THE GREAT COURSES®

Topic
Science

Subtopic
Engineering & Technology

Introduction to Machine Learning

Course Guidebook

Michael L. Littman, PhD
Brown University

THE GREAT COURSES®

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

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

LESSON GUIDES

SUPPLEMENTARY MATERIALS

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.

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/mlittmancs/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.

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.

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.

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.

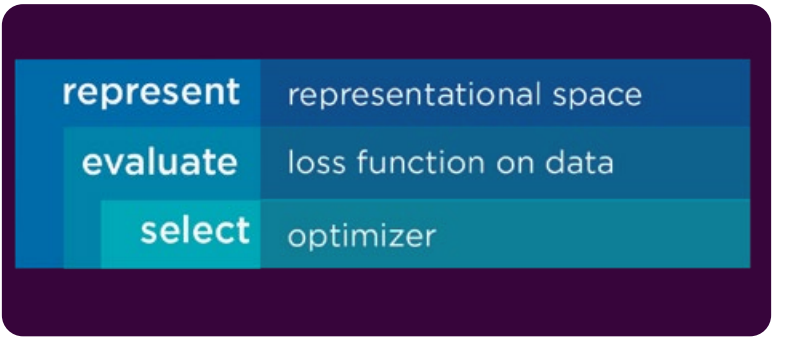
Once the loss function assigns scores to evaluate each candidate rule, an **optimizer** selects a rule with the best-available score from the space of candidate rules, given our dataset.

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 182

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/mlittmancs/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

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

Answers on page 182

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.

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.

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.

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 183

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.

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.

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.

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 183

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.

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

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

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 some what 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 184

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**.

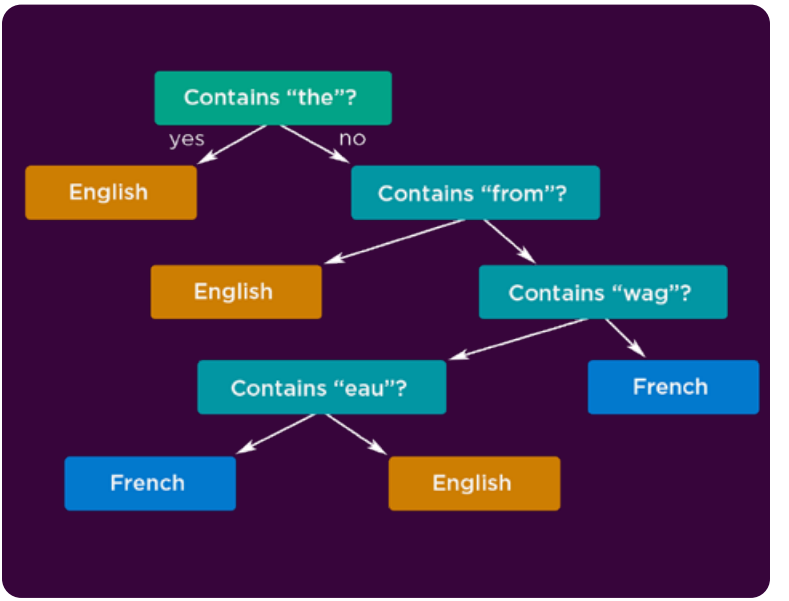
Text Classification

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."

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.

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* 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

1. When would it make sense to use a generative Bayesian approach instead of a discriminative neural network or decision tree approach?

2. 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?

3. 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 184

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 185

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 185

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.

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.

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.

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 186

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.

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.

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.

