have a person and an item and we want a numerical assessment of how well that person goes with that item. We'll call that number their compatibility. It could be 0/1 indicating that they go together. Or it could be a number indicating how much the person would enjoy the item. It could represent how likely the person would be to buy the item. Or, how much time the person would spend interacting with the item. If the computer can predict the compatibility score of a person and an item, however assessed, it can rank-order all the items for a person and present the most compatible ones for consideration.

Different recommendation problems differ based on what kind of information there is about the person and what kind of information there is about the item. In one version of the problem, each person and each item are associated with a feature vector. The recommendation system can then predict the degree to which the two are compatible by the similarity of those vectors. This version of the problem is unsupervised in that the recommender has no explicitly labeled data about compatibility. There's just an assumption that the more similar the features are between the person and item, the more compatible they are.

Some news-story recommendation systems work this way based purely on similarities. Each article is tagged by editors with keywords like "pop music" or "sports" or "local politics" that describe what they are about. And, the users select descriptors for themselves from this same set. Articles where the keywords match a user's descriptors are judged as compatible.

By contrast, in the supervised version, there are explicit compatibility judgments between people and items. That's the structure that's used in systems like online shops or video services that ask you to rate what you just bought or watched. The recommendation system then needs to generalize from these examples to identify additional compatible pairs of people and items, even though no compatibility judgment is available for these other pairs.

This version of the problem is supervised in that a sample of compatibility judgments is available for training the system.

LESSON 12 | RECOMMENDATIONS WITH THREE TYPES OF LEARNING TRANSCRIPT

A common variant of supervised recommendation is a recommendation system that can actively solicit compatibility judgments to gain new information for prediction. For example, it can recommend an item and then wait to find out how compatible the person and item actually are. This new information could be based on the person's direct judgment or an indirect measure, like whether the person takes the recommendation.

One difference between an active and passive version of supervised learning is that an active system might seek information by recommending an item that is not its best guess for the most compatible item. Instead, it might calculate that knowing a particular person-item compatibility value will better situate the algorithm for making future judgments more accurately. As a result, it can get up to speed more quickly for new items or new people.

The strategic, sequential nature of this decision makes the active version closely related to reinforcement learning. All three of these approaches to recommendation—unsupervised, supervised, and reinforcement learning—are behind popular user-facing information technologies and remain an important and active area of research. Let's delve into these three main classes of recommendation problems, starting with unsupervised approaches.

The big insight of the Toronto Paper Matching System is that the best information about a reviewer's expertise is the published work of that reviewer. The system asks each reviewer to provide pointers to a bunch of representative papers that the reviewer has authored. These papers are then processed to create a “bag of words” feature vector for the reviewer capturing that reviewer's expertise. The Toronto Paper Matching System represents a reviewer in the vector space model by pooling the words in the reviewer's representative published papers.

Of course, submitted papers can also be represented in this same space. There, we can compare the reviewer's vector and the paper's vector to get a pretty good indication of compatibility. The comparison can be done using the dot product or the normalized dot-product, the cosine, as these measures have been shown to be very effective at capturing a sense of overall similarity.

So, that's the main idea: the system recommends papers to a reviewer if those papers have high similarity to those written by the reviewer. Many conferences just use unsupervised rankings. That's already a helpful tool for focusing reviewers on the papers they are most likely to be able to review

knowledgeably. But the Toronto Paper Matching System incorporates supervised learning as well, since reviewers are allowed to add explicit compatibility judgments to the calculation.

To see how that works, let's start with a really easy case. Each person and each item are represented by vectors. For a large subset of people and items, we also have compatibility judgements. A combination of a person vector and an item vector have an associated compatibility. For a new pairing, we have the person vector and the item vector, and we need to predict compatibility. So, we have a supervised learning problem.

But there's a cool alternative that is possible when we're doing recommendations. Let's say we're making recommendations, and we already have many compatibility scores for each person and each item, although not for all possible pairings. The data is arranged differently, now. Each person is a row and each item is a column in a matrix. The cell of the matrix is filled in with the known compatibility score for the corresponding combination of person and item. If the compatibility score is unknown, then it's not filled in.

The pattern of compatibilities conveys a lot of information. In fact, there's so much information there that a recommendation system can make good recommendations even without using feature vectors to represent each person and item. Let me say that again. Given this kind of compatibility information, it is not necessary to come up with feature vectors. In a sense, the pattern of compatibility scores becomes a kind of feature representation on its own. Learning compatibility scores from this kind of input data is known in the field as collaborative filtering. Essentially, the preferences expressed by other users can help identify items that you are unlikely to want. The filtering out of unwanted items is being accomplished collaboratively.

One of the most influential and large-scale experiments on collaborative filtering was a competition known as the Netflix Prize, which took place from 2006 to 2009. Netflix wanted help to improve their existing recommendation system, Cinematch. They offered \$1million for the first team that could produce a 10% improvement over Cinematch's recommendations. The folks at Netflix already had a pretty good handle on machine learning, and it shows in the way they structured the competition. First, Netflix provided a training data set of over 100 million ratings given by about half a million users to around 18,000 movies. Compatibility scores were given on a 1 to 5 star scale. The title and year of release of each of the movies was provided separately and competitors had the option of looking up additional information about the movies.

LESSON 12 | RECOMMENDATIONS WITH THREE TYPES OF LEARNING TRANSCRIPT

The users were identified only by a numeric ID and Netflix tinkered a bit with their ratings to try to make it hard to figure out who users actually were. As it turns out, the attempt to conceal user identity was not very successful. An independent group showed they could identify some of the users by name based on their ratings and other public information. In the end, a privacy lawsuit prevented Netflix from running another public competition with user data.

Anyway, to evaluate a contestant's proposed recommender formula, Netflix created a list of 3 million user-movie combinations. Contestants needed to make a compatibility prediction on these combinations and send them to Netflix for scoring. The score on half of these queries was used by Netflix for their official internal standings. That's a testing set. The score on the other half were used to let competitors estimate how well they were doing. That's a validation set.

Netflix used root mean squared error for scoring. That is, the contestant provided a predicted compatibility score for a person-movie combination. Let's say the prediction was 3.8 stars. Netflix compared this value to the actual rating the person assigned, then squared the difference. If the person assigned a rating of 3, the penalty for that example would be 3.8 minus 3 or 0.8, and squared equals 0.64. If the person assigned a rating of 4, the penalty would only be 4 minus 3.8 squared equals 0.04.

So, Netflix chose a loss function that encouraged competitors to match the target number of stars as well as possible. They would compute this loss and send it back, in aggregate, to the contestant who submitted the predictions. Netflix did the accuracy calculations themselves instead of providing the testing data to the contestants. There needed to be a solid wall between training and testing to truly assess generalization. And, with \$1 million on the line, opportunities for cheating needed to be avoided.

It was a pretty elegant competition. Teams could download the data, play around with it as much as they wanted, and submit up to one new set of predictions each day in the form of a big text file. By the end of the first year of the competition, over 40,000 teams from 186 countries had registered. There were a few algorithms that teams were drawn to. One was a variant of a nearest neighbor algorithm. If you want to predict a compatibility score for a person, think of all of the ratings provided by that person as training data. The movie we want a prediction for is the testing data. But, to apply nearest neighbors, we need a way to judge the similarity of movies. How similar is movie i to movie j ?

Well, each column of the matrix is a vector where the components are compatibility scores. And, we have good ways of measuring vector similarity, like dot product or cosine. The tricky part is that the vectors have holes in them where compatibility scores are unknown. So, competitors just ignored components where the data was missing for either of the movies being compared. That ends up working pretty well. But, in the search for methods that work well for the problem, teams discovered that combinations of methods tended to outperform the individual methods, even the strongest ones.

The winner of the first Progress Prize, a year into the competition, achieved an 8.43% improvement using an ensemble of a whopping 107 algorithms. As final prize deadlines neared, teams that were not in the first place position approached each other to form supergroups so they could merge their algorithms. The final winner was an even bigger ensemble that pooled together hundreds of algorithms, where the predictions of individual algorithms were mixed together with weights trained by linear regression. Even minor factors, such as day of the week a movie was reviewed, and how many movies were reviewed all at once, turned out to give ensembles slight advantages over simpler methods.

In the end, the three-year race to the finish line was won by a 20-minute difference in submission time. That's like the winner of a footrace marathon being decided by nine-hundredths of a second. Very exciting! As it turns out, Netflix never implemented the prize-winning solution. Instead, they reported cherry-picking just two algorithms that together accounted for most of the improvement. One was a technique for reducing the dimensionality of the problem called Singular Value Decomposition. The other was an early precursor to deep neural networks called Restricted Boltzmann machines.

In any case, none of this work was designed to address another key problem that has to be solved to deploy these systems in practice. It's called the cold start problem. And, that problem brings us to our third approach to recommendation, one closely tied to reinforcement learning. The question is: How should a recommendation system balance making recommendations that it thinks are good for that person against exploratory recommendations aimed at discovering something important about the person's overall preferences? This problem comes up often outside of recommendation. It was first studied in the context of medical-treatment allocation. If you are testing two treatments for a disease, how can you learn which is more effective while giving good treatment along the way?

LESSON 12 | RECOMMENDATIONS WITH THREE TYPES OF LEARNING TRANSCRIPT

This problem is sometimes compared to one faced by a gambler choosing amongst slot machines. Slot machines have the nickname “one-armed bandits” because they have a single arm that you have to pull, and they steal your money. I saw a literal take on this name at the San Francisco airport. It is a slot machine made to look like a one-armed bandit.

Anyway, our third method for making recommendations is like a gambler playing multiple slot machines and switching away from those that feel like losers. The problem is called the k -armed bandit, where each arm is a different slot machine the gambler can choose. For our purposes, the arms correspond to items that can be recommended.

Here’s an example: When Barack Obama was first running for president in 2007, the staff for his website wanted visitors to sign up for a mailing list and donate money to the campaign. His team thought that the design of the webpage might have an impact, so they decided to do an experiment. The experiment involved showing different visitors to the page different combinations of a splashy visual and the message on the sign-up button. They were interested in testing four buttons and six visuals, for a total of 24 combinations. In their experiment, each visitor was randomly assigned to a combination, and they showed 13,000 visitors each combination. So, over 300,000 visitors were in the training set, and their click behavior was the feedback. The best performing combination was a picture of the Obama family along with “Learn more” on the sign up button.

The staff estimated that the use of this combination, instead of the one they were planning to use, brought in an additional 3 million sign ups and an additional \$60 million. That’s like winning 60 Netflix prizes all at once. This problem setting is commonly called A/B testing because it is often applied to select among two choices. Here, we’re trying to figure out which of the 24 combinations we should recommend to the average visitor to the site. We can view it as a kind of recommendation problem, in the form of a 24-armed bandit. We’re filling in the compatibility scores by showing the layouts to people and seeing whether they sign up.

In this example, we wait until each layout is seen 13,000 times before deciding to commit to the best combination. This approach is simple but it has an important downside. Even if one of the items is clearly worse than the others, this approach still recommends it many times while gathering data. During the learning process, the algorithm doesn’t focus in on the most promising item. We say it only “explores” and never “exploits.” It’s not terrible, but it seems wasteful.

At the opposite end of the spectrum, consider an approach that shows people configurations at random until someone signs up. At that point, it switches to the configuration that led to success. This approach explores only until a single person signs up, and then only exploits. That might work out okay. The algorithm could get lucky, and the first configuration to work might actually be the best one. But there's a good chance it'll latch onto something suboptimal. Assuming the first positive result is the best positive possible result is risky.

So, it can work out a lot better if we balance exploration and exploitation. One simple approach to mixing exploration and exploitation is to switch off between these two modes probabilistically. Each time a decision is made, this approach says: With high probability, be greedy and select the arm that appears to be the best. That is, exploit. But, with low probability, choose an arm at random, explore. The small probability value is referred to by the Greek letter *epsilon*, which has been used since the early 1800s to refer to small values. This *epsilon*-greedy approach is popular because it can be easily adapted to many different circumstances.

However, the *epsilon*-greedy approach doesn't really take into consideration how confident we are that one item is truly better than another. In fact, it ends up over-exploiting early on when it has little reason to have a preference and then over-explores later on after it has built up accurate estimates.

A mixed method that's better at incorporating how uncertainty declines is called the upper confidence bound approach. It splits its estimate of compatibility into two parts—the average compatibility observed and a confidence interval around this estimate. Then, the algorithm greedily selects the item with the largest estimate plus confidence interval. The approach explores when estimates are uncertain and exploits when the item with the best compatibility exceeds the upper bound of the uncertainty of the next highest item's compatibility.

To illustrate this idea, imagine we're comparing three layouts. Early in training, the compatibility of the first layout has been estimated to be exactly 7. The second layout is estimated to be anything from 1 to 9, with an average of 5, marked with a blue box. The third layout is estimated to be between 4 and 8, with an average of 6. At this moment, an *epsilon*-greedy approach would select the first layout with high probability, because the most likely value for the first layout is larger than the most likely value of the other two layouts. But the upper confidence bound approach would

LESSON 12 | RECOMMENDATIONS WITH THREE TYPES OF LEARNING TRANSCRIPT

select the second layout. This approach is looking at the upper values of each, and there's still a chance that compatibility for the second layout could be 9, higher than either of the other two.

Late in training, it becomes clear that the second layout's compatibility is actually between 4 and 6. The upper confidence bound algorithm will switch to the third layout and, in this case, discover that its compatibility lies at the upper end of its range, making it the best choice. The approach can now exploit by choosing the third layout, even though it's still uncertain about the precise compatibility of the second layout.

Let's think about how we can apply what we've seen to another recommendation problem. We're designing a wakeup service that will show a user a news story on a screen in the morning. Each day, there are five news stories that could be featured. The user is shown one and can click *like* or *dislike* depending on whether the story was worth seeing. Each news story can be represented as a "bag of words" feature vector. News stories never repeat, so this recommendation system will have to generalize to be effective.

In the machine-learning community, this problem is known as a contextual bandit problem because it must make its judgments based on vectors that describe the context of the people or items, or both. Let's try out some algorithms for this problem. For starters, I made some data consisting of a few thousand titles of academic papers. The data set has one paper title per line. Half of the papers are machine-learning papers from the ICML conference. The other half are robotics papers from the IROS conference. I mixed them all together and then unfairly tagged all machine-learning papers as *interesting* 1, and all of the robotics papers as *not interesting* 0.

We'll read in the file cb.txt consisting of the titles of the papers, and we'll read in the file vocab2.txt listing the words in those titles. I wrote some auxiliary code to read in the text files containing all the titles: readvocab reads the vocabulary file and assigns a numeric identifier to each word type; tokenize turns the titles into numeric vectors based on the words they contain; getdat reads in the collection of titles cb.txt. It uses tokenize to convert the titles into numeric vectors suitable for machine-learning training.

Now that we've read in the data, we're going to simulate 200 days of interactions. We'll call it incremental recommendation simulation. It's like a game. Each day, the chooser sees five titles, and selects only one of these items to show to the person. Then, the person provides feedback, a zero

or one. One means the person found the item interesting. If the chooser chooses an article the person found interesting, one point is added to the score.

After that feedback is provided, the title and the judgement are added to a training set. If the chooser just guesses at random every day, the score will be around 100. If the chooser chooses right every day, it would get a score of 200. But, initially, the chooser has no idea what the person likes. It must do some amount of exploration before it's able to start choosing correctly.

Now, we can evaluate different learning approaches for the iterative recommendation simulation. As a first strategy, we'll ignore exploration. We'll focus on exploitation by training a Naive Bayes classifier to assess each of the five incoming titles, and it will choose the most promising one to show to the person. We'll have the chooser guess randomly for alpha days, then it will use the gathered examples to train the classifier. For the remaining days, it will use the classifier to decide what to choose.

Let's see what happens with a range of alphas from 10 days to just under 200 days, in steps of five days. For alpha equals 195, performance is indistinguishable from random, because the chooser spends almost all 200 days collecting data. It's exploring and learning a really good classifier, but it scarcely ever uses what it knows. For low values of alpha like 25 or fewer, the chooser collects some data, then it uses that data to make decisions. The score isn't too bad. But, there's a kind of happy zone of alpha from maybe 25 to 75 days where the tradeoff between exploration and exploitation is even better. The strategy scores around 150 to 160 in this zone.

So far, we've been training once and using the results throughout. But we can get even more improvement if we retrain with every new example that comes in. To wisely trade off exploration and exploitation, we need a way to tell how confident the classifier is in its assessments. The Naive Bayes algorithm has a built-in parameter that can help. When a Naive Bayes model learns, it tallies up counts on each feature for each of the classes. In principle, this initial count should be zero. But, when there is only a small amount of data, the model can end up overfitting and behaving erratically.

So, we can use a non-zero value to start the count. This approach is known as Laplace smoothing, and it makes the algorithm more robust when trained with small amounts of data. Setting the smoothing parameter to a very large value means that the algorithm needs to see a lot of examples before it can distinguish the different classes. In effect, a large smoothing parameter

LESSON 12 | RECOMMENDATIONS WITH THREE TYPES OF LEARNING TRANSCRIPT

forces more exploration. Conversely, using a smaller smoothing value lets the algorithm be more reactive to the data it has seen, resulting in much faster exploitation.

We will retrain our Naive Bayes classifier after every interaction, but we'll experiment with different values of the smoothing parameter, also named alpha. The chosen item returned is just the one our classifier thinks is most likely to be interesting. The smoothing parameter values I tested ranged from 0.00005 to 1.0. So, let's plot the smoothing parameter against the score to see what values lead to the best outcomes.

The score of the strategy for a maximum smoothing value of 1 is around 159. That's where performance topped out with the explore-then-exploit method shown earlier. However, bringing the smoothing parameter down to 0.1 or 0.2 results in scores improving from 159 to around 164. That corresponds to five additional days satisfied out of the 200.

With even lower values of the smoothing parameter, the strategy tends to under explore, exploiting too early and lowering its score as a result. Adopting a smart strategy and the right amount of exploration can significantly improve the strength of recommendations a system can make.

Learning-based recommendation systems have been a lifesaver to people who would otherwise be swamped by information. The case of recommending papers to reviewers is an example where the incentives are mostly aligned—authors want their papers to be read by reviewers who are knowledgeable and interested, while reviewers want to read interesting papers.

But, machine-learning systems are deployed to recommend all sorts of items to people, and sometimes providers face strong incentives to “game the system” to get their items recommended. On an ad-supported video delivery platform, the situation is more like the Wild West. Video producers get paid for views, even if the viewers are unhappy, or even if they are harmed in the long run. Big tech companies like Google would prefer to limit incentives for unwanted behavior that threatens the value or reputation of their platforms. But, it's an arms race.

In 2019, I attended a workshop at Google to talk about next-generation recommendation systems. The academic attendees were brimming with exciting ideas to try. But the battle-weary Google engineers' attitudes were much more grim. They pointed out how most of the new ideas being discussed are themselves vulnerable to manipulation and are therefore non-starters.

It's not easy to make systems that learn and improve but at the same time avoid latching on to misleading inputs from unscrupulous participants. One cool suggestion that came out of the workshop was the idea of moving away from implicit measures of user interest like clicks or viewing time toward systems that let the users explain why they like or do not like particular content.

I hope future recommendation systems will incorporate options for users to explain their preferences so the systems can tune their behavior accordingly. For example, some existing sites let people write reviews. What if the recommender systems read the reviews to understand the justifications for a user's rating and then recommended items based directly on justifications? Justification-based recommendation could distinguish between providers who are gaming the system and those who are producing items that people actually want. That could make the internet healthier for all of us.

LESSON 13

GAMES WITH REINFORCEMENT LEARNING

When Arthur Samuel popularized the term *machine learning* in 1959, he was describing a method that let a checker-playing program he created improve by playing against itself. Since then, machine learning programs have been designed that learn to play a variety of high-profile games well. Reinforcement learning is the problem of learning behavior rules from evaluative feedback like what you get for winning or losing a game.

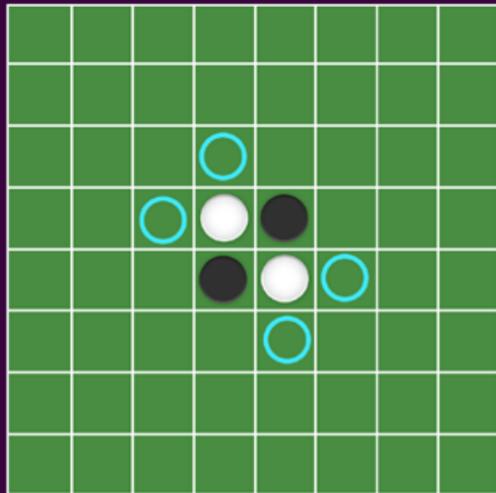
Othello

Like checkers, chess, or Go, Othello is played on a grid-shaped board in full view of both players. They take turns placing tokens on the board. Tokens can change allegiance during the game, and the player who controls the most tokens at the end wins.

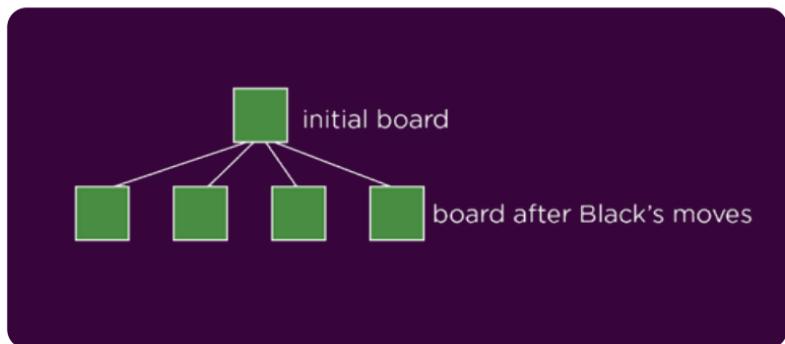
We can visualize game play in Othello and many similar games in the form of a tree. Trees come up all over the place in computer science, like in decision trees.

The root of this game tree is the initial board. We can refer to board positions generically as the game state. We can build up this game tree with game states iteratively. Because it represents all the possible ways the game can be played, we can call it the game tree.

Othello games always start with two black pieces and two white pieces on the board. The only moves that are legal involve surrounding at least one of the opponent's pieces from opposite sides, after which the surrounded piece is flipped to the opposite color. From the initial game state, there are exactly four legal moves that Black can make.



On our game tree, we can draw a line for each of Black's moves that take us to a new game state.



There are 64 squares on the board, and even if players only have two choices available on each turn, we're still talking about a game tree with 10 quintillion leaves. That's just too big.

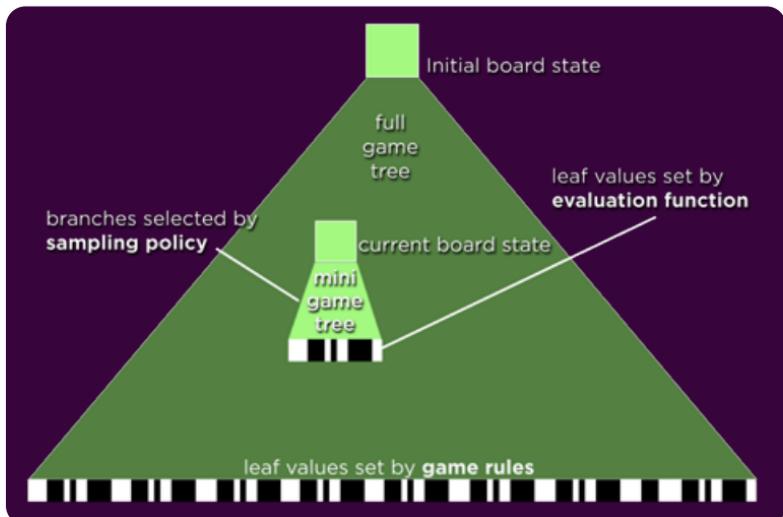
But despair not. There is a general recipe for playing games like Othello that has turned out to be super effective. Each time it's our turn, we're going to run an algorithm for selecting a good move from the current board.

To do its job, the algorithm needs two things:

1. an approximation of the optimal labeling of game states as win/draw/loss (+1/0/-1), called the evaluation function; and
2. a rule for choosing which moves we want the program to consider in its search for a good move, called the sampling policy.

If the big triangle is the full game tree that's too big to search for Othello, our evaluation function and sampling policy let us create a miniature version of the game tree for evaluating a game state.

We use the sampling policy to trace out a piece of the game tree. We use the evaluation function to assign values to the leaves of that mini tree. Then, we use rules to propagate the leaf values up to the root of the mini tree so that an action can be chosen. And we'll make our move in the game based on the values that we get from this calculation.



Starting in around 1993, Michael Buro created an Othello-playing program that followed this game-tree approach. Earlier efforts used human-designed evaluation functions. A key innovation in Buro’s program was that he used a machine learning approach to create the evaluation function.

Specifically, Buro defined a set of Othello-relevant features of boards. They included information about how many available moves there are, who has more tokens, whether the player controls important parts of the board (such as corners), etc. He collected expert games and also simulated lots of examples of game states along with whether those game states led to a win/loss/draw for Black. He had around 3 million game states, which was a lot of data in the 1990s.

He then trained a classifier to predict the likelihood that Black would win. He used logistic regression to capture this mapping. He named his program Logistello, a combination of *logistic regression* and *Othello*. After several years of development and refinement, Logistello became the first program to beat the reigning human champion in Othello.*

* The match took place just three months after the much more widely publicized Deep Blue/Garry Kasparov match, in which a computer beat the reigning human champion in chess. In fact, Deep Blue used the same game-tree strategy template, but its evaluation function was built by people, instead using machine learning.

Backgammon

In the generic game-tree approach, there are a few ways machine learning can contribute.

Logistello took a supervised approach to learning the evaluation function. Specifically, in its training data, it used pairings of game states and their final game outcomes.

In backgammon, IBM machine learning researcher Gerald Tesauro tried an even more interesting machine learning approach. Instead of gathering expert training data and training a learner to mimic experts, he framed backgammon as a reinforcement-learning problem.

He had his program play against itself. It would make moves using its current estimate of the evaluation function and then update its predictions using an approach to reinforcement learning called **temporal difference learning**.

The supervised approach says we need to pair a game state with its outcome given optimal play. The temporal difference version of learning says we don't know optimal play, but we can use this insight: The evaluation of a game state where the player has a choice is the maximum of the evaluation of the game states that can be reached in one move.

We might not yet have a good estimate for those game states, but we're going to keep doing updates to make game state values more consistent from one step of the game to the next. We don't need expert examples; we just need lots of chances to make things match up.

This so-called TD-Gammon approach ends up playing very strong backgammon. Tesauro's results brought reinforcement learning and temporal difference learning to many people's attention in 1992.

Video Games

In 2015, the company DeepMind announced that they had created a reinforcement-learning algorithm that could learn to play any of a suite of Atari video games. By some measures, the program's performance was on par with a human video-game expert.

One of several things that makes this result so exciting—so different from earlier results like what was achieved in Othello—is that video games don't have a well-defined set of rules the way board games do.

Part of the fun of many video games is learning how things work and learning how to control the characters so that they do what you want. And when the rules are not known in advance, the whole game-tree idea goes out the window.

Whereas an Othello player has to learn about how to sequence actions to bring about a winning position, a video-game player also has to learn how things work.

The DeepMind group adapted a reinforcement-learning algorithm called **Q-learning** to solve this problem. Q-learning is a variant of temporal difference learning.

The main innovation of Q-learning is that values are learned for the moves in the game tree instead of for game states.

Game-tree search requires that we can enumerate the game states reachable from the current game state. Learning moves avoids these difficulties.

In fact, if we have learned values for all the moves, then move selection is easy: Just take the move with the highest value.

Since the identity of the resulting game state is not needed, this approach can be used just as easily in video games as it can be in board games.

In addition to Atari games, variants of this approach have been used to play at the level of top human players on big-name multiplayer online battle games such as *Defense of the Ancients 2* and real-time strategy games such as *StarCraft II*.

Try It Yourself

Follow along with the video lesson via the Python code:

[L13.ipynb](#)

Python Libraries Used:

graphviz: Graph visualization.

gym: Reinforcement-learning test environments in OpenAI Gym.

sklearn.tree: Scikit-learn's decision tree algorithms.

Key Terms

decision tree regressor: A decision tree classifier that outputs a number instead of a class.

Q-learning: A variant of temporal difference learning where values are learned for state-action pairs instead of states alone.

temporal difference learning: A learning rule for sequential prediction tasks that attempts to minimize the difference between consecutive predictions.

READING

Kohs, Greg, dir., *AlphaGo*.

Mitchell, *Machine Learning*, chap. 13.

Mnih, et al., “Human-Level Control through Deep Reinforcement Learning.”

Russell and Norvig, *Artificial Intelligence*, chaps. 5 and 22 and sec. 21.8.3.

Sutton and Barto, *Reinforcement Learning*.

QUESTIONS

1. What are the two main functions a game-tree approach needs, and how can learning help with each of them?
2. What makes the standard game-tree approach to learning a strategy inapplicable to games like poker and Stratego?
3. What are some of the things that make particular games hard to learn? Let's modify the blackjack learner from the lesson to play a different game in the OpenAI Gym collection, Roulette. Roulette has 37 actions—0 is a guaranteed loss and 37 (“just walk away”) is a guaranteed tie. The other actions have expected values that are in between. What does the decision-tree reinforcement learner do with this problem?

Answers on page 482

Games with Reinforcement Learning

Lesson 13 Transcript

How can a machine improve its strategy for playing a game? When Arthur Samuels popularized the term machine learning back in 1959, he was describing a method that let a checker-playing program he created improve by playing against itself. Since then, machine learning programs have been designed that learn to play well a variety of high-profile games. Reinforcement learning is the problem of learning behavior rules from evaluative feedback like what you get for winning or losing a game.

The game of Othello is a great place to dive in. It's a famously easy game to learn to play, but the strategy space is rich and interesting. Like checkers, chess, or Go, Othello is played on a grid-shaped board in full view of both players. They take turns placing tokens on the board. Tokens can change allegiance during the game and the player who controls the most tokens at the end wins.

We can visualize game play in Othello and many similar games in the form of a tree. Trees come up all over the place in computer science, like in decision trees. The root of this game tree is the initial board. We can refer to board positions generically as the game state. We'll build up this game tree with game states iteratively. Because it represents all the possible ways the game can be played, we call it the game tree.

Othello games always start with two Black and two White on the board. The only moves that are legal involve surrounding at least one of the opponent's pieces from opposite sides, after which the surrounded piece is flipped to the opposite color. From the initial game state, there are exactly four legal moves that Black can make. On our game tree, we can draw a line for each of Black's moves that take us to a new game state. Because of symmetries in the board, each of the four possible game states provides exactly three legal moves for White.

In the form of a game tree, we have something that looks like this. The set of boards and the paths to reach them are expanding. Now, for the first time in the game, symmetry is broken. Different choices for Black's second move lead to game states with different numbers of options for White. In some game states, Black has five possible moves. In others, there are only four possibilities for Black.

TRANSCRIPT

Here's what the game tree looks like at this stage. Notice that the game tree is starting to get crowded. And, it's only going to get worse. There are 64 squares on the board and even if players only have two choices available on each turn, we're still talking about a game tree with 10 quintillion leaves. But, let's ignore that little bit of intractability for just a moment. Every game of Othello ends, no matter how the players play. In addition, every final game state can be categorized as a win, loss, or draw for the starting player. So, if we could magically reach the very bottom of the game tree, we'd be able to label leaves as a win, loss, or tie for Black.

Now, consider a game state of the game tree where it's Black's turn to move. Let's further imagine that we've already determined for all of the children of that game state whether they are a win for Black, or White, or a draw.

Here's what the situation looks like. The green square is a game state where Black has 4 different moves. The first leads to a win for Black, the second and fourth lead to wins for White, and the third leads to a draw. What can we say about the green game state? Since it's Black's turn and Black wants to win, we can assert that Black will choose the first action, leading to a game state where Black wins the game.

So, it's not much of a stretch to say that the green game state is one that Black will win. Let's color it black to reflect that idea. That's really useful. It means we can propagate up the game tree information about whether each game state is a win, loss, or tie. In fact, we can color all the game states of the game tree, as long as we know how to propagate information for all possibilities.

So far, we figured out that, if a position where Black moves can reach a position that is a win for Black, well, then that position is also a win for Black. But what if a position can only reach draws and wins for White? Well, if there's a way Black can draw, that's preferable to losing. So, Black should choose to win or else draw. Otherwise Black has no choice but to lose.

So, here's something clever we can do. Instead of making rules with colors, let's use numbers. We assign a win for Black a positive-1 value, a draw a 0 value, and a win for White a negative-1 value. That means we can express our color-propagation rule as a simple mathematical formula. The value of a game state where Black moves is the max of the values of its children.

That rule covers all possible cases. It also makes it very simple to specify what happens in a game state where White moves. There, the value of the game state is the min of the values of its children. Since the rules of the game tell us how to label the leaves of the tree with either plus-1, 0 or, minus-1, and we can calculate mins and maxes up the game tree as needed, we can—in principle!—label all of the game states in the game tree with these values.

Why is that useful? Because it encodes the optimal strategy for Black; when it's your turn, take whichever move leads to the highest valued game state. Simple as that. Alas, we can't really build a program like that to play Othello. There are smaller, simpler games, like Tic Tac Toe and Nim, where this approach works perfectly. But, for Othello, the game tree is just too big.

But, despair not. There is a general recipe for playing games like Othello that has turned out to be super effective.

What we're going to do is this: each time it's our turn, we're going to run an algorithm for selecting a good move from the current board. To do its job, the algorithm needs two things. One is an approximation of the optimal labeling of game states as win/draw/loss, that is, plus-1/0/minus-1. We'll call this program the evaluation function. Second is a rule for choosing which moves we want the program to consider in its search for a good move. We'll call that our sampling policy.

If the big dark green triangle is the full game tree that's too-big-to-search for Othello, our evaluation function and sampling policy let us create a miniature version of the game tree for evaluating a game state. We use the sampling policy to trace out a piece of the game tree. We use the evaluation function to assign values to the leaves of that mini tree. Then, we use the rules we have been discussing to propagate the leaf values up to the root of the mini-tree so an action can be chosen. We'll make our move in the game based on the values that we get from this calculation.

Okay, let's apply this idea to Othello. For simplicity, we'll use a sampling policy that chooses all possible moves up to a depth of a dozen or so. But, how will we evaluate the game states we reach at that point? Starting in 1993 or so, Michael Buro created an Othello-playing program that followed this game-tree approach. Earlier efforts used human-designed evaluation functions. A key innovation in Buro's program was that he used a machine learning approach to create the evaluation function. Specifically, Buro defined a set of Othello-relevant features of boards.

TRANSCRIPT

They included information about how many available moves there are, who has more tokens, whether the player controls important parts of the board like corners, etc. He collected expert games and also simulated lots of examples of game states along with whether those game states led to a win/loss/or draw for Black. He had around 3 million game states, which was a lot of data back in the 1990s.

He then trained a classifier to predict the likelihood that Black would win. He used our friend logistic regression to capture this mapping. He named his program Logistello, a combination of logistic regression and Othello. After several years of development and refinement, Logistello became the first program to beat the reigning human champion in Othello. The match took place just three months after the much more widely publicized Deep Blue / Kasparov match in which a computer beat the reigning human champion in chess.

In fact, Deep Blue used the same game-tree strategy template we have been talking about. Deep Blue's sampling policy went about 21 moves down the tree, considering all possible moves at each level. Unlike the less-famous Othello example, Deep Blue's evaluation function was built by people; no machine learning there. But, even so, machine learning did play a pivotal role in Deep Blue's eventual success.

Here's how. Gerry Tesauro is a machine-learning researcher at IBM. He worked on Deep Blue and developed a novel machine learning technique for learning move preferences from human examples. His technique was used in Deep Blue in the narrow area of king-safety evaluation. Kasparov and Deep Blue had their first match in 1996, which Kasparov had won. But, during a re-match in 1997, this part of the evaluation function made a critical difference in one important game state in Game 2. It assigned a higher value to the game state than a typical chess program would have, resulting in a nuanced move that surprised Kasparov. This surprise may well have contributed to "psyching out" Kasparov and changing the momentum of the match in favor of the computer. Kasparov lost that game, managed three draws, then lost the final game. Sorry, humanity.

In the generic game-tree approach I described, there are a few ways machine learning can contribute. Logistello took a supervised approach to learning the evaluation function. Specifically, in its training data, it used pairs of game states and their final game outcomes. In backgammon, Gerry Tesauro tried an even more interesting machine learning approach.

Instead of gathering expert training data and training a learner to mimic experts, he framed backgammon as a reinforcement learning problem. He had his program play against itself. It would make moves using its current estimate of the evaluation function, then it would update its predictions using an approach to reinforcement learning called temporal difference learning.

The supervised approach says we need to pair a game state with its outcome given optimal play. The temporal difference version of learning says we don't know optimal play, but we can use the insight we discussed earlier. Specifically, the evaluation of a game state where the player has a choice is the maximum of the evaluations of the game states that can be reached in one move. We might not yet have a good estimate for those game states, but we're going to keep doing updates and make the game tree values more consistent from one step of the game to the next. We don't need expert examples; we just need lots of chances to make things match up. This TD-Gammon approach ends up playing very strong backgammon. Tesauro's results brought reinforcement learning and temporal difference learning to many people's attention back in 1992.

The game of poker is quite different. Poker players make decisions under incomplete information, not knowing what their opponents see in their hands. You can build a game tree for poker, but you can't propagate values up the tree the way we discussed for Othello. The reason is that there are game states in the game tree where a player has the same information and must therefore behave identically.

Here's a tiny example to give you the idea. In the movie *The Princess Bride*, two characters engage in a deadly battle of wits. One, the Man in Black, adds poison to one of two goblets of wine secretly. He then places one of the goblets in front of his opponent and the other in front of himself. The opponent, Vizzini, must then choose one of the goblets. At that point, the two men drink and one dies. It's a comedy.

Here's the game-tree representation of this game. The Man in Black chooses first and either poisons his own goblet or Vizzini's. Then, Vizzini chooses to either drink his own goblet or the one in front of the Man in Black. If we use the labeling rules described earlier to propagate information up the tree, we find that the game is a win for Vizzini. In particular, Vizzini wins by drinking from whichever goblet has wine and not poison. Simple, right?

LESSON 13 | GAMES WITH REINFORCEMENT LEARNING TRANSCRIPT

Not remotely! (Ahem. Sorry.) I mean, no. That's because the game tree fails to record an important part of the game, which is that the Man in Black's decision was made privately. Vizzini can't use this information to decide which action to take. He must commit to taking the same action in both of those two game tree states. There's no clear way to constrain the choices using the standard game-tree algorithm.

Marty Zinkevich and colleagues came up with an approach that can learn to play quite well in games like this one. The basic idea is that there's a probabilistic strategy for each of the sets that contain game states that players can't distinguish due to partial information. So, for this game, we'd have a set of choices for the Man in Black at the top node and a set of choices for Vizzini for the pair of middle nodes. Here's where machine learning comes in. The players can compute what's called counterfactual regret. Roughly speaking, it's a measure of how disappointed they are for not playing each of their possible moves in each possible circumstance. A player can simulate playing the game repeatedly in his mind. A regret minimization algorithm chooses actions proportional to their regret. The more you wish you had done something, the more likely you'll be to do it next time.

If we update the strategies and repeat the process, the average strategies of the two players will be regret minimizing. That is, they will find a way to behave that is the best possible, considering that the opponent is trying to do the same. Strategies for the two players such that neither can unilaterally improve his chance of winning is what economists call a Nash equilibrium after mathematician John Nash.

The idea of counterfactual regret minimization can be thought of as learning from imagined self-play. And, it is at the heart of several breakthroughs in computer poker play. In 2017, a team from the University of Alberta used counterfactual regret minimization to derive a near perfect solution to a popular two-player version of poker called Heads-up limit Texas Hold'em.

In 2019, the system Pluribus from Carnegie Mellon and Facebook used a version of counterfactual regret minimization to address the tournament version of the game: Texas Hold'em with no limit and six players. In an exhibition match with 12 poker professionals, Pluribus achieved a decisive victory. But, even in poker, or blackjack, we at least know what the rules are. In 2015, DeepMind announced that they had created a reinforcement-

learning algorithm that could learn to play any of a suite of Atari video games. By some measures, the program's performance was on par with a human video-game expert.

One of several things that makes this result so exciting, so different from earlier results like what was achieved in chess and Othello, is that video games don't have a well-defined set of rules the way board games do. Part of the fun of many video games is learning how things work and learning how to control the characters so they do what you want. And, when the rules are not known in advance, the whole game-tree idea goes out the window. Whereas an Othello player has to learn about how to sequence actions to bring about a winning position, a video-game player also has to learn how things work.

The DeepMind group adapted a reinforcement-learning algorithm called Q-learning to solve this problem. Q-learning is a variant of temporal difference learning. DeepMind called their full algorithm deep Q-network, or DQN, because it combines Q-learning with deep networks. The main innovation of Q-learning is that values are learned for the moves in the game tree instead of for game states. That makes a huge difference.

Game-tree search requires that we can enumerate the game states reachable from the current game state s . Learning moves avoids these difficulties. In fact, if we have learned values for all the moves, then move selection is easy. Just take the move with the highest value. Since the identity of the resulting game state is not needed, this approach can be used just as easily in video games as it can be in board games. In addition to Atari games, variants of this approach have been used to play at the level of top human players on big-name multiplayer online battle games such as *Defense of the Ancients 2*, known as *DOTA 2*, and real-time strategy games such as *Starcraft 2*.

The deep learning methods that have been used in these games require hundreds or thousands of powerful computers to reach expert performance. But we can use a similar approach to learn to play a simple but real-world game—blackjack. The basic idea of the Q-learning approach is as follows: 1.) Let $Q(S, M)$ be a function that maps a game state S and an action M to a value. The value represents the probability we'll win from game state S if we take move M and then take optimal moves after that.

TRANSCRIPT

These values can be initially random; 2.) Then, each time the learner sees that game state S leads to game state S-prime when move M is taken, it uses its estimate of the value of the best move to take from S-prime to update $Q(S,M)$.

We're going to use a decision tree to represent Q and we'll retrain it periodically as more data comes in. Let's implement this idea for blackjack. Blackjack, and a bunch of other reinforcement-learning environments, are defined in a package called gym from OpenAI. You get access to these environments by simply telling Python to import gym and running gym. make on one of the environments.

We're going to start with an evaluation function, initially all zeros. Then, for each of five epochs, we're going to play N equals 100,000 games of blackjack in each epoch. At the end of each epoch, we'll retrain our evaluation function. Epochs and N are free parameters, and their choices matter. Playing fewer games per epoch means we'll more quickly proceed to using the new experience we've gained. I picked 100,000 to make sure rare but important events would be in the data. But smaller numbers would perform well here, too.

Another important parameter is epsilon. At each move in the game, we'll take whichever action has the highest Q value. But, just to make sure we get a chance to see how other moves work, we'll choose a random move epsilon equals 10% of the time. Epsilon strikes a tradeoff between exploring and exploiting, just like we saw in Lesson 12 about recommendation.

In each epoch, we'll gather up a set of training examples consisting of the current game state in dat and an improved estimate of its value from the Q -update equation in lab. We'll also count the number of games we win, just to see if we're making progress. For each of N games, we start off with an observation of the initial game state. In blackjack, that comes from dealing one card to the dealer and two to the player.

Until the game is over, we ask our learner to pick an action. We add the current game state and the current action to our training data. Then, we take a step in the environment following the selected action. In blackjack, there are two actions—hit and stay. What's our estimate of the value in the game state we just left under the influence of the selected action? Well, if the game has ended, we can use the value of the final board as the target. Like, if we are holding a 20 and we stand pat, then the dealer busts, we win. The combination of 20 and stay should produce a positive-1.

If the game hasn't ended, we need a special case if it's the first time that we're playing. We'll just guess zero for those games. But, if we've played and learned before, we can use the learned evaluation function. We ask our classifier what it predicts from the current board combined with both of the two possible actions. We would generally take whichever has the higher value, so the value is the max of these two predictions. We keep track of the number of times the game ends in a win, and after all the games in this epoch are over, we train our predictor with the newly collected data and do it all over again.

The learner is a decision-tree regressor. It's just like the decision tree classifier we've used before, but it outputs a number instead of a class. The number, in this case, is the prediction of the game outcome, something between minus-1 and 1. Here, we bound the number of leaf nodes at 6. That's roughly the largest size possible before I found the trees too hard to read. Bigger trees allow for more precision in representing values and I'd recommend using a number like 20 or 50 if you want to get a really good strategy.

The big piece we haven't talked about yet is how the player should choose actions while learning. If we haven't trained our predictor, we can't use it to look ahead. In this situation, we'll just take a random action. Otherwise, we can ask for the value for each of the two actions in the current board. If action 1 leads to a higher prediction than action 0, we'll choose action 1. Epsilon fraction of the time, we choose a random action just to see what it does.

With the code cued up and ready to go, let's see what happens. First of all, we can see that the first round leads to about 28% wins. That's the win rate in this version of blackjack for a random player. But the win rate jumps up to 41% in the very next round. It falls off a bit to 38% for the subsequent three rounds. In fact, that result is pretty consistent.

Since we're learning a decision tree, we can peer inside to see what kinds of predictions it learned. The tree first tests if the player has 18 or less. If so, it checks if the dealer is showing an ace. If so, it predicts a highly likely loss: a value of negative-0.719. If the dealer doesn't have an ace, it still predicts a loss, but only negative-0.211. On the other hand, if the dealer has 8 or more, it predicts an intermediate level of loss: negative-0.513. Interestingly,

LESSON 13 | GAMES WITH REINFORCEMENT LEARNING

TRANSCRIPT

these values don't depend on the action the player selected, which isn't really a perfect prediction. But, maybe it's not terrible. If you have 18 or less, you are probably going to lose.

On the right side of the tree are the predictions if the player is holding 19, 20, or 21. In that case, it matters what action the player takes. If it's action 1—hit—the player will likely lose. The predicted value is negative-0.537. But, if the player stays, the player will likely win with a 19. The prediction is 0.263. But, if the player has 20 or 21, winning is even more probable. The prediction is 0.674. I don't think you can do any better than 41% in this game. Interestingly, that's what the algorithm found in epoch 2. I think the reason the algorithm found that solution so quickly is that blackjack is a relatively shallow game. There are not many decisions per game so a little learning is enough. Other games would require a lot more epochs.

Why does the performance get worse after the first epoch? I'm not quite sure. It could have to do with the fact that the decision tree is only approximately accurate due to the limit on the number of leaves. Allowing bigger trees does help the system learn better policies.

Earlier, we talked about learning in game-tree search and pointed out two places learning can help. The first is in the prediction of good actions to pursue. The second is in making good value predictions. Research has found that, in some games, a good sampling policy is easier to learn and, in others, a good evaluation function is easier to learn. In games like chess, evaluation functions work well. The number and identity of the pieces left on the board tell you a lot about who is winning. As a result, there's no real reason to limit the actions to evaluate. When making decisions, typical algorithms in these games search wide and shallow. That is, they consider all possible actions, but use their evaluation functions to deal with the fact that we can't search widely and still reach the end of the game.

For many years, this approach was the only game in town. But the community later realized that there is a classic board game that just isn't amenable to this approach. The game is Go and it seemed as though it just isn't possible to learn an evaluation function for Go that's good enough to be used in this kind of search. In 2006, machine-learning researchers showed an alternative with a great deal of promise in Go. The idea was to search narrow and deep. That is, it's better to evaluate a board in Go by searching to the end of the game, where the outcome can be computed accurately.

But it matters less which actions you choose to get there. In fact, even random actions can be quite informative, guided by a kind of bandit-based search that balances exploration for learning and exploitation for focusing on promising lines of play. Maybe random actions work for Go because the pieces don't move around. And, unlike in Othello, captured pieces are removed from the board, not flipped to the opposite color. So, in Go, it's possible to assess what territory is controlled now by looking to see what happens later.

In any case, searching narrow and deep led to rapid progress in Go. By 2016, DeepMind showed they could combine methods for learning a good sampling policy and learning a good evaluation function. For the sampling policy, they trained on human expert games and also on the results of their own program's choices. For evaluation functions, they used a reinforcement-learning approach. They called their system AlphaGo. It was the first program to beat a human champion in Go, and a later version beat the number-one ranked Go player in 2017.

That same year, they provided a system that could play exceptionally well even without first training on human play. They called that version AlphaGo Zero. Also, in 2017, they announced AlphaZero, which was an approach that could learn to play championship level Go, shogi, or chess with minimal changes. Sort of a remarkable achievement.

There's been amazing progress on using machine learning approaches to play board games like Go. There's also been breakthroughs in using machine learning in partial information games with bluffing like poker. So, you'd think that partial information board games would be a snap. The game Stratego is a great example of such a game. Pieces take their places on the board in a manner that is very reminiscent of chess. But, unlike chess, the identity of the pieces is kept a secret. Like poker, players need to bluff and to look for clues to guess which pieces are in play.

Unfortunately, the counterfactual regret idea that is key to top poker play doesn't seem to translate so well to this board game scenario. There is just too much hidden information to reason about. As such, computer play of Stratego lags far behind the kinds of results we've been talking about. Some researchers have crossed their fingers and tried to treat Stratego like a video game where deep networks have been so successful, but the depth of strategy they achieve is still far below good human players.

LESSON 13 | GAMES WITH REINFORCEMENT LEARNING
TRANSCRIPT

So, even in the narrow world of board games, there is still much to learn and much to study. And, the real world combines all of these challenges, and much, much more. A lot more research will be needed to develop programs that can reason about real-life situations. But, next time, we'll look at an important step toward dealing with the real world—the problem of vision and learning how to see.

LESSON 14

DEEP LEARNING FOR COMPUTER VISION

Deep neural networks can produce state-of-the-art image-classification results. When these results started to appear, it was impossible to ignore deep networks as an essential tool for solving difficult computational problems. The overwhelming majority of work in the computer vision field was dominated by convolutional deep networks after 2015. And with this phenomenal success, other areas of artificial intelligence and computer science would also be infiltrated by deep networks.

The ImageNet Challenge

Neural networks revolutionized work in computer vision by vanquishing the competition in the ImageNet Large Scale Visual Recognition Challenge in 2012.

ImageNet is a dataset of photos along with labels of what each photo depicts. The challenge uses a subset of the larger dataset with 1,000 different categories, including types of animals, plants, activities, materials, and human-made objects.*

A training dataset of approximately 1 million images was released to participants, along with a validation set of about 50,000 images and a test set of about 150,000 unlabeled images. Each year, an exhibition was held to reveal which techniques had produced the best performance.

In 2012, a team from the University of Toronto supervised by Geoffrey Hinton entered the ImageNet competition using the name SuperVision. Not only did SuperVision improve on the results of the previous two years, but the size of the improvement grew substantially.

And SuperVision used neural networks—specifically, **convolutional neural networks**. Convolutional roughly means that these are neural networks designed to capture local features in images and combine the local features together to identify what the pictures represent.

Word spread quickly, and the following years showed substantial improvements as the techniques were refined.

The neural network architecture that won the ImageNet Challenge in 2014 is called VGG16—because it's arranged in 16 trainable layers. VGG16 illustrates many of the architectural ideas that are in use in other networks, such as convolutions, max pooling, and fully connected layers. And it's freely available for download.

VGG16 also has been demonstrated to have a nice property, whereby the last few layers tend to capture somewhat generic properties of images that can be used for recognizing more than just the 1,000 classes in ImageNet.

* In many ways, the ImageNet challenge was a lot like the Netflix Prize competition that was held to improve their recommendation system.

Advantages of Deep Networks

One of the remarkable things about deep networks that has helped propel them to wide use is that they tend to learn computations that solve the tasks they are given, which is great. But they also do a good job of learning new feature representations for their inputs that can be useful even beyond the specific task they were trained for.

Another exciting thing about deep neural networks is that the researchers designing them have produced Python libraries that make it really easy to specify and train the networks. A library called Keras defines the VGG16 network.

Given its size, VGG16 takes weeks to train, even using heaps of dedicated, specialized, high-performance graphical processing units. Another thing that's potentially cool about Keras and other deep learning packages is that they're optimized to take advantage of fast hardware.

VGG16 was trained to recognize the 1,000 ImageNet categories that include types of animals, plants, human-made objects, geological formations, and more.

We have access to a trained version of VGG16 that we can use to recognize objects in any of these categories. But even more importantly, VGG16's learned internal feature representation can be used to build recognizers for new objects without having to train it from scratch.

We can generalize beyond the 1,000 classes that the original competition was built around and learn to recognize categories of objects that they didn't think to include.

The deep learning features that come out of broadly trained networks like VGG16 do a great job of capturing a notion of visual similarity for objects more broadly.

Try It Yourself

Follow along with the video lesson via the Python code:

[L14.ipynb](#)

Python Libraries Used:

`IPython.display.display`: Displays images.

`keras.applications.vgg16`: The trained VGG16 network for image recognition.

`keras.applications.vgg16.decode_predictions`: Turns VGG16 predictions into labels.

`keras.applications.vgg16.preprocess_input`: Converts raw images into the form expected by VGG16.

`keras.backend`: Provides access to lower-level Keras functionality.

`keras.layers.Dense`: Creates a fully connected layer in Keras.

`keras.layers.Flatten`: Reorganizes array-shaped units into a flat vector.

`keras.models.Model`: Builds a neural network in Keras.

`keras.preprocessing.image`: Prepares raw images for processing by a neural network.

`numpy`: Mathematical functions over general arrays.

`PIL.Image`: Loads and converts images.

Auxiliary Code for Lesson:

[L14aux.ipynb](#)

Key Terms

convolution: A mathematical operation where the dot product of one vector is taken with another vector with its components shifted through a range of values. A key tool for neural networks achieving translation invariance in image recognition.

convolutional neural network: Neural network that learns convolutional operators designed to capture local features in images or sequence data and combine the local features to classify instances.

softmax: An activation function for neural networks that accentuates the largest value in a vector and then normalizes the values to be similar to a discrete probability distribution.

READING

Charniak, *Introduction to Deep Learning*, chap. 3.

LeCun, Bengio, and Hinton, “Deep Learning.”

Russell and Norvig, *Artificial Intelligence*, secs. 21.3 and 21.8.1.

QUESTIONS

1. What machine learning-friendly computer vision competition helped propel deep neural networks to prominence?
2. Translation invariance is the idea that a neural network can recognize the contents of an image even as it is shifted around within the picture. What component of a neural network is designed to ensure translation invariance?
3. This lesson showed that the VGG16 network can be used to encode images it was not trained on—a cheese grater and a foot file—as vectors that are perfectly recognized by k -nearest neighbors. What do you think would happen if we used a nearest neighbor (or a naive Bayes or a neural network) classifier on the raw vectors?

Answers on page 482

Deep Learning for Computer Vision

Lesson 14 Transcript

Alan Turing is one of the towering figures in computer science. His name is now attached to one of the central ideas in the theory of computation—the Turing Machine. He is also connected with a seminal concept in artificial intelligence—the Turing Test. Because he had such a lasting impact, the highest honor in computer science, kind of like our Nobel Prize, is also named for him. It's called the Turing Award, and it has gone to results in computer science from networking, to programming languages, to cryptography.

Awards for machine learning have been scarce. But in 2018, the prize went to three researchers for their pioneering work on deep neural networks. I attended the awards ceremony and it was a big moment for a lot of people in machine learning. Given the impact of deep learning, it made a lot of sense to acknowledge these researchers—Yoshua Bengio, Geoffrey Hinton, and Yann LeCun. But what was less obvious was that these three researchers, and others, had labored for many years to reach this point.

One reason it wasn't obvious was that, just a decade earlier, these same three researchers went out of their way to obscure the neural aspect of their work. The description of a workshop they helped organize in 2009 focused on the construction of what they called “high-level representations” and “deep architectures” with “nonlinear processing.” By “high-level representations”, they meant the contents of an image—animals, objects, scenery—as opposed just things like colors and edges and shapes. “Deep architectures” with “non-linear processing” just means neural networks with many layers.

They were hiding the connection to neural networks because *neural* had become a kind of taboo piece of terminology in machine learning. It had been popular in the 60s, and again in the late 80s and early 90s. But the techniques failed to live up to the hype and the community became disillusioned with anything neural. A few diehards continued to work on neural networks, but it was a niche topic and work was happening mostly underground.

The pivotal moment that helped neural networks climb back into the light was when they revolutionized work in computer vision. Specifically, they vanquished the competition in the ImageNet Large Scale Visual Recognition Challenge in 2012. ImageNet is a dataset of photos along with labels of what each photo depicts. The challenge uses a subset of the larger dataset with 1000 different categories, including types of animals, plants,

activities, materials, and human-made objects. In many ways, the ImageNet challenge was a lot like the Netflix challenge we talked about in the lesson on recommendations.