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 186

LESSON 11

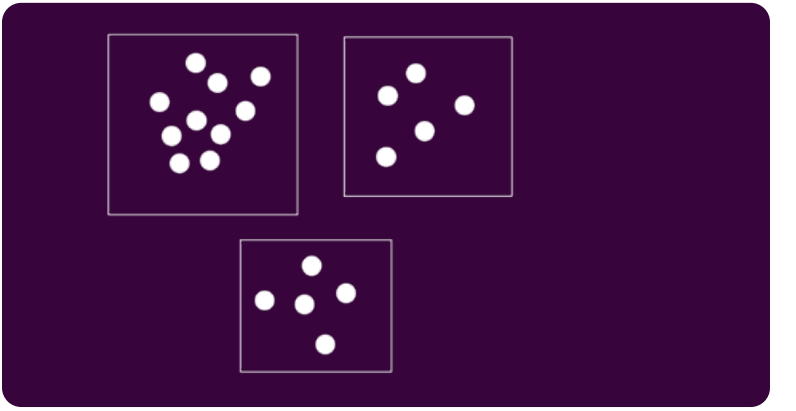
CLUSTERING AND SEMI-SUPERVISED LEARNING

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

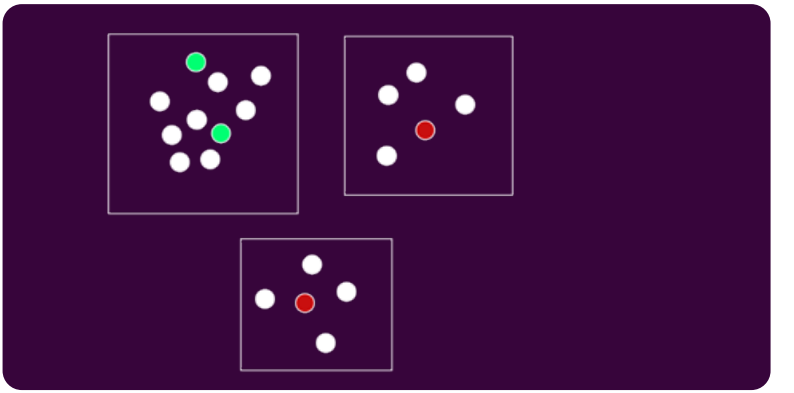
Imagine we have a set of data points in two dimensions. The data is unsupervised; there are no labels.

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.

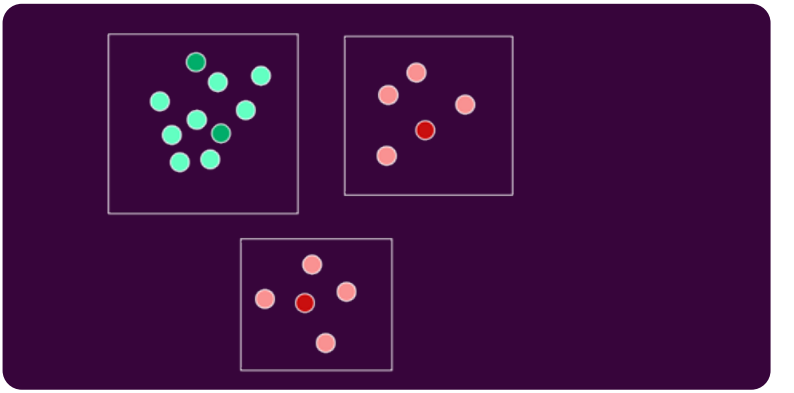


We don't know what labels the points have, but we might guess that, within each of these clusters, points share the same label. That's reminiscent of what happens in a nearest neighbor classifier or a support vector machine: Unlabeled instances get their labels from nearby labeled instances.

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.

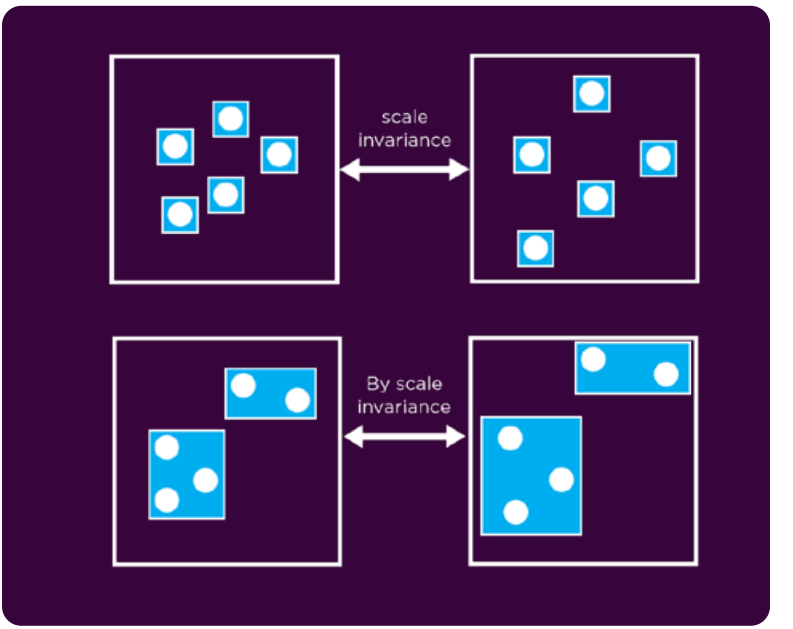
In this approach to semi-supervised learning, the supervised part is a very easy computation: To apply the supervision, the approach just diffuses labels to other instances in the same cluster. The clustering part is where the machine learning action is.