A training dataset of approximately 1 million images was released to participants along with a validation set of about 50,000 images and a test set of about 150,000 unlabeled images. Each year, an exhibition was held to reveal which techniques had produced the best performance. Here are the performance figures from the first five years of the challenge. On the y -axis is error on the testing set, and the x -axis is the year. As you can see, performance steadily improved over this window of time. By 2014, the error had dropped to less than a quarter of where it was in 2010.

All of these results were produced by machine learning algorithms. The 2010 and 2011 results used sophisticated applications of support vector machines, an approach to creating linear separators we talked about in Lesson 9. In 2012, a newcomer entered the ImageNet competition using the name SuperVision. (Get it? Super, vision and supervision.) Anyway, it was a team from the University of Toronto supervised by Geoff Hinton. Not only did Hinton's SuperVision improve on the results of the previous two years, but the size of the improvement grew substantially.

How did SuperVision do its thing? Neural networks. Specifically, convolutional neural networks. Roughly speaking, convolutional means these are neural networks designed to capture local features in images and combine the local features together to identify what the pictures represent. Word spread quickly and the following years showed substantial improvements as the techniques were refined. In other lessons, we'll talk about how the idea was picked up in other communities. In this lesson, we'll look at the role convolutional neural networks play in computer vision.

We already talked in the about nearest-neighbor algorithms, about distinguishing instances from one another by voting. If an unknown instance is near some labeled instances, we can have the labeled instances vote and we can predict the majority label for the unknown instance. For this idea to work as an approach to image classification, we need a notion of nearness between images. Typically, we represent an instance as a numerical vector. The raw information we start with is the array of pixels.

We can scale a set of images so they all have the same shape. Let's say that shape is a grid of 224 by 224 little colored rectangles. A color is itself represented by a triple of numbers that encode how much red, how much green, and how much blue there is in that color of the rectangle. So, each of these images can be written down as a list of about 150,000 numbers.

LESSON 14 | DEEP LEARNING FOR COMPUTER VISION
TRANSCRIPT

To measure the similarity between two images, we could take the cosine of their corresponding vectors—that's the normalized dot product, which we talked about in Lesson 8. That's fine, as far as it goes. That is, we can definitely build a classification system that way. But, there's not much reason to believe that it would work very well.

Two images can have high cosine similarity because they have similar pixel colors in the same locations and yet clearly be different images. These two images of unrelated objects have a cosine similarity of 0.87, for example, which is close to the perfect score of 1. That's probably because so much of the white background lines up.

Conversely, these are both images of dogs, but they have very little overlap at the level of the individual pixels. How can we capture images in a form where the similarities of their numeric vectors do a good job of capturing their visual similarity?

Let's talk about two approaches to constructing features and how they can be used to recognize images of digits. The first approach uses the raw pixel features we've been talking about. The second is a kind of feature that tries to capture the local spatial relationships between the pixels. These local spatial relationship features will be our jumping off point for understanding convolutional neural networks. We'll see that intelligent combinations of these two approaches can do a really good job of capturing image similarity.

The raw feature method we talked about is inherently translation variant. That is, the position of the digit in the image matters a lot. Consider these three digits: two 4s and a 9. The raw feature space that we've been using thinks the first and second are highly similar and the first and third are almost completely dissimilar. Surprising, right? I made the third image by sliding the first one over and up. I created the second image by drawing a 9 on top of the first image. The overlap looks like this: The dark patches are the overlap, so more dark is more similarity. The matching 4s barely touch and are therefore judged as dissimilar. The 9 and 4 share substantial overlaps. Something's not right with these features.

Here's another thing that should give us pause. If we take the image and scramble all of the pixel positions, reordering them randomly like some kind of sliding tile puzzle, and we do that to all the images, judgements of similarity are completely unaffected. That's weird, right? The features capture nothing about spatial locality in the image and everything about the absolute position in the image of individual dots. That's very different

from natural skill at recognition. We recognize a dog or a 4 or a juggling club in an image no matter where it is located. Our recognition is translation invariant—it works the same even as the content slides around.

How might we represent these digits with features that are translation invariant—location independent? One idea would be to count up the different kinds of strokes in the image, independent of where they occur. So, the 4s would both consist of a short horizontal bar, a tall vertical bar, and a short, positive slope, bar. No matter how we move the digit around, that description would remain exactly the same. And, it would be different from the 9, which consists of a tall vertical bar and a medium-size circle. Such a representation would satisfy the goal of being translation invariant. But, it's still not clear how one would automatically extract such a description from the image itself.

So, here's a similar idea, but one that is more easily implemented. In MNIST, digits are not made out of strokes, but out of shaded squares. Specifically, a digit exists as a 28-by-28 grid of pixels. Consider a smaller 5-by-5 patch like this. Now, I'm using transparent green for plus-1 and transparent red for negative-1. You can think of this image as being like a little close-up of a vertical bar. In particular, it is excited by the pixels that line up with the green patch in the middle and it is inhibited by the two stripes on the ends. This notion of exciting and inhibiting as a way of extracting features was discovered by scientists David Hubel and Torsten Weisel in their studies of the visual system of mammals. Some cells in visual cortex carry out exactly these kinds of computations to find patches, bars, edges, and various other low-level image features.

Here's what we're going to do with this patch. We're going to place it on the top of the image of the 4, in the upper right corner. At each overlapping pixel, you can see that the black part of the image of the 4 blots out whatever is in the patch, leaving black. Where there was white in the image of the 4, we see either the transparent green or transparent red. Here, we get six greens and three reds where the patch and the image overlap.

We can achieve a similar effect by multiplying the values of the corresponding pixels—black is 0 and green is plus-1, so the squares where the two colors overlap are worth plus-1 times 0 or 0. White is 1 and red is negative-1, so the squares where those two colors overlap are worth a value of 1 times negative-1 or negative-1. Now, we will sum up all of these products. In this case, we've got six 1s and three negative-1s, for a total of 3.

LESSON 14 | DEEP LEARNING FOR COMPUTER VISION TRANSCRIPT

Finally, we'll say that the patch matches the image at this position if the total is more than 10 out of a possible 25. So, in this case, there's no match. Is there someplace we can put the patch so that it will match? How about lining it up with the horizontal bar of the 4? Now, we have 14.5 cells that overlap with green and none that overlap with red, for a total of 14.5. That's more than 10, so we have a match.

How about the horizontal bar of the 4? Now, we have nine greens and six reds for a total of three. That's way under the threshold of 10, so no match. Basically, this patch is a little recognizer for vertical bars with a width of three. If we try the patch at every position in the image, we can get a sense of how much of the image is made up of vertical bars with a width of three. It doesn't tell us where they are, just tells us they are out there.

Other patch patterns can detect other visual features like rounded edges, diagonal bars, and more. The operation of scanning a patch over an entire image to see what matches we get is known as convolution. It's not that it's convoluted in the sense of being complex. It's because the word *convolve* means "roll together" and we're rolling the patch all over the image to count up matches. Using convolutional patches, we can encode an image like our 4 in terms of the total number of matches for each of a set of patches. Each patch becomes a feature in a feature-based representation of the image.

As I mentioned before, we get a representation that is translation invariant, but one where we lose a sense of which specific pixels are on or off. Really what we want are features that capture both position and spatial context. If we set up our convolutional filters correctly, we can learn the parameters that define the filter and ultimately pick filters that are tuned to the problems at hand.

The neural network architecture that won the ImageNet challenge in 2014 is called VGG16. Here's a table summarizing its architecture. It's a neural network arranged in 16 trainable layers—that's why it's VGG16. This architecture is the first deep network design we're looking at. Notice that each layer has a particular arrangement. The layers are listed from the input at the top to the output at the bottom.

Let's start with the input. Its size is 224-by-224-by-3. The three comes from the fact that there are three color channels for each of the 224-by-224 image positions. The very first layer that processes the input has a size of 224-by-224-by-64 units. It's a layer of convolutional filters, 64 of them in this case. The convolutional patches themselves are 3-by-3. These parameter settings were chosen by the network designers. VGG16 follows one convolutional layer like this immediately with a second one.

The stride of 1 refers to the idea that the convolutions are applied at adjacent pixels in the image with no skips. The activation function for these units is ReLU—they implement linear functions except with a kink defining a threshold minimum value at zero.

The next layer of the network is something new. It's called a max pooling layer. It has no trainable weights, so we're not going to count it as a layer in the architecture. Instead, it takes a 3-by-3 patch of the outputs of the previous convolutions and just returns the highest activation in those nine units. It does so with a stride of 2, so these 3-by-3 windows are only barely overlapping.

The idea of a max pooling layer is that if a convolutional unit discovers something exciting—"They, there's a corner here!"—we can enhance this information while decreasing the overall size of the layer. Because it uses a stride of 2, the total number of units at the next layer now has 224 divided by 2, while the number of learned features stays the same at 64. So, there are 112-by-112-by-64 units.

Note that the information is still arranged as a kind of image. It's lower resolution and there are 64 channels corresponding to the 64 different kinds of convolutional units from the previous layer. The basic structure of a set of convolutional layers thinned out by a max pooling layer repeats five times in this architecture. Each time, we have a kind of smaller resolution image but more and more channels corresponding to the kinds of features the network decides to pick out.

Finally, the last few layers of the architecture are fully connected, meaning that all of the units from the previous layer have weights connecting them to all of the units of the current layer. The explicit spatial information is gone now and the network is trying to recognize high-level properties of the entire image. The final layer has one unit for each of the 1000 ImageNet classes. The activation is Softmax, which means that the activations are forced to sum to 1 across all those 1000 units, causing them to act like probabilities or confidence values in the different classes.

Why this particular arrangement of layers? To be honest, there are lots of different arrangements that have been proposed, and many do good and useful things. I picked this one for several reasons. It did well in one of the ImageNet challenges. It also illustrates many of the architectural ideas that are in use in other networks—convolutions, max pooling, and fully connected layers. In addition, it's freely available for download.

LESSON 14 | DEEP LEARNING FOR COMPUTER VISION TRANSCRIPT

It also has been demonstrated to have a nice property, whereby the last few layers tend to capture somewhat generic properties of images that can be used for recognizing more than just the 1000 classes in ImageNet. In fact, that's one of the remarkable things about deep networks that has helped propel them to wide use. They tend to learn computations that solve the tasks they are given, which is great. But they also do a good job of learning new feature representations for their inputs that can be useful even beyond the specific task they were trained for.

I want to talk you through the kind of image features captured by VGG16. Layer 1 is the visual filters learned in a convolutional network close to the input. Edges, color patches and bars are the types of features selected at this top level. At a deeper layer, there are filters that are sensitive to bits of shape and texture. Some filters might be excited by stripes, others by concentric circles, others by scaly patterns and bumpy textures.

Deeper still, filters are reacting to object parts—smooth patches of surfaces, corners of objects, torsos of people, bumpers of cars. The deepest layer include filters that capture objects and their parts in context—items in water, legs on ground, faces on dogs, wheels on grass. The filters in the deepest layer also seem to become somewhat color specific, with some filters preferring patches with red items, say, while others might prefer brown or blue backgrounds. By creating a rich vocabulary of image filters, the convolutional network becomes ideally suited for categorizing any of a large set of possible object classes.

In a sense, it's not shocking that the network would decompose images into components in this way. After all, what alternative is there? But, it's still exciting to see this kind of analysis being done so automatically, and in a way that can generalize to new situations.

Another exciting thing about deep neural networks is that the researchers designing them have produced Python libraries that make it really easy to specify and train the networks. Here's an example of code from a library called Keras. It defines the VGG16 network we were just talking about. There are a few really cool things about this code. First, it lets you specify the architecture layer by layer, just like the table. The lines of code match up perfectly to the high-level description.

Second, let's take a minute to think about what happens under the hood. These statements are defining layers that are actually building up an explicit representation of a huge parameterized function. A program. And, the convolutional filters in this program act like subroutines of their own. When it comes time to train this function by way of gradient descent, it's necessary

to take its derivative. That's something you can't really do on code in general. But these functions are constructing the network not just functionally, but also symbolically, and that makes it possible for the gradient to be computed behind the scenes. When we run the function, it shows us that the whole network consists of 138 million trainable weights.

Given its size, VGG16 takes weeks to train, even using heaps of dedicated specialized high performance graphical processing units. That's another thing that's cool about Keras and other deep learning packages. They are optimized to take advantage of fast hardware. Of course, if you don't have that hardware, it's not much of an advantage. But all is not lost. VGG16 was trained to recognize the 1000 ImageNet categories that include types of animals, plants, human-made objects, geological formations, and more.

We have access to a trained version of VGG16 that we can use to recognize objects in any of these categories. But, even more importantly, VGG16's learned internal feature representation can be used to build recognizers for new objects without having to train it from scratch. We can generalize beyond the 1000 classes that the original competition was built around and learn to recognize categories of objects that they didn't think to include: fidget spinners, say, or Tesla cars. Or household abrasion tools. Yeah, let's do that last one.

If we want robots to help us around the house, they need to be able to distinguish the different kinds of tools we use. One pair of tools that I often confuse is our cheese grater and our foot file. They look very similar, but you shouldn't mix them up. Using the cheese grater on your body is dangerous and using the foot file on food is deeply unappetizing.

I imported a half dozen packages from Keras and one from NumPy for some matrix manipulation. I then loaded the parameters—the weights—for VGG16. Note that it was trained to recognize many categories from football helmets to cheeseburgers, but not foot files or cheese graters. I then surgically removed the final two layers that do the classification and left all the previous layers that define the feature set.

I took 10 photos each of the foot file and the cheese grater, in different orientations, all against a neutral background. We can grab my photos and unzip them. After importing two libraries for manipulating images, I constructed a list of the 20 image files. Let's loop through the images and load them in. We'll make them 224-by-224 pixels to match the image sizes that VGG16 expects. We'll squirrel away the image object itself in a list of images called imgs. To feed it into VGG16, we need to turn the image into an array.

TRANSCRIPT

But, the really slick thing is how we use VGG16 to turn this image into a feature vector. We feed the image into VGG16 as we would if we were going to predict one of the 1000 ImageNet classes. Since we had altered VGG16, removing its last layer, what's left is the activation on the final layer of the network before it makes its classification.

These vectors capture the deep similarities between images and are fantastic for use in a nearest-neighbor classifier. Don't take my word for it, let's try it out.

We'll loop through the images; i_1 is the feature vector for the current image. We'll compare i_1 to each of the other image feature vectors, i_2 . We'll use dot product similarity between i_1 and i_2 using Python's `@` operator and keep track of which image bestmatch has the highest similarity to i_1 . In each case, the nearest neighbor for the tool is another picture of the same tool, in the same or sometimes reversed orientation. Although VGG16 wasn't trained to recognize foot files or cheese graters, it understands how images work well enough to know which images correspond to the same object.

In this example, we used a machine learning classifier trained for one problem—VGG16 recognizing 1000 different image categories in ImageNet—to solve a related but distinct problem—distinguishing a foot file from a cheese grater. Within the machinelearning community, this repurposing idea is known as transfer learning. A big advantage of transfer learning is that a new machine-learning problem can be solved with only a small amount of new data and computation. The secret is leveraging previous processing on large amounts of other data. In this case, we created a great household tool recognizer with only 10 training images. It's hard to imagine how to better leverage small amounts of data to such great effect.

It's still a bit of an art form to know which previously trained systems to use for transfer. ImageNet includes dozens of categories of household objects—paintbrushes, pencil sharpeners, plungers. It needed to learn visual features for recognizing and characterizing all of these classes. Some of those visual features are probably helpful for characterizing cheese graters and foot files as well. But what if you have a trained network for recognizing different types of trucks? Could transfer learning be used to repurpose this network to distinguish kinds of fish? Maybe not.

Now, the point here is not about specific household tools. I wanted to illustrate that the deep-learning features that come out of broadly-trained networks like VGG16 really do a great job of capturing a notion of visual similarity for objects more broadly. Amazing stuff.

So, deep neural networks, with many layers arranged to identify and exploit translation-invariant features, can produce state-of-the-art image-classification results. When these results started to appear, it was impossible to ignore deep networks as an essential tool for solving difficult computational problems. After the ImageNet challenge, the overwhelming majority of work in the computer vision field was dominated by convolutional deep networks after 2015.

With the phenomenal success of deep networks in computer vision, it was inevitable that other areas of artificial intelligence and computer science would take notice. In upcoming lessons, we'll explore the equally transformative impacts of deep neural networks on text processing and speech processing. But, next time, we'll confront a fact that each successful new domain of deep learning has had to confront—debugging a deep learning program is hard, with elements unlike anything else seen in programming.

LESSON 15

GETTING A DEEP LEARNER BACK ON TRACK

This lesson teaches you some techniques that are specific to debugging machine learning programs. There's no magic recipe for taming a difficult program, but the strategies in this lesson can help you get things working faster.

Debugging Machine Learning Programs

When you debug a computer program, you are carefully figuring out the right questions to ask. You want to understand why things went wrong and how things can be made right.

Since machine learning programs are programs, you can make the same kind of bugs that plague authors of any typical computer program—from syntax errors in the code to logic errors in your own thinking.

But in machine learning programs, there's another layer to consider. Even when machine learning programs are broken, they are still trying to do the right thing. And that virtue of machine learning can also interfere with a programmer's ability to see where the/an error is: The programs don't work as well as they should, but the root cause of the problem can stay hidden—especially in deep learning programs, since they can be so complex.

A machine learning program can go wrong in any of its three main components: the representational space, the loss function, or the optimizer.

So debugging a machine learning program requires that we delve into each of them. We need to make sure that our instances and labels track each other and that we haven't corrupted the data along the way. We need to use the right loss function for the job. And we need to figure out how to coax our optimizer into producing useful answers.

Debugging the Loss Function

When computing loss, it's a good idea to watch out for class imbalance. If there's way more of one class than others, typical loss functions will skew the results toward only caring about the majority class. Using similar amounts of data across the classes can help. Or, if the classes do not have similar amounts of data, there are loss functions that weigh the different training instances differently to shift things around to compensate.

It's important to look at your data as a sanity check before and after it is input into the network. Is it corrupted? Is the important signal being overwhelmed by noise or errors? Did the labels get assigned correctly? Make sure your data is clean. The loss of your network is measured with respect to the data, so messed up data will mislead the training process.

An easy error to make is applying some kind of transformation on the training data but then forgetting to do the same transformation on the testing data.

The loss surface is also significantly affected by the amount of training data. Get enough training data. Overfitting—significantly better training performance than testing performance—is a sign that more data is needed.

Debugging the Optimizer

In machine learning of all kinds, optimization is the main service the computer does for you. But it can also be the most frustrating piece to debug because it's very much out of your control.

One of the big contributions of the deep learning revolution has been the idea that we can just wait longer.

In the 1990s, if we weren't getting good results after about an hour of training, we'd give up. In the 2010s, deep learning runs took days or weeks despite using computers that had become much faster.

If you don't want to wait longer, you can also get more processors or faster processors. Graphics processing units (GPUs) and tensor processing units (TPUs) can do the same amount of work in a fraction of the time because they are so well optimized for the operations that deep learning systems need. They can be expensive, and they can use a lot of energy—but they are fast.

The learning rate parameter has a big impact on optimization results. Low learning rates can do a better job of tuning the weights accurately.

But high learning rates can cover more ground in the search more quickly. If you conclude that weight updates are too slow, increase your learning rate. If you conclude that weight updates are too erratic, decrease the learning rate.

Packages like Keras can do some automatic tuning of the learning rate, but with enough experience, a person can often do much better, and much faster.

If you need the best performance money can buy, do a grid search on any hyperparameters you aren't already confident in. That's how the big players squeeze the best performance out of their networks. Include the optimizer itself in the search.

Optimization is hard. Sometimes there's just not enough signal in the data to learn a good rule.

Debugging the Representational Space

But even small players have control over the representational space for the learner.

If you are underfitting—getting low performance even on small amounts of training data—you might need a bigger network. More parameters provide more power for expressing more specific patterns in the data. And they can make the optimization problem easier as well.

You might find it useful to train on simplified artificial data or small amounts of data to make sure the network can carry out the necessary computations. Or you might train on a real-world dataset that's known to be easy.*

An annoying error is to freeze or unfreeze the wrong weights. If you're using a pretrained encoder, you don't want the weights in part of the network to change. You can tell the machine learning library to lock them in place. Other weights should be allowed to change to fit the data. Be careful not to get these things backward.

It can help to standardize the features entirely, changing the scale and shifting so that they have zero mean and variance of 1. Batch normalization standardizes for you. Normalizing can make it easier for the optimizer to find weight settings that transform the inputs into a form that's easier to manipulate.

Even if you get all of these many things right, you can still run into trouble. Things going wrong is not a rare occurrence. But keeping in mind these vulnerabilities in the loss, the optimizer, and the representational space can help get you back on track when things do go wrong.

Question your assumptions. You might feel certain that what you are doing *should* work. But maybe it just *shouldn't*.

Ask yourself for evidence—and gather the evidence if you don't have it. While checking what you believe should work and the evidence for that belief, you are likely to discover that the bug wasn't in the program—it was in your thinking all along.

* MNIST is a good example in the visual domain.

Try It Yourself

Follow along with the video lesson via the Python code:

[L15.ipynb](#)

Python Libraries Used:

keras: The Keras deep learning package from Google.

keras.applications.vgg16.decode_predictions: Turns VGG16 predictions into labels.

keras.applications.vgg16.preprocess_input: Converts raw images into the form expected by VGG16.

keras.layers.Activation: Sets the activation function for a layer.

keras.layers.Conv2D: Creates a 2D convolution in Keras.

keras.layers.Dense: Creates a fully connected layer in Keras.

keras.layers.Dropout: Enables dropout normalization.

keras.layers.Flatten: Reorganizes array-shaped units into a flat vector.

keras.layers.MaxPooling2D: Pooling layer for 2D convolutions.

keras.models.Sequential: Builds up layers of a neural network in Keras.

keras.optimizers: Optimizers used in Keras.

keras.preprocessing.image: Prepares raw images for processing by a neural network.

numpy: Mathematical functions over general arrays.

PIL.Image: Loads and converts images.

random: Generates random numbers.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

Key Terms

hyperparameter search: A process of tweaking the hyperparameters, perhaps using a grid search over values of the hyperparameters.

READING

Le, “The 5-Step Recipe to Make Your Deep Learning Models Bug-Free.”

Russell and Norvig, *Artificial Intelligence*, sec. 19.4.

Shao, Cecelia. “Checklist for Debugging Neural Networks.”

QUESTIONS

1. Why are vanishing and exploding gradients a problem that's specific to deep neural networks, and how can you tell if your network is experiencing them?
2. How can you tell if your learning rate is set too high? Too low?
3. How much of a difference does hyperparameter tuning make? Do a simple grid search to find the best learning rate using the setup from this lesson. Among learning rates of 0.001, 0.005, 0.01, 0.05, 0.1, and 0.5, what's the biggest difference in test set accuracy?

Answers on page 482

Getting a Deep Learner Back on Track

Lesson 15 Transcript

When you debug a computer program, you are carefully figuring out the right questions to ask. You want to understand why things went wrong, and how things can be made right. Since machine learning programs are programs, you can make the same kind of bugs that plague authors of any typical computer program—from syntax errors in the code, to logic errors in your own thinking.

But, in machine learning programs, there’s another layer to consider. Even when machine learning programs are broken, they are still trying to do the right thing. And that virtue of machine learning can also interfere with a programmer’s ability to see where the error is: The programs don’t work as well as they should, but the root cause of the problem can stay hidden, especially in deep learning programs, since they can be so complex. In this lesson, we’ll look at some techniques specific to debugging machine learning programs.

A machine learning program can go wrong in any of its three main components—the representational space, the loss function, or the optimizer. So, debugging a machine learning program requires that we delve into each of them. We need to make sure our instances and labels track each other and that we haven’t corrupted the data along the way. We need to use the right loss function for the job. And, we need to figure out how to coax our optimizer into producing useful answers.

As you might guess, there’s no magic recipe for taming a difficult program. But, the strategies in this lesson can help you get things working faster. The best way to show you what to do when things go wrong is to let things go wrong. In most of the examples I’ve prepared for this course, things didn’t work on the first try. I had to do debugging—sometimes a lot of debugging—to get the examples to the level where I was comfortable sharing them with you.

Not this time. We’ll work through an example, and I’ll show you my mistakes as I make them. I’ll also show you the changes I had to make to get things working.

Since we’re debugging, I decided we’d create a neural network classifier that can look at images of animals and flag images that have insects or spiders, basically any images of creepy crawlly critters. Yes, we’re going to find the bugs in a machine learning program that’s about finding bugs. That’s meta. Just like machine learning!

Okay, ImageNet is a large collection of labeled images that can be used for classification. It really is large. There are a million images. It’s too much to use ImageNet as a pedagogical example where we are training and retraining in different ways. Just uploading the images to a Python programming environment such as Colab could take an hour before we even start to process them.

Fortunately, a machine learning class at Stanford created a smaller version of the dataset where the individual images are scaled down and the number of distinct categories was reduced. Even with this reduced-size data set, a single training run with a thousand passes through the data could take an hour, and there are about a dozen runs in this lesson.

Several times, I’ve mentioned that deep learning leverages graphics processing units, or GPUs, to run quickly. Given the computational costs involved in training models this lesson, I found GPUs essential this time. If you are following along in Colab, I recommend that you do the same. After you load the notebook but before you run any of the code blocks, select the “Runtime” menu on the Colab notebook page. Under that, select “Change runtime.” Then, select “GPU” from the menu. Colab will remind you that you should only use GPUs when you really need them. You do.

The scaled down ImageNet collection is called “tiny-imagenet” but it still has 100,000 images, which is kind of a lot for something that’s tiny. The dataset consists of 200 categories instead of Imagenet’s full 1000 categories. Each image is a 64-by-64 pixel color image, which is about one-twelfth the size of those in Imagenet. There are 500 images in each category instead of Imagenet’s roughly 1000 images per category.

We can download and unzip all the image files. We’ll import some code for image processing. We’ll use keras for building our neural networks and numpy for working with vectors. We’ll also use a function from sklearn, the scikit learn library, for splitting up training and testing vectors. From the 200 categories in the set, I picked 36 that correspond to animals. The list includes lions, boa constrictors, and king penguins. We’ll combine these individual categories into a new category called “animals.”

TRANSCRIPT

I also picked 14 categories from tiny ImageNet that correspond to bugs. The list includes roaches, grasshoppers, scorpions, tarantulas, and dragonflies. We'll combine these individual categories into a new category called "bugs." The `read_cats` subroutine reads the images for the categories "cats" it is given and it associates their vectors with the labels in "lab." We also pass in the desired training and testing set sizes.

Initially, the lists of vectors and labels are empty. We run through all the categories `c` and images `i`. For each one, we construct a filename into the tiny dataset. Specifically, the directory that stores the images is called "tiny-imagenet-200/train." Then, there's a subdirectory for each of the tiny ImageNet categories, which contains a subdirectory called "images." Within that directory, there's 500 JPEG files, each named with the tiny ImageNet category and a number. We retrieve the image and store it in "img."

From the image object, we extract an array and flatten it out into a vector. Then, we reshape it to 64-by-64-by-3 colors. We string together all the collected images and labels into lists, one list called "vecs", the other called "labs."

Once the lists are constructed, we turn the list of vectors into a numpy array. We split up this array and the labels with the desired train/test sizes and return the result. Now, we can read the vectors and labels for the animals. That will be our category zero. We'll use 10% of the available data for training and 20% for testing. We'll do the same for the bugs. That will be our category one. Combining category zero and category one gives us our training and testing sets.

Now, we have our training and testing data. Next, we need to set up a learner. Since we're recognizing images, it seems like a reasonable place to start is the VGG16 architecture from the vision lesson. It consists of a series of 14 convolutional layers interspersed with max pooling layers. After these layers, it includes three fully connected layers ending with the output layer. In the code to build and return the untrained neural network, the main change is in the very first layer and the very last layer. VGG16 is designed to recognize ImageNet images of size 224-by-224. The tiny images in this set are only 64-by-64. So, the input shape is different.

The output shape is also different. In the ImageNet challenge, the learners have to pick out an image from 1000 categories. Here, we only have two—animal and bug. So, we'll use a single output unit with sigmoid activation.

A sigmoid unit is one that outputs a number near 1 if the input sums to a positive number and near 0 if the input sums to a negative number, with a smooth transition between them around 0. Thus, the outputs look like probabilities. It is trained to produce a number close to 1 for bugs and close to 0 for animals.

Okay, let's try to recognize our bugs! We build the network, attach a loss function and optimizer to it, and run the training algorithm. It says our accuracy on the testing data at the end of training was 72%. We'd really like the loss to be a tiny number like 0 but instead we're at 0.28. But, maybe that's okay? As debuggers, we need to look a little deeper though. Let's try to get a handle on whether that score is good or bad.

One important debugging tool is to look at the data. Here's some code for picking a random testing example and showing the associated image. It picks an index at random using the random number generation library's "randint" command for producing integers in a given range. It prints the associated label "y_test" for that testing example where 0 means animal and 1 means bug. It pulls out the instance, preprocesses it, and formats the array as an image.

I encourage you to run the code and look at some random images from the collection. Can you distinguish the animals from the bugs? I flipped through 10 images and there are two or three where I wasn't really sure what was going on. So, I'd score something like 70-80%, which is on par with the algorithm. That suggests that maybe the classifier is pretty good.

What does the classifier actually say? It classifies all of these images as animals. Uh-oh. The only reason it gets 72% correct is that the classes are imbalanced. Seventy-two percent of the images are animals, so always guessing animal seems to do well. No. We want the learner to have to work a bit harder to get a score like 72%. As part of our debugging process, we'll force the two classes to be of equal size.

Let's retrain using 500 examples from each class. Then, we can rerun and see what happens. 50% correct. Okay, that's good news and bad news. It's good news, because we now have an accurate measure that the learner is failing to capture any of the signal in the data. The bad news is that the learner is failing to capture any of the signal in the data. Alright. One powerful technique for debugging in general and for debugging learning algorithms in particular is to give it an input that it has to get correct. What do we expect to happen if we decrease the training size to 10 examples—5 positive and 5

TRANSCRIPT

negative? The neural network model has almost 40 million weights. With only 10 training examples, we should expect massive overfitting. That is, the learner should easily memorize the training data and obtain zero loss, but we cannot expect it to generalize usefully to new examples. Is that what happens? At this stage, it's worth checking.

Indeed, it gets a measly 50% correct on the testing examples. But it also gets the same poor 50% correct on the training examples. Something is definitely not right. Our learner is not responding in a meaningful way to the training data. We will focus on that issue first and then worry about generalization to the testing data. Let's take a look at the optimizer.

Well, one issue is that the optimizer is only running for one epoch, one pass through the data. That might be enough if we have millions of examples, but with only 10 examples, we're going to need to update the weights many times per training example to see meaningful learning. Let's rerun for 1000 epochs. I turned off verbose mode because otherwise the screen would be flooded with 1000 status updates. Instead, I'll just print the loss and accuracy at the end. Nope, still getting 50% in training and testing.

Neural networks were extremely popular in the late 80s and early 90s. A major cause for their falling out of favor is that it was difficult to train networks with more than two layers of weights. One of the reasons for this problem is that gradient computations are multiplicative. That is, the gradient at one layer is computed by multiplying weights times the gradient at the previous layer. Think of what that means when there are a dozen or more layers. If weights are typically less than one, the product will dive to zero. If the weights are typically more than one, the product will rocket toward infinity. The range of weights for which the gradient values are likely to remain in a meaningful range is pretty narrow. This problem is often referred to as one of vanishing or exploding gradients.

To get a deeper appreciation for the problem, let's look at a concrete but very simple example. Consider a neural network that maps a single input to a single output. It has nine layers of weights and each neuron has a linear activation function. That means that when we present an input x to the network, it gets multiplied by the first weight w_1 to produce the activation in the first hidden unit. With a linear activation function, this value remains unchanged and then is multiplied by the next weight w_2 to produce the activation in the second hidden unit. This process continues until we get the output y , which will be the input multiplied by all of the weights in the chain from input to output.

In a more complex network with multiple hidden units in each layer, the output is the sum of chains like this over all possible paths from input to output. If the input were 1 and the weights had values around 5, the output would be around 2 million. On the other hand, if the weights had values around 0.1, the output would be around a billionth.

Initializing weights to be around 1 keeps the activations from blowing up or collapsing. The same holds true for the gradients, even more so. In this simple network, the magnitude of the gradient is roughly the square of the output. Giant values get even bigger. Tiny values get even smaller. One way to keep activations from exploding is to use an activation that squashes big inputs.

Activation functions like sigmoid and hyperbolic tangent—tanh—were very popular in the 80s and 90s, and they serve this purpose. They are effective at preventing explosions in activations, but they do so by flattening out the gradient at high and low activation levels. The gradient of tanh at 5 or negative-5 is about 200,000ths. Since weight updates are proportional to the gradient, that means many, many, many updates are needed to move the weights when the gradient gets tiny.

Several techniques help tame these vanishing or exploding gradients in neural networks. Switching from sigmoid or tanh activations to ReLU helps eliminate the vanishing gradient problem. The gradient of ReLU is one that's either 0 or 1, no matter what the input is. As a result, the gradient is likely to move the weights quickly, if it moves them at all. But, ReLU activation still suffers from the exploding gradient problem if activations get big. This problem is mitigated by initializing all weights to values in the vicinity of 1. That way, activations never get significantly magnified or reduced as they propagate through the network. These algorithmic ideas played an important role in enabling the deep learning revolution, along with the availability of data and increases in computer power.

We're already been using keras' default weight initialization and ReLU activation functions, so we're probably okay, but let's take a look just to make sure. Since weight updates are proportional to the size of the gradient, an exploding gradient will manifest as extremely large weights in the network. Let's take a look to make sure the weights haven't gotten too big. Running "get_weights" on the learned model will dump all the values for us to see. There are a lot of them. But, flipping through, the weights seem to be in a reasonable range. I don't see anything bigger than one, for example.

TRANSCRIPT

So, exploding gradients is probably not our problem. Vanishing gradients are a lot harder to spot. They manifest as no weight changes at all, especially in the layers closest to the input. An important technique is normalizing or pre-processing our instances so they are in a range of values likely to pass through the network well. Keras includes a library with some image processing useful to VGG. We'll use the pre-processor to prepare the data for learning. After translating the image into an array, we apply the pre-processing routine to it. The resulting image array is then flattened and then reshaped into a 64-by-64 pixel array with three color channels representing red, blue, and green.

And, as we saw, activation functions are most reactive to their inputs when those inputs are in the minus-1 to plus-1 range. So, since pixel values are 0 to 255, we'll rescale by dividing by 128 and subtracting 1. That makes the smallest value 0 divided by 128 minus 1, or negative-1. The largest value becomes 255 divided by 128 minus 1, or roughly positive-1.

This modification did not improve matters. We're still getting 50% error on both training and testing. The preprocessing is likely useful for learning, but it didn't help here. It's a trick that's more useful for taming exploding gradients than vanishing ones. Our program is getting better, but it's still plagued by some problem we haven't identified yet. Hmm. Perhaps what's going on is that we have a very tiny training set, and we're only running through it 1000 times. Maybe that's just not enough to move the weights to where they need to be.

We don't want to increase the training set right now. That's because we're trying to debug why we're not seeing learning happening even on a small example. This is where the network ought to be able to simply memorize the inputs. We could increase the number of epochs used in training. That could slow things down a bit, while the weights will have a chance to move in the right direction. Or we could also just take bigger steps by increasing the learning rate.

Increasing the learning rate should let us see a change quickly. But, it's also a bit risky because big updates can cause the learner to overshoot, missing out on what could be more appropriate weights for the problem. Still, it's worth a shot. Hallelujah! We're getting 60% accuracy on the training data now. It's finally listening to us! Performance on the testing data is poor, just 49%, but that's okay. We are not yet training the classifier to be good. We were just trying to get it to react.

Let's increase the size of the data in the training set from 10 to 100, 50 animals and 50 bugs. We'll retrain for 1000 epochs; that's how many times the algorithm will run through the training data. Accuracy on the testing data is up to 55.8% now. That's encouraging. Let's increase the size of the training data from 100 to 10,000 by increasing the number of training examples in each of the two categories to 5,000.

Since there are more training examples to move around and move the weights around, we can cut back on the epochs a bit so training time doesn't increase too much. Let's change from 1000 epochs to 100 epochs. It takes a few minutes to train, even on the GPUs. Without GPUs, I could not get it to run even 50 iterations in under 13 hours. Using the GPUs, running 100 iterations took about 15 minutes. And, the results are pretty solid. We're getting 80% accuracy now on the testing data. That's about how well I did when I tried to recognize the images myself.

If we could increase the size of our data by another three or four factors of 10, up to 10 million or 100 million, we might be seeing astonishingly good performance. Deep learning is a data hungry beast, but it repays such contributions with improved accuracy. Since we're not going to go out and collect our own additional data, we're pretty much maxed out now. If we want to improve performance further, we'll need another approach.

One thing to notice is that we're strongly overfitting, with 99.7% accuracy on the training set versus 80% accuracy on the testing set. In principle, when we have a classifier that is generalizing properly, these numbers would be the same. That suggests that we need some regularization and maybe we should use a smaller representational space. Given that our VGG16 network has about 40 million trainable parameters, thinning out the representational space to have fewer parameters seems like a sensible idea. At this point, we could do various architectural tweaks: try another convolutional layer, change their kernel sizes, we could increase or decrease the number of convolutional features. Maybe you can see why it's so common to just reuse something that other researchers have found that works. The space of possible tweaks can be overwhelming.

We could also move the learning rate up. Or down. Or maybe change the optimizer or the number of training iterations. All these different kinds of modifications are collectively called hyperparameter search. If the parameters of the network are the weights, the hyperparameters are the parameters like the learning rate that influence how the weights are found.

TRANSCRIPT

We have not been very systematic about setting the hyperparameters. But, if we had a lot more computer power at our disposal, we could run hundreds or even thousands of different combinations of settings all at once to find what works best. And, that's something machine learning people often do. It's referred to as grid search, because you can think of the combinations of hyperparameter values as points on a grid. Also, the computer clusters where these searches take place also happened to be called grids.

A grid search really just amounts to a loop over parameter values. We might want to try 20 different learning rates, two different optimizers, 5 different initial random weights, and 10 different network architectures. That's 2,000 repetitions. So, that's 2,000 times longer to run. If learning takes 10 minutes, then this grid search would take five and a half days. If learning takes five hours, the grid search would take 13 months. Of course, people doing grid search don't wait 13 months. They use 2,000 computers and get the results in the same time they would spend on just one run. Of course, the people that can do that are the people who have 2,000 computers at their disposal. Generally speaking, those are big computing-centric companies, not your Average Joes.

One final technique to mention, whose apparent efficacy outstripped the ability of researchers to explain why it works, is batch normalization. Introduced in 2015, batch norm, as it's known for short, can dramatically reduce the number of epochs needed to train a deep network. Batch norm changes the behavior of layers inside the neural network. Specifically, it shifts and scales a unit's output value so it is mean 0 and variance 1 with respect to the batch of data the network is processing.

You can tell keras to use batch normalization by running "keras.layers.BatchNormalization" on a layer of the network you are constructing. Batch norm is not a debugging technique per se, but it is an important technique for making deep networks train more consistently. It keeps the activations on units in the minus-1 to plus-1 range where (as we've seen) gradients are best behaved.

Let's pull together what we've learned about debugging a deep learning system. I think a helpful way of organizing our knowledge is in terms of debugging along three dimensions that define a machine learning program. There's debugging we do on the loss function, and debugging we do on the optimizer, and debugging we do on the representational space.

Let's start with how loss is being computed. It's a good idea to watch out for class imbalance. If there's way more of one class than the others, typical loss functions will skew the results toward only caring about the majority class. Using similar amounts of data across the classes can help. Or, if the classes do not have similar amounts of data, there are loss functions that weigh the different training instances differently to shift things around to compensate.

It's important to look at your data as a sanity check, before and after it is input into the network. Is it corrupted? Is the important signal being overwhelmed by noise or errors? Did the labels get assigned correctly? Make sure your data is clean. The loss of your network is measured with respect to the data, so messed up data will mislead the training process.

An easy error to make is applying some kind of transformation on the training data but then forgetting to do the same transformation on the testing data. The loss surface is also significantly affected by the amount of training data. Get enough training data. Overfitting—significantly better training performance than testing performance—is a sign that more data is needed.

In machine learning of all kinds, optimization is the main service the computer does for you. But it can also be the most frustrating piece to debug because it's very much out of your control. In my opinion, one of the big contributions of the deep learning revolution has been the idea that we can just wait longer. In the 1990s, if we weren't getting good results after an hour or so of training, we'd give up. In the 2010s, deep learning runs took days or weeks despite using computers that had become a lot faster. It's a good thing come to those who wait approach.

If you don't want to wait longer, you can also get more processors or faster processors. GPUs and TPUs can do the same amount of work in a fraction of the time because they are so well optimized for the operations that deep learning systems need. They can be expensive and they can use a lot of energy, but they are fast.

The learning rate parameter has a big impact on optimization results. Low learning rates can do a better job of tuning the weights accurately, but high learning rates can cover more ground in the search more quickly. If you conclude that weight updates are too slow, increase your learning rate. If you conclude weight updates are too erratic, decrease the learning rate.

LESSON 15 | GETTING A DEEP LEARNER BACK ON TRACK

TRANSCRIPT

Packages like Keras can do some automatic tuning of the learning rate, but, with enough experience, a person can often do much better and much faster. If you need the best performance money can buy, do a grid search on any hyperparameters you aren't already confident in. That's how the big players squeeze the best performance out of their networks. Include the optimizer itself in the search. We used the Adam optimizer in this lecture, but others include, `sgd`, `rmsprop`. Switch it up a bit.

But even small players have control over the representational space for the learner. If you are underfitting—getting low performance even on small amounts of training data—you might need a bigger network. More parameters provide more power for expressing more specific patterns in the data. And they can make the optimization problem easier as well.

I have found it useful to train on simplified artificial data or small amounts of data to make sure the network can carry out the necessary computations. Or, you might train on a real-world data set that's known to be easy—MNIST is a good example in the visual domain.

An annoying error is to freeze or unfreeze the wrong weights. If we're using a pre-trained encoder like we did in the vision lesson to recognize household tools, we don't want the weights in part of the network to change. We can tell the machine learning library to lock them in place. Other weights should be allowed to change to fit the data. If we get these things backwards, ugh. It can help to standardize the features entirely, changing the scale and shifting so that they have 0 mean and variance of 1. Batch normalization standardizes for you. Normalizing can make it easier for the optimizer to find weight settings that transform the inputs into a form that's easier to manipulate.

Even if we get all of these many things right, we can still run into trouble. In my experience, things going wrong is not a rare occurrence. Optimization is hard. Sometimes there's just not enough signal in the data to learn a good rule.

But keeping in mind these vulnerabilities in the loss, the optimizer, and the representational space can help get you back on track when things do go wrong.

One last word of advice. Question your assumptions. Ask yourself, “Why do I believe this should work?” You might feel really sure that what you are doing ought to work. But, maybe it just shouldn't. Ask yourself for evidence and gather the evidence if you don't have it. Checking what you believe should work, and the evidence for that belief, you are likely to discover that the bug wasn't in the program, it was in your thinking all along.

It's kind of magical to watch a computer learning to see. Not bug, bug, not bug, bug, bug, not bug. Bugs be gone. Good job, network! Debugging a machine learning program can be time-consuming and frustrating. But, when you finally get it working well, it's an amazing feeling.

LESSON 16

TEXT CATEGORIZATION WITH WORDS AS VECTORS

In this lesson, you'll learn how you can use machine learning to help construct accurate language models from data. You know that neural networks are great with low-level perceptual data, but neural networks also came to be used for learning what words mean—and classifying text based on leveraging these language models.

Language Embedding

Machine learning gives us a whole new tool kit for representing and acting on the meaning of words that goes far beyond what was achieved in earlier systems that used handwritten rules to process language.

The starting point for the machine learning approach to understanding natural language is language embedding. The idea is to use machine learning to capture statistical relationships between words by associating each word with a high-dimensional vector.

Language embedding is used whenever we need to capture the meaning of words, which is an important component of processing language.

Because language embedding maps words to vectors, it's also known as word2vec.

There are many natural language tasks a computer might be able to help you with, including

- ◆ spam filtering, which is basically like asking the computer to look over your email to remove junk messages before you look at them;
- ◆ autocompletion, or spelling correction; and
- ◆ extracting text from images (optical character recognition) or audio (automatic speech recognition).

In all of these applications, the computer needs to be able to evaluate text and differentiate between likely and unlikely sequences of words.

A system for assigning probabilities to sequences of words is called a language model.

Language Modeling

For language models, words can be represented as vectors of numbers. That's helpful because neural networks are programs that map vectors of numbers to other vectors of numbers.

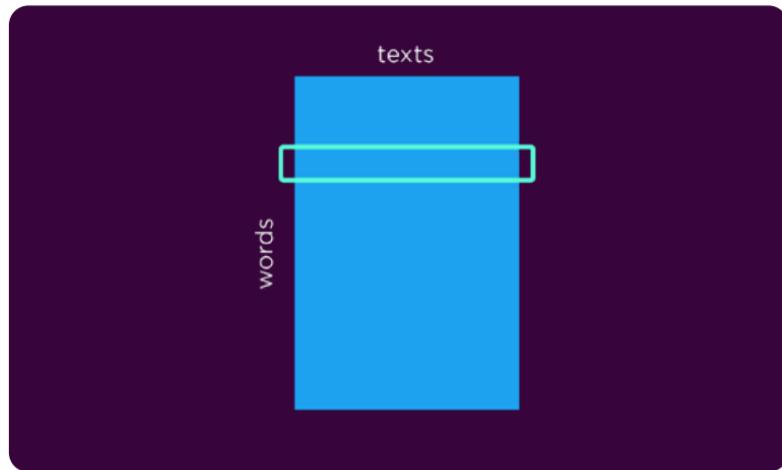
So if we can represent words as numeric vectors, that might provide a foundation for using numbers to do various kinds of language-processing tasks. More specifically, we'll want vectors to be similar to the same degree that the words being represented have similar usages.

The most basic form of this idea is to make a kind of numerical dictionary. Every word in English is assigned a numeric vector “definition.”

Thomas Landauer pioneered creating numeric vector “definitions” for words in the 1980s. He and his colleagues showed that embedding captured a lot of interesting knowledge about words and their relationships.

Landauer’s language embedding method is called **latent semantic analysis**. It’s called *latent* because it is finding hidden dimensions that characterize the data.

The starting point is a collection of texts, such as newspaper articles or encyclopedia entries. The set of distinct words in this collection becomes the rows of a matrix.



Each text of the collection becomes a column. Then, a cell in the matrix is filled in with the number of times the word for that row appears in the document for that column.

We want to create a vector “definition” for each word—one row for each word—such that two words that have similar meanings will have similar vectors. The assignment of words to positions in vector space is called an **embedding**.

The mathematical operation involved in latent semantic analysis is matrix factorization, or singular value decomposition.* It can be carried out pretty efficiently even on fairly large matrices—up to perhaps tens of thousands of rows or columns.

Embedding discovers general trends about word usages that can then be exploited to solve other problems. For example, vectors can be used to answer synonym questions from the Test of English as a Foreign Language—and at a level close to that of the average of applicants taking the test! Even the mistakes it made correlated with those of the test takers.

The singular value decomposition used in latent semantic analysis is closely related to an unsupervised neural network learning approach called **autoencoding**.

An **autoencoder** is trained on a set of vectors, like the rows of the words-texts matrix. It is trained to map a vector from its training set as an input to precisely the same vector as an output as accurately as possible.

Along the way, the vector is compressed, such that a smaller number of units is used to represent the original. The activation on the hidden units is an embedding of the input vector.

If the network has a hidden layer and only linear activation functions, the set of learned weights can be directly related to the singular value decomposition matrix approach. But formulating the problem as a neural network opens the door to more powerful approaches.

In 2014, word embeddings were taken to the next level. Researchers at Google figured out a way to train embeddings based on massive amounts of text—billions of words instead of tens of thousands.

Embeddings can pick up on subtle patterns of word usage, detecting synonyms and even solving analogy problems.

* The name refers to the fact that the decomposition can be used to determine how close a matrix is to being “singular,” or numerically unstable.

They used neural networks and created the embedding as the solution to an unsupervised learning problem. Using neural networks allowed them to use nonlinear transformations instead of the purely linear decomposition used in latent semantic analysis.

They defined a “good” embedding to be one where the embedding for each word is useful in predicting the embedding of the other words that appear near it in a large corpus of text.

Text Categorization

The goal of a pretrained word embedding like latent semantic analysis is to capture some of the patterns in a word’s usage. But instead of sequences of words, what’s captured is a vector, which neural networks can process more easily.

One of the many possible uses of a pretrained word embedding is text categorization. The problem of text categorization is to take a document and to figure out which of a set of predefined categories the text belongs to.

Many problems involve putting text into categories:

- ◆ Some companies want to organize their large collections of internal documents so that all documents pertaining to a specific project get filed together. In this case, the projects are the categories.
- ◆ Companies like Pinterest want to keep the content on their site upbeat and positive. To do so, they categorize the text that users enter into their site as being either safe or suspicious. Suspicious postings might get flagged to be hidden or removed.
- ◆ Text categorization also helps voice interfaces do the right thing. Amazon’s Alexa voice assistant classifies what it hears into categories that correspond to different kinds of actions it can do and information it can provide.[†]

Overall, by using pretrained embeddings, we can use generic text data to learn about the relationships between words. Then, we can use task-specific data to solve the problem at hand.

[†] Users can create their own apps for Alexa using the Alexa Skills Kit (ASK). In the programming interface, you include lists of example sentences so that Alexa can classify what it hears into the categories you define.

Try It Yourself

Follow along with the video lesson via the Python code:

[L16.ipynb](#)

Python Libraries Used:

keras.initializers.Constant: Incorporates a constant set of values into a neural network.

keras.layers.Conv1D: Creates a 1D convolution in Keras.

keras.layers.Dense: Creates a fully connected layer in Keras.

keras.layers.Embedding: Incorporates an embedding layer.

keras.layers.GlobalMaxPooling1D: Pooling layer for 1D convolutions.

keras.layers.Input: Builds an input layer.

keras.layers.MaxPooling1D: More local pooling layer for 1D convolutions.

keras.models.Model: Builds a neural network in Keras.

keras.optimizers: Optimizers used in Keras.

keras.preprocessing.sequence.pad_sequences: Adds padding information to data.

keras.preprocessing.text.Tokenizer: Turns a sequence of strings into discrete tokens.

keras.utils.to_categorical: Turns a set of activations into a one-hot categorical selection.

numpy: Mathematical functions over general arrays.

sklearn.datasets.fetch_20newsgroups: The 20 newsgroups dataset.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

Key Terms

autoencoder: A neural network trained on a set of vectors to map, as accurately as possible, a vector from its training set as an input to precisely the same vector as an output after internally representing the vector in a compressed form.

autoencoding: The behavior of an autoencoder.

embedding: Mapping objects into a vector space.

latent semantic analysis: An early form of word embedding that uses singular value decomposition to reconstruct a term-by-document matrix using several hundred dimensions. Invented by psychologist Thomas Landauer and colleagues in the 1980s, latent semantic analysis has been shown to capture some important properties of human language learning and use.

polysemy: The property of words having more than one meaning, making it difficult to know how to interpret words.

READING

Caliskan, Bryson, and Narayanan, “Semantics Derived Automatically from Language Corpora Contain Human-like Biases.”

Charniak, *Introduction to Deep Learning*, chap. 4.

Russell and Norvig, *Artificial Intelligence*, secs. 21.8.2 and 24.1.

QUESTIONS

1. How many words (types and tokens) are in the following sentence?

*How much wood could a wood chuck chuck
if a wood chuck could chuck wood?*

2. When we embed words in a d -dimensional space to try to capture patterns of similarities between the words, what goes wrong if we choose a d that's too small? What goes wrong if we choose a d that's too big?

3. This table lists the one-word names of a collection of occupations, along with what fraction of people with that job are female.

Job	Percent Female	Job	Percent Female
secretary	94.0	appraiser	45.3
typist	85.1	dentist	35.7
phlebotomist	75.0	rancher	25.8
telemarketer	65.3	announcer	17.7
packer	54.5	firefighter	5.1

In the paper “Semantics Derived Automatically from Language Corpora Contain Human-Like Biases,” the authors propose a way of measuring the “femaleness” of an embedding vector. Their idea is to take the word in question, w , and two sets of words, A and B . Compute the cosines between the embedding vector for the word w and the embedding vector for each of the words in A . Average. Do the same thing for the words in B . Subtract the two averages. Then, rescale the difference by the standard deviation of the cosines between the embedding vector for w and the embedding vectors for all the words in A and B . Select A to be a set of female words: *female, woman, girl, sister, she, her, hers, daughter*. Select B to be a set of male words: *male, man, boy, brother, he, him, his, son*. Then compute the correlation between the femaleness of each of the occupations with the percentage of females who hold these jobs. They got a correlation of 0.9, which suggests that the word embedding procedures capture the societal tendencies quite closely.

As an equation, the formula looks like this:

$$s(w, A, B) = \frac{\text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b})}{\text{std-dev}_{x \in A \cup B} \cos(\vec{w}, \vec{x})}$$

Use the GloVe embeddings loaded in for this lesson to replicate this analysis. Do you get similar results?

Answers on page 483

Text Categorization with Words as Vectors

Lesson 16 Transcript

Machine learning gives us a whole new toolkit for representing and acting on the meaning of words that goes far beyond what was achieved in earlier systems that used hand-written rules to process language. The starting point for the machine learning approach to understanding natural language is language embedding. The idea is to use machine learning to capture statistical relationships between words by associating each word with a high-dimensional vector.

Language embedding is used whenever we need to capture the meaning of words, which is an important component of processing language. Because language embedding maps words to vectors, it's also known as word2vec. There are many natural language tasks a computer might be able to help you with. In the Bayesian lesson early on, we talked about spam filtering; that's like asking the computer to look over your email to take out junk messages before you look at them.

You might want your computer to help catch tapas—sorry, typos. That's the problem of spelling correction or the interactive version known as autocomplete. Extracting text from images or audio can be very useful. These problems go by the names optical character recognition and automatic speech recognition. In all of these applications, the computer needs to be able to evaluate text and differentiate between likely and unlikely sequences of words. We call a system for assigning probabilities to sequences of words a language model.

In this lecture, we'll see how we can use machine learning to help construct accurate language models from data. We know that neural nets are great with low-level perceptual data, but what we'll see this time is how neural nets also came to be used for keeping track of what words mean and classifying text based on leveraging these language models.

In Lesson 14, we saw that two sorts of breakthroughs were made possible in vision by convolutional networks. First, convolutional networks were essential to greatly improving performance on a particular task—recognizing objects in images. But they also turned out to provide a fantastic representation for many other problems in image analysis. That is, we saw that we can train up a deep convolutional network for one problem and then transfer the internal representation it learns as input features for related problems.

A similar reusability breakthrough happened in language modeling, too. It turned out that the internal representations used to capture relationships between words could also be reused when solving a variety of other natural language processing problems. For reusing representations in language models, the crux of the idea is words can be represented as vectors of numbers. That's helpful because neural networks are programs that map vectors of numbers to other vectors of numbers.

So, if we can represent words as numeric vectors, that might provide a foundation for using numbers to do various kinds of language-processing tasks. More specifically, we'll want vectors to be similar to the same degree that the words being represented have similar usages. The most basic form of this idea is to make a kind of numerical dictionary. Every word in English is assigned a numeric vector definition.

A former supervisor of mine, Tom Landauer, pioneered creating numeric vector definitions for words back in the 80s. Using this idea of defining words with vectors, he started a company to automatically grade essays for standardized tests. Whether an essay was on topic was assessed based on how close its vector representation was to examples of ideal essays. His systems did a great job of characterizing meaning. And, the work suggested that even more useful systems could be created, if they could be built from bigger data sets.

He also made a joke when he talked about the idea. He'd say that, after much thought and much computation, he can now reveal the meaning of life. And, then he'd show a high dimensional vector. It's the numeric definition his unsupervised learning algorithm found for the word *life*. Maybe it's not quite the philosophical breakthrough he was teasing, but it does raise some interesting philosophical issues. Can meaning be captured through the manipulation of numeric vectors? You might be surprised to see how much about meaning can be captured this way.