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.

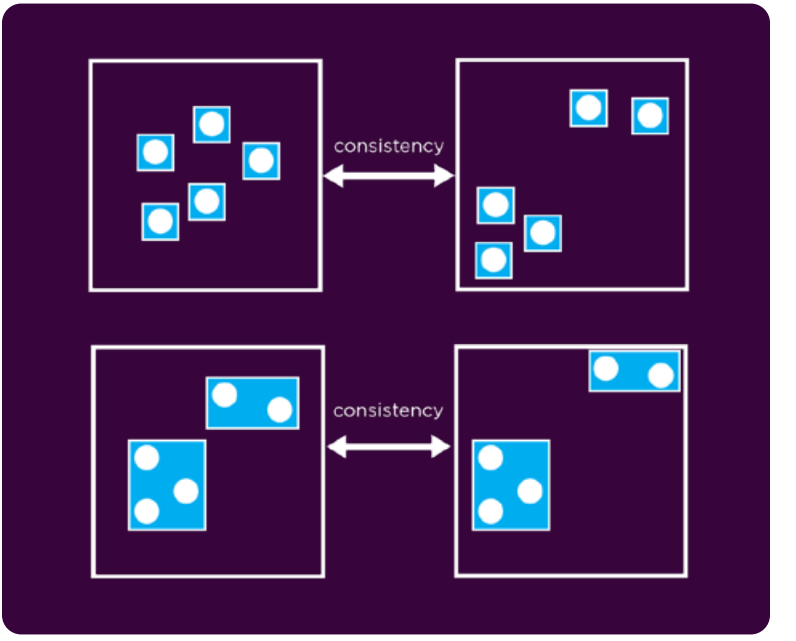
Returning to our simple example, the clusters that we see in a picture remain the same even if we zoom in or out. An algorithm that violates this rule would, for example, merge clusters as we zoom out.



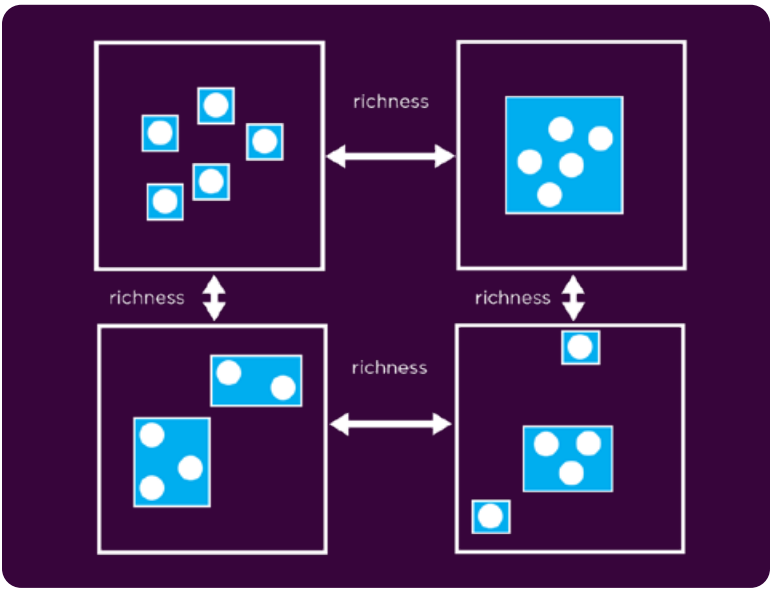
This property of clusters remaining the same before and after a zoom operation is called scale invariance, according to Kleinberg. It's the first of three properties that make for a "good" clustering algorithm.

Whenever it's cheap to gather large amounts of unlabeled data but obtaining labels is expensive, semi-supervised learning applies.

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.



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 187

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.

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.

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.

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 187

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.

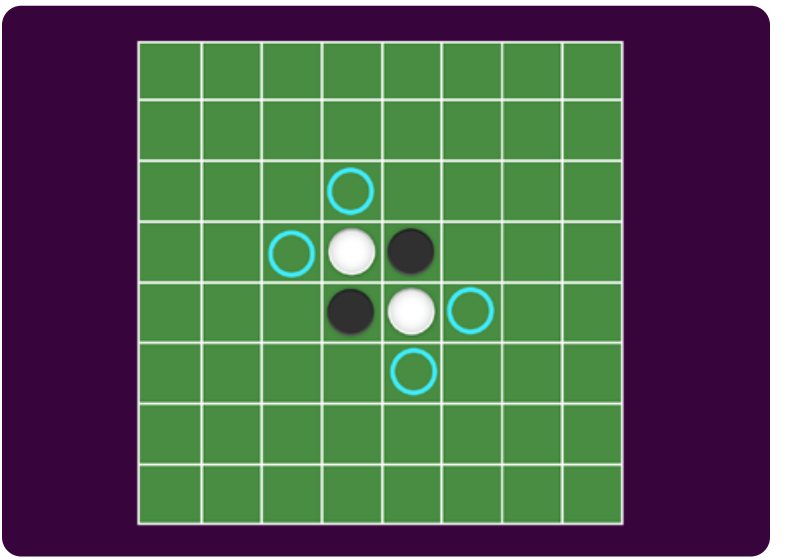
Othello

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.

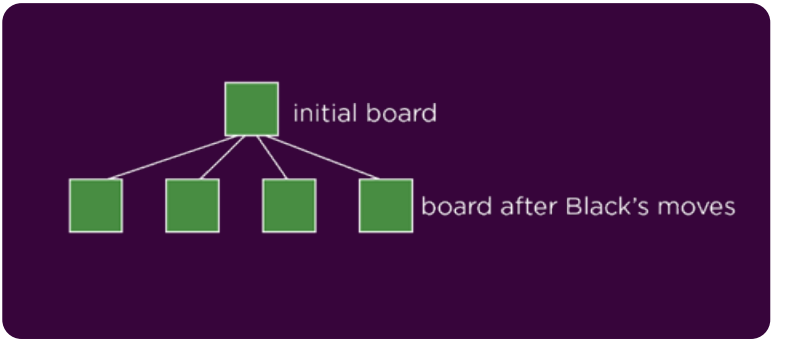
We can visualize game play in Othello and many similar games in the form of a tree. Trees come up all over the place in computer science, like in decision trees.

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.

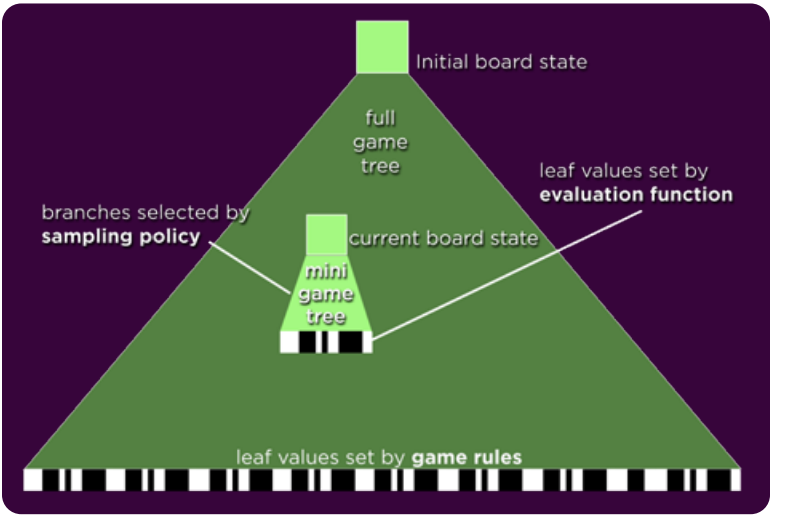
But despair not. There is a general recipe for playing games like Othello that has turned out to be super effective. Each time it's our turn, we're going to run an algorithm for selecting a good move from the current board.

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 188

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 188

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.

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.

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.

* 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 188

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.

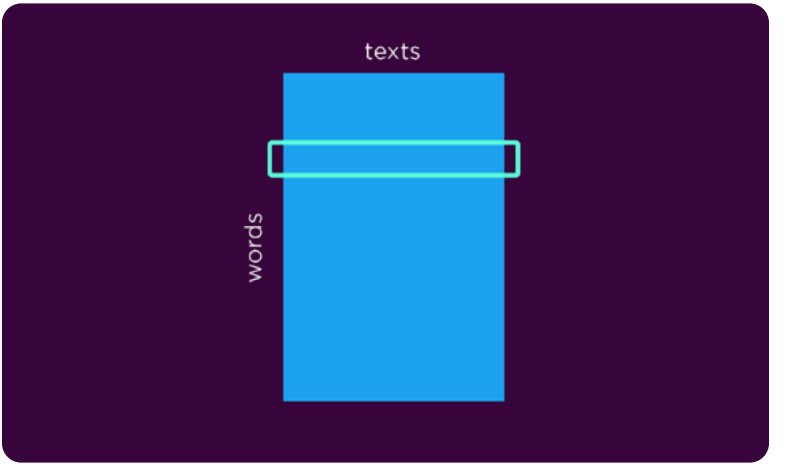
So if we can represent words as numeric vectors, that might provide a foundation for using numbers to do various kinds of language-processing tasks. More specifically, we'll want vectors to be similar to the same degree that the words being represented have similar usages.