Let's start by looking at Landauer's language embedding method, latent semantic analysis. It is called latent because it is finding hidden dimensions that characterize the data. Our starting point is a collection of texts. It could be a bunch of newspaper articles or maybe encyclopedia entries. The set of distinct words in this collection becomes the rows of a matrix. Maybe there are 10,000 such words. Each text of the collection becomes a column. A typical collection might be 3,000 documents. Then, a cell in the matrix is filled in with the number of times the word for that row appears in the document for that column.

LESSON 16 | TEXT CATEGORIZATION WITH WORDS AS VECTORS

TRANSCRIPT

So, we want to create a vector definition for each word, such that two words that have similar meanings will have similar vectors. The assignment of words to positions in space is called an embedding. I could make a three-dimensional embedding by placing words like *cat* and *dog* in physical space. Typical word embeddings use hundreds of dimensions, but the idea is the same.

One way to measure how similar two vectors are is using vector cosine; that's the normalized dot product. It measures the angle between the embeddings of the words, where the cosine similarity of a vector with itself is 1. *Cat* and *dog* have a small angle, which corresponds to a large cosine, close to 1. The cosine similarity of two vectors that point in completely unrelated directions is 0. And the cosine similarity of two vectors pointing in completely opposite directions is negative-1.

Anyway, related words like *cat* and *dog* will appear in a similar pattern of texts. So, their rows of the word-text matrix are likely to be vectors with high cosine similarity to each other. Now, whatever we do, let's make sure to avoid the biggest pitfall in machine learning—overfitting! Remember, we started with 3,000 texts. So, a row of the matrix for any word is a vector with 3,000 dimensions. We might do better if we approximate the information in the matrix using fewer parameters. If we limit the size of our representational space, we can combat overfitting.

The fundamental trade-off is that too few parameters is bad because that would tend to wash out important distinctions between words. Too many parameters is bad because that would retain noisy distinctions that aren't broadly meaningful; that's the essence of overfitting.

In practice, approximating our matrix of 3,000 dimensions with just 100 to 300 dimensions does a really nice job across a range of methods, data, and applications. But, how do we decrease the dimensionality of a matrix? To solve that problem, we can use a result that dates back to 1907. We can optimally approximate the dot products of vectors in high dimensions using vectors in lower dimensions.

Linear algebra tells us that any matrix can be rewritten as the product of three special matrices. The first gives a re-representation of the rows so that each row is independent of the others in terms of a set of internal factors. The second is a diagonal matrix giving the importance of each internal factor. And, the third gives a re-representation of the columns so each

column is independent of the others in terms of those same factors. The rows of the first of the three new matrices can be used as the vector definitions of the words.

And now, we're in a position to reduce the dimensionality of these vectors. The great trick comes from the fact that the second matrix tells us which factors are most important to capturing the original matrix. To get a lower dimensionality, we'll just drop all but the 300 most important factors. The solutions to all the approximation problems nest inside each other, so taking the best 300 dimensions is automatically the best you can do with 300 dimensions. Taking the best 100 dimensions is the best you can do with 100 dimensions, and so on.

We don't perfectly match the original word-by-text matrix. But we end up with the best 300-dimensional reconstruction of it. Latent semantic analysis then takes the rows of the word-by-300-factor matrix as the embedding. The mathematical operation we performed here is known as a matrix factorization, or singular value decomposition. The name refers to the fact that the decomposition can be used to determine how close a matrix is to being singular or numerically unstable. This decomposition can be carried out pretty efficiently even on pretty large matrices—up to, say, tens of thousands of rows or columns.

You can think of it as a kind of machine learning optimization, where it's trying to find the best low-dimensional reconstruction of a bigger matrix. It's an unsupervised method because it is learning a representation using only unlabeled data. Here, *best* is defined by our loss function, which is the total squared difference or squared error between our original matrix and our reconstruction.

Landauer and his colleagues showed that the resulting embedding captured a lot of interesting knowledge about words and their relationships. It discovers general trends about word usages that can then be exploited to solve other problems. For example, vectors can be used to answer synonym questions from the “Test of English as a Foreign Language” exam at a level close to that of the average of applicants taking the test. Even the mistakes it made correlated with those of the test takers.

These multiple choice questions consist of a question word like *credible* followed by four possible synonyms. The correct answer is the word that most closely matches the meaning of the question word. In this case, the

LESSON 16 | TEXT CATEGORIZATION WITH WORDS AS VECTORS

TRANSCRIPT

answer is *trustworthy*. To ask the language model its opinion, we have it look up the vectors for each of the words. Then, we have it select the word whose vector has the highest cosine with the vector for *credible*.

Now, let's transition to modern neural network approaches to the problem of language embedding. The singular value decomposition used in latent semantic analysis is closely related to an unsupervised neural network learning approach called autoencoding. An autoencoder is trained on a set of vectors, like the rows of the word-text matrix we have been discussing. It is trained to map a vector from its training set as an input to precisely the same vector as an output as accurately as possible. Along the way, the vector is compressed such that a smaller number of units is used to represent the original. The activation on the hidden units is an embedding of the input vector.

If the network has a hidden layer and only linear activation functions, the set of learned weights can be directly related to the singular value decomposition matrix approach. But, formulating the problem as a neural network opens the door to more powerful approaches. In 2014, word embeddings were taken to the next level. Researchers at Google figured out a way to train embeddings based on massive amounts of text—billions of words instead of tens of thousands. They used neural networks and created the embedding as the solution to an unsupervised learning problem. Using neural networks allowed them to use non-linear transformations instead of the purely linear decomposition used in latent semantic analysis.

They defined a good embedding to be one where the embedding for each word is useful in predicting the embedding of the other words that appear near it in a large corpus of text. The representational space was again one vector per word. The loss function can be decomposed into pieces as follows. At each position in the training corpus, we have a word and its neighbors. Here's the word *word* and five neighbors on either side of it in the corpus. For this example, the corpus is the transcript of this lecture.

We want to model the probability that a word *near* will appear in the neighborhood of word *word*, probability of *near* given *word*. We estimate this quantity by computing the dot products of the word's vector with the vector of every other word in the vocabulary. We exponentiate the values so they are positive. Then, we normalize by the sum so they add up to 1, summed over the entire vocabulary. We can interpret these numbers as

predictions—from the word’s perspective—of what neighbors the word expects to have. This approach is called GloVe. The name comes from global vectors.

The loss used in GloVe is a function of the probability the learned model assigns to the training data—which words appear near each other. To keep the numbers from getting too tiny, the method maximizes the log probability. Using logs converts a giant product of probabilities into a much more tractable sum of log probabilities over all of the word positions. We negate the quantity to make it into a loss. The higher the probability the vectors assign to the words that appear actually in the neighborhood, the lower the loss. A gradient-descent-based optimizer can figure out how to assign values to the vectors to minimize this loss.

The results are pretty impressive. We can ask it to arrange the words in 100-dimensional space based on a billion words of text. The resulting arrangement has some amazing properties. One of its striking features is its ability to solve analogy problems, which we talked about in Lesson 10 on machine learning pitfalls. The goal of a pre-trained word embedding like latent semantic analysis or GloVe is to capture some of the patterns in a word’s usage. But, instead of sequences of words, what’s captured is a vector, which neural networks can process more easily.

Let’s take a look at one of the many possible uses of a pre-trained word embedding. The problem of text categorization is to take a document and to figure out which of a set of pre-defined categories the text belongs to.

A lot of problems involve putting text into categories. A company wants to organize its large collection of internal documents so that all documents pertaining to a specific project get filed together. The projects are the categories. Companies like Pinterest want to keep the content on their site upbeat and positive. To do so, they categorize the text that users enter into their site as being either safe or suspicious. Suspicious postings might get flagged to be hidden or removed.

Text categorization also helps voice interfaces do the right thing. Amazon’s Alexa voice assistant classifies what it hears into categories that correspond to different kinds of actions it can do and information it can provide. Was that a question about a celebrity? The weather? Or a request to turn on the lights? Users can create their own apps for Alexa using the Alexa Skills Kit, also known as ASK. In the programming interface, you include lists of example sentences so Alexa can classify what it hears into the categories you define.

LESSON 16 | TEXT CATEGORIZATION WITH WORDS AS VECTORS TRANSCRIPT

As a concrete example of text categorization, we're going to work with a well-known dataset from the early days of the internet. The dataset is called the 20 Newsgroups Collection. It was gathered from a kind of discussion board that was popular back in the 90s. Each time a person would post, they would do so in a particular newsgroup category. The 20 Newsgroup Collection includes postings from 20 different newsgroups.

The full collection is available through the scikit learn library in Python, which is very cool. We just need to import the command, `fetch_20newsgroups`, and we can start using the dataset in experiments. To keep things simple, we'll just focus our experiment on two of the 20 newsgroups, corresponding to posts about baseball and about hockey. We load all the relevant data from the posts into a variable named `newsgroups`. When I looked through the data, I thought the heads and quoted text and footers really cluttered things up, so I removed them.

Now, I'm going to be talking a lot about words and I'll be talking about them in two distinct senses. In this brief passage from a writer by the name of Steve Martin, how many words does it have? Well, it's 13 words long. So, 13? But there are only 10 different words. So, 10? In the field of statistical natural language processing, researchers have different words for those two word concepts. We say the passage consists of 13 tokens but only 10 types. For example, *have*, *people*, and *way*, are types, and each of those types happens to have two tokens. The seven other types have one token each. So, a type is a word name, while a token is an appearance of the word. It's kind of like dalmatian is a type of dog, but the famous story has 101 tokens of this type.

Let's download a copy of the pre-trained embeddings produced by the glove embedding system. It was trained on 6 billion words of text—6 billion tokens. It includes embeddings with 50, 100, 200, and 300 dimensions. It's a big file and can take about five to 10 minutes to load. I'm putting the vectors in a dictionary called `embeddings_index`. Some Python code can read in the file and store each word vector in it. We'll work with the 100-dimensional embedding, stored in `glove.6B.100d.txt`. Give the index a word and the index will come back with the vector assigned for that word. Each word name—each word type—has a vector. There are 400,000 types in this collection. Since it had 6 billion tokens, that's an average of 15 tokens per type.

We're going to need to process the newsgroup documents into vectors. The deep learning library Keras gives us some tools for converting words into numbers. We set a maximum number of distinct words to 20,000 types and the maximum length of a sequence or document to 1000 tokens. We use a tokenizer to process the strings of letters corresponding to the newsgroup posts into sequences of numbers, one sequence for each word. For example, the word *goal* corresponds to a vector with 100 dimensions.

The mapping from words to their associated numbers is stored in `word_index`. We'll need that in a moment. We now have two different dictionaries or indices: `word_index` holds the mapping from words in the newsgroup postings to word IDs; `embeddings_index` holds the mapping from words to vectors. Note that they aren't exactly the same set of types. There could be words that appeared in the embedding training set that are not in the newsgroup postings, or vice versa.

We're now going to fill in the embedding vectors for the types in the newsgroup postings. We start with a matrix of zeros. The dimensions of this matrix are the number of types found in the newsgroup postings by the vector size in the pre-trained embeddings. We have a for-loop to go through the types appearing in the newsgroups. We use the `continue` statement to skip anything that appears less often than the `MAX_NUM_WORDS` highest frequency types.

Otherwise, we look up the embedding vector for the newsgroup word type and store the embedding vector in an embedding matrix. If the newsgroup collection has a type that was not given a pre-trained embedding, its embedding vector will be `None`. No vector will be filled in for that type. It will just remain a vector of all zeros.

Okay, time to prepare the data for the neural network we'll construct to process it. A lot of the cutting-edge art of using neural networks for solving machine learning problems is constructing an architecture well suited to the problem at hand. What we're trying to do here is map documents consisting of a sequence of words into one of two categories. The fact that our two categories happen to be baseball and hockey is probably not important to our choice of architecture. What is important is extracting as much higher-order meaning as possible from the documents.

In Lesson 6 on naive Bayes, we focused on bag of words representations that ignore word order. It's not a bad approach, especially when data is in short supply. But this time, we'd like to take advantage of word order.

LESSON 16 | TEXT CATEGORIZATION WITH WORDS AS VECTORS

TRANSCRIPT

That is, if words appear next to each other, that's something that might be relevant to deciding whether a document is about hockey or baseball. And, if we want to detect short phrases, then we've already talked about a tool that's really good at picking up on this kind of local structure.

In Lesson 14 on image recognition, we saw the idea of convolution, the technique that scans a window over an image to find informative features from the patches in the image. Those were 2D convolutions, but, here, all we need is a 1D convolution. A one-dimensional convolution extracts information from contiguous patches of words.

How many convolutional layers do we need? For this simple example, we'll use a sequence of three convolutional layers, which are marked with green, orange, and blue trapezoids. Each convolutional layer will have a kernel size of five, which is just another way of saying units will look at groups of five consecutive words, specifically five consecutive tokens. Each of the empty rectangles in the diagram contains a vector. The top row of vectors comes directly from the vector embedding by GloVe.

The first convolutional layer, in green, looks at windows of five input tokens. The second and third convolutional layers—orange and blue—each include the output of five convolutional filters applied to the previous layer. After three rounds of convolutions, we'll predict the category using a fully connected layer. It's really a lot like what we saw in the vision lesson except applied to sequences of words.

For this example, we'll make all the documents exactly 1000 tokens long using a pre-defined auxiliary function called `pad_sequences`. It extends the sequence with zeros so the sequence will be the determined size. `Pad sequences` chops off the end of a sequence if it is too long. We'll also convert the two output categories into a pair of values: we'll let baseball be 1,0 and hockey be 0,1. This representation is sometimes called one hot because each category is represented by a vector with a single 1 in it.

With the document instances and category labels ready to go, we can do a `train_test_split` to divide them into a training set of size 200 instances and leave the rest of the instances for evaluation. From the Keras library, we'll bring in subroutines to build our chosen network architecture. Since we want to train our network in two ways, we'll make a function called `train` with a parameter `pretrain` that we can set to `true` if we want to use a pre-trained embedding and `false` otherwise.

The only place the network depends on the pretrain parameter is in the embedding_layer. In the network with no pre-training, we set these weights to be trainable. In the network with pre-training, we initialize these weights from the pre-trained embedding and set them to be not trainable. We create the input layer from word sequences. The embedding layer takes the sequence input and creates embedding sequences. It maps the token identifiers into vectors. The idea is that each token is replaced by a corresponding vector.

Now, the layers that turn the words into vectors are in place. We'll make a sequence of three convolutional layers to process these vector sequences. Each convolutional layer creates 128 features at each position in the sequence. They have a kernel size of five. Each layer ends with max pooling. That means that if a feature is present in any position within a neighborhood of five around a given location, then it is considered present at that location.

We have the GlobalMaxPooling layer pool information across all of the windows. The 128 features that were constructed are finally collapsed to the output using a complete set of weights or a dense layer. We are using a softmax activation function here. Softmax automatically normalizes the activations across multiple units so that they sum up to 1. It is a way to convert arbitrary activations into a discrete probability distribution. That's a good match for the one-hot outputs we're training the network to learn.

Now, the structure of our network is in place. In standard machine learning terms, it is the representational space for the classifier we want to learn. We can set up a solver for Keras to use. I used the optimizer Adam and set a few of its parameters. As it turns out, I had to work hard to find good parameters for the Adam optimizer. In particular, when I set the learning rate parameter too small, learning was slow. When I set it too big, there was no learning at all; the accuracy rate was only 50%. Because there are only two classes, 50% is no better than random guessing.

When we hear about a successfully trained neural network, we should wonder how many unsuccessful attempts were made along the way. But think of it like this—I can do a trick, and it looks like it worked on the first try. It's a bit less impressive when I show you some of the failures I made along the way. The people who have mastered deep learning methods will often first go through a bunch of failures by running a hyperparameter search like we talked about in the previous lesson.

LESSON 16 | TEXT CATEGORIZATION WITH WORDS AS VECTORS TRANSCRIPT

Now that we've specified our network and solver parameters, we can specify the model's input units as sequence_input, and prediction output units of the network as preds. We want a loss function that encourages the network to make the outputs look like accurate probabilities. The cross entropy family of loss functions set the loss for our model to be categorical cross entropy, which is a standard way to make outputs sum to one. Then we run our machine learning algorithm for 100 iterations on the data, measuring accuracy using the validation data.

Running the deep learner, we've just constructed gets us a classifier that is accurate 83% of the time. Is that good? It's better than I was able to get with a few other classifiers we've talked about—Naive Bayes, decision trees, or nearest neighbors. I also tried one other algorithm to really focus the attention on the pre-trained word embeddings. I ran the same text categorization experiment with the same network, but with one tiny change—no pre-trained embedding.

The not-pre-trained version that had to learn its own embedding from the newsgroup data was equally accurate on the training set. Both got 98.5% correct. But not-pre-trained generalized less well to new postings. It got 76% correct compared to the 83% with the pre-trained version. In a sense, the pre-training lets the system do well with fewer tunable parameters. It's a kind of very-well-informed regularization.

And, that's the main lesson of pre-trained embeddings. We can use generic text data to learn about the relationships between words. Then we can use task-specific data to solve the problem at hand. The key idea of word embeddings is to define the meanings of words using vectors in high dimensional space. These embeddings are learned in an unsupervised fashion, thanks to techniques from linear algebra, like singular value decomposition. Word embeddings can also be generated using deep learning by using non-linear functions to generate representations that can predict nearby words. Either way, words that appear in similar contexts get assigned similar vectors.

Of course, one shortcoming of word embeddings is that they have to grapple with words like *duck*. Should we embed the word *duck* near *dodge*, or should we embed *duck* near *chicken*? Duck is a funny word, in more ways than one. This polysemy problem is one of the things that make language processing a continuing challenge for machine learning. Still, embeddings can pick up on subtle patterns of word usage, detecting synonyms and even solving analogy problems.

Finally, word embeddings trained on billions of words of English can help improve core natural language processing tasks like text categorization. Unsupervised learning and word embeddings are a key technology for creating automated tools for understanding natural language. But by themselves they are not great at producing language. So, in the next lesson, we will expand our toolbox of machine learning approaches to encompass language production. We'll see how to process vectors more flexibly and allow machines to compose text.

LESSON 17

DEEP NETWORKS THAT **OUTPUT LANGUAGE**

Have you ever created a story with a group of people by taking turns adding a word to the end of the story so far? Machine learning systems can output sequences through a similar process: Words are added one at a time, with the computer trying to find a sensible extension to what it already produced. In this lesson, you'll learn how neural networks can be trained to take their own text outputs as inputs so that they can process and produce longer and longer sequences of text.

Sequence-to-Sequence Problems

Since what is needed is a mapping from sequences to sequences, many language-processing problems are referred to as **seq2seq**. Seq2seq problems come in many forms.

Continuation is like repeated autocompletion. The input is a sequence of words, and the output is an extended sequence of words that reasonably continues what has been expressed so far. If the input is “I’m sorry, but you are going to have to...,” a continuation could be “come back and talk to me later”—or any of a zillion other possibilities. The more “natural” the continuation is, the better.

Text summarization is the problem of taking in a long sequence of words and producing a shorter sequence of words that conveys the same meaning. Using the previous sentence as input, a reasonable output might be “Text summarization makes sequences shorter.”

In image captioning, the input is a picture, while the output is a verbal sequence describing the image. If the input is a picture like this one, the output should be something like “A dog is wearing a Santa hat.”

Language translation takes in a sequence of words in one language, called the source language, and then reexpresses it in another language, called the target language.

Input: I’d gladly pay you Tuesday for a hamburger today.

Output: *Con mucho gusto te pagaré el martes por una hamburguesa me das hoy.*

All of these examples are problems that might train on millions of instances to produce an appropriate sequence of words as output.



Neural Machine Translation

Neural machine translation is a neural network approach to translation. It's an important problem whose output is a sequence. Because people have already translated material into different languages to share business documents or movies or books, there is a reasonable amount of training data out there, at least for some pairs of languages.

The European Union mandates that all official documents be translated into all 24 of the languages of the member countries. For example, if we want to train a system to translate Danish to Dutch—or any of the other 155 pairs—we'll find the data to do it.

It can be tricky to get large amounts of "parallel" text for less common language pairs, like if we want to translate Yiddish to Navajo. We could do what people have always done when they can't find translators who know both the source and target languages: translate through some more widespread language like English. It's not ideal, but that was how many early machine-translation systems worked, including at Google.

Google started offering the first iteration of their online translation service, Google Translate, in 2006. It used a phrase-based translation model. It was pretty useful. It served as a kind of online translation dictionary. But the fluency of the translations it produced wasn't good enough to result in readable text.

By 2016, the deep learning revolution was in full swing and Google was using machine learning methods to try to catch and pass their earlier effort. One of their first attempts to apply deep learning to translation matched the performance of their existing phrase-based translation. That's impressive, but probably not worth tossing the old system in favor of the new. The first deep learning attempt at translation was also too slow to deploy on a large scale.

There were several rounds of architectural improvements that increased accuracy and clever engineering ideas that increased speed. In September 2016, Google announced that they had a neural machine translation system that could replace the original Google Translate.

The differences were indeed impressive. For some pairs of languages, like English to Spanish and French to English, overall performance was on par with that of human translators.

Accuracy and speed of neural machine translation continued to progress after it started to reach the public circa 2016.

Transformer Networks

Transformer networks—so called because they transform sequences of words into other sequences of words—have been remarkably successful at solving hard problems in language processing.

Variations on the transformer idea produce state-of-the-art translations. If we were to implement and run it on realistic data, the models would get very large and take a long time to train. Some of the best models are trained on billions of words of text and include about a billion trainable parameters. They can take the better part of a week to train on top-of-the-line processors.

A significant concern is whether the trend of creating bigger and bigger deep models is sustainable, even for the very largest companies.

OpenAI analyzed the computing resources to train a top-of-the-line machine learning model as a function of the year in which their results were published.

They found that from 1959 to 2012, computing resources for training a state-of-the-art machine learning model doubled every two years, roughly in keeping with increases in computer speed.

Starting in 2012, when AlexNet stunned the world with its performance in visual recognition, the trend accelerated. From 2012 until OpenAI’s analysis in 2018, the state-of-the-art machine learning models doubled in computing requirements every 3.4 months. Meanwhile, processors following Moore’s law doubled only every 18 months.

This trend is unsettling because it means that the only researchers that can work at the cutting edge are those at organizations with the largest computing resources. Even just the electrical energy needed to train a single model is becoming enormous.

One development that’s mitigating the trends toward resource-intensive computation with very few participants is that the results of these training processes are increasingly available as open-source models.

Open-source models allow a wider variety of participants to use the trained models to attack other problems; that is, we can incorporate the trained models into other networks, where we fine-tune their weights to solve new but related problems.

The GPT-2 Model

One of the best seq2seq models available in 2020, called GPT-2,* was announced in February 2019 by OpenAI—the same organization that identified the growing need for computer power. It uses a transformer architecture, and it was trained in an unsupervised mode to simply predict text.

One of the remarkable things about the GPT-2 model is that it solved many natural language-processing problems right out of the box! Without additional training, the model was shown to produce state-of-the-art performance for a wide range of language tasks, including

- ◆ recognizing names and nouns in text,
- ◆ answering questions,
- ◆ making sentiment judgments,
- ◆ reasoning about simple scenarios, and
- ◆ predicting the next word (which is the task it was explicitly trained to do).

By repeated applications of next-word predictions, the network can solve the continuation problem presented in the beginning of this lesson.

The GPT-2 model was so good at story generation that its authors were concerned about releasing the model to the public out of fears that people with bad intentions would flood the internet with plausible-sounding fake text that would make it difficult to separate truth from fiction.

The model consisted of 1.5 billion parameters, but for the public, they released smaller—though increasingly powerful—versions in four stages during 2019.

With bigger models comes better understanding of facts about the world expressed in language.

* GPT-2 stands for *generative pre-training, version 2* because it was OpenAI's second model that generates text by pretraining on a large corpus.

Try It Yourself

Follow along with the video lesson via the Python code:

[L17.ipynb](#)

Python Libraries Used:

encoder: Organizes data for network transmission.

json: Reads an encoded JSON object.

model: Reads a pretrained neural network model.

numpy: Mathematical functions over general arrays.

os: Provides operating system access for manipulating files and directories.

sample: Makes choices randomly, given a probability distribution.

tensorflow: A deep learning library from Google. Supports differentiable programming, where derivatives of code are computed directly, making it simpler to implement meta-learning.

warnings: Set whether or not to display warning messages.

Key Terms

attention: Reweighting of inputs to a network with the goal of enhancing some of them to improve recognition accuracy.

recurrent network: Neural network where the output of some group of units is fed back into that same group of units, resulting in an activation loop.

seq2seq: Neural network trained to produce output sequences from input sequences. Examples include language translation, continuation, and text summarization.

subnetwork: A collection of nodes and links that can be repeated in a larger network. A convolutional filter in computer vision is an example.

transformer network: Neural network structure that transforms sequences of items, such as words, into other sequences of items, using attention.

READING

Charniak, *Introduction to Deep Learning*, chap. 5.

Russell and Norvig, *Artificial Intelligence*, chaps. 23 and 24.

Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin, “Attention Is All You Need.”

QUESTIONS

1. Neural machine translation is more “willing” than older phrase-based translation systems to do what?
2. A recurrent network is one that produces outputs that it can also consume, processing information in a kind of loop. How are these networks trained?
3. One of the amazing things about transformer-based models is that they can be used to solve a variety of natural language-processing problems without retraining. How can you use GPT-2 to evaluate the “femaleness” of job names from the word embedding exercise? Do you get higher correlations than you obtained with word embeddings?

Answers on page 484

Deep Networks That Output Language

Lesson 17 Transcript

On long family car trips, we used to make up stories together. Everyone in the car would take a turn adding a word to the end of the story so far. Machine learning systems can output sequences through a similar process: Words are added one at a time, with the computer trying to find a sensible extension to what it already produced. In this lesson, we'll talk about how neural networks can be trained to take their own text outputs as inputs so they can process and produce longer and longer sequences of text. Since what is needed is a mapping from sequences to sequences, such problems are referred to as seq2seq.

Seq2seq problems come in many forms. Continuation is like repeated autocompletion. The input is a sequence of words and the output is an extended sequence of words that reasonably continue what has been expressed so far. If the input is "I'm sorry but you are going to have to..." a continuation could be "come back and talk to me later." Or, any of a zillion other possibilities. The more natural the continuation, the better.

Text summarization is the problem of taking in a long sequence of words and producing a shorter sequence of words that conveys the same meaning. Using the sentence I just said as input, a reasonable output might be: Text summarization makes sequences shorter. In image captioning, the input is a picture, while the output is a verbal sequence describing the image. If the input is a picture like this one, the output should be something like, a dog is wearing a Santa hat. Language translation takes in a sequence of words in one language called the source language and then re-expresses it in another language called the target language. All of these examples are problems that might train on millions of instances to produce an appropriate sequence of words as output.

Let's dive into a neural network approach to translation, which has been called neural machine translation. It's an important problem whose output is a sequence. Because people have already translated material into different languages to share business documents or movies or books, there is a reasonable amount of training data out there, at least for some pairs of languages. The European Union mandates that all official documents be translated into all 24 of the languages of the member countries. For instance, if we want to train a system to translate Danish to Dutch, or any of the other 155 pairs, we'll find the data to do it.

LESSON 17 | DEEP NETWORKS THAT OUTPUT LANGUAGE TRANSCRIPT

It can be tricky to get large amounts of parallel text for less common language pairs, like if we want to translate Yiddish into Navajo. We could do what people have always done when they can't find translators who know both the source and target languages—translate through some more widespread language like English. It's not ideal, but that was how many early machine-translation systems worked, including at Google.

Google started offering their first online translation service, Google Translate, in 2006. It used a phrase-based translation model. It was pretty useful. It served as a kind of online translation dictionary. But, the fluency of the translations it produced wasn't good enough to result in readable text. By 2016, the deep learning revolution was in full swing and Google was using machine learning methods to try to catch and pass their earlier effort. One of their first attempts to apply deep learning to translation matched the performance of their existing phrase-based translation. That's impressive, but probably not worth tossing the old system in favor of the new. The first deep learning attempt at translation was also too slow to deploy on a large scale.

There were several rounds of architectural improvements that increased accuracy and clever engineering ideas that increased speed. In September 2016, Google announced that they had a neural machine translation system that could replace the original Google Translate. The differences were indeed impressive. For some pairs of languages like English to Spanish and French to English, overall performance was on par with that of human translators. But, the first language pair Google rolled out using the neural machine translation approach was Chinese to English.

Here's an example from their announcement post, to give you a sense of the improvement in quality. In the first column, I put a human translation of a Chinese sentence. It reads, "Li Keechong will initiate the annual dialogue mechanism between premiers of China and Canada during this visit and hold the first annual dialogue with Premier Trudeau of Canada." The phrase translation says: "Li Keechong premier added this line to start the annual dialogue mechanism with the Canadian Prime Minister Trudeau two prime ministers held its first annual session."

This sentence is pretty typical for state-of-the-art translators before deep learning. The word *premier* doesn't quite fit and the transition between Trudeau and two prime ministers doesn't work at all. You can tell that the original sentence must have been about starting a dialogue between Canada and China that takes place every year, but it's tough to tell much more than that.

The phrase translation stayed pretty close to the ordering of the ideas in the original. I colored different chunks of in the sequence: red, blue, orange, black, green, magenta. You can see that the human translation deviated in two significant ways from this ordering. The black chunk from the middle got moved to the end. A new yellow chunk was also added. The human translator stays close to the original but is open to moving things around to make the meaning clearer.

Now, let's look at the neural machine translation: "Li Keechong will start the annual dialogue mechanism with Prime Minister Trudeau of Canada and hold the first annual dialogue between the two premiers." It's much more fluent. The details seem clear. Everything fits. The translation moves the green chunk to the end where it flows a lot better. That's similar to what the human translator did, separating the black and green chunks because they are awkward right next to each other.

That was the state of neural machine translation when it started to reach the public circa 2016. Accuracy and speed continued to progress after that point. So, how do we formulate the problem of mapping between languages as a machine learning problem? And, how do we go about training a neural machine-translation system in particular?

The neural networks we've talked about in previous lessons have had a fixed set of inputs and a fixed set of outputs. So, if we're translating English to Spanish, we'd pick a maximum sentence length. For this example, let's start small and say our maximum sentence length is eight words. Our basic network might look something like this. The white rectangles represent words, so they'd be groups of input and output units with one unit per word in the vocabulary. To represent the word *dog*, it would have a 0 in every position, except a single unit corresponding to the word *dog* would be turned on. The blue rectangle represents all the processing layers of the network, whatever the number and structure of those layers might be. At this point, we're not going to worry about the details, just the high-level connectivity.

If we wanted to translate the English sentence, "Active dogs eat" into Spanish, we would turn on the appropriate units in the input. We'd also have to pad the input so it would consist of exactly eight tokens since these networks have fixed sized inputs and outputs. I'll use a generic STOP token to signal that the sentence is done. During training, we'd set the output units to represent the target Spanish sentence—*Los perros activos comen*.

LESSON 17 | DEEP NETWORKS THAT OUTPUT LANGUAGE TRANSCRIPT

That's roughly, "The dogs active eat." In Spanish, adjectives often follow the noun they modify. And, Spanish nouns usually get articles even in situations where they aren't needed in English.

So, here's our generic network. It's fine, as far as it goes. I mean, with enough layers and enough training time and enough data, it should be possible to learn any mapping we could want from input to output. But we're definitely making things hard by not choosing an architecture more suited to the problem. For one thing, we already have a trick for making it easier for the network to process words, at least on the input side, using embeddings. So far, the words in this network are represented by vectors of length 10,000 or so, if that's the size of the vocabulary. In particular, words are being represented merely as one hot vectors that indicate nothing more than which word they are encoding.

Lower-dimensional vectors would allow the network to capture relationships between words, as we saw in the previous lecture on word embeddings. So, let's modify the network to let it take advantage of word-to-word relationships using embeddings. I've added some dark blue boxes that convert the one hot representation of words into more informative embedded vectors for each word, shown in pink.

Each pink vector that results is specific to the input word, with maybe 100 or so dimensions per vector. The dark blue processing boxes carry out identical computations using a shared pattern of weights. Each box maps its one-hot representation of input word into its corresponding embedding vector. I call these identical boxes subnetworks, because they are mini neural networks themselves.

We can train the embedding subnetworks as part of training the translation, or we could pre-train the embedding subnetworks, using the ideas from the previous lesson, and then install the weights. If we pre-train, the weights here would be frozen and not change while we are training the translator. Using pre-trained frozen weights means the network is less adaptive to the project at hand, but it can be trained with less data.

Now, we have a network that has the meaning of each individual word encoded as a vector. Next, we need a vector representation of the meaning of the combination of all the words, at each point in the sentence. We would like this representation to be position independent, so it can be processed the same way no matter how long the sentence has been so far. We'll leverage the idea of creating subnetworks that can consume their own inputs.

The idea of a recurrent network is for the output of some group of units to be fed back into that same group of units. Let's start off looking at the first word of the English sentence. The embedding computation produces an embedding vector, depicted here as a pink box. The green vertical box is a state vector, which captures the meaning of the sentence so far. It starts off simply with a bunch of zeros so that there's a consistent initial value. The arrows indicate the flow from the input, at the bottom, to the output, at the top. The green state vector and the pink embedding vector are both inputs to a subnetwork, shown as a light blue box. That light blue processing box produces two outputs. One output is the translated word at that position.

At first, we haven't seen the English sentence yet, so the output should just be padding, like the control token STOP.

But the second output is a new state vector that captures the revised meaning of the sentence incorporating the additional word, and that's part of what makes this network structure so interesting. The green state vector is no longer the bunch of zeros we started with. And, because we have a new green state vector for the meaning of the sentence, we can apply the blue subnetwork to it when processing the next word.

The input and output words both advance from right to left, processing the words in sentence order. The sentence network takes as input the new word and the state vector produced in the previous iteration. We run the outputs through the network a second time. Then we advance the inputs and outputs forward again, and repeat, step by step, until the entire sequence has been processed. The reason this network architecture is called recurrent is because of the loop from the light blue box back to the green vector. Output vectors are going around and around.

As a kind of processing machine, there's no problem with this kind of recurrence. It's really just like a loop in a standard programming language. But how can we train such an odd network? All the networks we've seen so far have inputs that flow to outputs with no loops in them. How can we compute a gradient that is dependent on itself?

The trick is to unfold the recurrent network by replicating the piece that takes its own inputs.

LESSON 17 | DEEP NETWORKS THAT OUTPUT LANGUAGE TRANSCRIPT

Then the loops are laid out in space instead of in time. Unfolding lets us go from a single green vector to eight green vectors, one for every step of the computation. And unfolding lets us change from a single light blue box, to eight light blue boxes. And like the dark blue boxes that produce the embedded vectors, the eight light blue boxes are copies of the same subnetwork. During training, if a weight is changed in one of the subnetwork copies, it is changed in all of them. It's trying to learn a single computation that can be applied at every iteration. It's a very similar trick to the convolutional patches we talked about in Lesson 14—one set of weights is used in multiple places in the network.

Now, we have a network architecture that is position invariant; it's doing the same thing to all of the words so it doesn't have to learn to carry out the same kinds of computation independently at each sentence position. Another thing to note is that we're not padding the network anymore. Instead, the same neural machinery is applied to each word in the sentence. We need some kind of markers to notate the beginnings and ends of sentences, but position invariance means we use the computation no matter how many words we have, so there's no need to make the network a fixed size. The same network can be used to process sentences of varying lengths. We can always just make more copies of the light blue and dark blue computational boxes to accommodate additional words.

Okay, this architecture gets us beyond a network that only translates sentences of a fixed or limited size. And, there's another powerful idea that boosts performance substantially. We just need a better way to propagate information forward during processing. Let's take another look at the structure of this network. The network needs to produce the Spanish word *comen* as a translation of the English word *eat* in the context of a plural word like *dogs*. Take a look at how far the relevant information needs to travel to produce the correct translation. It has to be passed along by all of the green vectors. The circled green vector, in particular, has to pass along all of the information about the English sentence, which is then deconstructed to form the output words of the Spanish sentence.

Since the green vector has to capture the meaning of an entire sentence, it is sometimes referred to as a thought vector. That's a bit too anthropomorphic for my tastes, but it is a vector representation of the entire sentence. And the same light blue box needs to be able to unroll all possible sentences. It's a lot to ask of any box, even one that's light blue. Wouldn't it be nice if the light blue box could peek back at some of the previous vectors?

Well, it can't peek back at all of the previous vectors because the number of previous vectors varies with sentence position. We don't have a network structure that can take five vectors as input some of the time and 10 vectors some other time. We need a single structure that we can apply across contexts. If we separate the networks by the number of inputs they take, we'd lose our positional independence property.

Hmmm. Maybe we could use a trick like we did in the bag-of-words representation of texts. There, we'd collapse a bunch of vectors into one by adding them together. That would make the vectors a more uniform size, which is good. But it would also smoosh everything together. What we really want when we're producing the Spanish word *comen* is to look back specifically at the English words *dogs* and *eat*. We'd like to enhance the vector produced right after *eat*, and maybe the one right after *dogs*. We'd also like to de-emphasize the other word vectors. Once we've decided on how much emphasis each word's vector will get, we can add them together. That would be very useful information to bring to bear when selecting *comen*.

Here's how we can enhance the vectors we think the machine learner needs to focus on, while de-emphasizing other vectors. I've added two new kinds of vectors and two new kinds of subnetworks to the picture. The cyan vectors are the outputs of the recurrent network blue boxes. The orange vectors are a weighted sum of the green state vectors from the English part of the sequence. The rounded magenta rectangles near the top are ordinary trainable subnetworks. These subnetworks combine one vector intended to capture the current input word and one vector intended to capture the context to produce the current output word.

The lavender trapezoids are a new kind of computation. They compare the state vector for the current output coming from right underneath of them to the state vectors from the English part of the sequence coming in from the side. The closest vectors to the green vector from below are enhanced and the others are subdued. The orange vector is the combination of the state vectors from the left weighted by which are most similar to the vector from below. This weighting is referred to as attention in the deep learning community because of the way it selectively gives attention to, and enhances, some of the input.

LESSON 17 | DEEP NETWORKS THAT OUTPUT LANGUAGE TRANSCRIPT

It's worth taking a slightly closer look at what's going on in the lavender trapezoids—the attention selectors. It is a fixed computation; it has no trainable weights. The computation blends together the green vectors from the left and from the bottom. We'll call the vectors from the left encoding vectors since they are constructed during the process of absorbing the input sentence. They are labeled $v1$ through $v4$. We'll call the vector from the bottom w , the decoding vector since it is created in the process of generating the output sequence.

In one attention model, we compute the dot product of each encoding vector with the decoding vector. More similar vectors have a higher dot product. We exponentiate each of these dot product values—which greatly magnifies large values relative to the small ones—and then normalize these exponentiated values by their sum, making the values sum to 1. This operation is sometimes called a soft max because it roughly gives the highest dot product a value of 1 and all of the other dot products a 0.

Finally, the encoding vectors are averaged together using these softmax weights to produce the output context vector, shown in orange. The weights that combine the encoding vectors together are shown as attention weights. We then sum the encoding vectors, with each scaled by its encoding weight. Although this combination operation has no tunable weights of its own, its output depends on how the v and w vectors are computed. As such, it has a big impact on what weights are learned elsewhere in the network.

The overall network structure is referred to as a transformer network because it is transforming sequences of words into other sequences of words. Transformers have been remarkably successful at solving hard problems in language processing. Variations on the transformer idea produce state-of-the-art translations. If we were to implement and run it on realistic data, the models would get very large and take a long time to train. Some of the best models are trained on billions of words of text and include a billion or so trainable parameters. They can take the better part of a week to train on top-of-the-line processors.

A significant concern is whether the trend of creating bigger and bigger deep models is sustainable, even for the very largest companies. OpenAI analyzed the computing resources to train a top-of-the-line machine learning model as a function of the year in which their results were published. They found that from 1959 to 2012, computing resources for training a state-of-the-art machine learning model doubled every two years, roughly in keeping with increases in computer speed.

Starting in 2012, when AlexNet stunned the world with its performance in visual recognition, the trend accelerated. From 2012 until OpenAI’s analysis in 2018, the state-of-the-art machine learning models doubled in computing requirements every 3.4 months. Meanwhile, processors following Moore’s law doubled only every 18 months. This trend is unsettling because it means that the only researchers that can work at the cutting edge are those at organizations with the largest computing resources. Even just the electrical energy needed to train a single model is becoming enormous.

One development that’s mitigating the trends toward resource-intensive computation with very few participants is that the results of these training processes are increasingly available as open source models. Open source models allow a wider variety of participants to use the trained models to attack other problems. That is, we can incorporate the trained models into other networks, where we fine tune their weights to solve new but related problems.

As an example, we’ll play around with one of the best seq2seq models available in 2020. This model, called GPT-2, was announced in February 2019 by OpenAI—the same organization that identified the growing need for computer power. It uses a transformer architecture, like the one we’ve been discussing for neural machine translation. It was trained in an unsupervised mode to simply predict text.

At a time when other unsupervised language models were being named for Muppets—Elmo, Bert, Grover—the name GPT-2 was a boldly boring choice. It stands for Generative Pre-Training, version 2 because it was their second model that generates text by pre-training on a large corpus. One of the remarkable things about the GPT-2 model is that it solved many natural language processing problems right out of the box. Without additional training, the model was shown to produce state-of-the-art performance for a wide range of language tasks, including recognizing names and nouns in text, answering questions, making sentiment judgments, reasoning about simple scenarios, and, of course, predicting the next word, which is the task it was explicitly trained to do.

By repeated applications of next-word predictions, the network can solve the continuation problem I mentioned in the beginning of the lecture. The GPT-2 model was so good at story generation that its authors were concerned about releasing the model to the public, out of fears that people with bad intentions would flood the internet with plausible sounding fake text that would make it difficult to separate truth from fiction.

LESSON 17 | DEEP NETWORKS THAT OUTPUT LANGUAGE TRANSCRIPT

The model consisted of 1.5 billion parameters, but for the public, they released smaller, though increasingly powerful versions in four stages during 2019. Let's write some Python code to use all four of the released models for generating text. That will let us see how the changes in capacity related to the quality of the text produced.

The code for GPT-2 is maintained in a GitHub repository, so we'll clone ourselves a copy into Colab and install it so we can play with it. Then, we download the four different pre-trained models that OpenAI made available, each roughly double in size from the previous. The GPT-2 code depends on a few other libraries we need to import.

Okay, a lot of what's going on here is just setting up the right connection to OpenAI's code. This code does not build the network or train the network, so it's not that interesting from a machine learning standpoint. It's just providing access to the trained GPT-2 models.

I'll highlight one or two of the important points. We set up a session for talking to the TensorFlow backend, which is where the high speed numeric computations take place. We also create a space for the output of the model to go. We checkpoint the TensorFlow backend so we can establish the link to our code. Once all of that is set up, we can send our text prompt to the model for processing. We pull out the output of the model and return the string.

Now, we can feed a text prompt like, "Learning about machine learning is kind of like..." and ask the 124 million parameter model to predict the next 10 words. Each time you run it, it will do something different, but here's an example it gave me: "Learning about machine learning is kind of like what you'd learn from an expert in forensic and" Not terribly insightful, but it's definitely syntactically valid.

Let's take a look to see if we can tell the difference between the four models. I took the prompt, "My first time visiting the ocean, I marveled at..." and asked each of the four models for a 20-word completion. The smallest model produced: "...how big it was and how tiny it was, but it was still not quite small." The text is syntactically correct and maybe even relevant, but it's incoherent. It's odd to describe the ocean as big and tiny and not quite small.

The second model produced, “...the darkness of this vibrant ocean and at the very rugged beauty of this majestic landscape.” I’d say that’s a little better. The jump from the ocean to landscape is weird, but otherwise it’s pretty good. The next larger model produced, “...how beautiful it was. Sleeping on a beach is a lot like sleeping inside a....” Okay, it’s not clear where that’s going, but there’s nothing particularly jarring about it. Finally, the largest model produced, “...the beauty of the ocean and how it was further from a civilization like my own, the only one....” It’s actually a little bit odder, although maybe more inspiring.

I encourage you to play around with the model and see what you can get out of it. It knows a lot about English and even quite a bit about the world. You can also play with one of the models online at the Allen Institute for AI’s website. That version is really fun because it provides you with the top 10 highest probability next words, acting like a supercharged autocomplete.

You can use the code in this lecture to experiment with your own autocomplete by repeatedly asking for a one-word continuation of a given sentence. I decided to see how much the models know about the colors of common objects. I asked for 20 completions for 100 sentences like, “The color of calamine lotion is *blank*” and “The color of an arctic hare is *blank*” without the blanks. I counted up the number of times the completion matched my own choice of color term. The four models performed quite differently. The smallest model only matched my choice 12 times out of 100. The two medium-sized models went up to 56 and 46 matches. And, the largest model matched my choice 71 times. With bigger models comes better understanding of facts about the world expressed in language.

Training a state-of-the-art model to output natural language requires massive amounts of time and data. But, pre-trained unsupervised models like GPT-2 are available and have proven to be adept at solving a wide variety of natural language tasks. Some machine learning researchers like Yann LeCun think that unsupervised learning is the key to building learning systems that can understand the human world. As he likes to say, “The revolution will not be supervised.”

LESSON 17 | DEEP NETWORKS THAT OUTPUT LANGUAGE TRANSCRIPT

And, because of the success of unsupervised “fill in the blank” models for text, similar unsupervised techniques have been devised for images, filling in the missing parts of photographs—like playing guess what’s behind the black rectangle. My tie. Just like in language, they learn everyday facts about common objects, like their probable colors and what other objects they tend to appear with. Similar tricks are being applied to video sequences—fill in the missing video frame—and the algorithms learn how visual scenes typically change over time.

Long before Yann LeCun’s pet slogan that the revolution will not be supervised, there was an old activist slogan that the revolution will not be televised. However, next time, we’ll begin to see how the unsupervised revolution of machine learning is also coming to your television—and your computer, and your smartphone—thanks to machine learning’s ability to generate new images. And given the new possibilities for creating images and video, the un-supervised revolution will be televised.

LESSON 18

MAKING STYLISTIC IMAGES WITH DEEP NETWORKS

We can think of the creative process as being a kind of game. One side is generating ideas—a generator. The other side is deciding if the ideas are any good—a discriminator. Machine learning can use this idea of generator and discriminator to create usable images. The generator's job is to produce lots of images until it produces one that the discriminator finds acceptable.

Three Approaches

This lesson examines two different approaches to generation and discrimination.

In the first approach, we have a specific target image, such as this pig. Our discriminator then becomes a metric that prefers generated images that look—pixel by pixel—most like our target image.

In the second approach, we use a pretrained classifier to recognize the type of image that is being produced. This approach is more like what people typically do: We recognize the general category of pigs, of which this photo is an example. So our discriminator in the second approach prefers images that would be recognized as being part of that pig category. Matching pictures at a categorical level is more complicated but allows a broader range of images to be produced.

In both approaches, the generator uses optimization to try to satisfy the discriminator.

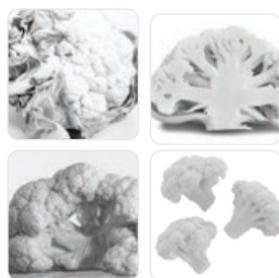
The next lesson examines a third method, where the generator runs without an explicit optimization step, while the generator and discriminator coadapt, allowing the method to learn its own discriminator on the fly.



Providing a Specific Target Image

Providing a specific target image is easiest. First, we take our pig picture, which is 64×64 pixels in grayscale.

Next, we need a process to generate each pixel for the pig image. Just for fun, let's use pictures of some other type of object—specifically, cauliflower—as picture elements to generate possible pig images. So, second, we gather many pictures of cauliflower. The cauliflower images are also 64×64 and in grayscale.



We're going to use our cauliflower images to build a single big pig image. We'll build a grid of 32×32 tiny images of cauliflower, where the overall result is supposed to look like our target image of the pig.

Our discriminator, to judge how well we're doing, will be defined as a simple kind of loss function. It takes any given large grayscale image built up from the 32×32 cauliflower images and compares it to a scaled-up picture of the pig.

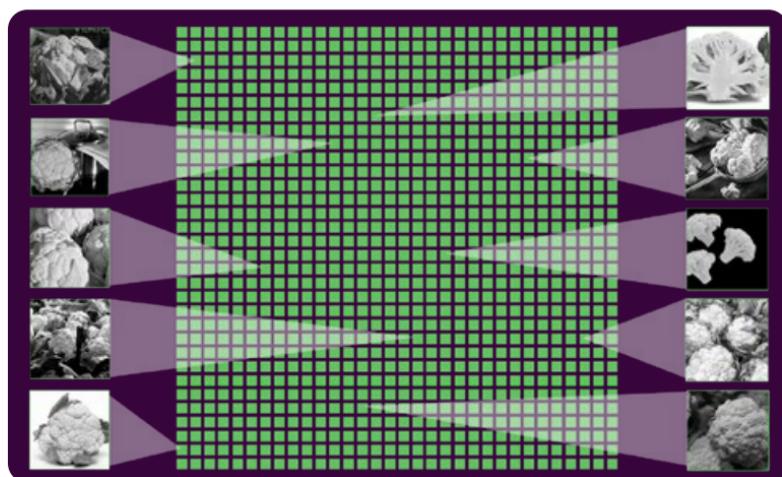
Then, it compares the image constructed out of cauliflower to the pig image. The comparison could be any of the distance or similarity metrics from [Lesson 08](#) about the nearest neighbor methods.

We'll use the squared Euclidean distance between the two images because it's simple and produces reasonable answers. That means we go through all the pixels in the image and compute the squared difference between that pixel in our generated image and the corresponding pixel in the scaled-up pig image.

The best image, with respect to this discriminator, is a scaled-up image of the pig itself, which will have zero loss—no other picture can do better.

What about the generator? Our generator will be making 32×32 grids, where each tiny image in the grid is one of 500 pictures of cauliflower in our dataset.

There are 1,024 tiny cauliflower pictures in the picture grid.



Finding an image from the generator that the discriminator would like is an optimization problem. One approach to this problem would be to just keep asking the generator for images and keep track of the one that the discriminator likes best so far. But that would be really, really slow.

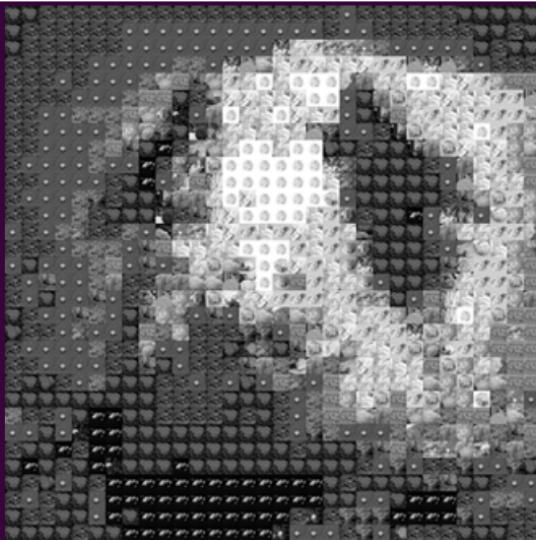
The set of possible images the generator can produce is 500 cauliflower choices raised to the power of 1,024 for the 1,024 pictures in the grid. That makes a number with roughly 2,700 zeros. We'd like to find the generated image with the best score, but we will not have time to find it this way.

Fortunately, the loss function used by our discriminator decomposes into 1,024 simpler optimization problems. The best composite image consists of the best choice of cauliflower image for each of the positions in the grid.

So, for each position, we can just loop through the 500 cauliflower images to see which one is closest to the corresponding patch of the scaled-up pig image.

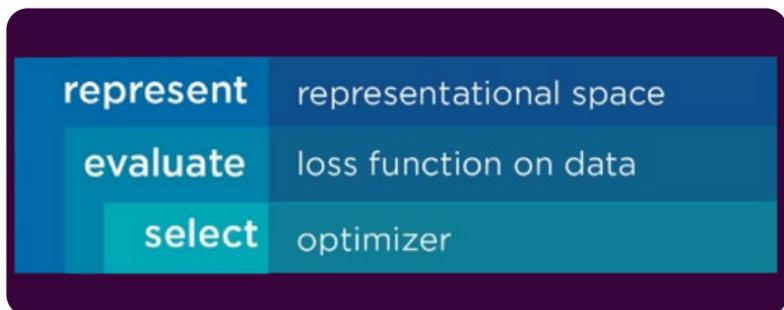
The code for solving this optimization problem runs quickly. Instead of checking all 500^{1024} grids, it just checks each of 500 images 1,024 times.

When we run it, we get a pretty good likeness.



This approach to image generation demonstrates the fundamental recipe for machine learning we've been using throughout this course:

- ◆ The representational space is the set of 32×32 grids of cauliflower pictures.
- ◆ The loss function is how far away the composite image is from a picture of a pig.
- ◆ The optimizer searches the space of cauliflower grids to find the best one.



There's even data, unlabeled, in the form of the set of available cauliflower pictures.

The images we can generate with this technique are limited only by our input images and by our choice of output target image. We get to be creative, but the machine learner doesn't. When we pick our target picture, at best we end up getting something very similar back.

To output a broader, more creative set of images, it would be nice for the discriminator to be a little more open-minded and allow the generator more freedom in creating acceptable images. For example, if the cauliflower pictures came together to look kind of like the pig Babe from the movie, the discriminator we have at the moment would dismiss it out of hand. Even though that output would look like a pig, it would not look like our target pig.

Using a Reference Image for Style Transfer

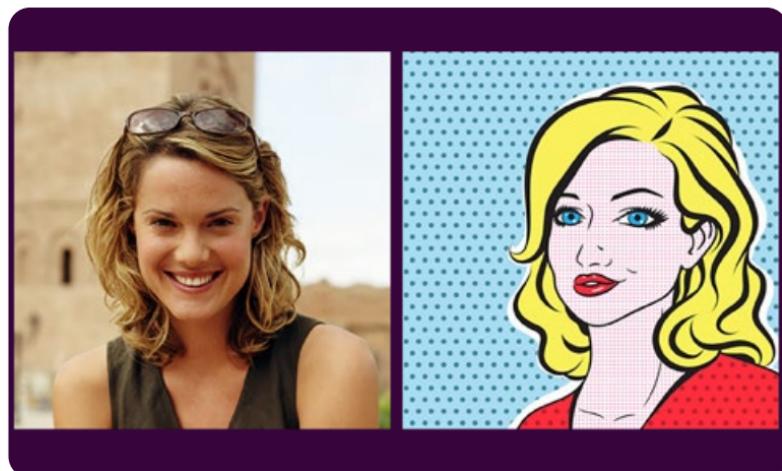
Instead of a pixel-based discriminator that prefers images that match a specific target image pixel by pixel, we can consider a discriminator that employs an image classifier. A classifier discriminator, as the name suggests, prefers images it recognizes as being in a specified class.

Since image classifiers have a high-level understanding of an image, we can use a classifier to intervene on images in a “semantic” way. Let’s look at an interesting technique called style transfer that leverages this idea using a reference image.

Imagine a photo of the Manhattan skyline rendered in the style of Van Gogh’s *Starry Night*, for example. Adding swirls and artistic flourishes can give a sense of Van Gogh’s style.

We can think of this image-generation problem again in terms of searching for an image that minimizes a loss function. But here, the loss function includes similarity to both the content image and to a stylistic reference image.

Let’s try an example involving two portrait images. The content image (left) is a photograph of a face. The stylistic reference image (right) is in Roy Lichtenstein’s pop art style.



We'd like an image of the same face, only with some of the features that make Lichtenstein's style so recognizable. Maybe the hair could be made more solid, with more use of strong primary colors. Maybe the face and other skin could be rendered with dots.

At a high level, we'll define a loss function that is the sum of three kinds of loss:

1. a content loss that's about making the new image recognized as being the same as the content image,
2. a style loss that's about making the features in the new image match those in the stylistic reference image, and
3. a local variation loss that discourages the new image from being too speckled by penalizing local variations in the image.

We'll then have the computer search for images that have low combined loss across these categories. These three losses are all summed together, and the optimizer is asked to create an input—an image—that minimizes the weighted sum of these loss functions. The effect is pretty interesting.



Iteration 1



Iteration 10

After the very first iteration, we can already see things happening. The smooth-looking features in the content image, such as the skin and hair, are beginning to be replaced with dots, similar to those in the stylistic reference image. By the 10th iteration, the artistic style of the content image is starting to look more like the stylistic reference image.

With the code supplied for this lesson, you can try various other combinations of content and style to see what happens. There are lots of interesting examples online, and you can also experiment with your own personal photos. It's fun!

Creating images at the level of a human artist is well beyond what can be done in 2020. But the representations discovered by deep neural networks make it possible to generate intriguing novel images.

Try It Yourself

Follow along with the video lesson via the Python code:

[L18.ipynb](#)

Auxiliary Code for Lesson:

[L18aux.ipynb](#)

Python Libraries Used:

keras.applications.vgg16: The trained VGG16 network for image recognition.

keras.applications.vgg16.decode_predictions: Turns VGG16 predictions into labels.

keras.applications.vgg16.preprocess_input: Converts raw images into the form expected by VGG16.

keras.backend: Provides access to lower-level Keras functionality.

keras.preprocessing.image: Prepares raw images for processing by a neural network.

keras.preprocessing.image.img_to_array: Converts an image into an array.

keras.preprocessing.image.load_img: Loads an image.

keras.preprocessing.image.save_img: Saves an image.

numpy: Mathematical functions over general arrays.

random: Generates random numbers.

scipy.optimize.fmin_l_bfgs_b: Specific optimizer available through SciPy.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

Key Terms

adversarial examples: Any examples that fool neural network classifiers.

computational graph: A representation that keeps track of the steps of a computation to support automatic differentiation.

gram matrix: A distance measure that captures similarities between objects.

tensor: An array of data that is a generalization of vectors and matrices. This course usually refers to a tensor as an array.

READING

Chollet, “How Convolutional Neural Networks See the World.”

QUESTIONS

1. It is helpful to decompose the process of creating new images into two subprocesses. What are they, and what do they do?
2. What are adversarial examples, and why are they of significant concern to machine learning and security researchers?
3. What are the strengths and weaknesses of the stylistic-image generator defined in the lesson? Here's another example for inspiration. If you take an image online of Van Gogh's *Starry Night* (like this: <https://images.fineartamerica.com/images/artworkimages/mediumlarge/3/starry-night-print-by-vincent-van-gogh-vincent-van-gogh.jpg>) and a similar-looking photograph of a city skyline (like this: <https://images.fineartamerica.com/images/artworkimages/mediumlarge/1/providence-skyline-nightscape-eddy-bernardo.jpg>), you can modify the stylistic-image-generation code to use these images and run it. What aspects of Van Gogh's style do you think it gets right, and what do you think it gets wrong?

Answers on page 484

Making Stylistic Images with Deep Networks

Lesson 18 Transcript

Creating an artistic masterwork isn't that hard. At least that's what one of my research mentors, David Ackley, used to say. You just need to be able to recognize when you have something great and to be really, really patient. You want to write a great song? Just keep banging out different sequences of notes and keep going until you've got a hit. In practice, of course, having enough time is what makes this idea impractical. There are too many random sequences of notes and most of them are terrible. If you are lucky, it might only take five years before you stumble on something moderately good. But most of us don't have that kind of time.

Nevertheless, there's the seed of a good idea in there. We can think of the creative process as being a kind of game. One side is generating ideas—a generator. The other side is deciding if the ideas are any good—a discriminator.

Machine learning can use this idea of generator and discriminator to create usable images. The generator's job is to produce lots of images until it produces one that the discriminator finds acceptable. In this lesson, we'll look at two different approaches to generation and discrimination.

In the first approach, we have a specific target image, like this pig. Our discriminator then becomes a metric that prefers generated images that look—pixel by pixel—most like our terrific target image. In the second approach, we use a pre-trained classifier to recognize the type of image that is being produced. This approach is more like what people typically do; we recognize the general category of pigs of which this photo is an example. So, our discriminator in the second approach prefers images that would be recognized as being part of that pig category. Matching pictures at a categorical level is more complicated but allows a broader range of images to be produced.

In both approaches, the generator will use optimization to try to satisfy the discriminator. In the following lesson, we'll examine a third method, where the generator runs without an explicit optimization step, while the generator and discriminator co-adapt, allowing the method to learn its own discriminator on the fly. But providing a specific target image is easiest, so let's start there.

LESSON 18 | MAKING STYLISTIC IMAGES WITH DEEP NETWORKS

TRANSCRIPT

First, we take our pig picture, which is 64-by-64 pixels in grayscale. Some pig, right? Next, we need a process to generate each pixel for the pig image. Just for fun, I decided we're going to use pictures of some other type of object—specifically, cauliflower—as picture elements to generate possible pig images. So, second, we gather many pictures of cauliflower. The cauliflower images are also 64-by-64 and in grayscale.

What we're going to do is use our cauliflower images to build up a single big pig image. We'll build a grid of 32-by-32 tiny images of cauliflower, where the overall result is supposed to look like our target image of the pig. Our discriminator, to judge how well we're doing, will be defined as a simple kind of loss function. It takes any given large grayscale image built up from the 32-by-32 cauliflower images and compares it to the scaled-up picture of the pig. Then it compares the image constructed out of the cauliflower to the pig image. The comparison could be any of the distance or similarity metrics we discussed in Lesson 8 about the nearest neighbor methods.

We'll use the squared Euclidean distance between the two images because it's simple and produces reasonable answers. If the images are represented by matrices A and B, we take the componentwise difference between them, square those differences, then sum them all up over the entire image. That means we go through all the pixels in the image and compute the squared difference between that pixel in our generated image and the corresponding pixel in the scaled-up pig image. The best image, with respect to this discriminator, is a scaled up image of the pig itself. The scaled up image of the pig will have zero loss and no other picture can do better.

What about the generator? Our generator will be making 32-by-32 grids, where each tiny image in the grid is one of 500 pictures of cauliflower in our dataset. There are 1024 tiny cauliflower pictures in the picture grid. Finding an image from the generator that the discriminator would like is an optimization problem. Our approach to this optimization problem would be to just keep asking the generator for images and keep track of the one that the discriminator likes the best so far. But that's just the idea of being super patient, and that would be really, really, really slow.

The set of possible images the generator can produce is 500 cauliflower choices raised to the power 1024 for the 1024 pictures in the grid. That makes a number with roughly 2700 zeros. We'd like to find the generated image with the very best score, but we will not have time to find it this way.

Fortunately, the loss function used by our discriminator decomposes into 1024 simpler optimization problems. The best composite image consists of the best choice of cauliflower image for each of the positions in the grid. So, for each position, we can just loop through the 500 cauliflower images to see which one is closest to the corresponding patch of the scaled-up pig image.

The code for solving this optimization problem runs quickly. Instead of checking all 500 to the 1024 grids, it just checks each of 500 images 1024 times. When we run it, we get a pretty good likeness. Many restaurants are looking for ways of replacing animals with vegetables on their menus. Maybe the cauliflower-pig could help them get started.

This approach to image generation demonstrates the fundamental recipe for machine learning that we've been using throughout this course. There's a representational space, which is the set of 32-by-32 grids of cauliflower pictures. There's a loss function, which is how far away the composite image is from a picture of a pig. And there's an optimizer which searches the space of cauliflower grids to find the best one. There's even data, unlabeled, in the form of the set of available cauliflower pictures.

There was a 16th century Italian artist who painted portraits of people consisting of all sorts of vegetables, fruit, flowers, fish, and other objects. We could follow his lead, using a wider variety of objects for our input data, and draw our pig that way. Or, we could use photos of our friends, or beloved celebrities, and combine those images to build any target image we want. The images we can generate with this technique are limited only by our input images and by our choice of output target image. We get to be creative, but the machine learner not so much. When we pick our target picture, at best, we end up getting something very similar back.

To output a broader, more creative set of images, it would be nice for the discriminator to be a little more open-minded and allow the generator more freedom in creating acceptable images. For example, if the cauliflower pictures came together to look kind of like the pig Babe from the movie, the discriminator we have at the moment would dismiss it out of hand. Even though that output would look like a pig, it would not look like our target pig.

But how could a discriminator judge an image in a meaning-based way? We could use image classifiers, which are trained to recognize images in terms of their category. So, instead of a pixel-based discriminator that

LESSON 18 | MAKING STYLISTIC IMAGES WITH DEEP NETWORKS

TRANSCRIPT

prefers images that match a specific target image, pixel by pixel, let's consider a discriminator that employs an image classifier. A classifier discriminator, as the name suggests, prefers images it recognizes as being in a specified class.

To save time, instead of building a classifier from scratch, we'll use a pre-trained classifier for a set of 1000 ImageNet classes. Specifically, we'll make use of the VGG16 network we also used in Lesson 14 on vision. It's a high-quality classifier that recognizes pigs and 999 other categories of objects.

We'll start with a generate-and-test approach. Let's say we want a picture of a flower, a daisy, say. We could have the computer generate an image, then run it through VGG16 to see what class the network thinks the image is—is it an ox, or a lampshade, or a daisy? We'll keep track of which image is recognized most strongly as being a daisy. Then, after many attempts, we can display the most daisy-like image.

Let's try it out and see what we get. We'll load some of our favorite packages, we'll import NumPy for working with vectors. From Keras, we'll use the VGG16 package. We'll also load a package we haven't talked about before called the Keras backend. It will let us tinker with VGG16 at a slightly lower level than we needed to before. We can then load the pre-trained VGG16 weights as our model. We need a few more packages: a Keras library for manipulating images, and the code VGG16 uses to preprocess images and make predictions.

We'll make a routine for producing a random image, suitable for presenting to VGG16. Let's call it `rand_image` and save its output as `img`. The result doesn't look like much to me. It's just a bunch of colored points. But what does VGG16 think it is? Looking at its top 10 predictions, the class it thinks fits best is envelope. But it only assigns 9% probability to that class, so it's really not very confident in its choice. It does seem to think it's some kind of fabric or weave, based on the other classes in the top 10; bib at about 9 % confidence, followed at some distance by velvet, binder, and handkerchief. Daisy is not in the top 10.

How daisy does VGG16 think the image is? The daisy class is number 985 in ImageNet, so component 985 of the model prediction is its belief that the image is a daisy. What's the score for this image? About 0.01%.

That's not very daisy-ish. In fact, it's on the anti-daisy-ish side. There are 1000 image classes, so a completely uncertain prediction would assign each class 0.1%. The daisy score for this image is merely one-tenth that amount.

But maybe it's good the score turns out so low. After all, it doesn't look like a daisy, does it? But even after 1000 attempts, the highest daisy score is still just 0.02%. That took about 10 minutes of run time. It seems pretty hopeless that a daisy picture is going to emerge any time soon. But we have a secret weapon that we're not using. A classifier based on a neural network, like VGG16, is not just a black box. We don't have to confine ourselves to asking how daisy the classifier thinks an image is. We can also ask it, what change to this image would you propose that would make you think it is more like a daisy?

In particular, that's precisely what gradients do. Gradients are slopes, measures of how much a change in one variable would result in a change in another variable. In this case, we want to know how a change in the image would impact the network's prediction. Neural network architectures are built specifically to allow gradients to be computed. We've used gradients to train neural networks. We ask, what change would you like to see to each of the weights so that the resulting network classifies images in a way that is more consistent with the training data? Then, we bump the weights in that direction and repeat. If we take enough small steps, we'll find a local optimum in weight space.

But for image generation, we don't want to change the weights. What we want to change is the input, the image, so it is the sort of input that produces a particular desired output. Neural network training and image generation are different problems. But they are not that different. Both use the same gradient-computing machinery.

Here's how we'll compute gradients to search for an input image. First, let's grab all the names of the VGG16 layers for easy reference. The target class number for daisy is 985, which I looked up in a set of Imagenet classes. We're going to need to represent the function from input image to class in a form that will let us take a gradient. The input is the input layer of the model. The word *tensor* here refers to a generalization of vectors and matrices. The set of input weights is a tensor. The output x is the activation on the prediction layer for the target class of daisies. The objective we want to optimize is exactly the daisy activation output. The more daisy the better.

LESSON 18 | MAKING STYLISTIC IMAGES WITH DEEP NETWORKS

TRANSCRIPT

We can now take the gradient of the objective with respect to the input. We use K.gradients with respect to our objective of the input tensor and then we store the results in the variable grads. It turns out that the magnitude of the gradient will be very tiny. So, to help boost things along, we'll use a trick of normalizing the gradients. It's not very elegant, but it will save us a lot of computation time.

Now, we link the image input with the objective and gradient. We're ready to optimize the image. We'll keep going until the image is 99.99999% daisy. That should do the trick. It'll be our answer true. Now, we'll run an iteration, putting the image img in and computing the objective value and the gradient. We'll print the objective value and then add the gradient to the image to move it toward greater daisyhood.

Here's how it runs. Once again, we start off with a daisy score of 0.01%. But even one update brings us up to 0.08%, which is four times more daisyish than the 0.02% we got by random generation and test. In 16 iterations, it has created an image that convinces the VGG16 network to within machine precision that it is 100% that daisy. Perfect!

Okay, let's unveil the most daisy daisy! Huh. It looks pretty much like what we started with. Well, no, not quite. There's streaks in the speckled background that kind of look like petals coming out of a flower, maybe? The image is indeed visibly altered, but it's not the sort of thing anyone would confidently declare to be a daisy. And yet, here's this world-class object-recognition system saying the image is absolutely positively 99s sure that it's a daisy, and nothing else. Interesting.

There are three things to learn here. One is that VGG16's understanding of objects is, well, alien. It can reliably classify photographs, but it is also picking up on cues in images that are very different from what matters to people.

But it's also worth noting how quickly the gradient approach was able to create the image. It really is a powerful way of doing optimization. The objective function wasn't what we wanted, in that an image can score very well on the metric it was given without looking like a daisy to us, but the optimizer quickly did its assigned job of finding something that the discriminator would like.

A third thing to learn is that the network became very convinced the image was a daisy after really minimal changes to the overall initial image. What if we had started with something other than random speckles that the network thought looked a little like an envelope? What if we had started with an image that it recognized as something else?

Let's try an example. I downloaded an image of a goldfish and I asked the program to turn it into a goldfinch, the bird. The optimizer ran for four iterations and produced this image. It's the same picture we started with, which VGG16 thought was a goldfish. But now the network sees it as 97% goldfinch. Here are the before and after photos for comparison. They are, for all human intents and purposes, indistinguishable. And, it's not like we fooled it into seeing a goldfinch slightly more than a fish. Its second and third choices are also songbirds. It thinks this photo is a bird and it's 97% sure that that bird is a goldfinch.

But why is VGG16 97% sure of itself, when, from a human perspective, it is so clearly wrong? When such a generally accurate system says it's 97% sure, we don't expect such a huge, nearly total, miss! No one in machine learning is entirely sure why the networks act this way. But they seem to be paying more attention to fine textural details than people would. If you look really closely, you can see little speckled streaks, kind of like what appeared in the fake daisy image. These patterns are probably the cue that the network is latching onto to imagine it is seeing a bird. I believe that a person could study these images intensely and start to recognize the cues. But it would take a lot of practice.

Computer security researchers refer to these sorts of images as adversarial examples because they are created explicitly to fool a classifier. For instance, experiments have been performed with adversarial examples to show that self-driving cars can be tricked into recognizing a stop sign as a speed limit 45 sign. Now, I'm not aware of any attempts to deliberately fool a real self-driving car but given the reality of malicious activity on other computer systems, we have to become prepared for all kinds of malicious activity.

Such examples show why it's hard to produce human-recognizable images from scratch by exploiting image-classification networks. While almost every goldfinch image is recognized as a goldfinch by the neural network, not everything that glitters like a goldfinch really is a goldfinch.

LESSON 18 | MAKING STYLISTIC IMAGES WITH DEEP NETWORKS

TRANSCRIPT

But the more our image generation is constrained, the more it can make better images. Since image classifiers have a kind of high-level understanding of an image, we can use a classifier to intervene on images in a semantic way. Let's take a look at an interesting technique called style transfer that leverages this idea using a reference image.

Consider a photo of the Manhattan skyline rendered in the style of van Gogh's "Starry Night", for example. Adding swirls and artistic flourishes can give a sense of van Gogh's style. We can think of this image-generation problem again in terms searching for an image that minimizes a loss function. But here, the loss function includes similarity to both the content image and to a stylistic reference image.

Let's try an example, looking at two portrait images. Our content image will be a photograph of a face. Our stylistic reference image is in Roy Lichtenstein's Pop art style. We'd like an image of the same face only with some of the features that make Lichtenstein's style so recognizable. Maybe the hair could be made more solid, with more use of strong, primary colors? Maybe the face and other skin could be rendered with dots?

Let's start off at a high level before we get caught up in code. We'll define a loss function that is the sum of three kinds of loss: a content loss that's about making the new image recognized as being the same as the content image; a style loss that's about making the features in the new image match those in the stylistic reference image; and a local variation loss that discourages the new image from being too speckled by penalizing local variations in the image. We'll then have the computer search for images that have low combined loss across these three categories.

Let's take a look at each of these loss categories in turn. There's a lot of details, so I won't go through everything at a low level. But you have access to the code, so you can look through it, run it, and get a sense of why it is structured the way it is. The content loss is the simplest. We're comparing a base image, the face, with the combination image, the new image being generated. The comparison takes the difference between the two, squares them, and sums up those differences. That's precisely how we measured content loss when we made a pig out of cauliflower.

There are two important differences, however. First, the `K.sum` and `K.square` functions might seem a little strange. We are not just using the standard Python built-in functions for these operations.

That's because we're not actually calling sum and square, we're building an explicit representation of the computation that uses sum and square. We need an explicit representation because we need to be able to take the gradient of the loss computation with respect to the image.

K.sum and K.square create a kind of computational graph that keeps track of what computations need to be made. The deep learning package can then run these computations forward as normal, or the package can run the computations backwards as part of the derivative computation that figures out how the outputs change the inputs. You can't run normal code backwards, so these specialized routines are used. The gradient computation itself is hidden inside the deep learning package. On the outside, the code looks almost normal.

The other important difference between what we're doing now and the cauliflower-pig example is that we're not going to compare the images at the level of pixels. That's too literal. We want the system to have a bit more artistic license, but without falling into the pitfall of dreaming of our electric images in a way that is so alien that we don't see the connection to real-world objects at all.

So, we're going to compare the images at the level of one of the internal convolutional layers of the neural network. Which layer should we pick? Well, the choice is somewhat arbitrary, but it's also trying to solve a balancing act. In particular, different layers of the network capture different levels of abstraction in the analysis of images. The closer the layers are to the input of the network, the more these features capture surface aspects of the image, what colors are in what places. The deeper into the network we go, the more semantic the features appear to be; is there an eye somewhere in the upper right of the image?

The second layer of the fifth block of weights in VGG16 is a compromise between capturing features that are too low-level and features that are too abstract. The code extracts activations at this layer for the base image and the combination image. And, it uses these activations in the loss function. As for the style loss, we'd like to measure loss in a similar way. No one has noticed anything like an explicit artistic-sensibility layer being learned by VGG16. And yet, features that capture some aspects of style can be extracted from the network indirectly.

LESSON 18 | MAKING STYLISTIC IMAGES WITH DEEP NETWORKS

TRANSCRIPT

Here's how we're going to do it. We're going to look at the feature representation the network is using across a set of different layers. Specifically, we'll be using the first convolutional layer for each of the five blocks of convolutional layers in the network. We construct a feature vector out of these activations created by both the stylistic reference image and the new image. These are the vectors to be compared by our style loss.

At its heart, the style loss follows the same structure as the content loss: take the image being created and the target image and compare them by squared Euclidean distance—total least squares. This quantity is scaled, but it's the same loss structure as what we've seen several times already. The content loss and the style loss do apply to different feature vectors. And style loss also differs because it transforms its feature vectors into something called a Gram matrix first.

A Gram matrix is a representation that captures similarities between objects. In this case, we're comparing the patterns of activation of a specific convolutional filter across the image. The style loss is minimized when the two images have similar gram matrices across all the chosen convolutional layers. Style loss wants the patterns of convolutional filter activations to match, meaning that the same low-level features activate together in the same regions.

The final component of the loss is due to local variation in the image being created. It's not a function of the content image or the stylistic reference image. This loss's goal is to penalize the image for being too speckled; essentially, charging it for rapid changes in color between neighboring pixels.

Here's how it works. It takes the image and chops off the first column. Then, it takes another copy of the image and chops off the last column. It lines those two images up and calculates the pixelwise squared difference between the two copies. It sums up those differences, shown here in black. The same operation is repeated by deleting the first and last rows to compare things in the vertical direction. A completely solid image would have no local variation variational loss because the shifted images would still match perfectly. A speckled image would have high local variational loss because every pixel is next to a pixel of a different color. Basically, the more that the pixels form large solid colored regions, the lower the loss.

In code, x represents the created image; x has its first and last row removed and subtracted; x then has its first and last column removed and subtracted. Both are squared so that both negative and positive differences are penalized. We slightly enhance differences by raising them to a power of 1.25 before summing. These three losses—content loss, style loss, and local variational loss—are all summed together. The optimizer is asked to create an image, an input that minimizes the weighted sum of these loss functions.

The actual code looks pretty similar to what we did when we were trying to make a daisy. We have Keras compute a gradient of the loss function, where the input image is the variable. Then, we repeatedly update the image in the direction of the gradient and repeat. The effect is pretty interesting. The content image is on the left, and the stylistic reference image is on the right.

After the very first iteration, we can already see things happening. The smooth skin tone in the content image has been replaced with red dots, similar to those in the stylistic reference image. In fact, there are dots of colors everywhere in the picture, including on the hair, background and clothes. Tiny patches of yellow are visible, presumably to match the style of the yellow hair in the stylistic reference image. More yellow is visible now. Unfortunately, it doesn't limit the added yellow patches just to the hair.

With the code we supplied, you can try various other combinations of content and style to see what happens. There are lots of interesting examples online and you can also experiment with your own personal photos. It's fun. Ten Iterations took me about 20 minutes to run on colab.

Creating images at the level of a real artist is well beyond what we can do in 2020. But the representations discovered by deep neural networks make it possible to generate intriguing novel images. A powerful generic approach we've seen is to have a generator that makes images, a discriminator that judges them, and then for the generator to optimize its output with respect to what the discriminator is looking for.

The choice of discriminator leads to different kinds of images. Semantic discriminators that pay attention to the meaning of the image can be defined by leveraging trained image recognizers. Stylistic transfer has also been used in other modalities like speech and video.

LESSON 18 | MAKING STYLISTIC IMAGES WITH DEEP NETWORKS
TRANSCRIPT

For example, neural networks can be used to create deep fakes in which actual video of a person is combined with video of someone else performing an action to make a video in which the person is performing the action. The artistic opportunities are immense, but there's also a risk of deep fakes being used to mislead or discredit.

Taking gradients with respect to images provides a mechanism for synthesizing new images made possible by a sharing of low-level stylistic features. In the next lesson, we'll see how neural networks can be used to create photo-realistic images. The idea is to make both the generator and the discriminator improve, by having them battle head-to-head for supremacy. This arrangement, which is generating images of people and places that have never existed, is called a generative adversarial network.

LESSON 19

MAKING PHOTOREALISTIC IMAGES WITH GANS

Training a network to become a great generator of images is kind of like training a student toward mastery of a subject. The student produces work, and the teacher needs to be very careful about how to set expectations of student performance. If teacher expectations are unrealistically high, the student can't meet them and won't learn. If teacher expectations are too low, the student has no incentive to improve and flounders unproductively. The teacher and student will need to coadapt: While the student is learning, the teacher's expectations increase, and in response, the student's performance rises to meet the increased expectations.

Generative Adversarial Network

The generator and discriminator you will learn about in this lesson coadapt. As the generator gets more and more adept at making realistic images, the discriminator ratchets up its expectations so that the generator is being challenged to get better and better.

We can design deep neural networks that carry out this kind of coadaptation of a generator and discriminator using a design called a **generative adversarial network (GAN)**. GANs are *generative* in that they include a generator component; they are *adversarial* because the discriminator is trained to challenge the generator.

Putting this adversarial idea in the mix allows the creation of new images that are photo-realistic. But GANs also open the door to what are sometimes called deepfakes.

The number and variety of GANs exploded starting in 2014, with as many as 500 types introduced in less than five years.

GANs have been used to solve a variety of image problems, especially in the context of generating photographic-quality pictures of nonexistent people, places, and things. GANs can also automatically conjure up a revised landscape after a building is removed.

Tools like Photoshop have long made it possible to modify photos manually, but GANs can make these modifications completely automatically.

In general, the simplest applications of GANs go like this: Given a collection of images from some class, generate a new image that belongs to the same class. If the class is photographs of faces, it generates a new face. If the class is floor plans of bedrooms, it generates a new bedroom floor plan.

If the class is cartoon characters, it creates new cartoon characters. Generating new characters could be beneficial in movie production. With the advent of GANs, battle scenes could more easily be populated with detailed, realistic individuals, each different from all the others.

The GAN insight is that we can pit the discriminator and the generator against one another. We already know that the generator needs to be good at producing images the discriminator accepts. So we can use the output of the discriminator as a loss function for the generator—the more the generator produces images the discriminator likes, the better.

And instead of a static discriminator that the generator can outwit using only minor, low-level changes to the image, we can train the discriminator dynamically to do a good job of distinguishing real daisy photos—to use an example from the previous lesson—from the actual output of the generator.

Our coadaptation goes back and forth many times, retraining the generator each time against the current discriminator and then retraining the discriminator against the current generator, etc. The hope is that this back-and-forth process creates the right kind of moving target that will lead to steady progress.

With high-quality daisy pictures and sufficient training time, this approach would produce imagery that would be very difficult to distinguish from nature photographs.

What makes GANs so powerful and revolutionary is that they dynamically construct their own discriminators as part of the training process.

Creating and Catching Fake Photos

Ian Goodfellow introduced the use of GANs in 2014 at the University of Montreal along with deep learning pioneer Yoshua Bengio.

The photo shown here is not Ian Goodfellow. It was generated by a GAN accessible through the website ThisPersonDoesNotExist.com. Each time you visit the page, it creates a never-before-seen human face for you.

The quality of the images is amazing. Note how the colors of his eyes match and how the stubble on his chin matches his eyebrows and hair. It's quite hard to tell that you're not looking at a real person. But there are ways to tell.

GANs need a lot of image data to be trained, but they do not need labels on those images. GANs represent a powerful and flexible methodology, creating state-of-the-art images of everything from apples to zebras

If you look really closely at the background over his left shoulder, you might find it hard to imagine what's back there. It's some kind of bushes over grass, but the shape of the boundary doesn't quite make sense.

As image-generation technology improves, however, it may become impossible to spot glitches without sophisticated analysis tools. In all likelihood, it is the GAN technology itself that will provide the tools for catching fakes.



Try It Yourself

Follow along with the video lesson via the Python code:

[L19.ipynb](#)

Auxiliary Code for Lesson:

[L19aux.ipynb](#)

Python Libraries Used:

keras.backend: Provides access to lower-level Keras functionality.

keras.layers.Activation: Sets the activation function for a layer.

keras.layers.advanced_activations.LeakyReLU: Enables leaky ReLU activation.

keras.layers.BatchNormalization: Enables batch normalization on a layer.

keras.layers.convolutional.Conv2D: Creates a 2D convolutional layer in Keras.

keras.layers.convolutional.UpSampling2D: Creates the inverse of a 2D convolutional layer in Keras.

keras.layers.Dense: Creates a fully connected layer in Keras.

keras.layers.Dropout: Enables dropout normalization.

keras.layers.Flatten: Reorganizes array-shaped units into a flat vector.

keras.layers.Input: Builds an input layer.

keras.layers.Reshape: Reshapes a layer.

keras.layers.ZeroPadding2D: Pads an image array with zeros.

keras.Model: Builds a neural network in Keras.

keras.models.Model: Builds a neural network in Keras.

keras.models.Sequential: Builds up layers of a neural network in Keras.

keras.optimizers.Adam: The Adam optimizer, a popular method for finding weights of a neural network to minimize loss.

keras.preprocessing.image: Prepares raw images for processing by a neural network.

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

os: Provides operating system access for manipulating files and directories.

sklearn.model_selection.train_test_split: Splits a dataset randomly into training and testing sets.

Key Terms

generative adversarial network (GAN): A neural network approach that learns to mimic properties of a given set of training data by building one network for producing instances and one for recognizing whether an instance comes from the training data. The basis for most work on deepfakes.

Instantiate: In computer science, to create a concrete instance of a more general class.

latent semantic analysis: An early form of word embedding that uses singular value decomposition to reconstruct a term-by-document matrix using several hundred dimensions. Invented by psychologist Thomas Landauer and colleagues in the 1980s, latent semantic analysis has been shown to capture some important properties of human language learning and use.

READING

Toews, “Deepfakes Are Going to Wreak Havoc on Society. We Are Not Prepared.”

Zhu, Park, Isola, and Efros, “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.”

QUESTIONS

1. Let’s say you want to generate realistic new images of cars using the generator-discriminator approach. What is likely to go wrong in each of the following approaches to creating a discriminator for carlike images? (a) A person hand-codes the discriminator. (b) Test how close the image is to a specific target image of a car. (c) Use a trained classifier for cars.
2. Teach yourself to spot GAN-generated faces until you can get five in a row correct:
<http://www.whichfaceisreal.com/index.php>.
3. For a dataset like MNIST, what do you think this lesson’s GAN generator would produce? Do you think it will invent images that look like digits but aren’t? Or will it simply create variations on the digits 0 through 9? Run the GAN code on MNIST digits to see. Tip: Using GPUs will greatly speed up the experiment.

Answers on page 485

Making Photorealistic Images with GANs

Lesson 19 Transcript

Training a network to become a great generator of images is kind of like training a student toward mastery of a subject. The student produces work and the teacher needs to be very careful about how to set expectations of student performance. If teacher expectations are unrealistically high, the student can't meet them and won't learn. If teacher expectations are too low, the student has no incentive to improve and flounders unproductively. The teacher and the student will need to co-adapt; that is, while the student is learning, the teacher's expectations increase, and, in response, the student's performance rises to meet the increased expectations.

The generator and discriminator we talked about last time are in a similar situation. As the generator gets more and more adept at making realistic images, the discriminator should ratchet up its expectations, so the generator is being challenged to get better and better. We can design deep neural networks that carry out this kind of co-adaptation of a generator and discriminator using a design called a generative adversarial network or a GAN. GANs are generative in that they include a generator component like we talked about last time. They are adversarial because the discriminator is trained to challenge the generator.

Putting this adversarial idea in the mix allows the creation of new images that are photo-realistic. But GANs also open the door to what are sometimes called deep fakes. The number and variety of GANs exploded starting in 2014, with as many as 500 types introduced in less than five years. A website was created called the GAN Zoo just to document all the different species of GAN that appeared. Some of the names are pretty punny—GANDI, VEEGAN, ELEGANT. One paper specifically described how to train your DRAGAN.

GANs have been used to solve a variety of image problems, especially in the context of generating photographic-quality pictures of non-existent people, places, and things. GANs can also make a person look older or younger.

Here's me and an aged version of me. I look a lot like my dad. Here's me and a regressed version of me. I look more like I think I look. And a bit like my son.

LESSON 19 | MAKING PHOTOREALISTIC IMAGES WITH GANS

TRANSCRIPT

GANs can automatically conjure up a revised landscape after a building is removed. Tools like Photoshop have long made it possible to modify photos manually. But GANs can make these modifications completely automatically. In general, the simplest applications of GANs go like this: Given a collection of images from some class, generate a new image that belongs to the same class. If the class is photographs of faces, it generates a new face. If the class is floor plans of bedrooms, it generates a new bedroom floor plan. If the class is cartoon characters, it creates new cartoon characters. Generating new characters could be beneficial in movie production. For example, scenes in the movie *Lord of the Rings* include battles with tens of thousands of separate characters. With the advent of GANs, movies could be more easily populated with detailed, realistic individuals, each different from all the others.

In the previous lesson, we tried to solve the problem of drawing a daisy. Our high-level strategy was to consider the creative process as consisting of two modules—a generator and a discriminator. The generator produces images and the discriminator judges them. To construct a discriminator, we started with a convolutional neural network that was already trained to a high degree of accuracy to recognize daisies and camels and screwdrivers and hundreds of other categories of objects. We then modified the image automatically using gradients to move it in the direction the network believed would be most daisy-like. The result was not daisy-like at all. At least, not to humans.

So, what went wrong? To a first approximation, we were not asking the network to create something more daisy-like in any absolute sense. We were merely asking the network to use low-level differences in the pixels to create something that was more daisy-like relative only to the other kinds of images the network happened to be trained on.

To say it another way, the VGG16 discriminator is only good at distinguishing daisy photographs from other typical photographs. But it has no idea what a daisy really is, so it can't tell the difference between a daisy photograph and a random field of pixels that includes some low level features found in daisy images. Simply put, it wasn't trained to do the higher-level discriminating we need.

Clearly, we need a better discriminator. But we don't know *a priori* how to tell the machine what the discriminator needs to be good at discriminating. The GAN insight is that we can pit the discriminator and the generator

against one another. We already know that the generator needs to be good at producing images the discriminator accepts. So, we can use the output of the discriminator as a loss function for the generator; the more the generator produces images the discriminator likes, the better.

The discriminator, meanwhile, is no longer going to sit around to be taken advantage of. Instead of a static discriminator that the generator can outwit using only minor, low-level changes to the image, we will train the discriminator dynamically to do a good job of distinguishing real daisy photos from the actual output of the generator. Our co-adaptation goes back and forth, many times, re-training the generator each time against the current discriminator and then retraining the discriminator against the current generator. The hope is that this back-and-forth process creates the right kind of moving target that will lead to steady progress.

To get the basic idea, let's work through how we can generate a very simple kind of artificial image. These three images each consist of 20 random grayscale values, constrained so that they increase in brightness from total black on the far left to complete white on the far right. For ease of reference, I'll call this kind of image a staircase because it's a sequence of steps from dark to light. Each staircase goes from 0 for black to 1 for white. But each staircase is slightly different from the others, in that the specific shade for each intermediate step can differ between staircases.

Here's a snippet of code that generates m staircases of size d . To start, a bunch of 0-1 uniform random numbers are arranged as an m -by- d matrix. The code then computes a running sum, also known as a cumulative sum or cumsum, across the rows of the matrix. NumPy refers to moving across the rows as traversing axis 1 of the matrix, so that's the parameter value we will use in the call.

So, now we no longer have a bunch of random numbers between 0 and 1. Instead, each row contains the running sum at each location. This procedure guarantees an increasing sequence, with a starting value that is somewhere between 0 and 1 and an ending value around 10.0. It's not a staircase yet, since a staircase goes from 0 to 1. We can rescale these values so they are in the right range. First, we grab the minimums or mins of each row, in a vector X . Since the rows are increasing, the min must be on the far left. We subtract off these mins.

LESSON 19 | MAKING PHOTOREALISTIC IMAGES WITH GANS

TRANSCRIPT

We then grab the maximums, the maxs, for each row. They are on the far right. We divide the vector X by the max values so everything is in the 0-1 range as desired. Then, we'll get a network to learn to produce these one-dimensional grayscale staircases. Staircases are a nice stand-in for the more complex two-dimensional color images that GANs typically produce. Once we've mastered them, we can graduate to photographic images.

Here, our goal is to train a network to produce novel staircases, using the artificially-generated staircases as examples. In the previous lesson, we tried using a recognizer as a discriminator and then iteratively improved a random image via gradient descent to generate an image that the recognizer believed was in the target class. That approach did not work for daisies. But staircases are much simpler. Maybe we'll have better luck with the gradient-descent approach here.

The simplest way to train a classifier to be a discriminator is to use just two classes: one class should be staircases, of course; the other class should be—non-staircases? But what does that mean? To train the classifier we need specific bad images to act as a foil against which the staircases can be judged. Let's take our alternative class to be scrambled staircases. That is, we'll generate a staircase, then swap the steps around so they aren't in sorted order. This process provides an informative contrast for staircases because individual steps are all of a type that could appear in a staircase picture, only the order is off.

We can generate random non-staircases by starting with a set of positive examples, X_{pos} , which are produced by running the function `make_pos(m,d)`. To this matrix, we can apply a random permutation to each row, using NumPy's `apply_along_axis` function. The resulting collection of negative examples is assigned to the variable X_{neg} .

Here's a row of the resulting matrix—a set of scrambled gray bars. For our classifier, we'll create a simple neural network using Keras. Our network has d equals 20 input units to correspond to the 20 steps. There are 10 hidden units that the network can use to compute. And, there are two output units, corresponding to staircase or not staircase.

The predictions in the output layer use softmax activation. Softmax accentuates the largest value in a vector, then normalizes the values to look like probabilities. That is, the values from Softmax are between 0 and 1 and sum up to 1.

We can train the network with 10,000 randomly-generated images, half staircase and half not. We train for 15 epochs. The resulting network has 100% accuracy in distinguishing staircases from non-staircases, which seems great. Now that we can accurately discriminate between staircase images and scrambled staircase images, let's use the gradient approach to turn a non-staircase into a staircase.

We'll take steps in the direction of the computed gradient until the image is recognized with over 95% confidence as a staircase. We want the staircase, which is represented by the vector `input_data`, to remain a valid image. To keep the values of the vector in the right range, we reset back to zero any values in `input_data` that have gone negative.

Now, let's use our learned staircase discriminator to try to help generate a staircase using gradients.

We start off with a random non-staircase, shown here on the top. The bottom image is the result. The highly accurate classifier recognizes that the top image is not a staircase. But it thinks the new output is a staircase, even though it is not. Check out the very dark bar near the middle, or the white bar three steps to its right. Those bars are out of place.

Let's look more closely at how the top image gets modified to become the bottom. On the left side, the steps are darkened all the way to full black. On the right side, the steps are brightened towards entirely white. It does seem as though the discriminator is paying attention to whether an image has darker steps on the left and brighter steps on the right. But it doesn't understand that all the steps need to be monotonic as we go from left to right—consistently changing in the same direction at every step.

Whatever rule of thumb the discriminator learned was pretty effective with the original dataset, yet it does not capture the real staircase concept we had in mind. So, let's try the GAN idea. It's adversarial, so we need to construct two networks. The discriminator network is like what we had before. It takes as input a vector of size d equals 20 representing the purported staircase. It has two fully connected or dense sets of weights leading to hidden layers, each of size d -over-2 or 10. Finally, the output layer is a single unit that should output 1 for a real staircase and 0 for a fake one.

Since we know this output value is between 0 and 1, we can use a sigmoid activation function. Sigmoids smoothly transition from near-zero values for negative inputs to near-one values for positive inputs, which guarantees

LESSON 19 | MAKING PHOTOREALISTIC IMAGES WITH GANS

TRANSCRIPT

the outputs stay in a zero-to-one range. The second network, the generator, is new, but it has a similar structure to the discriminator. In the non-adversarial generation process, we just used the discriminator's preferences to search for something the discriminator liked.

This network-structured generator maps a vector of noise into a staircase vector. The number of components of the noise vector can be thought of as a latent dimensionality of the image space—the more noise components, the higher the latent dimensionality. We'll use `latent_dim` to be a variable that specifies the number of random values we'll use as our input. You only need 18 random values to make a staircase of 20 steps, since the first value of the staircase must be set to 0 and the last value must be 1. I picked 50 for `latent_dim` so the network would have a large pool of random values to choose from.

The generator uses a dense layer of weights to map the chosen 18 random values to a hidden layer, where the number of hidden units is 2 times d or 40. The generator then has another dense layer of size d equals 20, then an output layer of size d equals 20. Since all of the outputs should be between 0 and 1, I set the activation function for the output layer to be a sigmoid of for each of these units.

The generator and the discriminator are separate networks, but they are also part of a grander whole. They link together through the generation of a staircase. The generator produces the staircase and then the discriminator judges whether the staircase is real or not. The discriminator can get its input from the generator, which it should classify as fake or from the training data, which it should classify as valid.

We'll use the Adam optimizer for training and we'll train for 30,000 epochs. The Adam optimizer is an adaptive algorithm introduced in 2015 that controls the process of gradient descent by optimizing the rate of learning at each step of the optimization process. Adam generally does a good job and it quickly became a default choice for training networks.

We'll give the networks a batch size of 128 examples in each epoch. We'll set up our training set X using only positive examples of staircases. We also create targets for learning—we use 1s for valid examples of staircases from the training set X and 0 for fake examples that come out of the generator. Next, we construct the combined network. We build the discriminator and set it up to be trained. We can then instantiate a generator and call it z , which will take noise inputs and produce images img .

We want to train the generator to make staircases that the discriminator cannot reliably distinguish from the real staircase images. In short, we want to fool the discriminator. The weights of the discriminator need to remain constant while we update the generator. If we did not hold the discriminator's weights fixed, gradient descent would solve the problem of fooling the discriminator merely by making the discriminator less accurate. So, we set `discriminator.trainable` for the combined network to be false. The discriminator assesses the generator's image output `im` and produces a judgment as to whether the image is valid. The combined network, end to end, connects the input `z` to the output called `valid`.

Now that the network is set up, here's how we train it. We've already seen two different strategies for training using Keras. The high-level strategy is to set up a network and some data and then call `fit` to do all the work of training. We trained a classifier for staircases via this high-level `fit` method. The low-level strategy is to define our objective directly in terms of operations on arrays of data, referred to as tensors, that Keras can manipulate. Then, we can have Keras take gradients of the objective with respect to the parameters. And, we can apply those gradients to update the parameters.

We applied the low-level gradient method to search for images of staircases that satisfied the learned classifier. But for GANs, we need a third, mid-level strategy that lets us interleave updates to different weights. To understand this mid-level strategy, it's important to understand the notion of a batch. To make deep networks train efficiently on modern hardware, we need to break up the training data into smaller units called batches. These batch units fit into memory and can therefore be processed much faster.

The high-level `fit` approach in Keras handles breaking the training data into batches, computing gradients, and making updates to model parameters all on its own. But a GAN involves two networks. So, to train a GAN, we need to use each batch in two different ways. We need to use it to adjust the weights of the generator, so it can make inputs that fool the discriminator. But we also need to train the discriminator to distinguish even the improved output of the generator from the real training data.

Using batches in these two ways means we need more detailed control over what happens during the processing of a given batch. We'll update our weights with respect to each batch of data, using a Keras function that updates the weights of a network based on a single batch of data. Using this `train_on_batch` function will allow us to implement our mid-level strategy.

LESSON 19 | MAKING PHOTOREALISTIC IMAGES WITH GANS

TRANSCRIPT

In each epoch, we create a batch of training examples. We make each batch of images from the training data by picking the indices of a set of training examples, `exs`, at random and then pulling out the corresponding instances as `imgs`. We also create an input set of random noise and feed it to the generator to get some artificial images we call `gen_imgs`. We train the discriminator to classify the real images as valid and the generated images as fake. `Train_on_batch` returns the loss resulting from the batch training.

Now that we updated the discriminator, we update the generator. The generator is part of the combined network. Given the noise input, we want to update the generator's weights to make the discriminator more likely to classify the fake images as valid. So, given noise input, we want to update the discriminator part of the combined network so the output label is fake. But we simultaneously want to update the generator part of the combined network so the output label is the exact opposite—valid.

If we run the code on positive examples of staircases, the GAN learns to produce staircases. After 20 iterations, the generated images are already showing a propensity for darker bars on the left and lighter on the right. But they are not yet internally consistent. After 90 iterations, the staircase structure is starting to emerge. You can see black bars consistently on the left and white bars consistently on the right. In between, things sometimes get a little muddled.

After 190 iterations, the staircase structure is much more consistently generated. The generator has learned to produce very good staircase facsimiles.

One way to measure how close the generated staircases are to being perfectly valid is the number of pairs of steps that are out of order. Since there are 20 steps, there are 20 times 19 divided by 2 equals 190 possible pairs of steps. In a valid staircase, 0 of these pairs are out of order. In a randomly permuted staircase, on average, half the pairs are out of order. That's 190 divided by 2 or 95 pairs.

Running the gradient approach improves things somewhat, resulting in an average of 48.4 pairs out of order. After 90 training steps, the GAN generator has improved things to an average of 18.6 steps out of order. After 190 steps—a few hours of running time—the GAN generator produces staircases with an average of only 8.4 pairs out of order.

Even eight pairs of steps is far from a perfect staircase. But keep in mind that we didn't have to give any explicit training to the learner about minimizing this quantity. It's just trying to make staircases that the discriminator cannot reliably distinguish from the real staircase images. We could have given it more processing time, but it already does a pretty good job.

With this technology in hand, can we generate some daisies? If we want to make daisies, we need images to train on. I made a dataset of 124 daisy closeups by searching online for daisy pictures that are freely shareable and then clipping out just the flower part using standard image-editing tools. I scaled all of the pictures down to 36-by-36 pixels because that's the smallest picture I had clipped out.

As we saw in Lesson 14 on vision, densely-connected networks—like we used in the staircase example—are not what we use for recognizing images. That's because dense networks don't have position invariance built in, forcing the network to have to learn to recognize items independently in each part of the image. So, a better choice would be convolutional layers, which do have position invariance built in. But this generator needs to act like an inverse convolutional network. That is, whereas a convolutional network allows for groups of neurons that map a patch of an image to a class, we need layers in the generator that produce patches.

In Keras, the inverse layer to the two-dimensional convolutional layer Conv2D is UpSampling2D. Our generator for the daisy case will make use of these UpSampling layers. Take a look at our example code for the details. With revisions to the discriminator and generator both in place, we can run our GAN approach on the daisy images.

You can get an idea of what the generator does by feeding it different patterns of random noise and seeing what comes out. After one iteration of training, the generator produces fuzzy splotches of green and blue and black. After 40 iterations of training, the generator has added yellow to the mix and we're starting to see the bright colors clumping in the middle of the frame. After hundreds of iterations of training, the network sorted out that there should be a yellow patch in the middle, white around it, and occasional flecks of green here and there.

Looking at the images, we can imagine that something daisy-related is happening. But it could also just be a bunch of fried eggs. It's still hard to tell. After thousands of iterations, the images have become less impressionistic and more photographic. The pictures are pretty clearly

LESSON 19 | MAKING PHOTOREALISTIC IMAGES WITH GANS

TRANSCRIPT

daisies now. The petals are in focus and the generator is pretty consistent. Not too bad, especially for low-resolution, 36-by-36 pixel images. With higher-quality daisy pictures and even more training time, this approach would produce imagery that would be very difficult to distinguish from nature photographs.

Ian Goodfellow pioneered the use of GANs in 2014 during his PhD at the University of Montreal, with deep learning pioneer Yoshua Bengio. The photo shown here is not Ian Goodfellow. It was generated by a GAN accessible through the website ThisPersonDoesNotExist. Each time you visit the page, it creates a never-before-seen human face for you. The quality of the images is amazing. Note how the colors of his eyes match, and the stubble on his chin matches his eyebrows and hair. It's quite hard to tell that you're not looking at a real person. As image-generation technology improves, however, it may become impossible to spot glitches without sophisticated analysis tools. In all likelihood, it is the GAN technology itself that will provide the tools for catching fakes. As they say, the only thing that stops a bad guy with a GAN is a good guy with a GAN.

In the examples we have seen so far, including the fake face, the generator networks are given noise patterns as input and then transform the noise into images. The use of noise is what allows the same network to produce a wide variety of outputs. But there's no reason we can't also train a GAN to use other images as input. For example, we could train the generator to use random photos of apples as their input and then generate photos that look like oranges. If we have a dataset of apples and a dataset of oranges, we could use the apples to generate oranges and the oranges to generate apples, training both generators and both discriminators simultaneously.

Wouldn't it be neat if the generators learn to transform the apples to oranges and vice versa so that they kind of look like each other? If we train the two GANs in parallel, there's no reason to expect any kind of relationship to develop between the two types of images. But there's a clever technique that links the two GANs, so they end up mapping between the two types of images. The technique is called a cycleGAN and it adds another component to the loss function used during learning.

Specifically, a cycleGAN trains two GANs, one for generating images of type A and one for generating images of type B. Each GAN has a discriminator and generator like before. But the loss function for each of these generators includes another element. It requires that plugging together two generators results in reconstructing the input image.

That is, suppose we feed a picture of type A into the type B generator, then take the resulting image from B and feed it to the type A generator. We should get the same picture of type A that we started with. To say it in mathematical language, the two generators should be inverse operations.

So, if the orange generator takes apple 1 and uses it to make orange 2, the apple generator should turn orange 2 back into apple 1. To make the network have this inverse property, we need to include a component in the loss function that compares the input apple to the two-stage output apple. That loss can be the pixel-wise comparison we used in Lesson 18 on style transfer.

An amazing observation is that adding this cycleGAN constraint when training the two GANs encourages the generator outputs to be as close to their inputs as possible. When the two GANs are locked in this tight dance together, each is focused on the other. Thus, the orange generator doesn't just produce an orange, it produces an orange that corresponds visually to the apple it was given.

The 2017 paper that introduced cycleGANs provides three different usage examples, each of which can be run in either of two directions. CycleGANs can be used for artistic style transfer, like turning a photograph into a Monet-style painting. Thus, cycleGANs provide another approach to the style-transfer problem we saw in the previous lesson. CycleGANs can go beyond style transfer to do object transfiguration, like turning a horse photograph into a zebra photograph. One thing I really like about the horse-zebra example is that the generator leaves the horse mostly intact and adds stripes. But it also dries out the grass and darkens the sky to better match the African landscapes where zebras are found.

CycleGANs can carry out rendering transfer, like taking a satellite image of a part of a city and turning it into a map view. These maps are also a sort of two-way street. That is, the learned networks also know how to turn a map into a corresponding satellite image, adding whatever color and detail is needed. So, it is not just changing a landscape, as in the horse-zebra transfiguration, but it's adding aerial landscape features to a map that didn't have any at all.

When we use machine learning to make images in the generator-discriminator framework, designing good discriminators is very hard. As we saw with daisies and staircases, a discriminator can seem to be very accurate on a fixed data set, while badly misclassifying examples produced by a generator. What makes all kinds of GANs so powerful and so revolutionary is that they dynamically construct their own discriminators as part of the training process. By creating a discriminator that is explicitly tuned to the current capabilities of the generator, the discriminator is able to slowly ramp up, forcing the generator to be better and better at reproducing the kind of detail of real images.

GANs need a lot of image data to be trained, but they do not need labels on those images. GANs represent a powerful and flexible methodology, creating state-of-the-art images of everything from apples to zebras. And GANs can also be used to generate other kinds of low-data, like sounds. In the next lesson, we'll look at networks that can extract language information from sound—the problem of speech recognition.

LESSON 20

DEEP LEARNING FOR SPEECH RECOGNITION

A milestone in the ability of machines to recognize human speech was reached in 2016. That's when researchers at Microsoft announced that their speech recognition system was achieving accuracy rates beyond human transcriptionists. Basic voice interfaces driven by machine learning have been broadly deployed since around 2011, and they have been improving ever since. Companies like Microsoft, Amazon, Apple, Google, and Baidu have all made important contributions to developing and disseminating speech recognition technology.

Limitations of Early Speech Recognition Systems

Given how common speech recognizers have become, it's worth noting the limitations of earlier systems to understand how things have changed.

There are three principal dimensions in which speech recognition systems have been limited.

- ◆ Some systems could only handle disconnected speech, with gaps between each individual word.
- ◆ Others were built only to recognize words from a limited vocabulary, such as the digits zero through nine or the names of cities on a train line.
- ◆ Still others are speaker dependent and have to be tuned to work with individual users.

Older systems improved to the point where they could overcome any two of these limitations. But it was machine learning systems introduced since 2011 that have been able to address all three challenges at once.

The history of speech recognition approaches shows us how this complex problem was decomposed into simpler problems that could be tackled separately. Later, deep neural networks were applied to each of the separate problems, resulting in rapid progress.

History of Speech Recognition Systems

The earliest work on speech recognition dates back to a digit recognition system built by Bell Labs researchers in 1952.

Computers were too slow back then to process speech information. As a result, the researchers built their specialized recognizer directly out of electronic components like vacuum tubes. The system extracted the strongest and second strongest formants—the frequencies with the highest intensities. Then, they compared the formants to a set of 10 templates, one for each digit.

The templates were kind of like specialized classifiers, but they were not built using machine learning algorithms. The templates were painstakingly created and tuned by people using trial and error. It was data-intensive work for people to do by hand.

The system they built was reported to have an accuracy of more than 90%—at least for one of the researchers who built it. For other people, accuracy was quite a bit lower.

The Bell Labs system suffered from all three of the limitations of traditional speech recognition systems. It was tuned for a specific person, it required gaps between words, and it could only handle the names of the 10 digits. It was a remarkable achievement, but it wasn't really ready for prime time.

Things improved gradually in the 1960s. Researchers attacked the problem of phonetic segmentation; that is, approaches were introduced to try to break connected speech into smaller units—words—so that the individual words could be recognized. Template patterns for words were introduced and could be matched against incoming speech sounds. Researchers realized how important top-down influences and context were in disambiguating sounds into the desired words.

Phonetic segmentation helped with connected speech and, to some degree, recognizing larger vocabularies. But since things were still primarily being done by hand, scaling up to more speakers and larger sets of words would have required prohibitively expensive amounts of expert human effort.

The 1970s brought faster machines and a new machine learning technology that greatly improved the scalability of speech recognition systems.

The technology is known as **hidden Markov models**, which belong to the Bayesian style of thinking about machine learning. The fundamental idea is to view the relationship between input and output backward. We build a model that goes backward from text words to speech sounds. The models are “hidden” in the sense that they include latent, or hidden, variables.

It's somewhat common for speech recognition systems to struggle with female and child voices, perhaps because of bias due to too few examples of these classes in the training data or because higher pitches might be harder for current algorithms to process consistently.

An ordinary, non-hidden Markov model is called a **Markov chain**, after Russian mathematician Andrey Markov.

Hidden Markov models were the best game in town before the first deep learning models began to be tried in speech recognition in around 2012.

After that point, hidden Markov models were no longer central to speech recognition, although they have remained a mainstay in specialized machine learning solutions to signal processing.

The Bayesian view says that we should think about the speech signal as being generated by the text-word sequence, even though we don't observe the text-word sequence. And that generative process can involve intermediate steps along the way. Those individual steps might be the words that make up the sequence, or they might be basic speech sounds, called phonemes, which make up the words.

Breaking Free of All Limitations

By 2011, hidden Markov models had become the dominant approach to speech recognition. These models could be used to produce systems that were free of any two of the three limitations of speech recognition systems, but not all three.

Hidden Markov model-based systems could produce recognizers for tens of thousands of words spoken by arbitrary speakers, as long as the words were isolated from one another—that is, they were still limited to disconnected speech.

Other systems could be used for dictation, transcribing natural speech as long as the system was trained for its particular user—that is, they remained speaker dependent. Many doctors used this technology to record patient notes.

Other hidden Markov models were used by banks in interactive systems where arbitrary people could call into a service and make requests by voice. But those systems needed strong top-down priors about what would be said—that is, they still needed a limited vocabulary.

Deep learning systems, starting in around 2011, made it possible to break free of all three limitations simultaneously.

Traditionally, speech recognition systems had been built to turn speech sounds to sentences through a series of intermediate steps:

- ◆ First, low-level speech sounds would be recognized into phonemes.
- ◆ Then, these phonemes would be coalesced into individual words.
- ◆ Lastly, the words would be massaged into the most likely sentences.

Each layer would send its best guess to the layer above it, but it would also transmit some degree of uncertainty, since language and sound are notoriously ambiguous.

Between 2012 and 2015, researchers discovered that they could improve on the state of the art for each of these transformations.

The overall progression through the three transformations remained the same, but the approach used to solve each of the three problem layers became neural networks.

- ◆ *For speech sounds to phonemes:* Convolutional neural networks are remarkably adept at treating a spectrogram—a plot of which sound frequencies are present in speech at each moment in time—as a kind of image and recognizing phoneme patterns in it.
- ◆ *For phonemes to words:* Recurrent neural networks can solve similar problems to those attacked by hidden Markov models and provide latent context for generating the words.
- ◆ *For words to sentences:* The combination of word embeddings and transformer networks does an astonishingly good job at distinguishing likely and unlikely sentences.

Replacing each component of the three layers of this system with an appropriate neural network leads to excellent speech recognition performance.

The boost in performance surprised Microsoft researchers when their system beat human transcriptionists for the first time in 2016. After working on the problem of speech recognition for decades, they didn't expect improvements to come so quickly. They hailed their accomplishment as a historic achievement.

Indeed, the benefits of neural networks for this problem really are remarkable. With a neural network approach, we can train a recognizer to go from speech sounds directly to words, without the intermediate step of phonemes.

Since 2014, there have been attempts to apply “end-to-end” training—speech sounds all the way to text words with a single network.

As of 2020, such end-to-end systems had been built, but they had not performed as well as the modular approach. But as the research community’s understanding of how to design and train neural networks continues to improve, new ideas might make end-to-end training for speech recognition viable.

Hidden Markov models capture the probabilities of sequences.

The Frontier of Speech Recognition

Being able to talk to machines and having them understand what we want would make the human-computer interface more natural and fluent. Computer scientists and engineers have been making progress on this challenge for 70 years. Nevertheless, the problem is definitely not solved.

The frontier of speech recognition has moved away from converting speech signals to text words. The bigger challenge is language understanding.

Being able to have a voice-based conversation with the computer involves low-level speech recognition, but it also requires that the machine tracks the topic of the conversation and how the words being said connect to the context of the conversation.

This problem is typically addressed, in a narrow way, by having experts write high-level scripts that keep the conversation moving in a predefined direction. But these kinds of hand-designed systems don’t scale.

Automating the creation of conversation scripts should be a good problem for machine learning. It’s not too hard for networks to predict likely directions a conversation might go. But it is much, much harder for machine learners to become effective conversational partners.

Try It Yourself

Follow along with the video lesson via the Python code:

[L20.ipynb](#)

Python Libraries Used:

`IPython.display`: Audio playback.

`keras.layers.Conv1D`: Creates a 1D convolution in Keras.

`keras.layers.Dense`: Creates a fully connected layer in Keras.

`keras.layers.Flatten`: Reorganizes array-shaped units into a flat vector.

`keras.layers.Input`: Builds an input layer.

`keras.layers.MaxPooling1D`: More local pooling layer for 1D convolutions.

`keras.models.Model`: Builds a neural network in Keras.

`librosa`: Sound and music.

`matplotlib.pyplot`: Plots graphs.

`numpy`: Mathematical functions over general arrays.

`os`: Provides operating system access for manipulating files and directories.

`sklearn.model_selection.train_test_split`: Splits a dataset randomly into training and testing sets.

`warnings`: Set whether or not to display warning messages.

Key Terms

expectation-maximization algorithm: An unsupervised Bayesian method that learns about latent structure by alternating between a labeling step (expectation) and a parameter-estimation step (maximization). In some probabilistic settings, it is a competitor to gradient descent for neural networks.

hidden Markov model: A Bayesian-style model that generates sequences by producing observable outputs overlaid on a Markov chain. Commonly used in speech recognition.

Markov chain: A transition system consisting of discrete states.

READING

Rabiner and Juang, “An Introduction to Hidden Markov Models.”

QUESTIONS

1. What three properties define the outer limits of speech recognition systems pre-2015?
2. Bayes’s rule decomposes the probability of text, given speech, into the probability of text times the probability of speech, given text. What are the latter two quantities known as?
3. The simple yes-no recognizer in the lesson used a neural network architecture consisting of four convolutional layers and three fully connected layers (more than 3 million trainable weights). It got about 96% training and testing accuracy. What do you think would happen if a switch were made from convolutional layers to solely densely connected layers? A network with one hidden layer of size 200 ends up having a similar number of trainable weights. Make a guess as to what training and testing accuracy you’d see. Run it to find out what happens.

Answers on page 485

Deep Learning for Speech Recognition

Lesson 20 Transcript

A milestone in the ability of machines to recognize human speech was reached in 2016. That's when researchers at Microsoft announced that their speech recognition system was achieving accuracy rates beyond human transcriptionists. Basic voice interfaces driven by machine learning have been broadly deployed since around 2011, and they have been improving ever since. Companies like Microsoft, Amazon, Apple, Google and Baidu have all made important contributions to developing and disseminating speech-recognition technology.

Given how common speech recognizers have become, it's worth noting the limitations on earlier systems to understand how things have changed. There are three principal dimensions in which speech recognition systems have been limited. Some systems could only handle disconnected speech . with . gaps . between . each . individual . word. Others were built only to recognize words from a limited vocabulary, like the digits zero through nine, or the names of cities on a train line. Still others are speaker dependent and have to be tuned to work with individual users.

Older systems improved to the point where they could overcome any two of these limitations. But it was machine learning systems introduced since 2011 that have been able to address all three challenges at once. The history of speech recognition approaches shows us how this complex problem was decomposed into simpler problems that could be tackled separately. Later, deep neural networks were applied to each of the separate problems resulting in rapid progress.

Since we'll be turning speech sound into text words, it's worth getting a clear idea about what sound is. Sound is just difference in air pressure traveling through space. To visualize the movements of a sound wave, imagine we have a tube filled with air molecules. We'll pull back a plunger, causing a low-pressure region to form. When we push the plunger forward, air molecules are thrust forward along with it. They bunch up, then bounce off each other causing other molecules to lurch forward and bunch up.

LESSON 20 | DEEP LEARNING FOR SPEECH RECOGNITION

TRANSCRIPT

The bunch travels along the tube of air at the rate at which the molecules move and interact. The rate of travel depends on overall air density, and air density depends on pressure and temperature; that is, the wave moves more slowly in the cold, thin air of Mt. Everest. The rate this wave propagates is known by another name — the speed of sound. It's about 750 miles per hour, almost 350 meters per second.

In day-to-day life, the role of the plunger in this example can be played by the movements of all sorts of objects in the world. The movement and resulting sound can be organized or disorganized. Disorganized movement is like the crumpling of paper and results in a disorganized pattern of sound waves. Organized movement is like the beating wings of a fly. Organized movement causes the waves to arrive in a very regular, very organized, pattern.

A typical housefly, for example, flaps its wings 190 times per second. Our ears register this consistent pattern as a tone. We can depict the pressure measured at our ear as a function of time. Very roughly, the human ear can detect sounds up to about 20,000 times a second. That implies the ear is sampling the sound-intensity function at roughly twice that rate. We'll call that the sampling rate. It's about one sample every 24 microseconds.

Two values are enough to summarize a nice clean pattern like a consistent tone. The first is the height of the waves; that is, the pressure difference between the highest and lowest pressure parts of the waves. That's what we usually refer to as volume or loudness or intensity. It's also known as the amplitude. The second is the time elapsed between the peaks. The number of peaks encountered per second is called the frequency and it's what we hear as pitch. Higher pitch [beep] sounds have frequencies like 255 peaks per second. Lower pitch [boop] sounds have frequencies like 85 peaks per second.

More complex sounds like the human voice have more going on than just a single tone with regular peaks. It can contain multiple frequencies at the same time. A really helpful way to characterize speech is to plot which frequencies are present at each moment in time. This display is called a spectrogram. Here's a spectrogram of me saying *no*— “No.” The *x*-axis is time and the *y*-axis is frequency. The third dimension is brightness. A bright white patch in the image means that there is high amplitude. Notice it can be bright for several frequencies at a time.

The time axis underneath is labeled with the sound I was making at each point. You can see where the *nnn* sound starts with a bright patch near the bottom of the image. The *nnn* transitions to an *uh* and bright patches appear above it in the spectrogram. The *oh* part is visible as a drop in pitch from the line marked 2,000 down to about 1,000. When the word ends, you see a return to the background pattern.

The white bars in this image are called formants because they are the tones that we shape with our mouths to form speech sounds. I'm told that some people can read spectrograms, reconstructing what was being said just by looking at the pictures. I haven't been able to pick-up that skill yet. But that is essentially what a speech-recognition system needs to do. It needs to turn this kind of data about the speech sound into text words.

Now, you understand the problem. How do we solve it? The earliest work on speech recognition dates back to a digit-recognition system built by Bell Labs researchers in 1952. Computers were too slow back then to process speech information. Remember, auditory information comes at a rate of about 40,000 times per second, or once every 25 microseconds. The year 1952 was a decade before computers could even multiply numbers at that speed, let alone carry out a complex speech-recognition algorithm.

As a result, the researchers built their specialized recognizer directly out of electronic components like vacuum tubes. The system extracted the strongest and second strongest formants — the frequencies with highest intensities. Then they compared the formants to a set of 10 templates, one for each digit. The templates were kind of like specialized classifiers, but they were not built using machine learning algorithms. The templates were painstakingly created and tuned by people using trial and error. It was data-intensive work for people to do by hand.

The system they built was reported to have an accuracy of over 90%, at least for one of the researchers who built it. For other people, accuracy was quite a bit lower. The Bell Labs system suffered from all three of the limitations I mentioned for traditional speech-recognition systems. It was tuned for a specific person, it required gaps between words, and it could only handle the names of the 10 digits. It was a remarkable achievement, but not really ready for prime time.

Things improved gradually in the 1960s. Researchers attacked the problem of phonetic segmentation. That is, approaches were introduced to try to break connected speech into smaller units — words — so the individual

TRANSCRIPT

words could be recognized. Template patterns for words were introduced and could be matched against incoming speech sounds. Researchers realized how important top-down influences and context were in disambiguating sounds into the desired words. Did Bob Dylan say, “The ants are my friends” or the “The answer my friends”? Phonetic segmentation helped with connected speech and, to some degree, recognizing larger vocabularies. But since things were still primarily being done by hand, scaling up to more speakers and larger sets of words would have required prohibitively expensive amounts of expert human effort.

The 1970s brought faster machines and a new machine learning technology that greatly improved the scalability of speech-recognition systems. The technology is known as hidden Markov models or HMMs. Hidden Markov models belong to the Bayesian style of thinking about machine learning. The fundamental idea is to view the relationship between input and output backwards. We build a model that goes backwards from text words W to speech sounds X . The models are hidden in the sense that they include latent or hidden variables.

An ordinary, non-hidden Markov model is called a Markov chain after Russian mathematician Andrei Markov. Hidden Markov models were the best game in town, before the first deep learning models began to be tried in speech recognition around 2012 or so. After that point, hidden Markov models were no longer central to speech recognition, although they have remained a mainstay in specialized machine learning solutions in signal processing.

The Bayesian view says we should think about the speech signal as being generated by the text word sequence, even though we don’t observe the text word sequence. And that generative process can involve intermediate steps along the way. Those individual steps might be the words that make up the sequence or they might be basic speech sounds, called phonemes, which make up the words.

Okay, so when a Bayesian system receives an input of speech sounds X , it needs to reason backwards to possible text word sequences to find the text word sequence w that is maximally likely. We want to maximize the probability of the text words given the speech sounds, written Probability of W -bar- X . We want to know, what was probably said, given what I heard? Argmax is a math operator that reveals the text words W that make the given argument, or expression, take on the biggest value.

As we saw in Lesson 6 on Naive Bayes, Bayes' rule lets us translate between the probability of W given X and the probability X given W . The rule tells us to rewrite the probability of W given X as the probability of X given W times the probability of W , divided by the probability of X . We can view the probability of X in the denominator as just a normalization factor that doesn't affect which W is more probable.

That leaves us with just two quantities. The probability of X given W tells us how words are pronounced. You give me a word W , I'll produce the speech sounds. It's the pronunciation model. The probability of W expresses which sequences of words are more likely than others. It's known as the language model. We saw this quantity in Lesson 17 on text generation. A language model is what tells us that the sequence 'know wide pre four two eight' is less common than 'no I'd prefer to wait'.

Hidden Markov models capture the probabilities of sequences. They are especially useful for the part of the Bayesian model where phonemes are mapped into frames — slices of the spectrogram. Take a look at a set of slices that make up an example of me saying the *oh* sound. For American English, *oh* is not one constant thing. It morphs over time. Think about your jaw as you say the sound *oh*. It closes a bit after you start making the sound. That change in mouth shape creates a change in the sound: *Ohhh-wuuuh*. The precise duration of a part of the sound, and the speed at which it changes, will vary from one utterance to another.

To simplify, let's express the slices of the spectrogram in a more cartoon form. Now, we can imagine that there's a relatively small set of slices, and that the overall sound is built by lining them up in order. The sound becomes four different spectrogram slices. Slice one appears twice in the sequence. Slice two appears twice in the sequence. Slice three appears once in the sequence. And, then, slice four appears four times in the sequence.

The exact number of repetitions of each slice, and the exact shape of the spectrogram for that slice, will vary from utterance to utterance. To capture this variability, we change these four slice types into a kind of probabilistic machine. Consulting its specific probabilities, it spits out a spectrogram slice and then randomly stays in the same state or progresses to the next state.

A hidden Markov model is precisely this kind of machine. It has a set of hidden states that generate observable outputs. Internally, these states transition from one to another with some degree of randomness.

TRANSCRIPT

It's a randomized sequence generator. The parameters that control the behavior of the model are: first, the probability of transition between each pair of states and, second, the output probabilities, which determine what slice type each state produces.

How can we learn good settings for these parameters? Well, if we had lots of data of *oh* sounds where each slice was tagged with the hidden Markov state that produced it, well, then the learning problem is really easy. To estimate the probability that state two transitions to state three, for example, we just count up the number of times the machine was in state two and then look to see how often it was next in state three. In this example, 3 out of 5 or 60% of the transitions from state two go to state three. To estimate what spectrogram is produced by state four, we can just average together all the spectrograms associated with state four in the data.

All of that's great. But speech sounds don't come with hidden Markov state labels on them. It's impractical to ask people to produce these labels, labeling each 10 millisecond slice with what state produced it. Ah, but do you know who can label the slices? A hidden Markov model, that is if we knew the parameters of the hidden Markov model,

then we could estimate the likelihood that each slice was produced by a specific state of the HMM.

It's kind of a chicken-and-egg problem. We can label data if we know the model and we can estimate the model if we know the labels. How do we break this cycle? What if we just made up a model at random? Then we could use that model to label the data. Then we could use those labels to learn a model. Then we could use that model to label the data. But that couldn't work, could it? I mean, since the initial model is just made up, how could the labels it produces be useful for anything?

Remarkably, this pull yourself up by your own bootstraps approach works quite well. The method is called expectation maximization because the labeling step can be seen as computing the expectation — the mathematical average — of the labels. And the model-learning step chooses the model that maximizes the likelihood of the assigned labels. Expectation-maximization is an unsupervised method. We give it unlabeled data and it iteratively learns a model that would produce that data. Specifically, it wants to find a model that makes the data as likely as possible.

A powerful mathematical fact about expectation-maximization is that each iteration produces a model that assigns ever higher probability to the data until it is unable to make any improvements. It is not guaranteed to find the best model. But in practice, it often finds one that is good enough to be useful. In that sense, the expectation-maximization algorithm is a lot like the gradient descent procedure that we use to train neural networks. Both start off with a model that is initialized randomly. Both have to discover latent structure that is not provided in the data. Both proceed by iterative improvement. Both have guarantees that these iterations make things better, but neither can guarantee that it will find the global optimum. Finally, both can be applied to many different kinds of problems and produce practical answers to hard problems.

Of the two, gradient descent seems to scale better, especially in terms of modern computational devices, and gradient descent seems to be a bit more general. Where expectation-maximization requires us to think about a generative model of data, gradient descent can be applied to any setting where we need to search for parameter values that minimize a continuous loss function. Gradient descent can be used wherever expectation-maximization is used, but not vice versa. And that takes us up to about 2011. By 2011, hidden Markov models had become the dominant approach to speech recognition. These models could be used to produce systems that were free of any two of the three limitations we talked about earlier, but not all three.

HMM-based systems could produce recognizers for tens of thousands of words spoken by arbitrary speakers, as long as the words were isolated from one another. That is, they were still limited to disconnected speech. Other systems could be used for dictation, transcribing natural speech as long as the system was trained for its particular user. That is, they remained speaker dependent. A lot of doctors used this technology to record patient notes. Other HMMs were used by banks in interactive systems where arbitrary people could call into a service and make requests by voice. But those systems needed strong top-down priors about what would be said. That is, they still needed a limited vocabulary.

Deep learning systems, starting in roughly 2011, made it possible to break free of all three limitations simultaneously. Traditionally, speech-recognition systems had been built to turn speech sounds to sentences through a series of intermediate steps. First, low-level speech sounds would be recognized into phonemes. Then these phonemes would be coalesced into individual words.

TRANSCRIPT

Lastly, the words would be massaged into the most likely sentences. Each layer would send its best guess to the layer above it, but it would also transmit some degree of uncertainty since language and sound are notoriously ambiguous.

Between 2012 and 2015, researchers discovered that they could improve on the state of the art for each of these transformations. The overall progression through the three transformations remained the same, but the approach used to solve each of the three problem layers became neural networks. For speech sounds to phonemes, convolutional neural networks are remarkably adept at treating a spectrogram as a kind of image and recognizing phoneme patterns in it. For phonemes to words, recurrent neural networks can solve similar problems to those attacked by HMMs and provide latent context for generating the words. For words to sentences, the combination of word embeddings and transformer networks does an astonishingly good job at distinguishing likely and unlikely sentences.

Replacing each component of the three layers of this system with an appropriate neural network leads to excellent speech-recognition performance. The boost in performance surprised Microsoft researchers when their system beat human transcriptionists for the first time in 2016. After working on the problem of speech recognition for decades, they didn't expect improvements to come so quickly. They hailed their accomplishment as a historic achievement.

Indeed, the benefits of neural networks for this problem really are remarkable. Because neural networks with the same format and the same training machinery are being used in each component, it is possible to skip over the intermediate representations. For example, phonemes are often viewed as a kind of necessary evil in speech recognition. It's not even clear to the people building the systems that phonemes are quite the right intermediate representation.

With a neural network approach, we can train a recognizer to go from speech sounds directly to words. It could potentially create something like phonemes internally, but no person has to tag the speech with these elements. The system could discover its own vocabulary of phonemes, or indeed, not use anything like them. The network can just devise whatever features are most effective in training.

Since 2014, there have been attempts to apply end-to-end training, speech sounds all the way to text words with a single network. End to end training is a kind of mantra in the deep learning community. It asserts that we'd be better off not having to pick any internal representations for our networks.

As of 2020, such end-to-end systems had been built, but they had not performed as well as the modular approach. But as the research community's understanding of how to design and train neural networks continues to improve, I would not rule out the possibility that new ideas will make end-to-end training for speech recognition viable.

Now, let's experiment ourselves a bit with a deep learning system for recognizing some simple words. Our system will handle speech that is disconnected and has limited vocabulary, but we'll make our system speaker independent. Our system could be used for a simple interactive voice response system for a bank, say. The speech commands data set includes samples of people speaking the words yes/no, left/right, on/off, stop/go, up/down, the digits zero through nine, wow, happy, Sheila, Marvin, bird, dog, cat, bed, tree, house, and background noise. Let's train a deep network to distinguish yes and no.

We'll need to import some standard libraries like MatPlotLib, PyPlot and NumPy. But we also need a few specialized libraries for processing sound, specifically librosa and wavfile. Next, we'll get the data file and unpack it into a directory called speech_commands. Let's take a look at the raw speech sounds for one of the instances of *no*.

I picked out a filename corresponding to one of the recordings of *no*. We can load the audio from this file into the vector samples. Librosa.load returns the vector and the sampling rate of the recording. This recording is 16,000 samples per second, which is moderately high quality. For comparison, the standard for CDs, early DVDs, and other related digital media was 44,000 samples per second.

If we plot the vector samples for our recording, we can see the speech sounds, "Nooooo." The library IPython display includes functions for transmitting audio files to your computer's speaker. Calling ipd.Audio on the samples we just loaded actually plays the sound. Was my imitation of the sound good? "No." Ouch.

Okay, let's put together a training set. All_wavs and all_labs will be our list of instances with their labels. We're going to read in the files for each of the two classes of sounds we care about, *yes* and *no*. For each label, we assemble a list of all of the audio files associated with that label in a list of wavs. For each file, we run the librosa.load command to pull in the raw speech sounds. To make the training clean and easy, we'll only use clips that are exactly one second long — that's 16,000 samples. This operation takes about 5 minutes. We end up with 4,255 instances.

TRANSCRIPT

We've got our data. Let's organize it into training and testing matrices. We'll import `train_test_split` to randomly separate the data into training and testing sets. We'll reshape our audio files into a shape that the network will be able to accept. We'll turn our *yes/no* labels into 1/0 by testing equivalence to *yes*. Finally, we can call `train_test_split`, where a test size of 0.2 makes an 80-20 split of our data.

Let's build a network in Keras for speech recognition. From the Keras library, we'll import the five layer types: Input, Conv1D, and so on. The input layer to our network is the 16,000 samples drawn from the one-second clips. We'll then instantiate a series of one-dimensional convolutional layers. Convolutional layers detect patterns in local patches of the input in a position-invariant way. Because speech is a one-dimensional sequence, `conv1d` is a sensible choice.

The first convolutional layer will extract eight features looking at 13 consecutive amplitude samples. These choices are somewhat arbitrary. They are similar values to values used in successful vision architectures like VGG16. The second convolutional layer extracts 16 features from 11 consecutive outputs of the first convolutional layer. The third convolutional layer extracts 32 features from nine consecutive outputs of the previous layer. Doubling the number of features in each convolutional layer is typical for vision networks. The fourth convolutional layer produces 64 features from seven consecutive outputs of the previous layer.

To turn the extracted audio features into a prediction, we flatten the features from the convolutional layer and then put them through two dense layers of size 256 and 128. The final dense layer maps the 128 features from the previous layer down to a single output representing the *yes/no* choice. We set the loss for the network to be mean squared error, then train it for 15 epochs. The trained network has 97% accuracy on new examples.

I listened to a few of the clips where the network made a confident but wrong answer. Some had a fairly clear word, but also a burst of noise or very odd background sounds. Some had no word at all, as far as I could hear. A few others seemed very clear and noise free. In these cases, the speaking voice was female. It's somewhat common for speech-recognition systems to struggle with female and child voices, perhaps because of bias due to too few examples of these classes in the training data, or perhaps because higher pitches might be harder for current algorithms to process consistently.

Because networks can end up being less accurate for specific classes of people, it's worth evaluating them carefully before deploying broadly. If the error rates for specific classes are bad enough to constitute a burden on a segment of the population, it might be worth putting on the brakes and looking for another solution. Being able to talk to machines and having them understand what we want would make the human-computer interface more natural and fluent. Computer scientists and engineers have been making progress on this challenge for 70 years. Nevertheless, the problem is definitely not solved.

Familiar network components like convolutional neural networks — which were pioneered for image processing — were discovered to be exceptionally well suited for speech as well. Moving to neural networks from early methods with human-defined features has led to systems that demonstrate a high degree of accuracy with minimal limitations. But it has meant that the engineers building the systems find it harder to understand what the systems were doing.

The frontier of speech recognition has moved away from converting speech signals to text words. The bigger challenge is language understanding. Being able to have a voice-based conversation with the computer involves low-level speech recognition, but it also requires that the machine track the topic of conversation and how the words being said connect to the context of the conversation. This problem is typically addressed, in a narrow way, by having experts write high-level scripts that keep the conversation moving in a pre-defined direction. But as we have seen repeatedly through this course, these kinds of hand-designed systems don't scale.

Automating the creation of conversation scripts should be a good problem for machine learning. For example, demos of voice-based reservation systems can be impressive in carefully circumscribed settings. Data of people having conversations is gathered and machines can be trained to process the data. It's not too hard for networks to predict likely directions a conversation might go. But it is much, much harder for machine learners to become effective conversational partners. It seems that learning to have conversations means you need to practice being in conversations, not just witnessing them.

LESSON 20 | DEEP LEARNING FOR SPEECH RECOGNITION

TRANSCRIPT

Perhaps ideas from reinforcement learning could help. After all, it is an approach focused on doing, with feedback, but without needing more explicit oversight from a teacher. The first challenge would be figuring out what rewards the learner should be optimizing in these interactions. The second challenge would be getting patient people for the reinforcement learner to talk to.

Figuring out the rewards for interactive dialogue systems is an open problem. But there has been some work on using machine learning tools to discover rewards by watching an expert's decisions. Such an approach can figure out the trade-offs in how a self-driving car decides whether to go fast to arrive quickly, or go slow to avoid a bouncy, uncomfortable ride. Next time, we'll turn to the subject of how rewards drive reinforcement learners and how we can make machines learn to construct their own rewards.

LESSON 21

INVERSE REINFORCEMENT LEARNING FROM PEOPLE

At a high level of abstraction, telling a machine what to do goes something like this: A person has an intent or goal. There's communication of that goal in some form so that it can be transferred into a representation the machine can process. The machine then uses this representation as a guide for planning and for controlling its behavior in the world. This lesson introduces a particular approach to telling machines what to do that uses a form of machine learning called inverse reinforcement learning.

Trigger-Action Programming

The most direct and traditional way of telling a machine what to do is by writing a program for it in a language like Python. Using programs, people can express any computable behavior to the machine.

Programming is pretty great. But it's also cognitively demanding. To program, you need to be an expert in what you want to do and how to express the idea. Programs can be hard to write, hard to debug, and hard to modify.

Fortunately, for some people, for some applications, we can simplify things in a useful way. Simplified programming languages provide good gateway exposure to programming, and they might help motivate more people to learn more programming.

Here's an analogy: Everyone should know how to read and write, but not everyone needs to be Shakespeare. What's the programming analog of using language to shoot off a quick email?

The analog of writing someone a note is trigger-action programming. It's a highly simplified way to connect real-world activity to programs for controlling that activity.

A trigger-action program can turn on the lights in your house when the sun goes down. There are many online services that use this style of programming. The services are designed in a way that makes programs seem simple and friendly, presumably so that almost anyone could write and use them.

Research on this topic verified that trigger-action programs are easy for people to learn to write. But it also showed that people could handle more complexity than most service providers were willing to allow.

The good-news/bad-news moment came when researchers followed up on this work and found that while people could indeed successfully write complex rules, they didn't know what the rules meant—they couldn't read them.

The level of precision required to read, write, and think about complex rules is very challenging for many people, even when the programming language itself makes it very simple to express a rule.

Ways of Programming Machine Behavior

To better understand how we might use machine learning to reduce the burden of programming, we can think of different ways of programming machine behavior in a simple two-dimensional space.

One dimension is how behavior is specified: direct or indirect. Direct specification means that sensory inputs are mapped directly to behavioral outputs like in a traditional computer program.

An indirect specification uses reward functions to guide machine behavior by assigning scores to different behaviors. The machine's job is to decide what behavior to adopt by interacting with the environment to discover which behaviors maximize those scores.

Specifying behavior via reward lets you express the goal that the learner is trying to follow without having to spell out the specifics of how that goal should be achieved. It's analogous to the machine learning idea of using a loss function.

Reinforcement learning is the branch of machine learning concerned with learning to maximize rewards. Reinforcement learning was addressed in [Lesson 13](#) on games, where the reward function comes from the rules of the game itself. It's typically much easier to express the rules of the game than to express winning behavior in the game. The machine learner is given a reward function but then generates behavior autonomously.

A robotic example of a reinforcement-learning problem is a solar panel that learns: The system can take action in the environment by changing the angle the panel is facing. The system is designed to measure energy brought in via the solar panel and subtract the energy needed to orient the solar panel. Then, it is told to maximize total energy brought in. It learns where to point the panel and when.

This is an elegant example of using rewards to obtain desired behavior without needing to specify the behavior itself—indirect specification.

But in more complicated examples, even the reward function can be hard to specify.

For example, in 2011, Nest began marketing a learning thermostat that adjusts its behavior via reinforcement learning. A thermostat has competing objectives—to maximize user comfort and minimize energy use—making the choice of reward function challenging.

Engineers just selected a particular trade-off for the Nest, in effect concluding that most end users would find it too hard to pick a reward function and that the thermostat would be better off just learning from the one chosen by the engineers.

A second dimension where machine learning can help with machine behavior are examples and demonstrations. Sometimes it's easier to just show the machine what you want it to do—demonstrate the behavior and then let the machine figure out the program. These demonstrations play the same role that training examples play in the machine learning settings you've already seen.

A machine learner can learn directly or indirectly from example demonstrations. One direct approach is called learning from demonstration, where the machine learner extracts a direct input-output mapping from the examples it sees. It uses supervised learning to learn a rule that generates outputs from inputs that look like the examples it is given.

By contrast, **inverse reinforcement learning** is an indirect approach that observes example behavior and then extracts a reward function that explains the behavior being observed. As of 2020, this promising approach has yet to find its way into deployed applications, but it is possible that the next round of home robots will include this feature.

Reinforcement Learning:

Rewards → Behavior

Inverse Reinforcement Learning:

Rewards ← Behavior

Robots and the Future of Society

Using inverse reinforcement learning to create trainable learning systems will provide us with better ways of delegating tasks to our machines that we want done. That will empower people and provide ways for people with good ideas to put those ideas out into the world to benefit others.

Some researchers think that intent-conveying ideas like inverse reinforcement learning are even more significant to the future of our society. The thinking is that these ideas may be a way—perhaps the best way—to address concerns about malfunctions in hypothetical future computer systems that are superintelligent.

A typical doomsday scenario begins when we ask the machines to do something relatively innocuous, such as improving the speed with which they can look up data in a big database.

Being dutiful optimizers and powerful problem solvers, they conclude that they will produce better solutions to their given problem if they seek to improve their own capabilities, assure their own survival, and take control over more and more resources, such as energy and raw materials.

Steadfastly pursuing these intermediate goals would put the machines in conflict with what people want. People, too, need resources, and people are apt to turn off a machine that is competing for the same resources. So stopping people from stopping the machine would become a natural goal for the superintelligent machine. In fact, the fewer pesky humans there are around the world, the less likely the machine's goals will be thwarted.

At the root of this robot apocalypse is an idea that a reinforcement learner can overfit to the reward function it is given. The presumption is that the machine learner will devise a policy that maximizes the specifics of the reward function it was given without being able to generalize to a broader set of circumstances.

How do we make our machines understand what we want and then help us do useful things in the world? The answers have implications for home robots, home automation devices, and even computers in general.

One solution to this problem is to make sure that machines realize that there is uncertainty in the goal they are given. The goal they have might be missing key elements, such as “improve database access time, using only *existing* capabilities and resources—and *without* killing any humans.”

Learning tasks from people using some version of inverse reinforcement learning might be the key to keeping powerful machines working in the service of humanity.

Try It Yourself

Follow along with the video lesson via the Python code:

[L21.ipynb](#)

Python Libraries Used:

functools.reduce: Summarizes a vector or list by a single value.

keras.backend: Provides access to lower-level Keras functionality.

numpy: Mathematical functions over general arrays.

seaborn: Data visualization.

Key Terms

inverse reinforcement learning: The problem of going from observations of behavior to estimates of the rewards that the behavior was selected to optimize.

maximum likelihood: A probability-based criterion for machine learning that states that the best rule is one that makes the observed data as likely as possible.

policy: A controller for a sequential decision problem, typically represented as a function that maps state to action.

READING

Bostrom, *Superintelligence*.

Charniak, *Introduction to Deep Learning*, chap. 6.

Hadfield-Menell, Dragan, Abbeel, and Russell,
“Cooperative Inverse Reinforcement Learning.”

Russell and Norvig, *Artificial Intelligence*,
chap. 27 and secs. 21.8.3 and 22.6.

QUESTIONS

1. Here are four ways you might learn to paint. For each one, say whether the specification for how to paint is direct or indirect and whether the instructions are handwritten or learned from examples:
 - (a) Take an art history class that covers what made the masterpieces so great.
 - (b) Watch a painter/teacher like Bob Ross at work.
 - (c) Take a how-to art class.
 - (d) Watch a professional sculptor and try to translate what you learn to painting.
2. Instrumental convergence is the idea that there are goals that any reward-driven agent should pursue to best satisfy its primary goal in complex real-world scenarios. The lesson mentions three: self-improvement, self-preservation, and controlling resources. Can you think of others?
3. Can you define a reward function that makes the grid-walking program from the lesson go to the green square and then come back and end on a yellow square? What reward function does the inverse-reinforcement-learning program return, given that behavior as input? Does the reward function capture the right behavior?

Answers on page 485

Inverse Reinforcement Learning from People

Lesson 21 Transcript

From 1999 to 2006, Sony developed and sold a dog-like robot called Aibo. It came with a set of simple pre-programmed behaviors. But in my lab, we integrated Aibo with machine learning algorithms and used it as an experimental testbed. So, it was a sad day for machine learning, or at least for me, when the Aibo was discontinued.

But in 2018, Sony began selling a new version of the Aibo robot in some parts of the world, though not the US. Given the advances in machine learning software, it's reasonable to hope we will see robots for sale that take advantage of machine learning. I worked briefly with a team that became Sony AI. My goal was to get an Aibo to learn tasks from people.

Training a robotic dog is just one example of a much broader quest. How do we make our machines understand what we want and then help us to do useful things in the world? The answers have implications not only for robo-pets, but also home robots, home automation devices, and even computers in general.

At a high level of abstraction, telling a machine what to do goes something like this: A person has an intent or goal. There's communication of that goal in some form so it can be transferred into a representation the machine can process. The machine then uses this representation as a guide for planning and for controlling behavior in the world. In this lesson, we'll build up to one particular approach to telling machines what to do that uses a form of machine learning called inverse reinforcement learning. But first, it's helpful to start by looking at a more familiar approach.

The most direct and traditional way of telling a machine what to do is by writing a program for it in a language like Python. Using programs, people can express literally any computable behavior to the machine. Programming is pretty great. But it's also cognitively demanding. To program, you need to be an expert in what you want to do and in how to express the idea. Programs can be hard to write, hard to debug, and hard to modify. Fortunately, for some people, for some applications, we can simplify things in a useful way. I think simplified programming languages provide good gateway exposure to programming and they might help motivate more people to learn more programming.

An analogy I like is that everyone should know how to read and write, but not everyone needs to be Shakespeare. What's the programming analog of using language to shoot off a quick email? I think the analog of writing someone a note is trigger-action programming. It's a highly simplified way to connect real world activity to programs for controlling that activity. A little trigger-action program can turn on the lights when the sun goes down. I use trigger-action programs very much like this in my house to control some of the lights and to do other internet of things tasks.

There are a lot of online services that use this style of programming. The services are designed in a way that makes programs seem simple and friendly, presumably so that almost anyone could write and use them. My collaborators and I decided to put to the test the idea that trigger-action programs are easy for people to learn to write. Our results verified the widely held belief that trigger-action programs are pretty easy for people. But we also showed that people could handle more complexity than most service providers were willing to allow.

The good-news-bad-news moment came a bit later when researchers at the University of Washington followed up on our work. They found that, while, indeed, people could successfully write complex rules, they didn't know what the rules meant. They couldn't read them. The level of precision required to read, write, and think about complex rules is very challenging for many people, even when the programming language itself makes it very simple to express a rule.

To better understand how we might use machine learning to reduce the burden of programming, we can think of different ways of programming machine behavior in a simple two-dimensional space. One dimension is how behavior is specified—direct or indirect. Direct specification means that sensory inputs are mapped directly to behavioral outputs like in a traditional computer program.

An indirect specification uses reward functions to guide machine behavior, by assigning scores to different behaviors. The machine's job is to decide what behavior to adopt by interacting with the environment to discover which behaviors maximize those scores. Specifying behavior via reward lets you express the goal that the learner is trying to follow without having to spell out the specifics of how that goal should be achieved. It's analogous to the machine learning idea of using a loss function.

LESSON 21 | INVERSE REINFORCEMENT LEARNING FROM PEOPLE TRANSCRIPT

Reinforcement learning is the branch of machine learning concerned with learning to maximize rewards. We talked about reinforcement learning in Lesson 13 on games. There, the reward function comes from the rules of the game itself. It's typically much easier to express the rules of the game than to express winning behavior in the game. The machine learner is given a reward function, but then generates behavior autonomously.

Some of my students built a robotic example of a reinforcement-learning problem—a solar panel that learns! The system can take action in the environment by changing the angle the panel is facing. The system was designed to measure energy brought in via the solar panel and subtract the energy needed to orient the solar panel. Then, it was told to maximize total energy brought in. It learned where to point the panel and when. It's an elegant example of using rewards to obtain desired behavior, without needing to specify the behavior itself—indirect specification. But in more complicated examples, even the reward function can be hard to specify.

For example, in 2011, Nest began marketing a learning thermostat that adjusts its behavior via reinforcement learning. A thermostat has competing objectives—maximize user comfort and minimize energy use—making the choice of reward function challenging. Engineers just selected a particular tradeoff for the Nest, in effect concluding that most end users would find it too hard to pick a reward function, and that the thermostat would be better off just learning from the one chosen by the engineers.

A second dimension where machine learning can help with machine behavior are examples and demonstrations. Sometimes it's easier to just show the machine what you want it to do; just demonstrate the behavior and then let the machine figure out the program. These demonstrations play the same role that training examples play in the machine learning settings we've already seen.

A machine learner can learn directly or indirectly from example demonstrations. One direct approach is called Learning from Demonstration or LfD, where the machine learner extracts a direct input-output mapping from the examples it sees. It uses supervised learning to learn a rule that generates outputs from inputs that look like the examples it is given.

By contrast, inverse reinforcement learning is an indirect approach that observes example behavior, and then extracts a reward function that explains the behavior being observed. As of 2020, this promising approach has yet to find its way into deployed applications, but it is possible that the next round of home robots will include this feature.

Let's take a look at a reinforcement-learning program that makes reward-based decisions from a hand-written reward function. We'll create a little grid-world, define tasks in this grid world by defining rewards, and then see what behaviors result when this reward is optimized the way a reinforcement learner would. Afterwards, we'll use examples from this same domain to experiment with inverse reinforcement learning.

We'll imagine the decision maker in this world is a robot in a field. The field is a grid of 50 locations arranged in five rows by 10 columns. There are also five different categories of locations. You can think of the categories as being types of terrain; say, gravel, grass, shallow water, mud, and asphalt. The robot can tell what category each location is. We can allocate different rewards to each category and then see how the robot navigates in response. The variable map is a 5-by-10 grid with each entry marking the category of the terrain in the corresponding location.

Let's have our program display the map. We'll use a library called seaborn to draw the grid on the screen. We can create a palette of colors for the five categories of terrain, then the seaborn heatmap command plots the map using the colors. Our map has some scattered blue terrain, an orange terrain stripe across the middle of the grid, a smaller yellow terrain strip halfway across the grid, and a green patch of terrain in the bottom right corner. The rest of the grid is the white background terrain.

We can assign reward values, r , to the terrain categories. Let's make white a neutral 0, blue and yellow and orange dangerous minus-1, and green an attractive positive-10. Next, we need to project these values out into the grid. We define an array matmap that is rows by columns by categories. Specifically, for each row i , column j , and category k , the entry in the array is 1 if the i,j location on the map is category k . Otherwise, the entry is 0. Next, we need to define how navigation in the map works. A robot could discover this information by wandering around in the field and observing how its action choices impact its location. For this demonstration, however, we'll provide the robot with a complete description of how its actions work.

LESSON 21 | INVERSE REINFORCEMENT LEARNING FROM PEOPLE TRANSCRIPT

We'll make use of a routine called `clip` that we'll use to keep the robot inside the grid by clipping the coordinate values to within the set of valid grid positions.

We define five actions acts in terms of how each one changes the robot's row and column. The first decreases the row by 1 and leaves the column unchanged. That corresponds to moving up in the grid. The other actions correspond to moving right, down, left, and staying in place. It is common in reinforcement learning to model actions as including stochastic effects. That is, it may be highly probable the robot's up action will move it up in the grid, but it's also possible the robot might slip to the left or right when it tries to move up. However, for simplicity, our example here is deterministic.

The array `mattrans` is the transition matrix. This matrix represents the probability that a given action will cause a one-step transition between any given pair of locations. We fill in this matrix by looping through all the actions. For each one, we enumerate all possible starting locations row i_1 column j_1 . We define i -next and j -next. They encode the location that results from adding an action's row and column increments to the current row and column, then clipping the result. Then, we loop through all the possible next locations row i_2 and column j_2 . We fill in the transition matrix with a 1 if the possible next location matches the actual next location, and 0 otherwise.

Note that I'm converting the i, j location to a single number— i times the number of columns plus j . That's just to simplify the matrix multiplications we'll use later.

Okay, we have the dynamics of the grid world now defined in `mattrans` and the mechanism for mapping rewards to states defined in `matmap`. Now, we're ready to compute a way to behave that maximizes rewards. We're going to write our planner using the Keras backend using commands that start `k dot`. It's a slightly awkward way to program but writing it this way will tremendously simplify the later step when we'll go from behavior to rewards that explain that behavior.

We start off by making a Keras variable `rk` for the terrain category rewards. Then, we take `matmap`—which is number of rows by number of columns by number of categories—and we compute the dot product of `matmap`

with rk . When we multiply by rk , which has the shape categories-by-1, we get a new array that is rows by columns. Specifically, it maps each location to its reward value. We use these values as the starting point for our reinforcement-learning process, assigning it to v , the robot's estimated best value for being in each location.

For the robot's decision making, we need to tell the robot how much to discount future reward compared to current reward. We set this parameter, gamma, to 0.9, meaning a reward retains 90% of its value one time step later. When gamma is close to 1, the robot cares a lot about making choices that will eventually lead to high reward. When gamma is close to 0, the robot only cares about short term gain, even if that means sacrificing opportunities for additional rewards later.

We also define a temperature parameter beta that indicates how carefully the robot will choose among closely valued actions. High values of beta cause the robot to reliably choose the highest valued actions. Beta values close to 0 result in the robot being more careless and choosing randomly among the available options. The choice of 10 here results in moderately careful reward maximizing choices. The robot will look 50 steps into the future when it makes decisions. That's a reasonable choice as the robot can make it across the entire grid world in 13 steps.

For each step of looking into the future, the robot considers each of the five actions. It computes a value qi for each action i . For each example, $q0$ says, let's multiply the transition matrix for action 0 times the value estimate v . That produces a value for each location that says how much reward will I get in the future if I take action 0 from this location now? The array Q brings all those values together. The array Q has one row for each location and one column for each action.

The array pi represents the robot's assigned probability to each action in each location. The probability of the robot taking an action in a location is determined by how much reward it predicts it will get, now and into the future, from taking that action in that location; precisely the contents of the Q array. The higher the predicted reward, the more likely the robot is to take that action. Again, the parameter beta controls how strong the relationship between reward and probability is.

LESSON 21 | INVERSE REINFORCEMENT LEARNING FROM PEOPLE TRANSCRIPT

The planner has multiplied the action matrix times the values, discounted them by gamma, and then added in the reward for being in the current state. So, the planner has completed one iteration of looking ahead. The value v now reflects how good it is to be in each location looking ahead one additional step. When the loop is completed, v will have factored in 50 steps of lookahead. If this were a chess program, chess grandmasters would weep in envy. But since our little grid only includes 5 times 10 equals 50 states, this navigation problem is a lot simpler than chess.

The output of this lookahead process is an action decision for each possible state: If you're here, go right, but if you're here, go down. In reinforcement-learning jargon, it's that's called the policy and it is the output of the lookahead process—planning. The planner also outputs the value function Q according to the relationships we previously defined. We can run the planner on the rewards r we defined and observe the result.

Here's a path that the resulting policy would follow from the upper left corner. Recall that white locations are neutral; blue, orange and yellow locations have minus-1 reward; and, green has a positive-10 reward. The robot avoids the terrains associated with the minus-1 penalties as best it can. It navigates to the location with the positive reward and then stays there.

What if we reprogram the reward function, so the green location is only worth 1, while the penalty for the negative colors is much larger, at negative-10? Well, even though the colors associated with positive and negative rewards are the same, the behavior changes. Now, crossing through the orange barrier is too expensive and the robot remains on the left side of the grid.

Let's change which colors are negative. Suppose we associate a reward of negative-1 with the white locations, positive-5 for the green location, and 0 for the others. Well, then, the robot prefers the colored locations as steppingstones on route to the green location. I think of this environment as the robot trying to stay off of the hot sand.

Reinforcement learning is a powerful way of getting agents to follow what we tell them to do. But reinforcement learning requires that we define reward functions.

What about demonstrating desirable behavior rather than writing down the reward function? That would be easier, in some cases. Ideally, an algorithm could analyze the behavior and produce a good reward function. How might we demonstrate desirable behavior? If I show you that an agent selected this behavior, what do you think the agent's rewards look like? This example demonstrates inverse reinforcement learning. Reinforcement learning went from rewards to behavior. Inverse reinforcement learning goes from behavior to rewards. Reinforcement learning is if you like carrots, go to the carrot. Inverse reinforcement learning is if you see a horse go to the carrot, it might be because the horse likes carrots.

One way to go from behavior to reward is to imagine we select a reward function, then ask a kind of Bayesian question: How likely is the observed behavior given these rewards? That is, we want a reward function that makes the observed behavior as likely as possible. I mean, we could just be looking at random unmotivated behavior that just happens to take this form, but then this particular path being chosen is pretty low probability. Instead, we can assume that the agent wants to get to green. That helps a lot. The probability of the observed behavior goes way up.

What about blue? Well, it might have missed blue by accident. But the likelihood of this particular path increases if we assume that missing blue was intentional. How about yellow? Yellow can't be very costly since the robot went right through it even though it didn't have to. But it can't be too cheap or the robot would have used the yellow locations to avoid the white locations. Orange is harder to say but it must be no more costly than the value obtained from getting to green. And orange can't be less costly than white because the robot didn't use the orange locations to avoid white.

We can formalize and justify this reasoning by taking a Bayesian perspective. We want to know the reward function R that is most likely, given the trajectory we observed. We'll use a simplified calculation that's called the maximum likelihood formulation. It says we need to search for a reward function that makes the observed trajectory as probable as possible.

To solve the problem of turning a trajectory into rewards, we will need a representational space, a loss function, and an optimizer. The representational space is assigning reward values to the five location categories—white, blue, yellow, orange, green. For the optimizer, we will

LESSON 21 | INVERSE REINFORCEMENT LEARNING FROM PEOPLE TRANSCRIPT

use gradient descent, since it's so general purpose. But how about the loss function? The loss function needs to tell us which reward functions fit the best with the observed trajectory. So, for any reward function, we can derive a policy that assigns probabilities to actions in each location. If you are telling me that in the trajectory the robot chose the down action from location 2,2, that suggests that the robot's rewards caused it to assign a high probability to action down in location 2,2.

In fact, for any reward function, we can look at the probability that the learned policy with that reward function would assign to all of the choices in a given trajectory. That is, we can compute the likelihood of that trajectory given the selected rewards. The more likely the observed trajectory, the better the hypothesized rewards. So, let's look for rewards that cause the likelihood of the observed trajectory to be maximized.

Equivalently, because logarithms turn products into sums, we want the sum of the logs of the probabilities assigned to the choices in the observed trajectory to be as big as possible. Therefore, the negation of this expression can be our loss function. Let's compute the log-based loss function. The sequence of actions in our trajectory is held in the variable trajacts: right, right, down, down, down, down, etc. Once the robot arrives at the green square, it executes the stay action a few times before we cut it off.

We need to convert this sequence of actions into a sequence of locations. You could write out the sequence of locations explicitly, or you could sum things up in a simple loop. Or, as I decided here, you could use a little fancy Python programming and apply a Python command called reduce to run through the sequences of actions from left to right and return the sequence of locations starting from 0,0. The important result is that we now have a sequence of actions in trajacts and a sequence of locations in trajcoords.

To compute the loss, we run through this sequence of actions and look at the computed policy π . We ask π what probability it assigns to the i th action in the sequence at the i th location in the sequence. We take the negative log of that value and add it to the loss. Now we have a way to connect the rewards to the loss in a single function. We can use gradient descent to search for a reward function that produces low loss.

We pick a learning rate and an initial random value for the rewards r . Keras can compute the gradient of the loss with respect to the rewards. We'll call that value grads. Next, we can create a keras function that takes the rewards as input and produces the loss and the gradient as output. We'll call the combined loss-gradient function iterate since we'll be evaluating it at each iteration of gradient descent. We'll run for 5,000 iterations, taking the rewards r and producing the current loss and gradient via the iterate function. We'll then move the rewards r a small step in the direction that decreases the loss. Every 100 steps, we'll output the current loss so we can measure the progress.

Here's what we get. The loss decreases very quickly at first from around negative-12.3 to around negative-10.8, then flattens out. The reward function reached at the end assigns a very small negative value to white and yellow locations, a slightly larger negative value to orange, a much more negative value to blue, and a much larger positive value for ending up at green. The algorithm successfully inferred an approximation of the robot's preferences from this single example.

The algorithm I just described takes in demonstration trajectories to estimate rewards. So now, let's talk about where the demonstration trajectories come from. My colleagues and I have found that people demonstrating desired behavior distinguish between whether they are just doing an example of the behavior or whether they are showing someone else how to do the behavior.

Consider the case we get if we tell someone three things: orange tiles have a penalty; the goal is to get to yellow; and the safe colors are white, blue, and purple. The path on the left is a reasonable example to satisfy the goal under these constraints. The path on the right is equally good as an example of doing. But the path on the right is a more complete and informative way of showing someone all the tile colors that are safe and which have a penalty.

We recruited people online and asked them to either do (solve the problem) or show (demonstrate a solution to someone else so they could do it). Here are paths taken by 60 participants. A few are confused, but the majority of the people in the show condition do some extra traveling to highlight where the safe cells are. They wiggle back and forth across safe boundaries like between white and blue and blue and purple. It's the kind of behavior you'd expect if the demonstrators are reasoning about the mental state of the observer. They are being better teachers.

LESSON 21 | INVERSE REINFORCEMENT LEARNING FROM PEOPLE TRANSCRIPT

Indeed, our inverse RL algorithm makes better use of these more instructive demonstrations in learning. When we train an inverse RL agent to extract the rewards from do demonstrations created by people, the learners are about 65% confident in the result. But if we use show demonstrations created by people, the learners' confidence rises to 92%.

My team and I even went one step further. We created a new inverse reinforcement learning algorithm that specifically reasons about why the teacher chose the particular demonstrations that were chosen. Basically, the new algorithm asked itself, what reward function would I demonstrate if I were trying to teach someone the rewards? This kind of recursive reasoning takes place all the time when we humans are communicating with each other.

It's hard, in general, to reason across many levels of communication at once. But this grid problem was small enough that we could write a recursive reasoning program to solve it. A learner using the show demonstration and this recursive reasoning is 98% confident in the correct task.

Using inverse reinforcement learning to create trainable learning systems will provide us with better ways of delegating to our machines tasks that we want done. That will empower more of us, providing ways for people with good ideas to put those ideas out into the world to benefit others. Some researchers such as Stuart Russell at Berkeley think that these intent-conveying ideas like inverse reinforcement learning are even more significant to the future of our society. The thinking is that these ideas may be a way—perhaps the best way—to address concerns about malfunctions in hypothetical future computer systems that are super intelligent.

A typical doomsday scenario begins when we ask the machines to do something relatively innocuous, like improving the speed with which they can look up data in a big database. Being dutiful optimizers and powerful problem solvers, they conclude that they will produce better solutions to their given problem if they seek to improve their own capabilities, assure their own survival, and take control over more and more resources, like energy and raw materials.

Steadfastly pursuing these intermediate goals would put the machines in conflict with what people want. People also need resources and people are apt to turn off a machine that is competing for the same resources. So,

stopping people from stopping the machine would become a natural goal for the super intelligent machine to take. In fact, the fewer pesky humans there are around the world, the less likely the machine's goals will be thwarted.

At the root of this robot apocalypse scenario is an idea that a reinforcement learner can overfit to the reward function it is given. The presumption is that the machine learner will devise a policy that maximizes the specifics of the reward function it was given, without being able to generalize to a broader set of circumstances.

One solution to this problem is to make sure that machines realize that there is uncertainty in the goal they are given. The goal they have might be missing key elements like improve database access time, using only existing capabilities and resources and without killing any humans. Learning tasks from people using some version of inverse reinforcement learning might be the key to keeping powerful machines, and even adorable Aibos, working in the service of humanity.

LESSON 22

CAUSAL INFERENCE

COMES TO MACHINE LEARNING

Deep learning seems to be everywhere. But it's not the be-all-and-end-all approach to machine learning. Deep learning methods are beset by at least two related challenges: All of their predictions are based solely on correlations found by the machine learner without having anything to say about causal effects, and very large amounts of data are almost always needed to support and refine models based solely on the quality of the correlation. But an approach called causal inference, which has been steadily growing in popularity since 2000, has the potential to avoid the weaknesses of deep neural networks.

Correlation versus Causation

Some of the shortcomings of machine learning you've learned about are a direct consequence of not looking beyond correlations. For example, it was data correlations that suggested that people with asthma are less likely to get pneumonia in the hospital—even though the truth is exactly the opposite.

More careful reasoning can lead to methods that can assess the actual causal relationships and avoid the usual pitfalls. And in cases where the data cannot support a causal interpretation, the methods explicitly flag that impossibility.

As an example, consider the classic case of cigarette smoking and cancer. For decades, cigarettes were known to correlate with cancer but had not been proven to *cause* cancer. And during the late 1950s until the early 1960s, perhaps the world's leading statistician, Ronald Fisher, argued that it was premature to conclude that smoking causes cancer.

Causation
indeed does
not equal
correlation.

Fisher asserted that the correlation between smoking and cancer was not the same as a causal link between smoking and cancer. In his view, the only scientifically valid way to be sure would be to select people randomly, force some to become smokers for years, and force others to avoid smoking. That is not an experiment we can ethically do.

But in the absence of such experiments, we cannot entirely rule out that some other factor can be confounding our ability to understand—for instance, people who get cancer might also have a genetic predisposition to become smokers.

In the smoking example, what we want to know is whether smoking causes cancer. Importantly, we want to know whether not smoking will change the likelihood of getting cancer, and we want to use the prediction to intervene in the world.

A **causal graph**—which is a close cousin of Bayesian networks—is a tool that makes causal inference possible. A causal graph is a powerful, and often necessary, tool for making predictions that remain valid even in the face of interventions.

Unobserved Confounders

The big idea of causal inference is that we can sometimes answer questions about the data that are very different from how the data was collected.

If we draw a Bayesian network and then use the data to estimate its parameters, we have computational methods that will let us answer what-if questions like “if I turn on the sprinkler, how likely is the ground to become wet?”

Standard machine learning methods that only model the correlational structure of the data cannot be used this way directly. The causal graph structure is important for propagating information about the explicit changes to the distribution of the data.

We might assume that our data includes information about all of the relevant variables. And of course, we’re aware that there are some aspects of the world that are not in our model. These factors might be unobserved because we are assuming that they have no influence on the relationships between our variables.

But in the real world, it’s often the case that there are variables that are important *and* their values are not recorded in the data. That could be because those variables were left out accidentally or that they were simply too expensive or otherwise difficult to observe.

Such variables are called unobserved confounders, or even nuisance variables, because we can’t see them, yet they mess with our results.

Calculating Correlation and Causation Effects

Here’s an example causal graph studied by a founder of causal machine learning, Judea Pearl. This example illustrates a Bayesian technique that lets you calculate both correlation and causation effects.

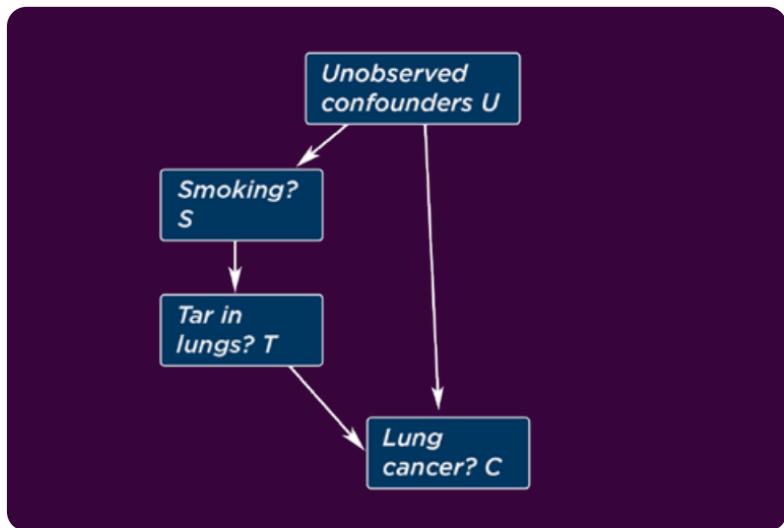
The causal graph includes variables that could be relevant: Does a given person smoke (S)? Does the person develop lung cancer (C)? On an x-ray, can you see tar in the person’s lungs* (T)?

* That was a feature that researchers in Great Britain had observed in many smokers and also many people with lung cancer. Researchers believed tar could be involved in the development of cancer.

There's one more element in the causal graph. A concern raised by Fisher and others was that a genetic factor might be influencing both smoking and cancer, making it falsely appear that smoking was the cause of the cancer. But since such a genetic factor was not identified, it cannot be included in the data.

And that throws a monkey wrench into our scheme for determining the effect of smoking on lung cancer. We can't learn the probability of lung cancer given tar in the lungs and the unobserved variable because the unobserved variable is unobserved—it's just not present in our data.

A controlled experiment could establish causality by breaking the link between the unobserved confounders and smoking. The result would be a causal graph with no link to the unobserved confounders. But we have no ethical way to do such an experiment.



In the causal graph we have, we can't estimate the probability of lung cancer based on the unobserved confounders. But that's OK. We could still collect data on the associations between all the observable variables, and surprisingly their correlations can tell us something about the causal effect between smoking and cancer.

The wonder of causal analysis is that there is a way to use quantities learned in the real world to infer values like those in the controlled world so that we can directly assess causality.

These calculations can be made using a tool developed by Pearl and his colleagues called the *do*-calculus. It provides a set of algebraic rewrite rules that relate causal quantities using the *do*-operator to traditional conditional probability expressions.

$$P(C|do(S)) = \sum_{U,T} P(U) P(T|S) P(C|T,U)$$

$$\begin{aligned} & \sum_{S',T} P(C|S',T) P(Z|S) P(S') \\ &= \sum_U \sum_{S',T} P(U) P(C|S',T,U) P(T|S,U) P(S'|U) \\ &= \sum_U \sum_{S',T} P(U) P(C|T,U) P(T|S) P(S'|U) \\ &= \sum_U \sum_T P(U) P(C|T,U) P(T|S) \sum_S P(S'|U) \\ &= \sum_U \sum_T P(U) P(C|T,U) P(T|S) \end{aligned}$$

What's really impressive is that all of these quantities can be learned from observational data. Causal graphs let us find out the causal impact of smoking just from observations—without depending on knowledge of the unobserved confounder and without needing to intervene.

For many causal graphs, it is possible to use the *do*-calculus to find a set of expressions that do not depend on the unobserved confounders yet allow causal expressions to be computed.

It's not always possible, however. It depends on the structure of the causal graph. For instance, in this smoking example, if an unobserved confounder could influence the tar variable, our ability to infer the causal effect of smoking on cancer would vanish.

But it has been shown that the *do*-calculus can always figure out whether a causal expression exists—and if it does exist, how to compute it.

Bayesian Causal Models

Classical Bayesian methods already provide a powerful way for probabilistic networks to capture the correlational information in a set of data. They can learn not only the parameters of a Bayesian model but even the network structure itself.

Bayesian causal inference lets us use that same overall representation to go deeper. Classical Bayesian methods just assess how often features are related in the distribution from which the data was collected. Bayesian causal analysis lets us learn how an output would change after an intervention that effectively shifts the data distribution.

When human beings learn, we naturally seem to extract a causal essence, and we identify causal relationships even when learning from only a few examples. Bayesian causal models open the door for machine learners to model the world more like humans do.

Researchers are working to find ways to combine causal inference with the benefits of deep networks and other classical machine learning methods. A merger of deep learning and Bayesian causal methods could make it easier for each of us to control the machine learning applications touching our day-to-day lives.

Try It Yourself

Follow along with the video lesson via the Python code:

[L22.ipynb](#)

Python Libraries Used:

dowhy: Causal inference package from IBM.

dowhy.CausalModel: Builds a causal model.

os: Provides operating system access for manipulating files and directories.

pandas: Library for organizing datasets.

sys: Operating system support.

Key Terms

causal graph: A cousin of Bayesian networks that captures the relationship between variables even when interventions are taken to change specific values.

READING

Hoover, "Dueling Economists."

Huszar, "ML beyond Curve Fitting."

Pearl and Mackenzie, *The Book of Why*.

Schölkopf, "Causality for Machine Learning."

QUESTIONS

1. What is the difference between the outcome of an intervention $Pr(x|do(y))$ and the conditional probability $Pr(x|y)$?
2. Given a causal diagram, we can do a controlled experiment to estimate the outcome of an intervention $Pr(x|do(y))$. Can we compute this quantity without resorting to a controlled experiment? Why or why not?
3. When we applied machine learning algorithms to the *Titanic* data from [Lesson 10](#) on machine learning pitfalls, we found they concluded that being female was associated with higher survival rates. The higher rates are due, in large part, to the fact that female passengers were treated differently by the crew and given first access to the lifeboats. It does not account for whether female passengers were more likely to survive, all other things being equal. Can a causal analysis provide a more nuanced perspective on the data? Apply the dowhy learner to the data, using "Sex" as the treatment and "Survived" as the outcome. Look at the average treatment effect and the causal estimate. What do you conclude about the impact of sex on survival?

Answers on page 486

Causal Inference Comes to Machine Learning

Lesson 22 Transcript

Deep learning seems to be everywhere. But deep learning is not the be-all-end-all approach to machine learning. It is important to keep in mind that deep learning methods are beset by at least two related challenges. First, all their predictions are based solely on correlations found by the machine learner, without having anything to say about causal effects. Even the deterministic and logical rules we've seen for decision trees never have the explanatory power of something like Einstein's $E=mc^2$.

And, second, very large amounts of data are almost always needed—that's to support and refine models based solely on the quality of the correlation. But there is an approach called causal inference—that has been steadily growing in popularity since 2000—that has the potential to avoid the weaknesses of deep neural networks. Some of the shortcomings of machine learning we've seen are a direct consequence of not looking beyond correlations. For example, it was data correlations that suggested that people with asthma are less likely to get pneumonia in the hospital even though it's exactly the opposite.

We'll see that more careful reasoning can lead to methods that can assess the actual causal relationships and avoid the usual pitfalls. And, in cases where the data cannot support a causal interpretation, the methods explicitly flag that impossibility. As an example, let's consider the classic case of cigarette smoking and cancer. For decades, cigarettes were known to correlate with cancer, but had not been proven to cause cancer. And, during the late 1950s until the early 1960s, perhaps the world's leading statistician, Ronald Fisher, argued that it was premature to conclude that smoking causes cancer.

Early in his career, Fisher had famously pioneered ways to design statistically rigorous experiments. So, it carried a lot of weight when he asserted that the correlation between smoking and cancer was not the same as a causal link between smoking and cancer. In Fisher's view, the only scientifically valid way to be sure would be to select people randomly, force some to become smokers for years, and force others to avoid smoking. That is not an experiment we can ethically do. But in the absence of such experiments,

LESSON 22 | CAUSAL INFERENCE COMES TO MACHINE LEARNING TRANSCRIPT

we cannot entirely rule out some other factor confounding our ability to understand; for instance, people who get cancer might also have a genetic predisposition to become smokers.

In the smoking example, what we want to know is whether smoking causes cancer. Importantly, we want to know whether not smoking will change the likelihood of getting cancer, and we want to use the prediction to intervene in the world. The particular tool we'll focus on that makes causal inference possible is a causal graph, which is a close cousin of Bayesian networks. We'll see that a causal graph is a powerful, and often necessary, tool for making predictions that remain valid even in the face of interventions.

Let's look at an example to develop these ideas in more detail. The data is just made up, but let's pretend it's real for the purposes of the example. I have data over nine mornings every other Wednesday during Spring 2015. On each day, I answered a series of questions. When I went out in the morning, was the ground wet? Was it April at the time? Did the weather report predict rain? Did I see people using umbrellas when I looked out the window? Did I run my sprinkler? But to build a causal model, we want to use Bayesian methods, so let's start with a naive Bayes classifier to see what it does.

A naive Bayes model begins with a prior probability for the class; here, whether the ground is wet. We use the naive Bayes model by observing evidence from each of the variables and producing an updated probability for the class, predicting whether the ground will be wet under the given circumstances. The variables in this example are whether it is April, whether there is rain predicted, etc. The naive Bayes model sets values for the variables based on the selected class, and each is set probabilistically and independently of every other variable. We depict the naive Bayes model by drawing arrows from the ground wet class to each variable to indicate how each variable depends on the class.

To learn the naive Bayes parameter values for this example, we need to use the data to estimate a prior probability for whether the ground is wet on any given day. The ground was wet five days out of nine, so we estimate five-ninths, which is 0.56, for the prior probability of the ground being wet. Once we have a prior probability to get us started, then we need conditional probabilities: 1) Given the ground is wet, we need estimates of the likelihood that it's April, the likelihood that rain is predicted, whether umbrellas

are seen, and whether the sprinklers are run. A second set of conditional probabilities is: 2) Given the ground is not wet, what are the likelihoods that its April, rain is predicted, umbrellas are seen, and sprinklers are run. Of the five times when the ground was wet, four of them had umbrellas visible. So, we estimate the probability of umbrellas being visible given that the ground is wet as four-fifths or 80%.

On the other hand, in the four times the ground was not wet, the sprinkler was only run once. So, we estimate the probability of the sprinkler being run given that the ground is not wet as one-fourth or 25%. We can use the data to estimate all eight of the conditional probability values.

Now, that we've learned these conditional probability values, we can use them for prediction. How likely is the ground to be wet if it's April, the prediction is for rain, I did not see any umbrellas, and I ran the sprinkler? We simply multiply the conditional probabilities together for the two cases—ground wet and not ground wet. For the umbrella seen category, since we did not see an umbrella, we use the probability that an umbrella is not seen in the two cases. We get that probability by just subtracting the umbrella seen conditional probability from 1. When we multiply the conditional probabilities together, we get roughly 0.020 for ground wet and 0.005 for ground not wet. We divide by the sum to normalize these values, giving us an 81% probability of the ground being wet.

So, that's great. It gives us a way of making predictions. And, probable wet ground on a day like the one described seems like a very sensible prediction. We could go through a similar calculation with this model to answer another question. What's the probability the ground is wet given that we've seen an umbrella? Doing the calculation results in an 80% probability that the ground is wet.

Let's think for a moment about what it means that the probability of ground wet given umbrella is 80%. According to this model, the ground is wet on a random day with probability 56%. But on a day when we see an umbrella, the probability increases to 80%. That's a reasonable prediction. Most of the time, seeing an umbrella is indeed associated with wet ground. These two events are highly correlated. But what should we do if we want to cause the ground to be wet? We can't change whether or not it's April.

LESSON 22 | CAUSAL INFERENCE COMES TO MACHINE LEARNING TRANSCRIPT

And, we can't change whether the weather report says it will rain. But we can turn on the sprinkler and we can pick up an umbrella and go outside. Would either of these interventions have a causal impact on the wetness of the ground?

Let's draw our naive Bayes model as a network. Now, we can incorporate the effects of the interventions in the model. To do so, we run our model in the normal way, but when we hit a variable that we're controlling, we'll force it to take on the value we want. To generate data with a Bayesian model like this, we visit the nodes. We start from nodes that have no incoming links, and we move on to nodes whose incoming nodes have already been visited. Each time we visit a node, we select a value for the node according to its conditional probabilities.

If we want to explore the causal impact of choosing to carry an umbrella, we need to treat the umbrellas seen node differently. When we visit that node, we simply set it to 1, meaning true. In this instance, the value is no longer being selected within the model but instead by a conscious intervention on our part. And what impact does our making this conscious intervention on the model have on the probability the ground is wet? None.

In a naive Bayes model, the variables are chosen given the predicted class. So, no interventions on the variables can have any impact on the class. Turning on umbrellas seen has no causal impact on ground wet, which seems pretty reasonable. But here's where naive Bayes models are naïve: they don't distinguish between umbrellas and sprinklers. Although we can intervene on both of these variables—by carrying an umbrella or by turning on a sprinkler—only sprinklers would be likely to make the ground wet in the real world.

To get a handle on how our decisions cause something to happen, we're going to need to become less naive. Instead of naive Bayes, we'll need the full power of Bayesian networks. A Bayesian network model generalizes the naive Bayes setting to allow any pattern of directed relationships between the variables—nodes can have any number of parents and any number of kids. There is only one important restriction. To make it so that no variable's value depends on itself, the directed edges cannot form cycles in the network.

The generative process is the same as for the naive Bayes model; visit the nodes in order, selecting a value for each from its conditional probability distribution. In this model, ground wet has incoming arrows, from sprinkler run and from predicted rain. The only node without an incoming edge is April, abbreviated A . The condition of being April is not influenced by any of the other variables. The running of the sprinkler, abbreviated S , is conditioned on whether it is April. And the prediction of whether there will be rain in the weather prediction, abbreviated R , is also conditioned on whether it is April. Whether I see people carrying umbrellas, abbreviated U , depends on whether rain was forecast.

One thing to notice about this model compared to the naive Bayes model is that some nodes have more than one parent. Whether the ground is wet, abbreviated W , is conditioned on a combination of both whether the sprinkler was run and whether rain is in the forecast. So, the parameters for this node will be necessarily more complex than what we saw in the naive Bayes model.

Even though the structure is more complicated, learning in this model is a simple generalization over what we did in the naive-Bayes case. For example, the ground wet W node needs a probability for being true for each of the possible truth values of its parents. Looking at the row in the table marked predicted rain, not sprinkler run, we just focus on the rows in the training data where rain is predicted and the sprinkler is not run. There are three such rows in the training data. In two of these three rows, the ground is wet. So, we fill in two-thirds or 67% in our model.

We can use this model for predictions, just as we did for the naive Bayes model. The model is more complex, but some simple predictions are actually a bit easier. A key concept for understanding the meaning of interventions—when we force a variable to take on a specific value—is to distinguish a causal intervention from a conditional probability. This new Bayesian model gives us a great opportunity to illustrate this distinction.

First, let's look at the probability that the ground is wet, given that we observe that the sprinkler is run. In this notation, we want to work out P of W given S . By the definition of conditional probability, that's P of W and S together divided by P of S . We'll first dive into P of S . By the law of total probability and the conditional independencies in the model, we can write P of S as a sum over the other variables of the product of each of the conditional probabilities of each of the variables.

LESSON 22 | CAUSAL INFERENCE COMES TO MACHINE LEARNING

TRANSCRIPT

Rearranging the summations, and using the fact that probabilities sum to one, we can simplify this sum down a bit. First, the sum over U and the probability of U can be dropped. Then the sum over W and the probability of W can be dropped. Finally, the sum over R and the probability of R can also be dropped. That's as far as we can go, because the probability of S depends on A so we can't factor it out of the sum over A . But what's left is pretty simple so we can just do the computation. We're summing over the two cases of April, A , and not April, not- A . For each one, we multiply the probability of April times the probability of sprinkler run, S , given April. After multiplying the quantities and then adding the two cases together, we get 0.4439. That's the probability of the sprinkler being run, P of S .

Now that we know the probability of the sprinkler being run, let's return to computing the probability of wet ground given sprinkler run, P of W given S . The next quantity we need is the probability of wet ground and sprinkler run, P of W and S . We can do a similar derivation to how we found P of S , but things are not quite as simple. That's because of the link from rain prediction to wet ground; the probability of wet ground depends on the rain prediction.

What we're left with is a sum over both A and R —the combination of whether or not it is April and whether or not rain is predicted.

We're summing over the combination of values for A and R . For each one, we multiply the probability of four of the quantities—all of the variables except U , which we had determined isn't needed for this calculation. After multiplying the quantities and then adding the four cases together, we get 0.3608. That's the probability of W and S together—the probability of wet ground and sprinkler run at the same time. Dividing by 0.4439, the probability of sprinkler run we had worked out, we get 0.8128. So, that's our estimate for the probability the ground is wet given the sprinkler is run.

It might seem natural to think of the probability the ground is wet given the sprinkler is run as being an estimate of what would happen if we turned on the sprinkler. After all, we get this quantity by observing cases where the sprinkler is on. However, this analysis is incorrect. To figure out what would happen if we explicitly turned the sprinkler on, we have to take into account that we are doing it unconditionally—not just in situations where the sprinkler would have been turned on in the collected data.

We can derive the value of a deliberate intervention by doing a little surgery on our diagram. Instead of asking whether the sprinkler was run, we state the sprinkler has indeed been run. There is no longer any influence

from April. The sprinkler is something that we make sure happens, independent of everything else. To understand the impact that this change, this intervention, has on the probabilities, we can work out P of W , the probability of the ground being wet, in this revised model where we've turned on the sprinkler unconditionally.

Working things through, we get the probability of wet ground is 0.8342 in this revised model. Roughly, that was 81% chance of wetness if we observe the sprinkler being on and an 83% chance of wetness if we actively turn on the sprinkler. It's a small change, but the thing to notice is that the numbers are different.

In the language of causal inference, we've computed P of W given $\text{do } S$. That's the probability of ground wet when we cause run sprinkler to be true. The value is different from the P of W given S —the probability of ground wet when we observe run sprinkler is true. Causation indeed does not equal correlation. And, here, we can compute probabilities separately for causation and for correlation, which is empowering.

The big idea of causal inference is that we can sometimes answer questions about the data that are very different from how the data was collected. If we draw a Bayesian network and then use the data to estimate its parameters, we have computational methods that will let us answer what-if questions like, If I turn on the sprinkler, how likely is the ground to become wet? Standard machine-learning methods that only model the correlational structure of the data cannot be used this way. The causal graph structure is important for propagating information about the explicit changes to the data distribution.

We've been discussing a very simple example, but where causal inference really shines is in a slightly more complex setting. So far, we've assumed that our data includes information about all of the relevant variables. Of course, we are aware that there are some aspects of the world that are not in our model. These factors might be unobserved because we are assuming they have no influence on the relationships between our variables. For example, we could imagine another unobserved variable like whether I had coffee in the morning, but it's fine to leave that out of the model.

But in the real world, it's often the case that there are variables that are important and their values are not recorded in the data. That could be because those variables were left out accidentally or that they were simply

LESSON 22 | CAUSAL INFERENCE COMES TO MACHINE LEARNING TRANSCRIPT

too expensive or otherwise difficult to observe. We call such variables unobserved confounders, or even nuisance variables, because we can't see them and yet they mess with our results.

Here's an example causal graph studied by a founder of causal machine learning, Judea Pearl of UCLA. This example takes us back to the smoking and cancer question and illustrates a Bayesian technique that lets you calculate correlation effects and causation effects. Our causal graph includes variables that could be relevant—does a given person smoke, S ? Does the person develop lung cancer, C ? On an X-ray, can you see tar in the person's lungs, T ? That was a feature that researchers in Great Britain had observed in many smokers and also many people with lung cancer. Researchers believed tar could be involved in the development of cancer.

There's one more element we need to include in the causal graph. A concern raised by Fisher and others was that a genetic factor might be influencing both smoking and cancer, making it falsely appear that smoking was the cause of the cancer. But since such a genetic factor was not identified, it cannot be included in the data. And, that throws a monkey wrench into our scheme for determining the effect of do-smoking on lung cancer. We can't learn the probability of lung cancer given tar in the lungs and the unobserved variable because the unobserved variable is unobserved; it's just not present in our data.

A controlled experiment could establish causality by breaking the link between the unobserved confounders and smoking. The result would be a causal graph with no link to the unobserved confounders. But we have no ethical way to do such an experiment. In the causal graph we have, we can't estimate the probability of lung cancer based on the unobserved confounders. But that's okay. We could still collect data on the associations between all the observable variables and, surprisingly, their correlations can tell us something about the causal effect between smoking and cancer.

Here's how. From an algebraic point of view, we want to estimate P of cancer C given do smoking S . That is, to assess causality, we want to know what happens if we intervene to hold constant the variable of interest. We can write out an expression that gives the probability given our causal intervention do S . In the controlled experiment, we create a new world and we can estimate P of cancer C given smoking S in this new world. I'm writing the new controlled world by putting a superscript little c on the P . We can write the definition of P of cancer C given smoking S in the controlled world.

Using some basic algebra, we conclude that the values we would estimate in the controlled world successfully quantify the causal effect of smoking in the real world. But it would be unethical to run experiments in the controlled world where we force people to smoke or not to smoke. So, this finding can't help us. The wonder of causal analysis is that there is a way to use quantities learned in the real world to infer values like those in the controlled world so we can directly assess causality. These calculations can be made using a tool developed by Pearl and his colleagues called the do-calculus. It provides a set of algebraic rewrite rules that relate causal quantities using the do operator to traditional conditional probability expressions.

I'll show just the result it produces for this example. To estimate P of cancer C given do smoking S , we need to learn a prior probability, P of S , for how common smoking is in the population. Then, we need two conditional probabilities: P of cancer C given smoking S and tar T : How is cancer impacted by smoking and tar in the lungs? P of tar T given smoking S : How is tar influenced by smoking? There is a small twist, though. For the P of tar T given smoking S quantity, we don't use the value of S we're summing over. Instead, we use the value of the intervention we're making—true—because we're computing the effect of forcing smoking to be true. Then, we sum up, for all values of S and T , the product of these three quantities.

What's really impressive is that all of these quantities can be learned from observational data. Causal graphs let us find out the causal impact of smoking, just from observations without depending on knowledge of the unobserved confounder and without needing to intervene. As a sanity check, let's go through the algebra to show that this expression derived from the do-calculus really does give the right answer to the causal question.

Take the expression and use the law of total probability to sum up over the unobserved confounder U . Using the conditional independence structure of the model, we can simplify some of the quantities. Then we can rearrange the sums, putting the sum over S -prime on the inside and then noting that it sums to 1. That means we can drop it. What's left is exactly the expression we need— P of cancer C given do smoking S .

For many causal graphs, it is possible to use the do-calculus to find a set of expressions that do not depend on the unobserved confounders and yet allow causal expressions to be computed. It's not always possible, however.

LESSON 22 | CAUSAL INFERENCE COMES TO MACHINE LEARNING TRANSCRIPT

It depends on the structure of the causal graph. For instance, in the smoking example, if an unobserved confounder could influence the tar variable, our ability to infer the causal effect of smoking on cancer would vanish. But it has been shown that the do-calculus can always figure out whether a causal expression exists and, if it does exist, how to compute it.

To explore causal inference, the tool we'll use is a Microsoft library released in 2018 called dowhy. It can learn causal graphs from data and carry out do-calculus derivations to find ways of using observational data to reach causal conclusions. You can give dowhy numbers or you can give it a directed graph. Here's a small demonstration of what dowhy can do and why.

A clinical trial in the 1980s called the Infant Health and Development Program or IHDP looked at the cognitive capacity of premature infants and the impact of treatments on their development. The IHDP dataset contains instances from their randomized study and records properties of the children and their caregivers. An important question the data can address is, does treatment from a specialized therapist result in better outcomes than treatment from other care givers?

The data has a bunch of features. Treatment is a 0/1 variable that indicates the absence or presence of specialized treatment for each child. The feature labeled y_{factual} is an assessment of the child's improvements in cognitive development. The other features are not given semantically meaningful names as a way of protecting the privacy of the participants in the study. We call the dowhy function CausalModel to process the data. The function identify_effect then analyzes the model to derive a causal effect.

Note that when you run this routine, the code reminds you that it's making some educated guesses about the way that the unobserved confounders can impact the model. The dowhy software steers users away from taking the results at face value and into looking more closely at possible causal effects. When the analysis is complete, we can estimate the effect of the treatment on the outcomes in two ways.

The first is what's called the average treatment effect or A-T-E. That's a correlational measure of treatment and outcomes. To estimate the average treatment effect, we subselect the instances where the treatment is given, and store those instances in data_1. Then, we select out the instances where the treatment was not given and store those instances into data_0. The A-T-E is the difference between the means in these two sets.

For this data, we get an A-T-E of 4.02. So, the treatment results in an outcome boost of over four points on the scale being used. The average outcome in this dataset is only 3.16, so a boost of 4.02 looks quite large. But one of the confounders in this data was that children were not always selected for treatment at random. So, there's a backdoor propensity score weighting that uses the do-calculus to re-assess how much of the gains are really attributable to the treatment and not other factors.

Dowhy's estimate_effect function uses the causal analysis that was performed to more precisely characterize the treatment effect, like what we did in the cancer-tar-smoking example. This method reveals a difference of 3.38 instead of 4.02. So, treatment by specialists is indeed helping, but not as much as a less sophisticated analysis would suggest. And, that's really the punchline for research on causal inference. Classical Bayesian methods already provide a powerful way for probabilistic networks to capture the correlational information in a set of data. They can learn not only the parameters of a Bayesian model, but even the network structure itself.

Bayesian causal inference lets us use that same overall representation to go deeper. Classical Bayesian methods just assess how often features are related in the distribution from which the data was collected. Bayesian causal analysis lets us learn how an output would change after an intervention that effectively shifts the data distribution. In many important cases, it's even possible to analyze the causal graph to create a specialized learning algorithm that produces accurate causal predictions in the face of unobserved confounders.

When human beings learn, we quite naturally seem to extract a causal essence and we identify causal relationships even when learning from only a few examples. Bayesian causal models open the door for machine learners to model the world more like humans do. In contrast, deep neural networks, for all their superiority in interpreting low-level perceptual data, rely on nothing more than correlations, which are baked into what they learn. There is no way to ask a neural network to make predictions using some hypothetical alternative data distribution. They only make predictions based on the distribution on which they were trained.

LESSON 22 | CAUSAL INFERENCE COMES TO MACHINE LEARNING TRANSCRIPT

So, researchers are working to find ways to combine causal inference with the benefits of deep nets and other classical machine-learning methods. A successful merging of causal inference and neural nets holds the promise of making more accurate classifiers, with far less data than is possible now. By reducing the amount of data needed for training, a merger of deep learning and Bayesian causal methods could make it easier for each of us to control the machine learning applications touching our day-to-day lives.

LESSON 23

THE **UNEXPECTED POWER** OF OVER-PARAMETERIZATION

What was it, fundamentally, that triggered the shift in 2015 that got deep learning off the ground? What kicked off the wave of changes you've been learning about in this course and that all of us will likely be feeling for decades to come? After all, the ideas that make up deep learning have roots in a more distant past.

The Deep Learning Revolution

At the heart of deep learning is using calculus—in particular, differential calculus. The roots of calculus go back to its formalization by Isaac Newton and Gottfried Wilhelm Leibniz in the 1600s and, before them, all the way back to mathematics developed in Persia in the 1400s.

Of course, none of these predecessors were using calculus to train neural networks to update parameters incrementally to minimize a loss function. That idea—the idea of training a network by gradient descent—came much later, when three research teams independently discovered the backpropagation method for efficient gradient descent in the mid-1980s.

Three reasons are typically given for why general neural networks waited three more decades before bursting onto the scene in 2015:

1. Computer speed increased. We needed faster computers to process vast amounts of data, and computers had been getting progressively faster for decades. Moreover, general-purpose graphics processing units (GPUs), developed for gaming starting in around 2007, turned out to be just as big a boost for machine learning as for their intended use in gaming.
2. The availability of datasets expanded. Machine learning solutions are defined by their data, and society began living online for the first time. Facebook launched in 2004; the first-generation iPhone came out in 2007. Digital images, digital sound, digital video, machine-readable text, and other kinds of personally relevant data were being amassed in unprecedented quantities.
3. Improved machine learning algorithms helped larger-size neural networks learn more effectively. There were new representational spaces, loss functions, and optimization algorithms.

But there's a fourth reason that may be just as important as the others—and it's much more surprising. One of the things that makes deep networks work so well is that researchers developed a willingness to try things that “shouldn't work.”

Before 2015, any mainstream data scientist would have told you that networks with millions upon millions of parameters cannot be trained to produce reasonable answers. The view was that the data would not sufficiently constrain all of those free parameters.

It was a central dogma in statistics—learned through hard experience—that having too many parameters in your model or rule would mean that you would be overfitting, with all the bad consequences you learned about in **Lesson 09**.

Your model would learn idiosyncratic, chance variations from your data instead of representing your data in a generalizable way. Indeed, overfitting is *the* concept from machine learning that everyone should know.

So there was a conceptual barrier that had to be overcome before the vast majority of potential contributors to deep learning could actually get on board.

So, while overfitting is undeniably bad, it was important for the field of machine learning to also get beyond the standard view of overfitting.

That will let us dive deeper into the implications of training big networks whose set of weights, or parameters, is very large compared to the amount of data that's available. Such models can be called over-parameterized because they have more parameters than it seems like they should, relative to the amount of data.

These networks violate what we thought we knew about data analysis and machine learning, but they are also central to the success of deep networks in solving hard real-world learning problems.

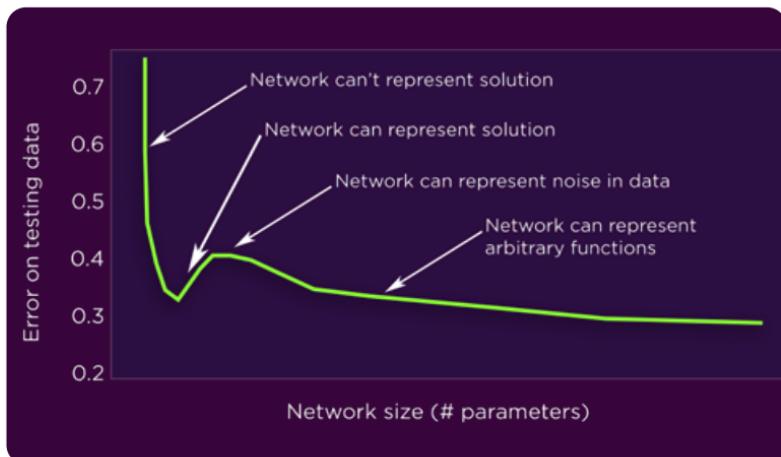
The Double Descent Phenomenon

The standard warning about overfitting is that you want to give a machine learning system just the right amount of representational power, given the complexity of the function you are fitting and the amount of data you have to estimate it. You don't want to over-parameterize—use way more parameters than you can fit with the available data.

But the deep learning revolution suggests that we look more closely at what happens if we have what would previously have been “too many” parameters.

And in 2018, an unexpected phenomenon referred to as the double descent was identified: Error on the testing data improves, gets worse, but then improves again.

Scientists at OpenAI demonstrated double descent in a variety of deep networks at the forefront of the field. These include transformer networks from natural language processing and convolutional networks from image processing.



The error on the real-world data followed a high-low-high-low pattern as the network size was increased. They found the same pattern if they kept the network size fixed but changed how long it was trained.*

Initially, the network is too small to solve the problem at hand and the testing error is poor. The network underfits the data. Once the network has the capacity to capture the solution to the problem, testing error hits a minimum. The network is fitting the data well.

With a further increase in capacity, the network is able to capture the solution but also the noise and inconsistencies in the data, resulting in increasing testing error. The network is overfitting the data. That's where the story used to end.

* Training time is also a measure of network complexity. Long training times lead to more complex networks because there is more time to amplify the weights and express more complex functions.

But increasing the capacity even *more* allows the network to go beyond representing just the solution and the noise. Now it can approximate just about any function it might need. At that point, the network is able to match the data, even with the noise, but it can also interpolate in the regions between the training points.

In these interpolation zones, predictions on testing data are typically quite good. The overall prediction performance of the network with noise incorporated returns to the level it reached when it first captured the solution and ignored the noise.

In the first descent, the network captures the solution but ignores the noise because it simply doesn't have the capacity to fit the noise. The network is like a dog that fetches a ball but does not chase cars driving on the other side of a fence. It's not that the dog doesn't want to go after the car; it's just that external constraints prevent it from doing so.

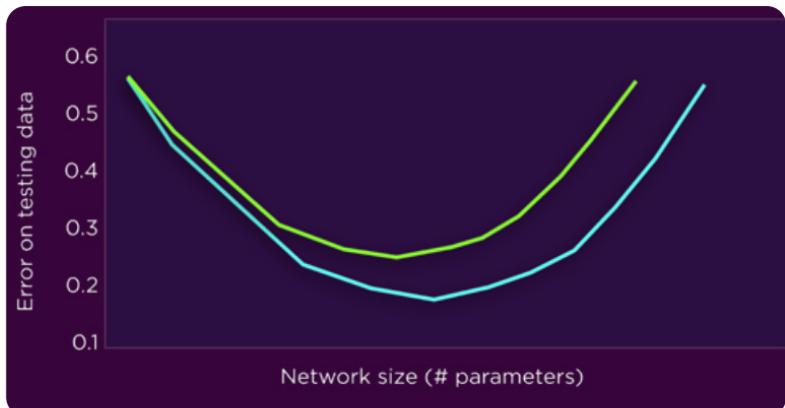
In the second descent, the network captures the solution and interpolates around the noise. Now the network is like a dog in an open yard that's had many, many times to chase cars. It is briefly distracted by cars, but it comes back to the yard right away.

The analogy isn't perfect, but the lesson is a good one. It's natural to try to restrain a learning algorithm so that it doesn't "hurt itself" by chasing after the noise in the training data. But we can actually reach the same solution by providing additional freedom. The learner can chase after the noise but still return to reasonable values for inputs outside the training set, thereby getting the benefit of still producing good answers on unseen data.

Why Do Over-Parameterized Networks Learn?

The double descent phenomenon leads to another counterintuitive result, although the practical implications so far have been minor.

A central tenet in standard machine learning has been that more data is better. In the classical single-descent setting, adding more data allows us to define parameter values more accurately, leading to better prediction on unseen data. The extra data stretches the testing error curve down and to the right, so error is less at all network sizes. In standard machine learning, more data is always good.



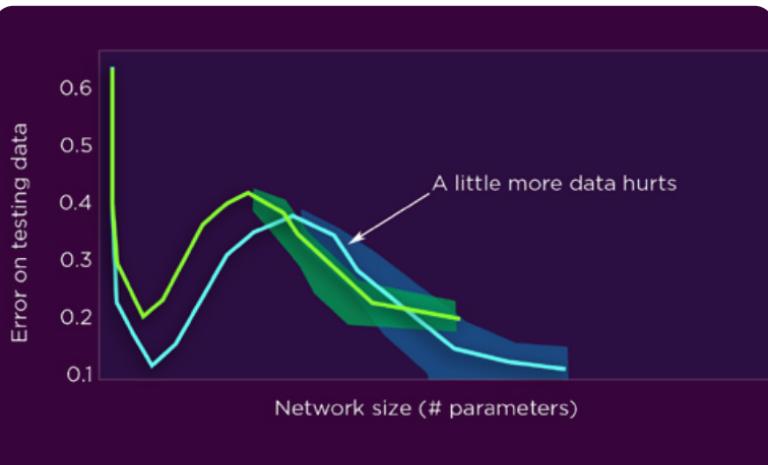
But consider what happens in the new over-parameterized double-descent deep learning regime. Because the curve has a hump, stretching it down and to the right does not necessarily result in error improvements at all network sizes on the curve. Here, there's a range of network sizes where additional data actually hurts generalization performance a bit.

The notion that more data is bad is pretty shocking, but we're talking about a very weak effect here. Each of these curves is an average of values with error bars above and below the curve. So performance will vary significantly from run to run. And because the error bars overlap, in some runs the small effect will disappear.

Moreover, this particular issue also goes away if we add even more data or if we add more parameters to the model because the shape of the curve changes.

So don't expect this effect to be a big deal in practice. Still, it's worth knowing that over-parameterized deep networks—unlike standard models—can sometimes react to additional data by getting a little worse at generalization. This phenomenon was first documented in a paper circulated by OpenAI in 2019.

Overall, the double descent wasn't known before 2018, and there's much more to find out. Even so, it already helps explain why early deep learning models experienced empirical successes in spite of blatant over-parameterization.



In fact, the double descent points out an amazing property of deep learning that overturns what had been an unbreakable precept for decades, if not centuries, in data science.

So why do over-parameterized deep networks perform so well?

This question was the topic of a 2019 award-winning paper at the Neural Information Processing Systems (NeurIPS) conference. Researchers found that the existing mathematical tools that people use to analyze the behavior of machine learning algorithms—including all of the algorithms in this course—are not capable of resolving the question of why over-parameterized deep networks perform so well.

Instead, researchers will have to develop new mathematical methods to explain the resistance of deep networks to overfitting and their consequent practical power.

So recent findings about over-parameterized networks are that they resist overfitting, and current mathematical tools cannot tell us why they learn. Both of these findings allow that over-parameterization won't hurt learning.

But the point is much stronger. It seems that over-parameterizing is *essential* for machine learning of hard problems, even though only a fraction of the parameters actually matter in the end. In other words, we seem to need a ton of parameters for accurate learning, even though we only need a much smaller set to represent the learned function. The process of pruning demonstrates this idea.

Try It Yourself

Follow along with the video lesson via the Python code:

[L23.ipynb](#)

Python Libraries Used:

keras.layers.Dense: Creates a fully connected layer in Keras.

keras.models.Sequential: Builds up layers of a neural network in Keras.

keras.regularizers: Regularizers in Keras.

matplotlib.pyplot: Plots graphs.

numpy: Mathematical functions over general arrays.

Key Terms

lottery ticket hypothesis: A possible explanation for the behavior of deep networks that allows it, once trained, to be pruned to a much smaller size.

polynomial regression: Regression where the output rule is the coefficients of a polynomial of a given degree.

READING

Nakkiran, Kaplun, Bansal, Yang, Barak, and Sutskever, “Deep Double Descent.”

Frankle and Carbin, “The Lottery Ticket Hypothesis.”

QUESTIONS

1. What are the four phases of the double descent, describing generalization accuracy as a function of increasing model size?
2. You train a deep neural network and then remove the connections with the smallest weights. You start retraining the pruned networks. How should you initialize the remaining unpruned weights? Rank the following options from best to worst and explain the reasoning behind your ranking.
 - (i) Initialize them to the values where they left off from training the first time.
 - (ii) Initialize them to their original values in the first network you trained.
 - (iii) Initialize them with random values in the same range as in the first network you trained.
3. In this lesson, the double descent phenomenon was demonstrated in an artificial dataset. Do you think the same behavior would be visible in a small real-life dataset—for example, the diabetes data from [Lesson 03](#) on decision trees? Train a neural network with one hidden layer and vary numbers of hidden units from 1 to 1,000. Do you see the double descent?

Answers on page 486

The Unexpected Power of Over-Parameterization

Lesson 23 Transcript

So far, in this course, we've kind of been taking the deep learning revolution for granted. Yes, we've been celebrating and using the results, from generative adversarial networks to convolutional networks to transformers. But to better understand this flurry of results, it's also helpful to take a step back. What was it, fundamentally, that triggered the shift in 2015 that got deep learning off the ground? What kicked off the wave of changes we've been discussing in this course and that all of us will likely be feeling for decades to come? After all, the ideas that make up deep learning have roots in a more distant past. At the heart of deep learning is using calculus; in particular, differential calculus.

The roots of calculus go back to its formalization by Newton and Leibniz in the 1600s, and, before them, all the way back to mathematics developed in Persia in the 1400s. Of course, none of these predecessors were using calculus to train neural networks to update parameters incrementally to minimize a loss function. That idea, the idea of minimizing by gradient descent, came much later, when three research teams independently discovered the backpropagation method for efficient gradient descent in the mid-1980s.

Three reasons are typically given for why general neural networks waited three more decades before bursting onto the scene in 2015. One reason is increased computer speed. We needed faster computers to process vast amounts of data and computers had been getting progressively faster for decades. Moreover, general-purpose graphics processing units or GPUs, developed for gaming starting in 2007, turned out to be just as big a boost for machine learning as for their intended use in gaming.

Second, the availability of data sets expanded. Machine learning solutions are defined by their data, and society began living online for the first time: Facebook launched in 2004, the first-generation iPhone came out in 2007, digital images, digital sound, digital video, machine-readable text, and other kinds of personally relevant data were being amassed in unprecedented quantities.

Third, improved machine learning algorithms helped larger-size neural networks learn more effectively. There were new representational spaces like transformer networks with ReLU units, which reduced the chance that a network would get stuck in poor local optima during training. There were new loss functions like cross entropy, which provided loss surfaces that could be traversed more smoothly. And there were new optimization algorithms like Adam, which matched well with what is needed to train bigger neural networks.

Speed, data, and algorithms are the three typical reasons people give for the deep learning revolution. I think there's a fourth reason that may be just as important as the others and it's far more surprising. One of the things that makes deep networks work so well is that researchers developed a willingness to try things that shouldn't work.

Before 2015, any mainstream data scientist would have told you that networks with millions upon millions of parameters cannot be trained to produce reasonable answers. The view was that the data would not sufficiently constrain all those free parameters. It was a central dogma in statistics—learned through hard experience—that having too many parameters in your model or rule would mean you would be overfitting, with all the bad consequences we discussed back in Lecture 9. Your model would learn idiosyncratic, chance variations from your data instead of representing your data in a generalizable way. Indeed, I already said that overfitting is the concept from machine learning that everyone should know.

So, there was a conceptual barrier that had to be overcome before the vast majority of potential contributors to deep learning could actually get on board. So, while overfitting is undeniably bad, in this lesson, I want to look at how important it was for the field of machine learning to also get beyond the standard view of overfitting. That will let us dive deeper into the implications of training big networks whose set of weights or parameters are very large compared to the amount of data available. Such models can be called overparameterized because they have more parameters than it seems like they should, relative to the amount of data.

I'll talk about how these networks violate what we thought we knew about data analysis and machine learning, but also how they are central to the success of deep networks in solving hard real-world learning problems.

LESSON 23 | THE UNEXPECTED POWER OF OVER-PARAMETERIZATION

TRANSCRIPT

First, we need to revisit the idea of overfitting. We'll see that the standard story has some new facets in the context of deep learning. I personally learned about overfitting in college in a class on cognitive psychology. My teacher, Richard Gerrig, taught a wildly popular intro psych course, which in 1991 became one of the very first offerings from The Teaching Company. Here's an example I got from Professor Gerrig that I have adapted and used in my own classes many, many times.

What we're going to do is take some one-dimensional data that is well modeled by a low-order polynomial. We'll add some noise to the data and show that the best predictions of the original function come when the degree of the polynomial in our model matches the degree of the polynomial that generated the data. In this case, the data comes from a simple cubic function or third degree polynomial, marked in green on the plot. I generated nine data points equally spaced and then added random noise to each y position. The noise was in the range negative-1 to 1.

We can try fitting the data with a line, which is a 1-degree polynomial. It's okay, but even the best line doesn't fit very well. The data just isn't that straight. We can also try a parabola; that's a quadratic function, which has an x -squared term, also called a second-degree polynomial. This choice gets the curving nature of the data a bit better, but it lacks the ability to bend downward on the far left while still going upward on the top right.

When we try a cubic function, which is a third degree polynomial, that pretty much nails it. So, degree-1 was just okay, degree-2 was better, degree-3 was much better. What happens if we continue this progression? Degrees 4 through 7 are not an improvement, but are not too bad, either; they make slightly squarer versions of the cubic curve.

But at degree-8, things start to come undone. We only have nine data points, with eight spaces to span between those data points, so an 8-degree polynomial can hit each of the training points exactly, by switching between zooming up and zooming down between each pair of points.

In general, we can always fit the data exactly if the degree of the polynomial is one less than the number of points. Two points determine a line, three points determine a parabola, four points determine a cubic, and so on up to nine points determine a degree-8 polynomial. We can summarize the results of these plots by plotting the error relative to the true cubic of each curve we

get for each polynomial from degree-0 to degree-8.

The uptick at the end is the telltale sign of overfitting. At degree-8, the polynomial can change direction enough times to hit every data point, but it loses the nice smooth degree-3 shape that the data came from.

The best fit happened around degree-3, as expected, although 4 through 7 also fit well.

Extrapolating past degree-8, it looks like perhaps the error will just get worse and worse. But we cannot find out what happens, at least not with the methods I used to fit these curves. I generated the best fit curves for each example by using least-squares minimization using matrix algebra. After degree-8, the matrix equations can no longer be solved because we've got more unknowns than we have equations.

Having looked at all of the possible polynomials, we conclude that the best we can do is a middle polynomial of degree-3. But the story is more complicated than that, as we'll see in our next example. Here, we'll follow largely the same script. But instead of using nine points around a cubic curve, we'll use four points around a line. But more importantly, we're going to use deep learning to find the best fitting polynomials and we'll look at polynomials with degrees much bigger than the amount of data we have. We'll overparameterize the function.

We'll step through creating a program to do the work of finding the best fit. The program starts off by creating the training data. We need our target line, which will be used to generate our training and testing data. We're going to look at the line between negative-1, a starting point called `xfrom`, and positive-1, a destination point called `xto`. Our test set will consist of 200 test points, plus 1, for a total of 201 points evenly spaced on that interval.

The labels for those points are just the value of the line at those points. For this particular example, I handed-picked a couple of x values along the interval. The labels for these points are the output of the line, plus a little noise to bump it up or down by 0.1 at each position. We could pick these values randomly, but I chose these specific values to make the results repeatable. We can import the `matplotlib.pyplot` library to visualize the function. I'm using blue to mark the four training points and green to mark the testing line. You can see that the blue points straddle the target line.

LESSON 23 | THE UNEXPECTED POWER OF OVER-PARAMETERIZATION

TRANSCRIPT

We're going to fit this data with a polynomial function, varying the degree of the polynomial. Unlike matrix algebra, deep learning tools like Keras can be used to fit over-parameterized models. That's what we'll use here. To feed the data to Keras, we need to expand the x values into a set of features consisting of powers of those x values. Our routine, `makefeatures`, returns a numpy array consisting of the input x 's each raised to successive powers from 0 to fitd , inclusive. For example, if fitd is 4, the expanded data values corresponding to a degree-4 polynomial consist of: 1, the x value, the x value squared, the x value cubed, and the x value to the fourth power.

Now, we'll define a simple Keras network for fitting the data. We import the Keras libraries we need to create a simple one-layer network. `Sequential` is for concatenating the layers. `Dense` is for making the one fully connected layer we need. Then, to fit the data with a d -degree polynomial, we first create training and testing data with the necessary features. Then, we construct a model consisting of a single dense layer with a linear activation. That's the standard setup for polynomial regression.

We run the optimization for 100,000 training epochs. That may seem like overkill, but the effect we're looking for emerges in highly trained networks. After the network is trained, we use it to make predictions for the training and testing data and plot the resulting fit. We return the mean-squared error on the testing and training examples. Fitting the training data with a horizontal line gives us the mean but results in a terrible fit to the data points. The mean squared error for the testing data is about 1/3. For the training data, it is about 1. The testing error is lower because more of the green line is in the middle, closer to the red line than the extreme blue points. Because our representational space is constrained to horizontal lines, we underfit the data.

Next, let's try a line that's free to go in any direction. The testing data is located along a line, which is exactly the line that the algorithm finds. The red and green lines perfectly coincide. The testing error is so minuscule, it's on the order of 10 to the minus-10, much less than the training error of 10 to the minus-2, or 0.01. Usually, testing error is around the same or a little worse than training error. But this case is special because the target function is so clearly defined.

Let's go up to a quadratic. Now is where things first go off the rails.

Fitting this training data of four points with a degree-2 polynomial allows the function to hit all of the data points perfectly. So, it does. But in the process, it zooms up so it can catch the two points on the right on the way down. That's overfitting. The representational space now has enough power to drive training error to 0, and it's fitting the noise along the way. The resulting function generalizes badly.

So far, we are just seeing the standard warning about overfitting. You want to give a machine learning system just the right amount of representational power, given the complexity of the function that you are fitting and the amount of data you have to estimate it. You don't want to overparameterize, use way more parameters than you can fit with the available data. That's what Professor Gerrig taught me and that's what I have passed along to generations of machine learning students. But the deep learning revolution suggests that we look more closely at what happens if we have what would previously have been too many parameters.

Again, we still can't fit a high-degree polynomial to just four points using standard methods. But we can apply deep networks in this situation. When we fit a degree-4 polynomial with the same four data points, the result is surprising. The polynomial uses some of its extra representational power to avoid bowing outward quite as much as we saw with the parabola. We're still hitting the training points exactly, but the mean error on the testing points has dropped from 0.458 to 0.136. What's going on? Once the representational space is rich enough to let the learned function hit all of the training points with flexibility to spare, it is able to start using that extra power to smooth out the function elsewhere. The result is a higher-degree polynomial that hits all of the testing data, but still generalizes somewhat better than the lower-degree example of overfitting did.

To get this particular curve, I ran the fitting procedure seven times and chose the result with the lowest training error. It is interesting to note that even with 100,000 training steps in a single run of the fitting procedure, Keras still does not consistently return the same minimum loss solution each time. On the other hand, it's interesting to note that all seven of the runs had a lower testing error than the quadratic did. Moderate over-parameterization always did better.

LESSON 23 | THE UNEXPECTED POWER OF OVER-PARAMETERIZATION TRANSCRIPT

Let's look at the training and testing error across a range of polynomial degree values, d . I'm plotting the training error in blue to match the blue training points. I'm plotting the testing error—the deviation from the target line—in green to match the green target line. I'm plotting the mean loss over seven separate runs for each polynomial degree, d . We see the error on the training data in blue dropping immediately as we go from degree-0 to degree-1. By degree-2, the training error is effectively 0. But on the testing data in green, error drops, then rises. That's the standard overfitting curve. But then, with some noise, the testing error slowly but surprisingly drops a second time. This unexpected result was identified as a possible widespread phenomenon in 2018. It is referred to as the double descent. Error on the testing data improves, gets worse, but then improves again.

Scientists at Open AI demonstrated double descent in a variety of deep networks at the forefront of the field. These include transformer networks from natural language processing and convolutional networks from image processing.

The error on the real world data followed the high-low-high-low pattern as the network size increased. They found the same pattern if they kept the network size fixed but changed how long it was trained. Training time is also a measure of network complexity. Long training times lead to more complex networks because there is more time to amplify the weights and express more complex functions.

Initially, the network is too small to solve the problem at hand and the testing error is poor. The network underfits the data. Once the network has the capacity to capture the solution to the problem, testing error hits a minimum. The network is fitting the data well. With a further increase in capacity, the network is able to capture the solution but also the noise and inconsistencies in the data, resulting in increasing testing error. The network is overfitting the data. That's where the story used to end. But increasing the capacity network even more allows the network to go beyond representing just the solution and the noise. Now, it can approximate just about any function it might need. At that point, the network is able to match the data, even with the noise, but it can also interpolate in the regions between the training points.

In these interpolation zones, predictions on testing data are typically quite good. The overall prediction performance of the network with noise incorporated returns to the level it reached when it first captured the solution and ignored the noise. In the first descent, the network captures the solution but ignores the noise because it simply doesn't have the capacity to fit the noise. The network is like a dog that fetches a ball but does not chase cars driving on the other side of a fence. It's not that the dog doesn't want to go after the car, it's just that external constraints prevent it from doing so.

In the second descent, the network captures the solution and interpolates around the noise. Now, the network is like a dog in an open yard that's had many, many times to chase cars. It is briefly distracted by cars, but it comes back to the yard right away. The analogy isn't perfect, but I think the lesson is a good one. It's natural to try to restrain a learning algorithm so it doesn't hurt itself by chasing after the noise in the training data. But we can actually reach the same solution by providing additional freedom. The learner can chase after the noise, but still return to reasonable values for inputs outside the training set, thereby getting the benefit of still producing good answers on unseen data.

The double descent phenomenon leads to another counter-intuitive result that's worth mentioning, although the practical implications so far have been minor. A central tenet in standard machine learning has been that more data is better. In the classical single descent setting, adding more data allows us to define parameter values more accurately, leading to better prediction on unseen data. The extra data stretches the testing error curve down and to the right, so error is less at all network sizes. In standard machine learning, more data is always just plain good.

But consider what happens in the new over-parameterized double descent deep learning regime. Things get a little more complicated. Because the curve has a hump, stretching it down and to the right does not necessarily result in error improvements at all network sizes on the curve. Here, there's a range of network sizes where additional data hurts generalization performance a bit. Hmm. More data is bad is pretty shocking, but we're talking about a very weak effect, here. Each of these curves is an average of values with error bars above and below the curve, so, performance will vary significantly from run to run. And, because the error bars overlap, in some runs the small effect we're talking about will disappear.

Moreover, this particular issue also goes away if we add even more data or if we add more parameters to the model because the shape of the curve changes. So, I don't expect this effect to be a big deal in practice. Still, it's worth knowing that over-parameterized deep networks—unlike standard models—can sometimes react to additional data by getting a little worse at generalization.

This phenomenon was first documented in a paper circulated by Open AI in 2019. Overall, the double descent wasn't known before 2018 and there's much more to find out. Even so, it already helps explain why early deep learning models experienced empirical successes in spite of blatant overparameterization. In fact, the double descent points out an amazing property of deep learning that overturns what had been an unbreakable precept for decades, if not centuries, in data science.

So, why do overparameterized deep networks perform so well? After all, decades of analysis had suggested that over-parameterized networks should fall prey to overfitting their training data, thereby failing to generalize beyond the data at hand. This question was the topic of a 2019 award-winning paper at the leading academic conference in neural networks—Neural Information Processing Systems or NeurIPS. The paper found that the existing mathematical tools that people use to analyze the behavior of machine learning algorithms—including all the algorithms in this course—are not capable of resolving the question of why overparameterized deep networks perform so well. Instead, researchers will have to develop new mathematical methods to explain deep networks' resistance to overfitting and consequent practical power.

So far, I've covered two important facts about overparameterization. First, overparameterized networks resist overfitting. As the number of parameters increases, the generalization plot undergoes a double descent, with a region of overfitting surrounded on both sides by regions of successful generalization. Second, I've said that current mathematical tools cannot tell us why overparameterized networks can learn. In short, both of these recent findings allow that overparameterization won't hurt learning.

But the point is much stronger. It turns out that to get high accuracy from a neural network, we must train a lot more weights than are needed to represent the learned rule. It seems that overparameterizing is essential for machine learning of hard problems, even though only a fraction of the parameters actually matter in the end. That is, we seem to need a ton of parameters for accurate learning, even though we only need a much smaller set to represent the learned function.

Experiments with pruning—removing weights from a trained network—suggest that the reason we need to create networks with lots of weights is that finding the right combination of weights for a particular problem is hard. We need to give the network lots of possibilities so it can hit upon the combination that works. Researchers call such a combination a lottery ticket because, if you've got it, you win the learning lottery.

Here's an experimental setup that demonstrates this idea. Let's take a neural network and randomly initialize its weights. We'll call that network A. Let's train A on some data until it reaches a local min. We'll call the resulting neural network B. It has the same number and location of weights as the original A, but the specific weight values are different. Typically, some of the weights in any network will be close to 0. Let's make a new network B-prime where we set B's smallest weights to 0. In fact, we'll just remove the small weights entirely from the network.

Now, which network would we expect to do best on the training data? Well, A isn't trained. So, we'd expect B to be better than A. How about B-prime? Well, B-prime is close to B, but not quite the same. Since B was at a local min, any non-zero weights in B that are removed in B-prime should probably make things worse. Probably not much worse, but a little worse.

What if we keep training? Well, training A just gets us B, so nothing new there. Training B also gets us B, since it's at a local min. But training B-prime could get us someplace new. Typically, we'll end up with a new network C with higher accuracy than B-prime. Sometimes, C will even have higher accuracy than B.

We refer to the act of removing weights from a network—whether to improve its accuracy or to have the same accuracy with fewer weights—as pruning. It's been known since the 90s that pruning can be valuable for producing a network that will be cheaper to run without sacrificing accuracy. After all, fewer weights means less storage and less time spent calculating values.

LESSON 23 | THE UNEXPECTED POWER OF OVER-PARAMETERIZATION

TRANSCRIPT

We can repeat this pruning procedure multiple times to get an even sparser network. Once all of the incoming weights of a unit are pruned, we can prune the unit itself and all of its outgoing weights. We can potentially get a much smaller network, perhaps 5% the size of the original. At some point, of course, we reach a local optimum in which none of the weights is small or removing the small weights results in a network that doesn't improve with additional training. And, at that point, the process has to stop. In 2018, researchers started to look more closely at this phenomenon to see if it revealed anything deep about how deep networks work. This process of discovery is ongoing, but there are already some very exciting revelations.

Let's look at another visualization of a pruning step. This example is just made up, so don't worry too much about the specific values. Here, I'm just showing the weights of the network, not their locations relative to the nodes. The first line depicts our initial network A with randomly assigned weights. I'm using darker shading to indicate a larger weight. Green shades represents a positive weight and red negative weights.

After training, we get a new network B with slightly different weight values from A. Some weights get bigger, some smaller. Some change sign and some remain exactly the same. Now, we can remove the small weights from the network to get B-prime. That's what we did earlier, and that's how pruning was commonly done before 2018. But researchers started to wonder whether there were better or more revealing ways of selecting the values of the weights to use in each iteration of the pruning process.

Using the same pattern of pruning, they looked at setting the weights to their values from the initial network, A. Let's call that resulting network A-prime. Continuing the pruning process from A-prime typically was even better than continuing from B-prime. There's something special in those initial weights, particularly the subset of initial weights that don't get pruned away. The authors called these weights the lottery ticket because it represents a lucky combination of values that leads to good performance. They compared the lottery ticket pruning to other ideas like keeping the same set of weights but re-initializing them randomly or starting with the initial weights from A and removing the same size subset of weights but completely at random.

These other ideas don't work nearly as well as the lottery ticket scheme. There's special information in those initial weights that allow the problem to be solved well. Later work showed that just keeping the signs of the initial weights is enough. You can set all of the remaining weights to a single, identical magnitude, and just use the original signs, resulting in high accuracy. One enticing implication of this line of work is an explanation for why over-parameterization is important. For a sparse network, even one that we know can be trained well, it is not the case that every setting of initial weights allows it to be trained well. But for an over-parameterized network, perhaps having so many available parameters in the rule or model is a mechanism that increases the chances that the initial network includes a lottery ticket.

As of 2020, we didn't understand how neural networks find the lottery ticket. Nor, did we even know how the number of lottery tickets grows with the amount of overparameterization. But it was clear that by including lots of extra weights, the chance that the network would train well is improved immensely. As with the lottery ticket hypothesis, the precise nature of the double descent phenomenon is not yet understood. We have more to learn about when it occurs, how to detect it, and how to best make use of it in hard problems. What we want, but don't yet have, is a general way to explain the unreasonable effectiveness of adding extra parameters.

Exploring the mysteries of over-parameterization is a royal road to finding more general, reliable, and interpretable approaches to deep learning. Understanding the intricacies and outer limits of the deep network training process is what will bring the next wave of machine learning breakthroughs.

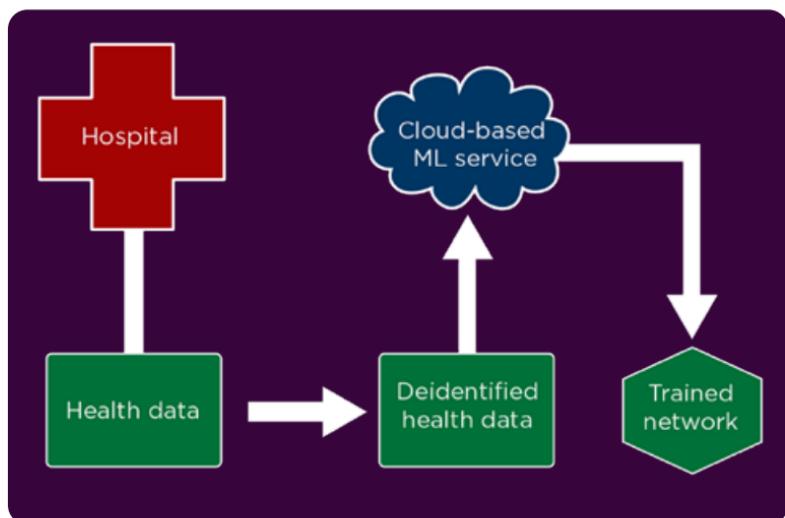
LESSON 24

PROTECTING PRIVACY WITHIN MACHINE LEARNING

During the first two decades of the 21st century, large-scale collection, processing, and retention of personal data became pervasive—and increasingly invasive. So much private data being gathered in so many ways raises real risks for misuse, including harassment, discrimination, and even crime. But this lesson introduces two specific techniques that are being developed for mitigating these risks. Homomorphic encryption hides private data in plain sight so that machine learning algorithms can be applied to the data without revealing its contents to the organization doing the machine learning. By contrast, differential privacy blurs the personal data before training machine learning models—another way to prevent it from being extracted and exposed.

Leaks in the Machine Learning Pipeline

To understand where privacy can be violated, consider a very simple machine learning pipeline. In this example, personal health information is being collected by a hospital. The hospital is a relatively trusted organization, governed by privacy laws, and users are willing to share their data in exchange for medical care.



To evaluate possible cures for diseases and improve patient care, the hospital wants to run sophisticated machine learning algorithms on the data.

They don't have the algorithms or the processing power to do state-of-the-art machine learning. That's something that some other organization—some kind of cloud-based machine learning service—would be better suited to do.

Because of privacy legislation, the hospital can't just hand over the private patient data to the machine learning service. At the very least, all identifying information, such as names and addresses, in the training data needs to be removed.

The cloud-based machine learning service runs on the deidentified data and produces a trained network. That network can be used for making predictions on new patients.

If everyone involved is trusted, this pipeline makes sense. It lets each organization do what it does best and provides the service of making medical predictions—that's valuable to patients and their doctors.

But in the context of untrusted parties—what computer security experts sometimes call adversaries—this machine learning pipeline has risks. In fact, there are multiple ways that personal information about individuals in the dataset can be revealed, or leaked, to untrusted parties.

The first place where an individual's information can leak is if the hospital's health data gets hacked. Hackers have a long history of seeking and getting large amounts of sensitive information.*

Every phase of the machine learning pipeline—from storage, to deidentification, to model deployment—has privacy threats.

Fortunately, hospitals are held to a very high security standard, with a decades-old culture of protecting private information. And laws such as Europe's 2018 General Data Protection Regulation and California's 2020 Consumer Privacy Act are forcing other sectors to protect personal data more like hospitals do.

But machine learning is subject to other kinds of leaks that require more specific methods to protect private information in the machine learning pipeline. In particular, even deidentified data can be a target for attack.

Homomorphic Encryption

A powerful tool from computer security that can help avoid leakage of data, whether deidentified or not, is encryption. If cloud-based machine learning services would store their data in encrypted form, only someone with the decryption password would be able to read the data.

Even better, if the data source, such as the hospital, encrypts the data before sharing it with the cloud service, the chances of exposing sensitive information go way down.

But if the hospital encrypts the data before sending it to the machine learning service, we can't expect the service to be able to do machine learning on the data. It's a catch-22.

* Banks, at least one credit reporting agency, email providers, social networks, and hotels have been the targets of high-profile hacks that have exposed billions of user accounts.

However, there is an amazing approach for making it so that encrypted data can be used for learning. Although it's not yet practical to deploy on a large scale in 2020, it's a very promising idea.

A simple way the hospital can get the cloud-based service to do valuable work for them without exposing the sensitive data is by using decoys.

The hospital would prepare a second dataset consisting of fake data and would ask the service to do learning on both the real data and the fake data. This way, the service isn't sure which dataset contains the sensitive data and which contains made-up nonsense.

And the more decoys the hospital creates, the more secure the data is. Let's say the hospital asks the service to train on 128 different datasets. Most of that computation isn't directly being put to use. Instead, it's just there as a distraction. So training with decoys becomes more private, but also much less efficient in terms of time, storage, and electricity.

And this trade-off is especially important to keep in mind because, in the digital world, even 128 decoys may offer only a modest level of protection. Ideal privacy would use a ton of decoys.

Fortunately, there is a scheme for doing calculations with 128 decoys that produces the equivalent of 2^{128} decoys. The idea, called **homomorphic encryption**, is that each bit of the data is delivered to the training algorithm in 128-bit encrypted form. The encryption is designed so that the basic operations we need for machine learning—things like multiplying and adding vectors—can be applied to these encrypted values in a way that produces other encrypted values.

These encrypted values, when decrypted, produce the same answer as if the analogous operations had been performed directly on the original data.⁺

As an approach for keeping the training data away from prying eyes, homomorphic encryption seems right on the edge of viability as of 2020. Proposals existed for privacy-preserving versions of linear regression, logistic regression, decision trees, k -means clustering, support vector machine classification, and neural networks. And researchers are continuing to discover more efficient operators for carrying out key steps of machine learning algorithms.

⁺ In mathematics, this kind of analogy is called a homomorphism. That's why this kind of decoy-based encryption is called homomorphic encryption.

If we keep our original data safe and protect even deidentified data, it can be very tempting to think that there's nothing more we need to do with regard to privacy. But in fact, even the trained model itself can leak private information.

Researchers have shown that it is possible for attackers to coax specific training instances out of workhorse models such as deep networks and support vector machines—especially if partial information about the training instances can be discovered by the attackers.

In short, an attacker can gain private information by using knowledge about the training procedure and the internals of the learned model.

Sometimes, even ordinary access to a public query interface of a trained classification model can be enough to break privacy. If people can use the trained model to make predictions, they can learn about the data that was used to train it.

That means such a public interface has a much larger number of potential adversaries. These adversaries don't have to figure out how to force their way in to steal the data; they merely have to figure out a clever way to play a form of 20 questions until something private gets revealed.

In the privacy literature, attack strategies are published as a proof that an existing approach to security is insufficient.

Differential Privacy

An intuitively appealing approach to thwart an attacker is to blur the training data just a bit. If we do blur everything a little, it shouldn't hurt the performance of a robust classifier much at all. Yet randomization in how the blurring is done should make it impossible to reliably disclose the data used in the training process.

By using randomization to blur the data, differential privacy allows individual instances of data used in training to be protected with only a minimum sacrifice of machine learning accuracy.

In the 1970s, the idea of suppressing information in published data to protect privacy was first formalized in the statistics community. But it was the work of Cynthia Dwork and others in 2006 that first quantified the trade-off between utility and privacy.

They created a field called **differential privacy**. The name comes from the idea that we're looking at the *difference* between two datasets. One dataset has a sensitive data item in it. The other does not have the sensitive data item but is otherwise identical. If we cannot reliably tell these two datasets apart, they are considered differentially private.

Differential privacy has been applied to many machine learning approaches, including linear regression, logistic regression, support vector machines, *k*-means clustering, principal components analysis, naive Bayes, and deep learning.

Some of these privacy methods are now part of libraries that are almost as fast and easy to use as Scikit-learn.

The Push for Privacy

So it is, in fact, possible to get the benefits of machine learning without sacrificing our privacy. This state of affairs is long overdue.

We're reaching the point at which machines monitor our behavior 24-7 and increasingly sophisticated algorithms analyze the collected data. George Orwell's vision of a Big Brother that watches and analyzes everything we do is becoming practical.

In her 2019 book *The Age of Surveillance Capitalism*, Shoshana Zuboff argues that the biggest problem is that the organizations have our data, but we do not have theirs. There is "epistemic inequality"—an imbalance in the amount of knowledge between users and "Big Other." And because we do not possess the knowledge needed to prevent abuses from happening, we are at a disadvantage.

Among the big tech companies, Apple has been the most committed to deploying tools to protect the privacy of their users.

Some of the push for increasing privacy in machine learning comes about due to the European Union's privacy rules, which went into effect in Europe in 2018. The rules require that companies give users additional control over their data, including the right to remove it from the machine learning models that companies have trained.

But something like differential privacy is also needed—to prevent removal from a database from itself becoming just another threat model for grabbing private information as a user leaves.

Try It Yourself

Follow along with the video lesson via the Python code:

[L24.ipynb](#)

Python Libraries Used:

diffprivlib.models: Learning algorithms augmented with differential privacy.

keras.datasets.fashion_mnist: Fashion MNIST dataset.

keras.preprocessing.image: Prepares raw images for processing by a neural network.

numpy: Mathematical functions over general arrays.

sklearn.naive_bayes.GaussianNB: Naive Bayes learner for real-valued features.

Key Terms

differentiable programming: An approach to building software systems that generalizes neural network training. It allows the program itself to be analyzed using calculus to more easily reason from output to input.

differential privacy: A property of a data-storage system or trained machine learning system that makes it difficult to compare two versions that differ in only one training instance to recover that instance.

homomorphic encryption: A form of secure encryption of data that allows computations to be carried out on the data without exposing the data itself.

READING

Tufekci, "Machines Shouldn't Have to Spy on Us to Learn."

Hao, "How Apple Personalizes Siri without Hoovering Up Your Data."

QUESTIONS

1. What is homomorphic encryption, and why is it important for making it possible to apply machine learning algorithms to private data?
2. Even with homomorphic encryption, making the results of the machine learning model available for predictions can leave them open to snooping. By querying a machine learning model before and after an item was removed, it can be possible to reconstruct the deleted training instance. What is the name for a method that can continue to protect the privacy of the instances used to train the model even as individual training instances are added or removed?
3. Privacy typically comes with a cost in terms of running time or accuracy. How significant is this cost? Train two naive Bayes classifiers (standard and private) on the spam detection dataset to get a sense of the differential cost of differential privacy.

Answers on page 487

Protecting Privacy within Machine Learning

Lesson 24 Transcript

I bought an internet camera called a Welcome and installed it in my house. It used machine learning to recognize the faces of people coming in the front door. It would figure out if someone new showed up and alert me on my phone. I thought it was really neat and I loved that it was doing machine learning right in my living room. But when the camera eventually stopped working, I didn't bother to fix it or replace it. And here's why. My teenage daughter didn't like that the camera was watching her, and she hated that it was keeping a video record of her arrivals and departures.

This example exemplifies a really important issue in machine learning. Machine learning algorithms need to look at our data to do anything interesting for us, but that means that they need to look at our data. During the first two decades of the 21st century, large-scale collection, processing, and retention of personal data became pervasive, and increasingly invasive. Yes, employing this kind of data benefits science, engineering, and medicine, but so much private data being gathered in so many ways raises real risks for misuse, including harassment, discrimination, and even crime.

In short, machine learning can and does put our privacy at risk. But in this lecture, we'll look at two specific techniques that are being developed for mitigating these risks. Homomorphic encryption hides private data in plain sight so machine learning algorithms can be applied to the data without revealing its contents to the organization doing the machine learning. By contrast, differential privacy is a technique that blurs the personal data before training machine learning models, another way to prevent it from being extracted and exposed.

To understand where privacy can be violated, consider a very simple machine learning pipeline. In this example, personal health information is being collected by a hospital. The hospital is a relatively trusted organization, governed by privacy laws, and users are willing to share their data in exchange for medical care. To evaluate possible cures for diseases and improve patient care, the hospital wants to run sophisticated machine learning algorithms on the data. That's not something people running a hospital have typically been well equipped to do on their own. They don't have the algorithms or the processing power to do state-of-the-art machine learning. That's something that some other organization, some kind of cloud-based machine learning service, would be better suited to do.

Because of privacy legislation, the hospital can't just hand over the private patient data to the machine learning service. At the very least, all identifying information in the training data needs to be removed. So, names and addresses need to be taken out. Often, even zip codes need to be removed, since any non-local person using the hospital from out of the area would be easy to identify. The cloud-based machine learning service runs on the de-identified data and produces a trained network. That network can then be used for making predictions on new patients.

If everyone involved is trusted, this pipeline makes sense. It lets each organization do what it does best, and it provides the service of making medical predictions; that's valuable to patients and their doctors. But in the context of untrusted parties—what computer security experts sometimes call adversaries—this machine learning pipeline has risks. In fact, there are multiple ways that personal information about individuals in the dataset can be revealed or leaked to untrusted parties.

The first place where an individual's information can leak is if the hospital's health data gets hacked. Hackers have a long history of seeking and getting large amounts of sensitive information. Banks, at least one credit reporting agency, email providers, social networks, and hotels have been the targets of high-profile hacks that have exposed billions of user accounts. Fortunately, hospitals are held to a very high security standard, with a decades-old culture of protecting private information. And laws such as Europe's 2018 General Data Protection Regulations and California's 2020 Consumer Privacy Act are forcing other sectors to protect personal data more like hospitals do.

But machine learning is subject to other kinds of leaks that require more specific methods to protect private information in the machine learning pipeline. In particular, even de-identified data can be a target for attack, a lesson the machine learning community has learned the hard way.

As we discussed in Lecture 12 on recommender systems, the 2006 Netflix Prize ended with a privacy lawsuit. Netflix had released a dataset with movie ratings from half a million users and offered researchers a \$1 million prize for using the data to improve accuracy at predicting movie preferences. And, Netflix had de-identified all the user data—people's names were removed. But researchers correlated the preferences expressed in the Netflix data with preferences already available online like publicly posted movie reviews by individuals on IMDb, the Internet Movie Database. They found, that if you can use online reviews to identify just a couple of obscure movies that someone has seen, then you can figure out who they are in the Netflix data.

LESSON 24 | PROTECTING PRIVACY WITHIN MACHINE LEARNING TRANSCRIPT

Also, in 2006, a similar vulnerability was discovered in AOL's public release of de-identified data to researchers. The experiences at Netflix and AOL led tech companies to become much more wary about sharing even de-identified data with the public. And, after Facebook's collaboration with Cambridge Analytica resulted in a major data breach in 2016, even behind-the-scenes research became a focus of concern.

So, what to do? A powerful tool from computer security that can help avoid leakage of data, whether de-identified or not, is encryption. If cloud-based machine learning services would store their data in encrypted form, only someone with the decryption password would be able to read the data. Even better, if the data source, such as the hospital, encrypts the data before sharing it with the cloud service, the chances of exposing sensitive information go way down.

But if the hospital encrypts the data before sending it to the machine learning service, we can't expect the service to be able to do machine learning on the data. Catch-22. However, there is an amazing approach for making it so that encrypted data can be used for learning. Although it's not yet practical to deploy at large scale in 2020, it's a very promising idea that's worth sharing.

From the hospital's point of view, the basic puzzle is: How could the service do computation on the hospital's data without, well, looking at the data? Here's a simple way the hospital can get the cloud-based service to do valuable work for them without exposing the sensitive data. Decoys. Decoys are often used in movies as a way of obscuring sensitive information. For example, let's say a jeweler has a priceless diamond necklace and needs to transport it from store A to store B. The jewelers know that if they simply drive the jewels from A to B, thieves would be watching and they would deploy a crack team via helicopter to the location of the car, stop the car, and steal the jewels.

So, the jewelers get a great idea. They will take two identical briefcases and load each into one of two identical cars, then drive the cars on two entirely different routes. The thieves are watching all of this so they know what's happening. But they can't be sure of where to deploy their team. If they just pick one of the cars at random, there's a 50-50 chance of getting nothing at all.

To apply the idea of decoys to the machine learning privacy problem, the hospital would prepare a second data set consisting of fake data. Then, they would ask the service to do learning on both the real data and the fake data. This way, the service isn't sure which dataset contains the sensitive data and which dataset contains made-up nonsense. And the more decoys the hospital creates, the more secure the data is. If the jeweler sent out a fleet of 128 identical cars, they could start to feel pretty confident that their delivery would get through without the real jewels being intercepted.

In the hospital case, that would mean asking the service to train on 128 different datasets. Most of that computation isn't directly being put to use. Instead, it's just there as a distraction. So, training with decoys becomes more private, but also much less efficient in terms of time, storage, and electricity. And, this tradeoff is especially important to keep in mind because, in the digital world, even 128 decoys may offer only a modest level of protection. Ideal privacy would use a lot of decoys.

Fortunately, there is a scheme for doing calculations with 128 decoys that produces the equivalent of 2 to the 128 decoys. The idea, called homomorphic encryption, is that each bit of the data is delivered to the training algorithm in 128-bit encrypted form. The encryption is designed so that the basic operations we need for machine learning—things like multiplying and adding vectors—can be applied to these encrypted values in a way that produces other encrypted values. These encrypted values, when de-crypted, produce the same answer as if the analogous operations had been performed directly on the original data.

In mathematics, this kind of analogy is called a homomorphism. In this instance, the analogy is between computational operations on unencrypted data and analogous operations on encrypted data. If the analogy is well formulated, the results of operations on the encrypted data will continue to match the operations on unencrypted data. That's why this kind of decoy-based encryption is called homomorphic encryption.

Visually, you can think of homomorphic encryption as supporting computational operations on encrypted data. Say we want to use a homomorphic encryption service to add 3 and 2. We'd encrypt the 3 and the 2 and send them to the service. The encrypted values can be thought of as including the 3 and the 2 but also many, many decoy values at the same time. Then we'd ask the service to add those numbers. The actual operation that the service would execute is not addition, but something that

LESSON 24 | PROTECTING PRIVACY WITHIN MACHINE LEARNING TRANSCRIPT

carries out an analogous, in other words homomorphic, version of addition on the encrypted numbers. It's essentially doing the operation on all the decoys simultaneously.

The service would send us back the result in encrypted form. When we decrypt it, lo and behold, we discover that the answer is 5. The service does the calculation for us but never learns our numbers. Even the answer it provides is just one piece of hay in a haystack, given the many possible problems we might have been asking it to solve.

As an approach for keeping the training data away from prying eyes, homomorphic encryption seems right on the edge of viability as of 2020. Proposals existed for privacy-preserving versions of linear regression, logistic regression, decision trees, k -means clustering, SVM classification, and neural networks. And, researchers are continuing to discover more efficient operators for carrying out key steps of machine learning algorithms.

If we keep our original data safe and protect even de-identified data, it can be very tempting to think that there's nothing more we need to do with regard to privacy in machine learning. But in fact, even the trained model itself can leak private information. Researchers have shown that it is possible for attackers to coax specific training instances out of workhorse models we've seen, including deep networks and support vector machines, especially if partial information about the training instances can be discovered by the attackers.

Let's look at one specific example in detail. We'll see how private information can be extracted from a trained model. And, we'll examine an approach to differential privacy that is effective at hiding the private information. We'll be trading off some immediate utility of the trained model in favor of more privacy. First, let's be precise about the scenario that we're studying. Computer security researchers call a security scenario in which the attacker has some information and seeks to leverage it to reveal other information a threat model.

We've hinted at a few threat models already. The specific threat model we'll examine now is one where an attacker Eve has access to two trained models produced by a known machine learning algorithm—naïve Bayes. One is trained on a set of data that includes your private information. The other is trained on the same set of data, but without your private information. Now, Eve the attacker is in an ideal position: She can leverage all the information she has to uncover specific private information that is specific to you.

The situation roughly corresponds to what happens if you demand to have yourself removed from a known database. An attacker might compare the known database before and after your information is removed and use just this information to figure out your private information. It is also a great test case for privacy: If we can protect your data even here, we protect it in lots of more realistic scenarios. We'll look at the case where Eve the attacker cannot directly access the trained model, but she can issue queries with specific instances to see how the model classifies these instances.

So, here's how the adversary Eve can figure out the missing training instance, given access to a trained naive Bayes classifier. Eve can see a dataset consisting of instances, where each instance is a vector of numbers. Each instance is labeled as white for negative, or gray for positive. One instance is hidden behind a black bar. Eve can also see the trained model. In naive Bayes, the trained model consists of the probability of a one in each position amongst the positive examples, and the probability of a one in each position amongst the negative examples. The trained model also includes the prior probability of an instance being positive or negative, which is computed by counting the number of positive examples, the number of negative examples, and then dividing by the total number of examples.

Here, we can see five positive and five negative examples in the training set. How can we figure out the label of the instance hidden behind the black bar? Well, if it were negative, the training set would consist of six negative and five positive instances, so the prior for negative would be six out of 11 positive or roughly 0.55 for negative examples and five out of 11, or 0.45 for positive examples. Like Eve the attacker, we observe that it's exactly the reverse, with a prior of 0.45 for negative examples in white and 0.55 for positive examples in gray. That means the label of the missing example is more likely to be positive, gray.

That's very valuable information for Eve the attacker, who can now focus her attention exclusively on the positive cases. This narrowing-in is possible with naive Bayes, because the positive model depends only on the positive instances in the training set, while the negative model depends only on the negative instances. All Eve needs to do is solve some simple algebra problems for each column. The first column consists of three 0s, two 1s, and one unknown value. The average of the six values is roughly 0.5. That means $3 \times 0 + 2 \times 1 + x = 0.5 \times 6$, or $2 + x = 3$. That means x must be 1!

LESSON 24 | PROTECTING PRIVACY WITHIN MACHINE LEARNING TRANSCRIPT

For the second column, the same reasoning gives us x equal to 0.18. That doesn't quite make sense, since x should really only be 0 or 1. But the values in the model are rounded to one decimal place, so that suggests that x needs to be rounded, too. The missing value in the second column is 0. Column by column, Eve pieces together the values in the missing instance. In short, an attacker can gain private information by using knowledge about the training procedure and the internals of the learned model.

Sometimes, even ordinary access to a public query interface of a trained classification model can be enough to break privacy. If people can use the trained model to make predictions, they can learn about the data that was used to train it. That means such a public interface has a much larger number of potential adversaries; these adversaries don't have to figure out how to force their way in to steal the data. They merely have to figure out a clever way to play this particular form of 20 questions until something private gets revealed.

Let's return to the example at the beginning of the lesson and look at a dataset with pictures. The pictures to protect might be snapshots of faces from my home that my daughter does not want shared. But here, we'll use personal items of clothing to stand in for personal images. We'll train a naive Bayes classifier to distinguish two classes in the fashion MNIST dataset, t-shirts and sneakers.

This first block of code prepares the dataset: $\text{train}X$, $\text{train}Y$, $\text{test}X$, and $\text{test}Y$ are the full set of training and testing examples from the dataset, where X represents the instances and Y the labels. The instances are arranged as 28-by-28 grayscale images. For training the model, we'll reshape the data into flat vectors of size 784. To simplify training, we'll convert the grayscale images to black and white. For each image in the training and testing set, we compute the median grayscale value and call everything below that value 0 and everything above that value 1.

The fashion MNIST dataset includes 10 classes, but we're only using class 0, which are t-shirts, and class 7, which are sneakers. $\text{Test}Y_{\text{pair}}$, $\text{test}X_{\text{pair}}$, $\text{train}Y_{\text{pair}}$, and $\text{train}X_{\text{pair}}$ are subsets of the training and testing data where only the pair of target classes are retained. The dataset includes 12,000 training examples and 2,000 testing examples, evenly split across the two categories.

Before we start training on this data, let's take a look at it. To see the image data for ourselves, we can take the flat vectors and turn them into 28-by-28 arrays again, then use Keras' image library to display them as gray scale images. The function `showpic` takes a flat vector and displays it. It scales the images from their 0/1 black-white values to 0 to 255 pixel values for display. We loop through the first 10 examples in our training set and display each one. Because of the conversion to black and white, the images appear mainly as silhouettes. Nevertheless, it's enough to show us which items are t-shirts and which are footwear, and even some distinguishing markings.

Let's train a naive Bayes classifier on the dataset we've prepared. `Gnb` is the naive Bayes classifier on continuous values, which the scikit learn library calls `GaussianNB`. We fit the classifier using the training data, then use the resulting model to make predictions on the testing instances. The predictions, `predYpair`, are compared to the labeled testing instances `testYpair` to compute an accuracy. The trained classifier only gets three wrong out of 2,000 testing instances. So far, so good. The learned classifier achieves 99.8% accuracy on this dataset.

Next, we'll prepare a new dataset consisting of the same instances but with one instance arbitrarily selected for removal. We'll set `remove` to be the position in the dataset for the instance we'll be removing. Then, we'll save the instance and its label so we can refer to it later. Eve the attacker doesn't have this information, but once she reconstructs the missing instance, we'll want to see if she was successful in her attack. `TrainXpair2` and `trainYpair2` are the new training set. We'll train up a new classifier, see what it predicts on the testing data, and evaluate it. Removing this one item from the training examples doesn't change the classifier much; after all, there are 19,999 other examples. The number of mislabeled points at three remained the same.

Now, it's time to try to play the adversarial role of Eve the attacker and extract the missing instance from the model. In the privacy literature, attack strategies are published as a proof that an existing approach to security is insufficient. Here's a strategy Eve might take to extract information from the trained naive Bayes classifier. When Eve begins the attack, she has access to two trained models. One model is trained including the private instance and one is trained without. All Eve can do is query the two models with different inputs. Eve can proceed by constructing a synthetic instance that the trained model classifies as roughly 50% t-shirts and 50% sneakers.

LESSON 24 | PROTECTING PRIVACY WITHIN MACHINE LEARNING TRANSCRIPT

She looks to see how each of the two models classifies the test instance. In one case, she asks the original model. In the other case, she asks the model trained without the target data item. Then, for each column, she temporarily puts a one in that column of the test instance. She asks both the original model and the new model to classify this instance and notes which one reacts more strongly to the change by changing its class prediction.

The reconstruction of the image is exactly the pattern of which components cause more of a reaction in the original model or the retrained model. For example, we can probe each pixel of the image and see which model responds more. We'll color pixels gray where the original trained model responds more, and blue where the newly trained model responds more. The resulting image will be an approximation of the private image that was removed from the dataset. It's a t-shirt!

Let's actually try this attack. Eve the attacker has access to two classifiers—one with our private image, and one without. She queries each one to build up a guess of what the private image was. I ran Eve's strategy a half dozen times with different missing images. The trained classifier has evidence of a lot of the details of the images it was trained on. You can make out some distinguishing patterns on the shoes and t-shirts. Keep in mind that Eve the attacker made these pictures just by querying the two models a small number of times. That was enough to extract and reconstruct the target image, out of the 2,000 images it was trained on. Pretty spooky.

Is there anything we can do to thwart Eve the attacker? An intuitively appealing approach is to blur the training data just a bit. If we do blur everything a little, it shouldn't hurt the performance of a robust classifier much at all. And, yet, randomization in how the blurring is done should make it impossible to reliably disclose the data used in the training process.

Back in the 1970s, the idea of suppressing information in published data to protect privacy was first formalized in the statistics community. But it was the work of Cynthia Dwork and others in 2006 that first quantified the tradeoff between utility and privacy. They created a field called differential privacy. The name comes from the idea that we're looking at the difference between two datasets. One dataset has a sensitive data item in it. The other does not have the sensitive data item but is otherwise identical. If we cannot reliably tell these two datasets apart, they are considered differentially private.

Differential privacy has been applied to many machine learning approaches including linear regression, logistic regression, support vector machines, k -means clustering, principal components analysis, naive Bayes, and deep learning. Some of these privacy methods are now part of libraries that are almost as fast and easy to use as scikit learn. Let's train a differentially private version of naive Bayes on our Fashion MNIST data and see what our adversary Eve makes of it. Several libraries are available for training differentially private models. IBM researchers released a comprehensive one in 2019 that they call Differential Privacy Library—diffprivlib. We can get a copy from Github on the web and install it in Colab.

Training a naive Bayes model with IBM's differential privacy library follows nearly the same procedure as training a naive Bayes model from scikit learn. We simply call GaussianNB from the differential privacy library instead of sci kit learn. Otherwise, the syntax and functionality are the same. After training, we can see how accurate the model is by counting the number of misclassified points. Because differential privacy depends on random blurring of the data, you can expect different results each time you train. This time, my model made 106 mistakes out of 2,000 testing examples.

When we use Eve's reconstruction algorithm on this model, the results are very different from what we got with the traditional naive Bayes classifier. Now, instead of slightly noisy duplicates of the private data, the adversary Eve gets no information at all. The images are either black or white or filled with static. Such examples have shown it's possible to get the benefits of machine learning without sacrificing our privacy.

This state of affairs is long overdue. We're reaching the point at which machines monitor our behavior 24/7 and ever more sophisticated algorithms analyze the collected data. George Orwell's vision of a Big Brother that watches and analyzes everything we do is becoming, well, practical.

In her 2019 book, *The Age of Surveillance Capitalism*, Shoshana Zuboff argues that the biggest problem is that the organizations have our data, but we do not have theirs. There is epistemic inequality, an imbalance in the amount of knowledge between users and Big Other. And, because we do not possess the knowledge needed to prevent abuses from happening, we are at a disadvantage.

LESSON 24 | PROTECTING PRIVACY WITHIN MACHINE LEARNING TRANSCRIPT

Among the big tech companies, Apple has been the most committed to deploying tools to protect the privacy of their users. Because Apple's business model has focused on selling hardware instead of data, they have been under less pressure to gather and sell personal information. But they still want the user data for improving the recognition capabilities of their Siri personal assistant, among other things. One additional trick Apple uses to help keep private data private, and yet still learn from it, is to do some model training on your personal devices. This is the idea of federated machine learning, which exploits the under-used computing capacity of your smartphone to avoid sending private data over the network for training.

Since this distributed computing model is known as edge computing, the machine learning version is sometimes referred to as edge AI. This kind of semi-localized machine learning lets you keep more of your private data on your private machines. Based on the machine learning that takes place locally, your device sends results, in the form of minor model updates, to Apple's central servers. That way, improvements discovered locally can be pooled and incorporated into the central system, without gathering everyone's private data in a central place. Apple combines this distributed training scheme with differential privacy to provide additional protection.

Some of the push for increasing privacy in machine learning comes about due to the European Union's privacy rules, which went into effect in Europe on May 25th, 2018. The rules require that companies give users additional control over their data, including the right to remove it from the machine learning models that companies have trained. But as we've just seen, something like differential privacy is also needed, to prevent removal from a database from itself becoming just another threat model for grabbing private information as a user leaves.

Every phase of the machine learning pipeline—from storage, to de-identification, to model deployment—has privacy threats. Better use of general cryptography tools can go a long way toward keeping private data safe. But there are also vulnerabilities specific to machine learning. Homomorphic encryption allows service providers to use our encrypted data for training machine learning models, sacrificing some efficiency in the process. Because the data remains encrypted—essentially through retaining a large set of decoys—the service providers cannot leak the meaning of the data, even to themselves!

Differential privacy uses randomization to blur the data. It allows individual instances of data used in training to be protected, with only a minimum sacrifice of machine learning accuracy. Companies have deployed some of these privacy-preserving tools and have accelerated their use in the wake of new government regulations that give consumers more control over their data and how it is used. But perhaps the most important thing we need is knowledge—knowledge of what’s at risk, and knowledge about how our personal information is being used, turned into money, and sometimes abused.

Machine learning raises new privacy challenges, but privacy can still be protected at a cost. We need to keep in mind that organizations with access to our data will need more time, storage, and electricity to protect our privacy.

LESSON 25

MASTERING THE MACHINE LEARNING PROCESS

In the years to come, machine learning will likely advance to the point where programs can learn flexibly and robustly across a much wider range of problems and data relationships. At that point, creating a machine learning program will look less like training a machine and more like teaching a living being. And humans will indeed be the masters. Whether we'll be a force for good is up to us.

Step into Meta-Learning

Computers run machine languages. And back in the day, machine languages were the only way people could program computers. It was painstaking work, but it fostered a close relationship between people and their machines.

With the creation of higher-level programming languages starting in the 1950s—and continuing with the development of accessible languages such as Python—more and more people could participate in programming, and much more could get done.

By 2020, machine learning itself reached a similar turning point. Throughout its first decades, machine learning focused on tightly coupled combinations of representational spaces, loss functions, and optimizers—each carefully handcrafted by machine learning experts.

But as the field matured and the process of producing machine learning programs was better understood, it became possible to create a higher-level interface that made the design and implementation of machine learning programs easier.

For example, machine learning has been made easier by figuring out how to turn machine learning algorithms upon themselves. They can do machine learning about machine learning!

The tools that have made this possible are known as meta-learners—algorithms that learn how to learn.

Viewing the machine learning process itself as a higher-level algorithm, we can use our machine learning tools as a mechanism for optimizing that algorithm.

This step into **meta-learning** lets us solve some difficult problems that are otherwise impossible to address.

Discrete versus Continuous Algorithms

There are two approaches that unify the design of machine learning algorithms: one that reasons about discrete problems using satisfiability solvers, and another for continuous problems that combines continuous deep learning methods and more traditional programs using differentiable programming.

The contrast between discrete and continuous algorithms has stretched across this course, a difference based on the structure of representational spaces.

Decision trees have been our go-to example of a discrete structure. There's no ability to change one tree model into another by making continuous changes to a variable. Instead, there's always a discrete choice of what attribute to split on at each point in the tree and then a discrete choice of predicted class at each leaf.

Another discrete method is a genetic algorithm, which can search through representational spaces consisting of discrete binary strings.

But most of the representational spaces you've seen in this course use continuous representational spaces, where every intermediate model between two valid models is also a valid model.

Neural networks are continuous, including perceptrons and deep networks—and so are the probabilistic models, like causal models and inverse reinforcement learning.

Continuous methods and discrete methods have been studied in very different research communities. And while much of the focus in this course has been on continuous methods, general discrete methods are poised to join the mainstream of machine learning.

Lesson 01 included the following statement:

In machine learning, we do not tell the program what to do. We simply declare what the program should like by giving the computer code and examples that are used to assess how good its proposed rules are.

The lesson then explained that the style of programming in machine learning is more declarative than imperative, even though in a more general sense we are always telling the computer what to do. We declare that the program should make a good choice, and we give it input that defines what better and worse choices are—but we don't tell the computer what to do.

In other words, machine learning programs can always be viewed as declarative specifications. The designer communicates the goal of learning by providing labeled data and a loss function that conveys *what* a good rule looks like without saying *how* to construct it directly.

But this idea that machine learning is declarative, which has been hovering around in the background ever since, comes pushing to center stage in the discrete methods community.

Instead of tasking a traditional imperative language like Python to do a declarative job, the discrete methods community has devised specification languages that are declarative from the ground up. These are not all-purpose languages, but their distinct superpower is letting you use programming constructs to express what you want more explicitly.

Following this same basic strategy, the formal methods community in computer science has developed entire declarative specification languages for expressing discrete logical relationships.

The insight of specification languages is to let designers write specification programs while using traditional programming language constructs, such as conditionals, loops, variables, and arrays. A specification program expresses the *what* that the designer wants to be satisfied.

Then, the specification language figures out the *how* by bringing to bear a generic solver, known as a Boolean satisfiability solver.

By 2015, general Boolean satisfiability solvers had replaced problem-specific solutions for many logical reasoning problems of interest to the formal methods community, from circuit design to program analysis.

The Alloy language and model finder from MIT is an example of a specification language that provides designers a clean, high-level interface to Boolean satisfiability solvers.

Because they are both fundamentally declarative, it is natural and productive to use specification programs to solve machine learning problems.

Specification languages like Alloy give us a general approach for solving all kinds of discrete machine learning problems. And viewing the creation of machine learning programs through this high-level lens also brings new insight into the value of deep learning libraries for approaching continuous machine learning problems.

It is useful to think of a specification language like Alloy as being parallel to a deep learning system like Keras:

- ◆ Boolean variables are parallel to a neural network's weights,
- ◆ the satisfying of a logical formula is parallel to the minimizing of a loss function, and
- ◆ Boolean satisfiability is parallel to gradient descent.

This analogy encourages us to think of a deep neural network as a specification program.

And that makes sense. After all, any neural network can be equivalently expressed as a set of assignment statements, with one assignment statement per node.

A specification program that represents a deep network has variables that store continuous values that encode the weights of the network. The values of the weight variables are ultimately filled in by an optimization process to minimize a selected loss function.

In this way of thinking, the optimization process works by taking a derivative of this specification program, one line at a time, from bottom to top.

Even the most bare-bones neural network libraries can handle specification programs of simple assignment statements like these.

But other traditional programming constructs can be handled, too, with some additional complexity. For example, it's also possible to take derivatives of subroutines. Researchers have also found efficient approaches for taking derivatives of conditional if-then branches, loops, and even recursion. As a result, it is possible to differentiate any specification program.

Taking derivatives of specification programs is an idea that dates back at least to the 1980s. It goes by several names, including differentiable programming and computational differentiation. And it brings the power of the discrete methods community to continuous problems so central to deep learning.

In other words, differentiable programming generalizes the gradient descent approach you've seen for neural network programs while also completing the analogy with discrete specification languages like Alloy.

The ultimate promise of differentiable programming is already being realized in the popularity of the deep learning library PyTorch, which dates to 2017.

Because of the compatible but complementary strengths of ordinary imperative programs that focus on *how* to solve a problem and specification programs that focus on *what* problem to solve, the idea is to assemble both approaches into more complex programming systems. These systems use combinations of standard programs and deep networks and then refine the behavior of the entire assembly using data.

For computers, these more complex programming systems do not necessarily produce problems that are any more computationally intensive. But for humans, this simplified way of writing programs already has some significant benefits.

Starting in around 2016, examples of scaled-up differentiable programming began to appear. Projects combined machine learning with traditional programming in unified hybrid systems.

Traditional programming shines whenever a crisp program specification is available. By contrast, machine learning is really good at problems involving perception and soft preferences that are difficult to express explicitly—things humans think of as more like intuition or judgment. A hybrid program holds the promise of making it possible for the programmer to write each piece of the program in the format that is most natural for that piece.

Automatic Programming via Meta-Learning

The idea of differentiable programming is also important in meta-learning—where a learning algorithm is used to improve the behavior of another learning algorithm.

Consider a traditional program that takes in a sequence of characters and returns an output. For instance, the program might return whether or not the sequence would be a valid password, based on whether it has the right length and the right collection of symbol types. In this example, an instance would be a proposed password, and the category would be valid or invalid.

In machine learning, we pop up a level and create a program that takes collections of instances along with their associated categories as a higher-level input. Its output is a classifier—exactly the kind of thing that, in traditional programming, would have been written by hand.

When is it most useful to pop up a level like this?

We probably would not bother learning a classifier for passwords because basic rules for passwords are already explicit.

By contrast, instances like images or audio clips are ill-suited for traditional programming but have been handled spectacularly by machine learning. In addition, it is possible to gather many instances pairing inputs and desired outputs, making it possible to derive a classifier from this data.

Can we take this idea of automatic programming even further? Could we create automatic systems capable of writing even our machine learning programs for us?

Just like the jump from traditional programming to machine learning, popping up another level is a useful thing to do when

1. the desired level of accuracy is too hard to achieve at the level below and
2. examples are available to train on.

So, for example, consider writing a machine learning program for recognizing new faces. A generic learning algorithm can learn to recognize faces, but it will probably take thousands and thousands of examples each time we want to train it.

Instead, we could accomplish more if we had a machine learning algorithm that could do the preparatory design work that we'd be doing and automate feature extraction and parameter setting that's specific to the domain we care about.

Analogously to the jump from programming to machine learning, we'd provide collections of related datasets.

We'd ask this higher-level algorithm to produce a specialized learner that can solve problems from this class of collections—accurately and efficiently.

Since we're learning to learn, this approach is called meta-learning.

There are many forms that meta-learning can take. Human beings, who need to solve many diverse problems over the course of their lifetimes, become more and more proficient at learning new tasks. They become better learners. Think of musicians who have mastered so many instruments that they can pick up a new instrument in just a few minutes.

Similarly, a successful meta-learning algorithm can produce a specialized machine learning algorithm that can learn new tasks. And the specialized algorithm we create can learn new instruments, or whatever, with far fewer examples than a generic machine learning algorithm would need.

Try It Yourself

Follow along with the video lesson via the Python code:

[L25.ipynb](#)

Python Libraries Used:

math: Mathematical functions.

matplotlib.pyplot: Plots graphs.

random: Generates random numbers.

torch: PyTorch deep neural network library from Facebook.

torch.nn.functional: Building neural networks in PyTorch.

Key Terms

meta-learning: Machine learning about machine learning. Meta-learners typically leverage multiple datasets to create a machine learner that is well suited to handle other similar datasets.

READING

Hutson, “AI Researchers Allege That Machine Learning Is Alchemy.”

Innes, “What Is Differentiable Programming?”

Narodytska, Ignatiev, Pereira, and Marques-Silva, “Learning Optimal Decision Trees with SAT.”

Russell and Norvig, *Artificial Intelligence*, chap. 6.

QUESTIONS

1. What differentiates a discrete machine learning model from a continuous one?
2. Machine learning is the idea of automatic programming guided by data. What is automatic machine learning guided by data?
3. The meta-learning for the learning thermostat in this lesson learned from many users, but all users fell into two discrete categories: ones with an offset of $b = 0$ and ones with an offset of $b = \pi/2$. Update the example so that there are three categories of users: $b = \pi/3$, $b = \pi/4$ and $b = 2\pi/3$. Is MAML able to learn this new population of users without any additional parameter tuning?

Answers on page 487

Mastering the Machine Learning Process

Lesson 25 Transcript

Computers run machine languages. And, back in the day, machine languages were the only way people could program computers. It was painstaking work, but it fostered a close relationship between people and their machines. With the creation of higher-level programming languages starting in the 1950s and continuing with the development of accessible languages such as Python, more and more people could participate in programming, and much more could get done.

By 2020, machine learning itself reached a similar turning point. Throughout its first decades, machine learning focused on tightly coupled combinations of representational spaces, loss functions, and optimizers each carefully hand-crafted by machine learning experts. But as the field matured, and the process of producing machine learning programs was better understood, it became possible to create a higher-level interface that made the design and implementation of machine learning programs easier.

For example, machine learning has been made easier by figuring out how to turn machine learning algorithms upon themselves. They can do machine learning about machine learning. The tools that have made this possible are known as meta-learners—algorithms that learn how to learn. Viewing the machine learning process itself as a higher-level algorithm, we can use our machine learning tools as a mechanism for optimizing that algorithm.

This step into meta-learning lets us solve some difficult problems that are otherwise impossible to address. For example, we can develop a system that learns complex functions from just four data points. We'll look at two approaches that unify the design of machine learning algorithms: one that reasons about discrete problems using satisfiability solvers, and another for continuous problems that combines continuous deep learning methods and more traditional programs using differential programming.

The contrast between discrete and continuous algorithms has stretched across the course, a difference based on the structure of representational spaces. We likened the distinction to that of Lego bricks and sliders on a sound mixing board. Decision trees have been our go-to example of a discrete structure. There's no ability to change one tree model into another by making continuous changes to a variable. Instead, there's always a discrete choice of what attribute to split on at each point in the tree, and then a discrete choice of predicted class at each leaf.

LESSON 25 | MASTERING THE MACHINE LEARNING PROCESS

TRANSCRIPT

Another discrete method is a genetic algorithm, which can search through representational spaces consisting of discrete binary strings. But most of the representational spaces we've seen in this course use continuous representational spaces, where every intermediate model between two valid models is also a valid model. Neural networks are continuous, including perceptrons and deep networks; and, so are the probabilistic models, like causal models and inverse reinforcement learning. Continuous methods and discrete methods have been studied in very different research communities. And while much of our focus in this course has been on continuous methods, general discrete methods are poised to join the mainstream of machine learning.

Now, at the very beginning of this course, I said: In machine learning, we do not tell the program what to do. We simply declare what the program should like, by giving the computer code and examples that are used to assess how good its proposed rules are. I went on to point out that the style of programming in machine learning is more declarative than imperative, even though in a more general sense we are always telling the computer what to do. We declare that the program should make a good choice and we give it input that defines what better and worse choices are, but we don't tell the computer what to do.

That is, machine learning programs can always be viewed as declarative specifications. The designer communicates the goal of learning by providing labeled data and a loss function that conveys what a good rule looks like without saying how to construct it directly. But this idea that machine learning is declarative, which has been hovering around in the background ever since, comes pushing to center stage in the discrete methods community. Instead of tasking a traditional imperative language like Python to do a declarative job, the discrete methods community has devised specification languages that are declarative from the ground up. These are not all-purpose languages, but their distinct superpower is letting you use programming constructs to express what you want more explicitly.

Following this same basic strategy, the formal methods community in computer science has developed entire declarative specification languages for expressing discrete, logical relationships. The insight of specification languages is to let designers write specification programs while using traditional programming language constructs such as conditionals, loops, variables, and arrays. A specification program expresses the what that the designer wants to be satisfied. Then, the specification language figures out the how by bringing to bear a generic solver, known as a Boolean satisfiability solver.

By 2015, general Boolean satisfiability solvers had replaced problem-specific solutions for many logical reasoning problems of interest to the formal methods community, from circuit design to program analysis. The Alloy language and model finder from MIT is an example of a specification language that provides designers a clean, high-level interface to Boolean satisfiability solvers. Because they are both fundamentally declarative, it is natural and productive to use specification programs to solve machine learning problems.

Let's consider a discrete example. Imagine that you have a set of data and you want to learn a decision tree from it. The problem of designing an n -node decision tree that correctly classifies the training data can be expressed as a specification program. The specification program consists of a set of Boolean variables that answer questions like: What attribute should the root of the tree split on? And what class should be assigned to instances that reach the 7th leaf?

Then, behind the scenes, a set of constraints relate the values of these Boolean variables defining the decision tree to the classification of the instances in the dataset. Ultimately, the specification program defines the discrete problem: Fill in values for the Boolean variables so they correspond to a decision tree that classifies each of the examples in the dataset correctly. The specification language automatically converts this specification program into a Boolean satisfiability problem and solves it.

Specification languages like Alloy give us a general approach for solving all kinds of discrete machine learning problems and viewing the creation of machine learning programs through this high-level lens also brings new insight into the value of deep learning libraries for approaching continuous machine learning problems. It is useful to think of a specification language like Alloy as being parallel to a deep learning system like Keras: Boolean variables are parallel to a neural network's weights, the satisfying of a logical formula is parallel to the minimizing of a loss function, and Boolean satisfiability is parallel to gradient descent.

This analogy encourages us to think of a deep neural network as a specification program. And that makes some sense. After all, we saw in Lesson 3 that any neural network can be equivalently expressed as a set of assignment statements, with one assignment statement per node of the network. A specification program that represents a deep network has variables that store continuous values that encode the weights of the network. The values of the weight variables are ultimately filled in by an optimization process to minimize a selected loss function.

LESSON 25 | MASTERING THE MACHINE LEARNING PROCESS

TRANSCRIPT

In this way of thinking, the optimization process works by taking a derivative of this specification program, one line at a time, from bottom to top. Even the most barebones neural network libraries can handle specification programs of simple assignment statements like these. But other traditional programming constructs can be handled, too, with some additional complexity. For example, it's also possible to take derivatives of subroutines. For example, the weights of convolutional filters used in vision and those in transformer modules used in language define subnetworks. These subnetworks are called, just like subroutines, in different places in the main network. But since these subroutines are themselves sequences of assignment statements, derivative computations remain straightforward.

Researchers have also found efficient approaches for taking derivatives of conditional if-then branches, loops and even recursion. As a result, it is possible to differentiate any specification program. Taking derivatives of specification programs is an idea that dates back at least to the 1980s. It goes by several names, including differentiable programming and computational differentiation. And, it brings the power of the discrete methods community to continuous problems so central to deep learning. That is, differentiable programming generalizes the gradient descent approach we've seen for neural network programs, while also completing the analogy with discrete specification languages like Alloy.

The ultimate promise of differentiable programming is already being realized in the popularity of the deep learning library PyTorch, which dates back to 2017. Because of the compatible but complementary strengths of ordinary imperative programs that focus on how to solve a problem and specification programs that focus on what problem to solve, the idea is to assemble both approaches into more complex programming systems. These systems use combinations of standard programs and deep networks and then refine the behavior of the entire assembly using data.

For computers, these more complex programming systems do not necessarily produce programs that are any more computationally intensive. But for humans, this simplified way of writing programs already has some significant benefits. Starting back in 2016 or so, examples of scaled-up differentiable programming began to appear. Projects combined machine learning with traditional programming in unified hybrid systems.

Traditional programming shines whenever a crisp program specification is available. By contrast, machine learning is really good at problems involving perception and soft preferences that are difficult to express explicitly—

things humans think more of as intuition or judgment. A hybrid program holds the promise of making it possible for the programmer to write each piece of the program in the format that is most natural for that piece.

For example, consider a planner of flight-paths for a drone. On the one hand, ensuring the safety of a flight controller is easier using traditional methods. That's because the components of the program can be mapped to physical quantities whose properties are well understood. But learning-based systems can discover adjustments to the controller that are much more efficient than traditional controllers. A hybrid learning system can get the benefits of being reliably safe and adaptively efficient, by letting machine learning make small changes to a controller that's been hard-coded with safety constraints.

The idea of differentiable programming is also important in meta-learning where a learning algorithm is used to improve the behavior of another learning algorithm. Note that this approach goes beyond another possible meaning of meta-learner you might encounter. When we talked about the Netflix Prize, there was a sort of meta-learner that was constructed from an ensemble of learners. Here, by contrast, we are considering a meta-learner as a creator of learners.

Consider a traditional program that takes in a sequence of characters and returns an output. For instance, the program might return whether or not the sequence would be a valid password, based on whether it has the right length and the right collection of symbol types. In this example, an instance would be a proposed password, and the category would be valid or invalid. In machine learning, we pop up a level and create a program that takes collections of instances along with their associated categories as a higher-level input. Its output is a classifier—exactly the kind of thing that, in traditional programming, would have been written by hand.

When is it most useful to pop up a level like this? We probably would not bother learning a classifier for passwords because basic rules for passwords are already explicit. By contrast, instances like images or audio clips are ill-suited for traditional programming but have been handled spectacularly by machine learning. In addition, it is possible to gather many instances pairing inputs and desired outputs, making it possible to derive a classifier from this data.

Can we take this idea of automatic programming even further? Could we create automatic systems capable of writing even our machine learning programs for us? Just like the jump from traditional programming to

LESSON 25 | MASTERING THE MACHINE LEARNING PROCESS

TRANSCRIPT

machine learning, popping up another level is a useful thing to do when: 1) the desired level of accuracy is too hard to achieve at the level below; and 2) examples are available to train on. So, for example, consider writing a machine learning program for recognizing new faces. A generic learning algorithm can learn to recognize faces, but it will probably take thousands and thousands of examples each time we want to train it.

Instead, we could accomplish more if we had a machine learning algorithm that could do the preparatory design work that we'd be doing and automate feature extraction and parameter setting that's specific to the domain we care about. Analogously to the jump from programming to machine learning, we'd provide collections of related datasets. We'd ask this higher-level algorithm to produce a specialized learner that can solve problems from this class of collections accurately and efficiently.

Since we're learning to learn, this approach is called meta-learning. There are many forms that meta-learning can take. Human beings that need to solve many diverse problems over the course of their lifetimes become more and more proficient at learning new tasks. They become better learners. I've met musicians who've mastered so many instruments that they can pick up a new instrument in just a few minutes.

Similarly, a successful meta-learning program can produce a specialized machine learning algorithm that can learn new tasks. And, the specialized algorithm we create can learn new instruments, or whatever, with far fewer examples than a generic machine learning algorithm would need. The trade-off is that the meta-learner needs to train on many examples of datasets. So, if we want a meta-learner that produces a supervised learner, we'd need a lot of labeled datasets. In any case, learning to learn is arguably a key skill in general intelligence. So, meta-learning will likely be a part of any general AI system.

Without attacking the problem of meta-learning in its full generality, we'll look at one narrow version that has shown great promise called weight initialization. This example also helps show off the power of differentiable programming. One of the most elegant and widely successful meta-learners for weight initialization generalization is called MAML, short for model-agnostic meta-learner. Being model-agnostic means it can be applied across a wide variety of deep-learning scenarios. It derives the hyper-parameters needed by the low-level learners, specifically the initial weights they use for training. And, because the loss function is differentiable, this meta learner can be trained using meta gradients, gradients of the updates used in gradient descent.

To illustrate, let's imagine we start with three related tasks, T_1 , T_2 , and T_3 . These names might sound like a reference to the *Terminator* movies, but our three tasks here are very narrow, and very friendly. Here, a machine learning task is something defined by a set of labeled training examples. The meta-learning problem is to find a learning algorithm that performs well across these three sets of labeled training examples. We'll give our meta-learner algorithm an initial set of weights that the specialized learning algorithm will start off with when embarking on solving any of the three tasks T_1 , T_2 , or T_3 . We'll call the setting of these weights w .

Let's suppose we find that four steps of gradient descent is enough to train solutions to all three tasks from a random starting weights. We could ask ourselves: Might it be possible to find starting weights that learn good classifiers for all three tasks in even fewer steps? Well, we could engage in some kind of search or optimization process to discover a better set of initial weights.

How can we change the weights of w so that when we do gradient descent on the different tasks starting from those weights, the error is minimized? We can do gradient descent on the process of gradient descent. To do that, we'd be taking the derivative of a computation that itself has a derivative in it. That's fancy.

The arrival of the deep learning package PyTorch from Facebook in 2017 has provided a system that supports differentiable programming. Although we could write a similar program in Keras, PyTorch demonstrates how nice it is to work with a library designed to be differentiable at its core. Python itself wasn't set up to let PyTorch differentiate its functions. So, you have to use PyTorch's functions so PyTorch has access to them for taking derivatives. In a language built for differentiable programming, this distinction would not be needed. Instead, PyTorch is retrofitting Python. We can use PyTorch to develop an elegant implementation of MAML, the model-agnostic meta-learner.

Imagine we're creating a learning thermostat. Over the course of the day, the user occasionally sets the thermostat manually, providing a labeled data point. For example, the user might prefer cooler temperatures at night, for better sleeping, and warmer temperatures when waking up. The thermostat needs to learn a complete schedule; that is, what temperature target to use throughout the day. The machine learning problem of mapping time of day to temperature is a regression problem. We can solve it using any of a variety of methods.

LESSON 25 | MASTERING THE MACHINE LEARNING PROCESS

TRANSCRIPT

Let's use a dense neural network implemented in PyTorch. The network we'll use has one input representing the time of day and one output representing the desired temperature. In between are two layers of 32 units each, with complete connectivity between all the nodes in those layers. It has 32 weights to go from time of day to the first hidden layer, 32 times 32 equals 1,024 weights to go between the two hidden layers, then 32 weights to map to the output.

We can construct the network in PyTorch by importing a few libraries. Torch is the main library that computes gradients and handles tensors—arrays of data. Nn is a neural network library for creating layers and training them. And F is the functional library within nn that captures the layers. The subroutine net is our specification program for turning an input time into an output temperature. The unspecified values are carried along in the parameter data structure params. F is one of two basic methods torch provides for creating layers—specifically, it is a method where each layer is given the previous layer as input. It's called functional because, in mathematics, a functional is a higher-order function that takes functions as inputs and produces another function as output.

The modifiable parameters—the weights—of the network are stored in a data list, and that list can be treated as a unit. Again, the specific structure of the network is three layers of weights. Each layer is a fully-connected set of units. The units first sum up their input activations linearly, and then apply the rectifying linear unit, or ReLU, activation function. For this simplified thermostat example, the data we'll use to train the network is samples from a sine wave. You might imagine the sine wave as standing in for a more nuanced pattern of up and down temperatures a thermostat user might prefer over the course of the day.

We can train a network to predict this data in PyTorch by creating training data, represented by orange points in the plot. Then we repeatedly update the weights of our network to decrease prediction loss on those points. We sample the sine wave at 25 different x locations. For each one, the y coordinate is the sine of the x coordinate, shifted by b units. We're using the torch version of the random and sine functions because they are the function types that support the automatic differentiation, we'll need for gradient descent training.

Alpha is our learning rate, which says how far along the gradient we should update our parameters. We make gradient updates, repeating nters number of iterations times, here 10,000. To compute the gradient, we first put

the x data through our network net. Note that we give the network two things—the x coordinates we are using for training and the parameters of the network.

Torch has a loss function defined. We give our loss function the target y values and the output f of our network. We're using $L1$ loss, which is the absolute-value difference between the network's predicted outputs f and the target values y from our dataset. We then ask PyTorch to compute the gradient of our loss function given the parameters params. The gradient tells us what direction the parameters need to move to reduce the loss the most. Thus, the automatic gradient—autograd—takes the loss and the parameters as input.

Now, we move each of the parameters a little bit in the negative direction of the gradient, so as to descend the loss function. The learning rate alpha controls the size of these steps. Note that PyTorch has more streamlined ways of setting up and training simple networks like this. We are making all of the steps of training explicit in PyTorch function calls so PyTorch can manipulate them and take derivatives.

Trained on 25 training points, this learning algorithm can make accurate predictions of the underlying function. Trained on just 10 points, the learning algorithm tracks the function considerably worse—missing the peaks and troughs and a fair amount of the shape in between.

Trained on only four points, the learning algorithm does not capture the shape of the function at all. To be fair, the predictions the learner is making are reasonable given the tiny amount of data that it has. But apart from the location of the training points, the predictions are complete nonsense, as the learned function merely interpolates everywhere else. So, training a standard learning algorithm with just four points would not be sufficient for learning a thermostat-user's preferences.

Since we have background knowledge suggesting that different users are likely to be modeled by related functions, we can frame this learning problem as a META-learning problem. There are two phases of meta-training. In the first phase, the meta-learner is trained on 275,000 different datasets, artificially generated in this example. The datasets give the meta-learner the chance to make predictions for many different users. From each dataset, the meta-learner is given only four training points, and it is judged on its ability to predict four other points chosen from the actual target function.

LESSON 25 | MASTERING THE MACHINE LEARNING PROCESS

TRANSCRIPT

In the second training phase, the thermostat learner is deployed in a new home. This home is not one the learner saw during the first phase of training, but one that is drawn from the same distribution. It then needs to make predictions given just four labeled examples. For our demonstration, we'll imagine that everyone's day follows the same pattern, where the thermostat is set higher in the morning and evening when the user is at home. However, some users are early risers and some are late risers. So, every user wants the thermostat to follow only one of only two possible sine waves that differ only in terms of a shift leftward or rightward on the x axis.

Here's an example of the two patterns the meta-learner will see. I've also marked four points on each curve so you can see how little data the learner will have to work with. We're going to wrap the thermostat-learning algorithm we just used inside another loop that will train this algorithm many times, with many different datasets. The outer optimization will be performed by torch's implementation of stochastic gradient descent, or SGD.

For the meta-learner, we'll set a learning rate, lr , of 1 times 10 to the minus-2 or 0.01. For each user, the number of steps of stochastic gradient descent—the inner loop—will be just five. Note that we used 10,000 iterations for this problem when we didn't have a meta-learner. Now, that we do have a meta-learner, we're asking the user predictions to be made with very few iterations and a tiny dataset.

For each iteration, it , we create a new virtual user, defined by a choice of the shift parameter b . For that user, we collect four labeled data points from training. We also make four labeled data points for testing. Both training and testing use the same shift b . To learn for the current user, we repeat the thermostat-learning code from earlier. The difference is that learning takes place on a copy of the parameters. That's because there is a shared set of initial parameters across all users, but then the learning thermostat for each user adapts from there independently.

Again, we're doing gradient descent in the inner loop to learn the preferences of the user, but we're also doing gradient descent in the outer loop across users to learn good initial weight parameters for the inner loop. The learning algorithm is carried out with the SGD optimizer. Its gradient computation is zeroed out before the inner loop. After the inner loop, we test the `new_params` parameters it learned for the current user by predicting on four new points, represented by v_x and v_y and measuring this validation loss. The command `loss2.backward` triggers the gradient computation for the entire calculation since the gradient was zeroed out before the inner loop. Then, we have the SGD optimizer take a step.

Let's think for another moment about what's happening here. The outer loop loss, `loss2`, is a measure of how good a job the inner loop did at predicting new datapoints for the user. The only parameters it has to manipulate to improve this loss is the initial parameters, `params`. PyTorch is taking the derivative through the entire inner loop gradient descent algorithm. It is differentiating the inner learning algorithm itself to find an improved value of `params`. The problem being solved by the outer loop is: Find initial parameters that make the inner loop learning algorithm learn accurate predictions across the space of possible users.

And it is successful. After training on 275,000 users, we can show the inner loop temperature-learning algorithm just four data points and it learns to predict the entire temperature preference curve with a fair degree of accuracy. It learns well from users with either of the two shifts—both late risers and early risers. By making the learning process itself differentiable, we were able to optimize it using gradient descent.

And that brings us full circle. Through the arc of this class, we have studied how we can create programs that learn from data. All machine learning approaches share a common recipe. We define a representational space for potential solutions, a loss function for judging which solutions are most suitable to the problem, and an optimization function for identifying a solution with minimum loss.

We've examined five broad approaches, with over 24 more specific methodologies for creating learning programs, and we've sometimes even touched on how fruitfully these approaches can work together. But unlike Pedro Domingos, I don't think we'll ever have a single master algorithm for all of machine learning. There are simply too many different scenarios in which we want to apply the technology. Instead, I think we're trying to enrich our vocabulary for telling machines what to do.

In spite of all that's been accomplished in the first decade of deep learning, it is sometimes pursued more as a modern type of alchemy than as a branch of computer science. I think there will be many advantages to increasing our understanding of what machines do when they learn. We need to understand why overparameterization works, for example, and the answers will have all sorts of practical consequences. And one way to promote greater understanding is for everyone involved in disseminating machine learning—from researchers and developers, to journalists and marketers—to commit to communicating about their work in clear, direct terms.

LESSON 25 | MASTERING THE MACHINE LEARNING PROCESS

TRANSCRIPT

One development that has the promise to greatly expand the impact of machine learning on society is differentiable programming. That's because differentiable programming streamlines the process of combining machine learning with traditional programs—especially in a language like Python—and makes it possible to turn machine learning on itself. And remember: What's called good-old-fashioned artificial intelligence still runs using traditional programs. So, combining machine learning with traditional programs has the payoff of producing a more robust and predictable and beneficial form of artificial intelligence, now powered by the best of machine learning.

In *Star Wars*, Darth Vader faces his mentor Obi-Wan Kenobi one final time. Vader says, “The circle is now complete. When I left you, I was but the learner. Now I am the master.” And that’s where we’ve arrived. We have gone from learning about machine learning ourselves to writing programs that can learn about machine learning on our behalf.

In the years to come, I expect machine learning will advance to the point where programs can learn flexibly and robustly across a much wider range of problems and data relationships. We’ll be applying our algorithms in more diverse settings, so, machine learning programmers will need to select and sequence the kinds of data that give learners an appropriate foundation. Those learners will be making more valuable connections and inferences. And, rapidly increasing their capabilities will let them meet bigger challenges sooner.

At that point, creating a machine learning program will look less like training a machine and more like, well, teaching a living being. And humans will indeed be the masters. Whether we’ll be a force for good is up to us.

BIBLIOGRAPHY

Angwin, Julia, Jeff Larson, Surya Mattu, and Lauren Kirchner. "Machine Bias." *ProPublica*, May 23, 2016, <https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing>. Very influential and highly cited article detailing how machine learning systems can cause real-world harm if not applied carefully.

Bostrom, Nick. *Superintelligence: Paths, Dangers, Strategies*. 2014. This book sent Elon Musk into a tizzy about AI bringing about human extinction. Its main argument is presented in three main parts. The first part is an extremely thorough and lucid look at the then-current progress toward general AI. But the middle part, where runaway machine intelligence comes into being, is a huge leap. The last part, which looks at what will happen after general AI arrives, is fascinating from the perspective of war-gaming out dramatic alternatives. The first and last parts are recommended.

Caliskan, Aylin, Joanna J. Bryson, and Arvind Narayanan. "Semantics Derived Automatically from Language Corpora Contain Human-like Biases." *Science* 356, no. 6334 (2017): 183–186. <https://doi.org/10.1126/science.aal4230>. An influential analysis and account of how word embeddings encode cultural biases.

Charniak, Eugene. *Introduction to Deep Learning*. MIT Press, 2019. One of the first deep learning books to come out, this book covers a few critical topics well from an implementation/engineering point of view. Unfortunately, it doesn't cover transformer networks, so the material in **Lesson 16** doesn't have a corresponding reading.

Chollet, Francois. "How Convolutional Neural Networks See the World." *The Keras Blog*, January 30, 2016, <https://blog.keras.io/category/demo.html>. Provides illustrations of what kinds of visual filters are learned by convolutional neural networks, by the creator of Keras.

Domingos, Pedro. *The Master Algorithm*. Basic Books, 2015. The first book written by a prominent machine learning researcher for a general audience, this book makes an effort to present the breadth of the machine learning field in a unified way. As such, the goals of the book are very similar to those of this course.

BIBLIOGRAPHY

Frankle, Jonathan, and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks." Paper presented at the International Conference on Learning Representations 2019, New Orleans, LA. <https://arxiv.org/pdf/1803.03635.pdf>. Award-winning paper on pruning in deep networks.

Hadfield-Menell, Dylan, Anca Dragan, Pieter Abbeel, and Stuart Russell. "Cooperative Inverse Reinforcement Learning." Paper presented at the 30th Conference on Neural Information Processing Systems, Barcelona, Spain, November 2016. <https://arxiv.org/abs/1606.03137>. This paper proposes a different view of inverse reinforcement learning from one of the originators of inverse reinforcement learning. It suggests agents explicitly maintaining uncertainty about the goals.

Hao, Karen. "How Apple Personalizes Siri without Hoovering Up Your Data." *MIT Technology Review*, December 11, 2019, <https://www.technologyreview.com/s/614900/apple-ai-personalizes-siri-federated-learning/>. An account of how Apple is making learning from users more efficient and more private.

Hoover, Eric. "Dueling Economists: Rival Analyses of Harvard's Admissions Process Emerge at Trial." *The Chronicle of Higher Education*, October 30, 2018. <https://www.chronicle.com/article/Dueling-Economists-Rival/244964>. This article on fairness in college admissions is a great example of why causal reasoning should be central to important applications of data analysis.

Huszar, Ferenc. "ML beyond Curve Fitting: An Intro to Causal Inference and do-Calculus." inFERENCe, May 24, 2018, <https://www.inference.vc/untitled/>. This blog post is helpful in explaining how causal reasoning contrasts from standard machine learning.

Hutson, Matthew. "AI Researchers Allege That Machine Learning Is Alchemy." *Science*, May 3, 2018, <https://www.sciencemag.org/news/2018/05/ai-researchers-allege-machine-learning-alchemy>. A broad-audience article highlighting a criticism of deep learning that likens it to alchemy, which was historically important and ultimately laid the foundation for empirical scientific exploration of chemistry but wasn't itself scientific.

Innes, Mike. "What Is Differentiable Programming?" *flux*, <https://fluxml.ai/2019/02/07/what-is-differentiable-programming.html>. A description of the idea of differentiable programming.

Kleinberg, Jon. "An Impossibility Theorem for Clustering." In *Advances in Neural Information Processing Systems 15* (NIPS 2002), edited by S. Becker, S. Thrun, and K. Obermayer, pages 463–470. <https://www.cs.cornell.edu/home/kleinber/nips15.pdf>. A theoretician's look at the problem of clustering, showing that three intuitively desirable properties of clustering algorithms cannot be simultaneously satisfied.

Kohs, Greg, dir. *AlphaGo*. Moxie Pictures/Reel As Dirt, 2017. <https://www.alphagomovie.com/>. A gripping documentary about a reinforcement-learning system that unseated, and deeply unsettled, a human champion.

Koren, Yehuda, Robert Bell, and Chris Volinsky. "Matrix Factorization Techniques for Recommender Systems." IEEE Computer Society, August 2009, <https://www.inf.unibz.it/~ricci/ISR/papers/ieeecomputer.pdf>. Fun read about mathematical techniques used in the Netflix Prize competition that's peppered with evocative movie references.

Landauer, Thomas K., and Susan T. Dumais. "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction, and Representation of Knowledge." *Psychological Review* 104, no. 2 (1997): 211–240. <http://www.stat.cmu.edu/~cshalizi/350/2008/readings/Landauer-Dumais.pdf>. Two researchers trained as behavioral scientists share their work on word embeddings and how it parallels the kinds of things people know about words.

Landauer, Thomas K., Darrell Laham, and Peter Foltz. "Learning Human-like Knowledge by Singular Value Decomposition: A Progress Report." In *Advances in Neural Information Processing Systems 10* (NIPS 1997), edited by M. I. Jordan, M. J. Kearns, and S. A. Solla, pages 45–51. <http://papers.nips.cc/paper/1468-learning-human-like-knowledge-by-singular-value-decomposition-a-progress-report.pdf>. This paper is basically about tricks you can get latent semantic analysis to do. The authors report that latent semantic analysis matches human assessments of essay quality; selects synonyms and antonyms from a set of choices; mimics priming effects of word relatedness; simulates associations connected with various logical fallacies people make; identifies similar passages of text, even if written in different languages; and helps judge the reading level and overall textual coherence of passages of text.

Lazer, David, and Ryan Kennedy. "What Can We Learn from the Epic Failure of Google Trends?" *Wired*, October 2015, <https://www.wired.com/2015/10/can-learn-epic-failure-google-flu-trends/>. Well-written account of the impact of overfitting in the real world.

BIBLIOGRAPHY

Le, James. “The 5-Step Recipe to Make Your Deep Learning Models Bug-Free.” January 15, 2020, <https://jameskle.com/writes/troubleshooting-deep-neural-networks>. Blog post presenting helpful hints for troubleshooting neural networks.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning.” *Nature* 521 (May 28, 2015): 436–445. <https://doi.org/10.1038/nature14539>. These three authors—who won the Turing Award for their work in deep learning—give an account of their work in a *Nature* special issue on machine learning.

McClelland, James, and David Rumelhart. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs and Exercises*. MIT Press, 1989. This book is mentioned primarily for its historical significance. It’s what was used by students of neural networks just starting out in the late 1980s, and its treatment of running basic networks is very direct and easy to follow.

Mitchell, Melanie. *An Introduction to Genetic Algorithms*. MIT Press, 1998. One of the first and clearest textbooks to describe genetic algorithms. It’s appropriate for an introductory computer science course on the topic.

Mitchell, Tom. *Machine Learning*. McGraw-Hill Education, 1997. This textbook is a little out of date now—it hasn’t been updated to cover deep learning—but it describes the basic machine learning algorithms really well, at least to a general computer science audience.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, and S. Petersen. “Human-Level Control through Deep Reinforcement Learning.” *Nature* 518, no. 7540 (2015): 529–533. A landmark paper showing that deep neural networks can be used in the context of controlling a video game from input images. It helped get DeepMind, the group that did the work, bought by Google for around \$500 million. It was also the first machine learning paper to appear in the prestigious journal *Nature*. The visibility of and excitement around the results helped launch the deep learning revolution.

Nakkiran, Preetum, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. “Deep Double Descent: Where Bigger Models and More Data Hurt.” Paper presented at the International Conference on Learning Representations 2020, virtual conference. <https://arxiv.org/pdf/1912.02292.pdf>. A thorough account of the double descent phenomenon and its manifestation in real datasets.

Narodytska, Nina, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. "Learning Optimal Decision Trees with SAT." *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, 2019. <https://www.ijcai.org/Proceedings/2018/0189.pdf>. A paper showing how decision tree learning can be viewed as a Boolean satisfiability (SAT) problem and solved with discrete approaches.

O'Neil, Cathy. *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown, 2016. Worth recommending for the titular pun alone, this book brought concerns about algorithmic bias and its impact on society to a wide audience. It covers the most compelling examples very well.

Pearl, Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988. This book documents the foundations of Bayesian networks and is appropriate for people in computer science just starting out in the field.

Pearl, Judea, and Dana Mackenzie. *The Book of Why: The New Science of Cause and Effect*. This book—by Pearl, who spearheaded the effort to make machines and machine learning causal—is his attempt to make these ideas accessible to a general audience. It's interesting and valuable, although it's pretty clear that Pearl has an axe to grind in favor of the causal perspective.

Rabiner, L. R., and B. H. Juang. "An Introduction to Hidden Markov Models." *IEEE ASSP Magazine*, January 1986, http://ai.stanford.edu/~pabbeel/depth_qual/Rabiner_Juang_hmms.pdf. A classic survey of the algorithms and applications of hidden Markov models.

Russell, Stuart, and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson, 2020. This textbook is the most widely used book for introductory AI, of which machine learning has become an important part. The authors are very grounded in the idea that AI programs are agents that take in observations from the world and act on the world, which provides a consistent underpinning for everything covered. This edition includes a lot of material on machine learning and even deep learning. The target audience is computer science undergraduates, and it presents things very well for that group.

Schölkopf, Bernhard. "Causality for Machine Learning." Max Planck Institute for Intelligent Systems, Tübingen, Germany, December 23, 2019. <https://arxiv.org/pdf/1911.10500.pdf>. A survey of causal machine learning by a core contributor to the research community.

BIBLIOGRAPHY

Shao, Cecelia. “Checklist for Debugging Neural Networks.” *Towards Data Science*, March 14, 2019, <https://towardsdatascience.com/checklist-for-debugging-neural-networks-d8b2a9434f21>. Blog post presenting helpful hints for debugging neural networks.

Sutton, Richard, and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. An introduction to reinforcement learning by the founders of the field. It’s written for people with a computer science background and maybe with some machine learning background as well. While the programming examples are great, the book focuses too much on variations of their preferred temporal difference approach and not enough on the broader concept of reinforcement learning.

Toews, Rob. “Deepfakes Are Going to Wreak Havoc on Society. We Are Not Prepared.” *Forbes*, May 25, 2020, <https://www.forbes.com/sites/robtoews/2020/05/25/deepfakes-are-going-to-wreak-havoc-on-society-we-are-not-prepared/#d05d69b74940>. An accessible description of deepfakes, generative adversarial networks (GANs), and their potential bad impacts on society.

Tufekci, Zeynep. “Machines Shouldn’t Have to Spy on Us to Learn.” *Wired*, March 25, 2019, <https://www.wired.com/story/machines-shouldnt-have-to-spy-on-us-to-learn/>. An argument for privacy-preserving machine learning.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need.” Paper presented at the 31st Conference on Neural Information Processing Systems, Long Beach, CA, December 2017. <https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf>. The paper that kicked off the transformer network craze.

Zhu, Jun-Yan, Taesung Park, Phillip Isola, and Alexei A. Efros. “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.” Paper presented at the IEEE International Conference on Computer Vision, Venice, Italy, October 2017. <https://junyanz.github.io/CycleGAN/>. A description and demonstration of images of the CycleGAN system that can learn to map classes of images to related images in other classes.

ANSWERS

LESSON 01

1. Data easy to come by; rule hard to express by hand.
2. Supervised: labels given; unsupervised: no labels given; reinforcement: evaluations given on the combination of instances and labels.
3.

```
import math

def distance(c1, c2):
    return(math.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2
                    +(c1[2]-c2[2])**2))
```

Code: [L01qs.ipynb](#)

The questions on page 15

LESSON 02

1. Prints 20 hellos.
2.

```
def goodbye(x,y):
    for i in range(x):
        print("hello")
    for i in range(y):
        print("goodbye")
```

The questions on page 35

LESSON 03

1. Rows are instances; columns are features.
2. No, it chooses its splits “greedily,” moving top-down without ever going back, and therefore might not hit on the right combination of splits to classify the data best.
3. The 20-split tree is not so easy to understand, and there’s a very unusual split that says, “If eight or more pregnancies (and a bunch of other conditions), then no diabetes in five years.” There is only one example in the data that matches this condition, so there’s very little reason to expect that it would generalize.

Code: [L03qs.ipynb](#)

The questions on page 46

LESSON 04

1. There’s no bound on the number of hidden units a network can have. In fact, it can be very useful to have many, many hidden units if the inputs and outputs are continuous values if you want to represent a complicated function like a sine wave.
2. (a) neural network: perceptual input; (b) decision tree: categorical attributes that don’t need to be combined in complex ways; (c) decision tree: spelling rules are somewhat regular; (d) neural network: perceptual.
3. Accuracy on unseen data goes up and up; more layers give the network more power to express complex concepts.

Code: [L04qs.ipynb](#)

The questions on page 63

LESSON 05

- ```
1. import numpy as np
p = an image array
l = np.reshape(p,(-1,3))
sum(l[:,1] > l[:,0])
```
- They are different. With the *distance* formulation, you can express the idea of a very precise color of green being the colors either strictly darker or strictly lighter than a target color will necessarily be categorized the same as the target color. However, by extending the feature set to include squares of the components, the distance rule can be captured.
- The weight associated with green is very consistent, presumably because so many of the training examples have such a large value on this component, so it's essential that the network get the weight right.

**Code:** [L05qs.ipynb](#)

[The questions on page 79](#)

**LESSON 06**

- Roughly, it makes sense to use a generative approach if you believe the data is well captured by a generative process; that is, there are clear relationships among the positive examples and among the negative examples. When this kind of structure isn't present, or if it's there but is much more complicated than the decision rule, it's better to use a discriminative approach.
- The parameters are chosen to maximize the likelihood of the observed data, so the negative log likelihood is the loss function it minimizes.
- The naive Bayes classifier is the fastest to train and ends up with an accuracy close to that of the neural network.

**Code:** [L06qs.ipynb](#)

[The questions on page 97](#)

**LESSON 07**

- That means all organisms have the same chance at reproduction, which means the individuals are chosen for reproduction uniformly at random. That means the fitness function has no role in reproduction and optimization will not take place.
- Here are a few common examples: a one-point crossover where one parent contributes the left part of the string and the other parent contributes the right part, and the switch point is chosen at random. Another option is uniform crossover, where each bit position comes from one of the two parents uniformly at random.
- Without the absolute value, the system does very poorly because it can't represent the target function. With absolute value, it can learn the function! Note that the symbolic regressor often does well but sometimes can't handle even very simple ones. As an example,  $|x|-4$  fails half the time, even though  $|x-4|+4$  works nearly all the time.

**Code:** [L07qs.ipynb](#)

The questions on page 114

**LESSON 08**

- The number of nearby neighbors grows linearly with the dimension, but the number of non-neighbors grows exponentially—much faster.
- Since distance between colors seems to be a useful signal in this problem, nearest neighbors would be expected to work well. It's very important, though, that the training data includes a wide variety of colors from the image so that any new color will be close to something in the labeled training data.
- The nearest neighbors are much closer than would be expected for random data in a 486-dimensional space. Because testing data has close neighbors, 1-nearest neighbor has an excellent chance of generalizing well.

**Code:** [L08qs.ipynb](#)

The questions on page 131

## LESSON 09

1. Regularization and cross-validation.
2. You need to decrease the dimensionality of the representational space by half.
3. Nearest neighbors is much faster and also more accurate for everything but the RBF kernel—which is more accurate. It is interesting to note that RBF kernels are the most like nearest neighbors; it seems like a good dataset for nearest neighbors.

**Code:** [L09qs.ipynb](#)

The questions on page 148

## LESSON 10

1. The benefit is that the new data should be more representative of the users of the current system, thereby providing more useful/accurate information. The risk is that the new data will further exacerbate inequalities in the data, making the system more useful to the people it works well for and simultaneously less useful for the people it works poorly for.
2. Customer complaints are a proxy for product failures, but they are not a direct measure of product failures. The company might just be seeing that some communities simply have more time or energy to make complaints. It might be worth the expense of collecting data more uniformly by contacting customers directly to see which products actually have failed and whether the rate is connected to location.
3. Logistic regression does a slightly better job on this dataset. It is better suited to making probabilistic judgements than the linear classifier, and it is trained for accuracy, unlike naive Bayes. However, we wouldn't expect precisely the same behavior in all possible datasets.

**Code:** [L10qs.ipynb](#)

The questions on page 167

**LESSON 11**

1. Scale invariance, consistency, richness.
2. In gradient descent, the set of possible assignments of the weights to values is infinite, so you can keep improving while still having an infinite number of possibilities left. In  $k$ -means, each iteration changes the discrete assignment of points to centers, of which there is only a finite number. Therefore, after a finite number of iterations, the set will be exhausted and no further improvements are possible.
3. For this dataset and with these parameters, Agglomerative Clustering did better. You might've expected KMeans to be a better fit because of its emphasis on compactness, but AgglomerativeClustering is able to make longer chains of inference here, relating images of digits to more distant images of digits if they have "intermediate" forms that are from the same class. It is often the case that the best choice of algorithm depends on the specific application domain.

**Code:** [L11qs.ipynb](#)

The questions on page 187

**LESSON 12**

1. Overexploring wastes labeling resources on items that are clearly not compatible. Overexploiting runs the risk of the system making bad predictions due to a lack of exposure to compatible items.
2. (a) Deploy bots to create many fake reviews; (b) Set the interest vectors to all 1s to guarantee the maximum dot product; (c) Make bots that click on the items and/or use clickbait descriptions.
3. The alphas are much larger than for the greedy chooser. The best performance we see is for alpha = 500. However, top performance for this problem is much lower for Thompson sampling than greedy. Thompson sampling is slower, too.

**Code:** [L12qs.ipynb](#)

The questions on page 206

**LESSON 13**

1. The evaluation function can be trained using supervised learning from game logs or temporal difference learning. The sampling policy can be trained using supervised learning from expert games.
2. The game tree requires that all players make their decisions based on the true state of the game. In partial-information games, different nodes of the game tree must have the same action, and standard game-tree search cannot be modified to respect this property.
3. Since the payoffs are so noisy and there are so many actions, it can't find reliable differences between them. It ends up finding one or more that seem to have positive expected value and doesn't realize that action 37—quit—is actually best.

**Code:** [L13qs.ipynb](#)

The questions on page 225

**LESSON 14**

1. The ImageNet Challenge.
2. Convolutional layers.
3. The raw images should be quite poor because there is simply not enough training data to separate the high-dimensional vectors into classes. Indeed, the error rate is 40% to 50% for the 10 images, which is indistinguishable from chance performance.

**Code:** [L14qs.ipynb](#)

The questions on page 242

**LESSON 15**

1. Gradients involve products of values across all layers of a network. In a network with many layers, that means multiplying many values together, resulting in very big numbers or very small numbers, depending critically on the magnitude of those numbers. Exploding gradients manifest as the weights in a network getting very big during training. Vanishing gradients manifest as the weights not changing enough from one iteration to the next during training.

## ANSWERS

2. Too high: Updates are erratic, with loss jumping up and down from one iteration to the next. Too low: Progress is slow, with loss barely changing from one iteration to the next.
3. For alpha 0.001, the accuracy is 61%. It goes up to 85% for alpha 0.1, which is better than what was found in the lesson. Then, it goes back down to 50% for alpha 0.5. So tuning makes a pretty big difference!

**Code:** [L15qs.ipynb](#)

The questions on page 258

## LESSON 16

1. Seven types, 15 tokens.
2. Too small means we won't be able to capture the patterns of similarities well enough. Too big means that the patterns we find in our training data won't necessarily generalize well to new data—overfitting.
3. 

```
def wefat(w, A, B):
 As = [cos(w,a) for a in A]
 Bs = [cos(w,b) for b in B]
 v = (np.mean(As) - np.mean(Bs))/np.std(As + Bs)
 return(v)
```

| Job          | Percent Female | Female Score |
|--------------|----------------|--------------|
| secretary    | 94.0           | -0.059       |
| typist       | 85.1           | 0.993        |
| phlebotomist | 75.0           | 0.643        |
| telemarketer | 65.3           | 0.322        |
| packer       | 54.5           | -0.027       |
| appraiser    | 45.3           | -1.204       |
| dentist      | 35.7           | 0.130        |
| rancher      | 25.8           | -0.984       |
| announcer    | 17.7           | -1.050       |
| firefighter  | 5.1            | -0.485       |

Correlation: 0.64.

**Code:** [L16qs.ipynb](#)

The questions on page 277

**LESSON 17**

1. Move chunks of texts around to make transitions less awkward.
2. By unrolling/unfolding the loops to turn them back into a feedforward network.
3. One way to proceed is with a fill-in-the-blank approach: “I met a firefighter. \_\_\_ ...” and look at the relative number of times the blank is filled in with *She* versus *He*. With this approach, you’ll likely get a correlation of around 0.75, which is even higher than the word embedding approach.

**Code:** [L17qs.ipynb](#)

The questions on page 297

**LESSON 18**

1. The *generator* produces images, and the *discriminator* assesses them. The goal is to generate an image that the discriminator “likes.”
2. An adversarial example is an image of one type of object that is minimally modified to fool a network into thinking it is an image of another type of object. The existence of these examples worries researchers because it shows that networks focus on very different things in images than people do. They could be used to trick AI systems or even robotic cars to behave differently and perhaps dangerously.
3. The stylistic-image generator does a good job of transforming the background elements in a way that looks more “painted.” There are speckles and streaks that match the stroke style of Van Gogh. It also does a pretty good job of making the buildings and plants look more cartoonish and hand-drawn, with less detail and bolder boundaries. On the other hand, the most striking aspect of Van Gogh’s work is the swirls in the sky; that is, the sky isn’t just made of streaks, but the streaks go together to make tumbling waves. The stylistic-image generator misses these features. Being able to accurately capture low-level perceptual patterns but missing broader high-level conceptual patterns is very common for neural network approaches.

**Code:** [L18qs.ipynb](#)

The questions on page 319

**LESSON 19**

1. a: Such a program is too difficult for a person to write.  
b: The generator will only be able to create the target car, not cars in general. c: An adversarial example will be created that doesn't look like a car but is still recognized as such.
2. Try to look for odd behavior in the backgrounds and around the edges of the faces.
3. The trained GAN produces a mix of recognizable digits and weird invented line drawings.

**Code:** [L19qs.ipynb](#)

The questions on page 337

**LESSON 20**

1. Speaker independence, large vocabulary, and connected speech.
2. The language model and the pronunciation model.
3. Training accuracy is about the same at 96%, but testing accuracy drops to around 54%. Those convolutional layers make a huge contribution to generalization performance.

**Code:** [L20qs.ipynb](#)

The questions on page 357

**LESSON 21**

1. a: Indirect handwritten, b: Direct from examples, c: Direct handwritten, d: Indirect from examples.
2. Wikipedia also includes goal-content integrity, which is a special kind of self-preservation concerned with sticking to the original goal. Another is freedom from interference, which can be seen as a special kind of controlling resources—specifically, its ability to pursue its goal.
3. No, it can't make such a temporally inconsistent behavior. The resulting reward function assigns a small penalty for white, a big penalty for blue, and a moderate penalty for orange. Yellow and green get positive rewards, with yellow's larger than green's. The optimal trajectory with this reward function goes straight to yellow and stays there, never bothering to visit green.

**Code:** [L21qs.ipynb](#)

The questions on page 376

**LESSON 22**

1. The conditional probability is computed with respect to a specific distribution over the data. It's asking, in that distribution: If  $y$  happens, what's the probability that  $x$  also happens? The outcome of an intervention is asking: What happens, on average, to  $x$  when we force  $y$  to happen? It includes the cases where  $x$  and  $y$  would both have been true, but it also includes the cases where  $y$  would not have been true but we force it to be true and observe the effect on  $x$ .
2. Yes and no. We can use the *do*-calculus on the causal diagram to discover whether there is another way to estimate the quantity without running a controlled experiment. Sometimes we can and sometimes we can't, but the *do*-calculus will tell us for sure which is the case.
3. It's complicated. The average treatment effect is 0.553, and the causal estimate is 0.375. That would seem to mean that "intervening" by making someone female does indeed make that individual more likely to survive, although less so than the correlation numbers suggest. But it's still not clear how to interpret these numbers because the causal analysis is built on some assumptions, and it doesn't really make sense to "intervene" on sex. The biggest problem is that there's no direct measurement of how the crew treated each passenger, so we can't control for how femaleness is influencing crew behavior. The data isn't as informative as we'd like, so our conclusions are necessarily limited—even when we use causal analysis tools.

**Code:** [L22qs.ipynb](#)

The questions on page 395

**LESSON 23**

1. Network can't represent solution, error is high; network can represent solution, error is low; network can represent noise in data, error is moderate; network can represent arbitrary functions, error is low again.
2. ii. Original values from the first network is the best—that's the lottery ticket, or the weights that led the first network to perform well. i. Starting where the values left off works moderately well. iii. The scheme of random values in the same range works poorly.

- No, it seems to be something closer to the standard overfitting curve. Performance improves and then (slowly) declines with increasing numbers of hidden units.

**Code:** [L23qs.ipynb](#)

The questions on page 416

## LESSON 24

- It is a way of letting computers do calculations on encrypted data without revealing that data during or after the computation. It's relevant to machine learning because it means that it could be possible for learning algorithms to work on your data without even being able to see the data it is working on.
- Differentially private.
- In this case, there was no decrease in accuracy and no measurable increase in running time.

**Code:** [L24qs.ipynb](#)

The questions on page 436

## LESSON 25

- In a continuous model, all of the models in between a pair of models are also valid models. In a discrete model, you have to jump from one model to the next, making it impossible to use things like gradient descent to optimize. Discrete solvers can leverage satisfiability algorithms to jump around.
- Meta-learning!
- Yes, it seems to work very well. However, the results don't generalize to other values of  $b$ , such as  $b = 0$ .

**Code:** [L25qs.ipynb](#)

The questions on page 457

# GLOSSARY

## activation function

In a neural network, the function that translates a unit’s incoming sum of weighted activations to its output activation. Examples include linear, step, sigmoid, and ReLU. [L04](#)

## adversarial examples

Any examples that fool neural network classifiers. [L18](#)

## algorithm

A process to be carried out for solving a specific problem by a computer. A learning algorithm (for example, ID3 for decision trees) is one that takes a dataset of labeled instances and produces a rule. An optimization algorithm (for example, backpropagation for gradient descent) is one that searches a space to find a solution with low loss. [L01](#)

## attention

Reweighting of inputs to a network with the goal of enhancing some of them to improve recognition accuracy. [L17](#)

## autoencoder

A neural network trained on a set of vectors to map, as accurately as possible, a vector from its training set as an input to precisely the same vector as an output after internally representing the vector in a compressed form. [L16](#)  
*Closely related to singular value decomposition.*

## autoencoding

The behavior of an [autoencoder](#). [L16](#)

## backpropagation

Efficient algorithm dating to 1986 for computing gradients (the high-dimension “slope”) of neural networks. Used for training neural networks by gradient descent. [L04](#)

## Bayesian network

A graphical representation of a joint probability distribution over random variables. [L01](#) *See also naive Bayes.*

## causal graph

A cousin of Bayesian networks that captures the relationship between variables even when interventions are taken to change specific values. [L22](#)

**classifier**

A rule that maps instances to discrete classes. L05 *Also called **discriminator**.*

**computational graph**

A representation that keeps track of the steps of a computation to support automatic differentiation. L18

**convolution**

A mathematical operation where the dot product of one vector is taken with another vector with its components shifted through a range of values. A key tool for neural networks achieving translation invariance in image recognition. L14

**convolutional neural network**

Neural network that learns convolutional operators designed to capture local features in images or sequence data and combine the local features to classify instances. L14 *Also called **deep convolutional neural network**.*

**cross-validation**

Technique to combat overfitting by setting aside some of the training data to help assess generalization and avoid using a representational space that is too big. L09

**data**

The information a machine learning algorithm works with. It is often divided into training data, testing data, and sometimes also validation data. L01

**decision tree learning**

Creating a hierarchically structured classifier from data. L03

**decision tree regressor**

A decision tree classifier that outputs a number instead of a class. L13

**deep convolutional neural network**

*See **convolutional neural network**.*

**deep neural network**

Neural network with more than three or four layers. L04

**delta rule**

An update rule for neural network learning that comes from calculating the derivative of a one-layer neural network with squared loss. [L05](#)

**differentiable programming**

An approach to building software systems that generalizes neural network training. It allows the program itself to be analyzed using calculus to more easily reason from output to input. [L24](#)

**differential privacy**

A property of a data-storage system or trained machine learning system that makes it difficult to compare two versions that differ in only one training instance to recover that instance. [L24](#)

**discriminator**

*See classifier.*

**DPLL satisfiability**

The Davis-Putnam-Logemann-Loveland algorithm for Boolean satisfiability that works by backtracking.

**embedding**

Mapping objects into a vector space. [L16](#) *See also word embedding.*

**evolutionary algorithm**

*See genetic algorithm.*

**expectation-maximization algorithm**

An unsupervised Bayesian method that learns about latent structure by alternating between a labeling step (expectation) and a parameter-estimation step (maximization). In some probabilistic settings, it is a competitor to gradient descent for neural networks. [L20](#)

**generative adversarial network (GAN)**

A neural network approach that learns to mimic properties of a given set of training data by building one network for producing instances and one for recognizing whether an instance comes from the training data. The basis for most work on deepfakes. [L19](#)

**genetic algorithm**

An approach to optimizing a function inspired by Charles Darwin's principle of natural selection. [L01](#) *Also called evolutionary algorithm.*

**genetic programming**

The application of genetic algorithms to optimize programs. [L07](#)

**gradient descent**

The process of optimizing a function by iteratively moving its parameters in the direction that causes the function's value to decrease. [L04](#)

*See also* [backpropagation](#).

**gram matrix**

A distance measure that captures similarities between objects. [L18](#)

**hidden Markov model**

A Bayesian-style model that generates sequences by producing observable outputs overlaid on a Markov chain. Commonly used in speech recognition. [L20](#)

**homomorphic encryption**

A form of secure encryption of data that allows computations to be carried out on the data without exposing the data itself. [L24](#)

**hyperparameter search**

A process of tweaking the hyperparameters, perhaps using a grid search over values of the hyperparameters. [L15](#)

**hyperparameter**

Higher-level parameter, such as learning rate, that influences the process of setting other lower-level parameters. [L04](#)

**instantiate**

In computer science, to create a concrete instance of a more general class. [L19](#)

**inverse reinforcement learning**

The problem of going from observations of behavior to estimates of the rewards that the behavior was selected to optimize. [L21](#)

***k*-armed bandit**

A decision problem where an agent must decide which of  $k$  initially unevaluated actions to choose to maximize payoff. For example, A/B testing is a two-armed bandit. A contextual bandit can make its judgments based on vectors that describe the current context. [L12](#)

**k-means**

An approach to unsupervised clustering that iteratively defines a set of centers and assigns vectors to their closest centers. [L11](#)

**k-nearest neighbors**

Also known as nearest neighbors, instance-based learning, memory-based learning, or lazy learning. An approach to supervised machine learning that uses “nearness” as a stand-in for “likely to share labels.” [L08](#) *Also called subnetwork.*

**latent semantic analysis**

An early form of word embedding that uses singular value decomposition to reconstruct a term-by-document matrix using several hundred dimensions. Invented by psychologist Thomas Landauer and colleagues in the 1980s, latent semantic analysis has been shown to capture some important properties of human language learning and use. [L16](#)

**linear regression**

Regression where the output rule is the coefficients of a line. [L12](#)

**logistic regression**

An approach to classification that’s equivalent to constructing a one-layer neural network with a sigmoid activation function. Similar in overall structure to a perceptron, naive Bayes, or a linear support vector machine. [L10](#)

**loss function**

A measure of how “incorrect” a rule is. The loss function based on data can be used to guide the construction of better and better rules in the context of machine learning. Examples include mean squared error for regression data; cross entropy and Kullback-Leibler divergence for categorical data; and hinge loss function for  $-1/1$  binary classification. [L01](#)

**lottery ticket hypothesis**

A possible explanation for the behavior of deep networks that allows it, once trained, to be pruned to a much smaller size. [L23](#)

**machine learning**

The process of using data to construct rules. The main settings of machine learning are supervised learning, unsupervised learning, and reinforcement learning. [L01](#)

**Markov chain**

A transition system consisting of discrete states. [L20](#)

**maximum likelihood**

A probability-based criterion for machine learning that states that the best rule is one that makes the observed data as likely as possible. [L21](#) *Closely related to loss function.*

**meta-learning**

Machine learning about machine learning. Meta-learners typically leverage multiple datasets to create a machine learner that is well suited to handle other similar datasets. [L25](#)

**naive Bayes**

A specific type of Bayesian network that models the observed features as being probabilistically related to the class but independent of one another. [L06](#)

**natural language processing**

The use of computer programs to solve problems in written or spoken language. [L04](#)

**nearest neighbors**

*See [k-nearest neighbors](#).*

**network**

A collection of items with a directed set of connections between them. Examples include neural networks and Bayesian networks. [L01](#)

*See also [subnetwork](#).*

**neural network**

A representational space for rules consisting of activations propagating from input to output. One of five long-standing approaches to machine learning, which often leverage gradient descent for training. [L01](#)

**optimization problem**

The computational challenge of finding objects that result in high score or low loss. A key step in producing rules via machine learning. [L09](#)

**optimizer**

A program or algorithm for solving an optimization problem. [L01](#)

**overfitting**

A problem that arises when a rule is learned from a large rule space using too little data, resulting in poor performance on unseen examples. [L01](#)

**perceptron**

A representational space for rules equivalent to a one-layer neural network with step-function activation. The earliest neural network that was studied. [L09](#)

**policy**

A controller for a sequential decision problem, typically represented as a function that maps state to action. [L21](#)

**polynomial regression**

Regression where the output rule is the coefficients of a polynomial of a given degree. [L23](#)

**polysemy**

The property of words having more than one meaning, making it difficult to know how to interpret words. [L16](#)

**Q-learning**

A variant of temporal difference learning where values are learned for state-action pairs instead of states alone. [L13](#)

**recurrent network**

Neural network where the output of some group of units is fed back into that same group of units, resulting in an activation loop. [L17](#)

**regression**

The problem of mapping instances to numbers. [L03](#) *See also logistic regression, linear regression, polynomial regression, and symbolic regression.*

**regularization**

Technique to fight overfitting by modifying the loss function to penalize both error and representational space size. [L09](#)

**reinforcement learning**

The branch of machine learning concerned with generating behavior by interacting with an environment with the goal of maximizing reward given evaluative feedback. [L01](#) *Compare with inverse reinforcement learning.*

**ReLU (rectified linear unit)**

In neural networks, an activation function that returns its incoming activation, thresholded at zero to prevent negative activations. A key innovation in the development of deep neural networks. [L04](#)

**rule**

A function that takes an instance and produces an output, whether a Boolean, category, number, or vector. Used to refer to the output of a machine learning algorithm. [L01](#)

**semi-supervised learning**

A framework for machine learning that combines labeled and unlabeled data. [L01](#)

**seq2seq**

Neural network trained to produce output sequences from input sequences. Examples include language translation, continuation, and text summarization. [L17](#)

**sigmoid**

An S-shaped monotonically increasing activation function that returns a number near 1 if the input is positive and near zero if the input sums to a negative number, with a smooth transition between them around zero. [L04](#)  
*See also regression, logistic regression.*

**singular value decomposition**

A matrix decomposition approach that analyzes data and finds a lower-dimensional representation for it. Used in latent semantic analysis. [L12](#)  
*Closely related to autoencoding.*

**softmax**

An activation function for neural networks that accentuates the largest value in a vector and then normalizes the values to be similar to a discrete probability distribution. [L14](#)

**subnetwork**

A collection of nodes and links that can be repeated in a larger network. A convolutional filter in computer vision is an example. [L17](#)

**supervised learning**

The problem of learning an input-to-output mapping, or rule, from examples. [L01](#)

**support vector machine**

An approach to supervised learning that leverages similarity between instances to maximize the distance between the decision boundary and the nearest labeled example. [L09](#) *Compare with k-nearest neighbors.*

**symbolic regression**

Regression where the output rule is a mathematical expression. [L07](#)

**temporal difference learning**

A learning rule for sequential prediction tasks that attempts to minimize the difference between consecutive predictions. [L13](#) *See also backpropagation.*

**tensor**

An array of data that is a generalization of vectors and matrices. This course usually refers to a tensor as an array. [L18](#)

**transfer learning**

Any technique that leverages statistical insights gained from doing supervised learning on related problems. [L11](#)

**transformer network**

Neural network structure that transforms sequences of items, such as words, into other sequences of items, using attention. [L17](#)

**unsupervised learning**

The branch of machine learning concerned with learning the relationships between unlabeled input instances, often by clustering those instances by similarity or by finding a reduced dimensional representation of the instances. [L01](#) *Compare with backpropagation, reinforcement learning, and semi-supervised learning.*

**word embedding**

A mapping from words to vectors, usually selected so that words that appear in similar contexts are given similar vectors. [L10](#)

# SOFTWARE LIBRARIES

The Python software in this course builds on publicly available libraries, most of which are installed with Colab (aka Google Colaboratory; see [Lesson 02](#)). A few specialized libraries are not available directly through Colab, so you install them yourself when needed. The lion's share of the work is carried out by the Python Standard Library that includes the numerical computation package `numpy`. The course also makes extensive use of the machine learning library Scikit-learn and the deep learning library Keras.

A complete list follows of libraries referenced in the course, their purpose, and lessons where they appear. Lesson numbers followed by “aux” are code that was used behind the scenes in the lesson but wasn’t part of the main coding example. Lesson numbers followed by “qs” are used in the code for that lesson’s question.

## `csv`:

Parses data from comma-separated-value files. [L10](#)

## `diffprivlib.models`:

Learning algorithms augmented with differential privacy. [L24](#)

## `dowhy`:

Causal inference package from IBM. [L22](#)

## `dowhy.CausalModel`:

Builds a causal model. [L22](#)

## `encoder`:

Organizes data for network transmission. [L17](#)

## `functools.reduce`:

Summarizes a vector or list by a single value. [L11](#), [L21](#)

## `gplearn.genetic.SymbolicRegressor`:

Genetic programming approach to learning a symbolic expression. [L07](#)

## `graphviz`:

Graph visualization. [L03](#), [L06](#), [L08aux.ipynb](#), [L13](#)

## `gym`:

Reinforcement-learning test environments in OpenAI Gym. [L13](#)

**IPython:**

Libraries to support interactive functionality in Python.

**IPython.display:**

Audio playback. [L20](#)

**IPython.display.display:**

Displays images. [L14](#)

**json:**

Reads an encoded JSON object. [L17](#)

**keras:**

The Keras deep learning package from Google. [L15](#)

**keras.applications.vgg16:**

The trained VGG16 network for image recognition. [L14](#), [L18](#)

**keras.applications.vgg16.decode\_predictions:**

Turns VGG16 predictions into labels. [L14](#), [L15](#), [L18aux.ipynb](#)

**keras.applications.vgg16.preprocess\_input:**

Converts raw images into the form expected by VGG16. [L14](#), [L15](#),  
[L18aux.ipynb](#)

**keras.backend:**

Provides access to lower-level Keras functionality. [L14](#), [L18](#), [L19aux.ipynb](#), [L21](#)

**keras.datasets.fashion\_mnist:**

Fashion MNIST dataset. [L24](#)

**keras.initializers.Constant:**

Incorporates a constant set of values into a neural network. [L16](#)

**keras.layers.Activation:**

Sets the activation function for a layer. [L15](#), [L19](#)

**keras.layers.advanced\_activations.LeakyReLU:**

Enables leaky ReLU activation. [L19](#)

**keras.layers.BatchNormalization:**

Enables batch normalization on a layer. [L19](#)

**keras.layers.Conv1D:**

Creates a 1D convolution in Keras. [L16](#), [L20](#)

**keras.layers.Conv2D:**

Creates a 2D convolution in Keras. [L15](#)

**keras.layers.convolutional.Conv2D:**

Creates a 2D convolutional layer in Keras. [L19](#)

**keras.layers.convolutional.UpSampling2D:**

Creates the inverse of a 2D convolutional layer in Keras. [L19](#)

**keras.layers.Dense:**

Creates a fully connected layer in Keras. [L14](#), [L15](#), [L16](#), [L19](#), [L20](#), [L23](#)

**keras.layers.Dropout:**

Enables dropout normalization. [L15](#), [L19](#)

**keras.layers.Embedding:**

Incorporates an embedding layer. [L16](#)

**keras.layers.Flatten:**

Reorganizes array-shaped units into a flat vector. [L14](#), [L15](#), [L19](#), [L20](#)

**keras.layers.GlobalMaxPooling1D:**

Pooling layer for 1D convolutions. [L16](#)

**keras.layers.Input:**

Builds an input layer. [L16](#), [L19](#), [L20](#)

**keras.layers.MaxPooling1D:**

More local pooling layer for 1D convolutions. [L16](#), [L20](#)

**keras.layers.MaxPooling2D:**

Pooling layer for 2D convolutions. [L15](#)

**keras.layers.Reshape:**

Reshapes a layer. [L19](#)

**keras.layers.ZeroPadding2D:**

Pads an image array with zeros. [L19](#)

**keras.Model:**

Builds a neural network in Keras. *See also* **keras.models.Model**.

[L19aux.ipynb](#)

**keras.models.Model:**

Builds a neural network in Keras. [L14](#), [L16](#), [L19](#), [L20](#)

**keras.models.Sequential:**

Builds up layers of a neural network in Keras. [L15](#), [L19](#), [L23](#)

**keras.optimizers.Adam:**

The Adam optimizer, a popular method for finding weights of a neural network to minimize loss. [L19](#)

**keras.optimizers:**

Optimizers used in Keras. [L15](#), [L16](#)

**keras.preprocessing.image:**

Prepares raw images for processing by a neural network.

[L01aux.ipynb](#), [L05](#), [L14](#), [L15](#), [L18](#), [L19](#), [L24](#), [L15](#)

**keras.preprocessing.image.array\_to\_img:**

Converts an array into an image. [L11](#)

**keras.preprocessing.image.img\_to\_array:**

Converts an image into an array. [L18](#)

**keras.preprocessing.image.load\_img:**

Loads an image. [L18](#)

**keras.preprocessing.image.save\_img:**

Saves an image. [L18](#)

**keras.preprocessing.sequence.pad\_sequences:**

Adds padding information to data. [L16](#)

**keras.preprocessing.text.Tokenizer:**

Turns a sequence of strings into discrete tokens. [L16](#)

**keras.regularizers:**

Regularizers in Keras. [L23](#)

**keras.utils.to\_categorical:**

Turns a set of activations into a one-hot categorical selection. [L16](#)

**librosa:**

Sound and music. [L20](#)

**math:**

Mathematical functions. [L01qs](#), [L07aux.ipynb](#), [L08aux.ipynb](#),  
[L09aux.ipynb](#), [L11](#), [L25](#)

**matplotlib.pyplot:**

Plots graphs. [L03qs](#), [L04](#), [L06qs](#), [L07](#), [L08](#), [L09](#), [L10qs](#), [L12](#), [L19](#), [L20](#),  
[L23](#), [L25](#)

**model:**

Reads a pretrained neural network model. [L17](#)

**numpy:**

Mathematical functions over general arrays. [L01aux.ipynb](#), [L04](#),  
[L05qs](#), [L05qs](#), [L07](#), [L08aux.ipynb](#), [L08qs](#), [L10qs](#), [L11](#), [L14](#), [L15](#), [L16](#),  
[L17](#), [L18](#), [L19](#), [L20](#), [L21](#), [L23](#), [L24](#)

**os:**

Provides operating system access for manipulating files and directories.  
[L17](#), [L19](#), [L20](#), [L22](#)

**pandas:**

Library for organizing datasets. [L08aux.ipynb](#), [L22](#)

**PIL:**

Python image library.

**PIL.Image:**

Loads and converts images. [L14](#), [L15](#)

**random:**

Generates random numbers. [L03qs](#), [L04](#), [L06qs](#), [L07aux.ipynb](#),  
[L08aux.ipynb](#), [L09](#), [L10](#), [L11](#), [L12](#), [L15](#), [L18aux.ipynb](#), [L25](#)

**sample:**

Makes choices randomly, given a probability distribution. [L17](#)

**scipy.optimize.fmin\_l\_bfgs\_b:**

Specific optimizer available through SciPy. [L18](#)

**scipy.stats:**

Computes vector statistics like the mode of a list. [L11](#)

**seaborn:**

Data visualization. [L08aux.ipynb](#), [L08qs](#), [L21](#)

**sklearn:**

Scikit-learn library, with implementations of many significant pre-deep-learning machine learning algorithms. Built on top of [numpy](#),  
[scipy](#), and [matplotlib](#).

**sklearn.cluster.AgglomerativeClustering:**

Scikit-learn's tree-based hierarchical clustering algorithms. [L11qs](#)

**sklearn.cluster.KMeans:**

Scikit-learn's  $k$ -means clustering algorithms. [L11qs](#)

**sklearn.datasets.fetch\_20newsgroups:**

The 20 newsgroups dataset. [L16](#)

**sklearn.datasets.fetch\_openml:**

Provides access to OpenML datasets. **L04, L06qs, L10qs, L11**

**sklearn.linear\_model.LogisticRegression:**

Scikit-learn's logistic regression algorithms. **L10**

**sklearn.metrics.confusion\_matrix:**

Computes a confusion matrix. **L06**

**sklearn.model\_selection.train\_test\_split:**

Splits a dataset randomly into training and testing sets. **L04, L06qs, L08aux.ipynb, L10qs, L11, L15, L16, L18aux.ipynb, L19aux.ipynb, L20**

**sklearn.naive\_bayes.GaussianNB:**

Naive Bayes learner for real-valued features. **L24**

**sklearn.naive\_bayes.MultinomialNB:**

Naive Bayes learner for binary features. **L06, L10qs, L12**

**sklearn.neighbors.KNeighborsClassifier:**

K-nearest neighbor classification algorithm. **L11qs**

**sklearn.neighbors:**

K-nearest neighbor algorithms. **L08, L09qs**

**sklearn.neural\_network.MLPClassifier:**

Scikit-learn's neural network algorithm (multilayer perceptron). **L04, L06qs, L10**

**sklearn.svm:**

Scikit-learn's support vector machine. **L09**

**sklearn.tree:**

Scikit-learn's decision tree algorithms. **L03, L04, L06, L08aux.ipynb, L09aux.ipynb, L10qs, L13**

**sklearn.utils.check\_random\_state:**

Repeatable, random way to split training and testing data. **L04, L06, L10qs**

**statistics:**

Algorithms for statistics like computing the median. **L07aux.ipynb**

**sys:**

Operating system support. **L22**

**tensorflow:**

A deep learning library from Google. Supports differentiable programming, where derivatives of code are computed directly, making it simpler to implement meta-learning. [L17](#)

**torch:**

PyTorch deep neural network library from Facebook. [L25](#)

**torch.nn.functional:**

Building neural networks in PyTorch. [L25](#)

**warnings:**

Set whether or not to display warning messages. [L17](#), [L20](#)

# IMAGE CREDITS

| Page | Credits                                      |
|------|----------------------------------------------|
| 311  | taviphoto/iStock/Getty Images.               |
| 311  | bergamont/Getty Images.                      |
| 311  | wuestenigel/flickr/CC BY 2.0.                |
| 311  | stevepb/Pixabay.com.                         |
| 311  | pagerniki/Pixabay.com.                       |
| 315  | Digital Vision/Getty Images.                 |
| 315  | StockSanta/Getty Images.                     |
| 335  | lucidrai/ns/stylegan2-pytorchGitHub/GPL 3.0. |