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.

Embeddings can pick up on subtle patterns of word usage, detecting synonyms and even solving analogy problems.

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.

* 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:

$$s(w,A,B) = \frac{mean_{a\in A}\cos(\vec{w},\vec{a}) - mean_{b\in B}\cos(\vec{w},\vec{b})}{std\text{-}dev_{x\in A\cup B}\cos(\vec{w},\vec{x})}$$

Use the GloVe embeddings loaded in for this lesson to replicate this analysis. Do you get similar results?

Answers on page 189

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 190

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.

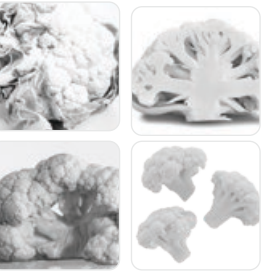
In the second approach, we use a pretrained classifier to recognize the type of image that is being produced. This approach is more like what people typically do: We recognize the general category of pigs, of which this photo is an example. So our discriminator in the second approach prefers images that 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.

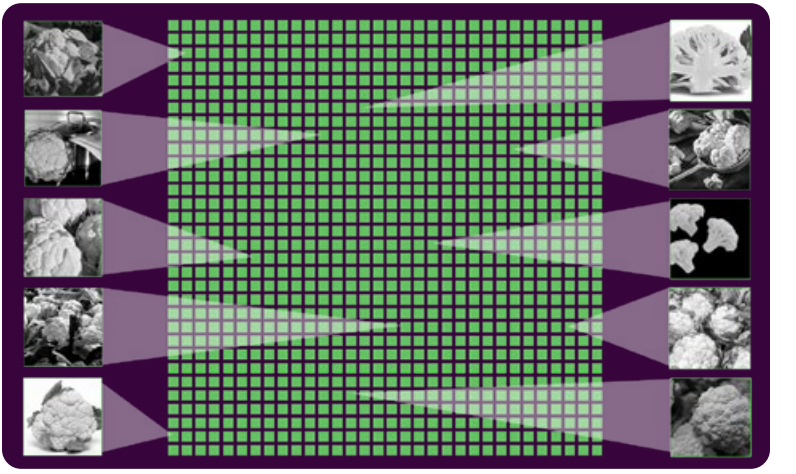
Then, it compares the image constructed out of cauliflower to the pig image. The comparison could be any of the distance or similarity metrics from **Lesson 08** about the nearest neighbor methods.

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.

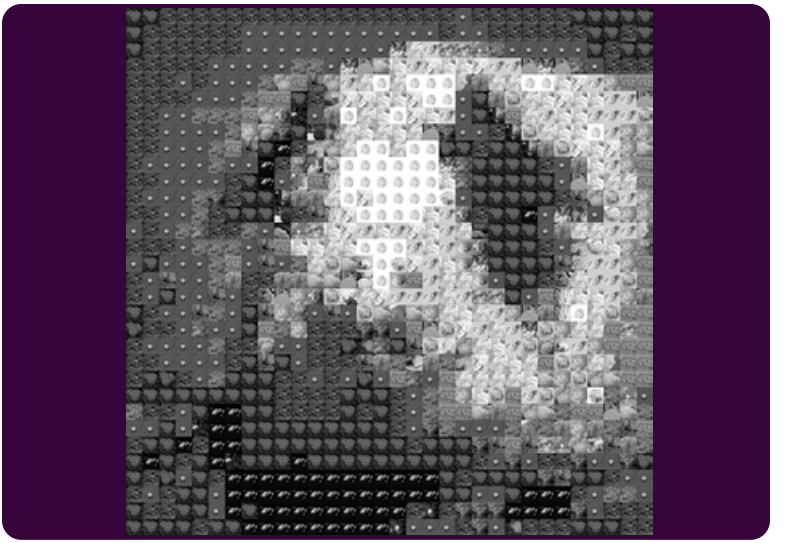
The set of possible images the generator can produce is 500 cauliflower choices raised to the power of 1,024 for the 1,024 pictures in the grid. That makes a number with roughly 2,700 zeros. We'd like to find the generated image with the best score, but we will not have time to find it this way.

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.

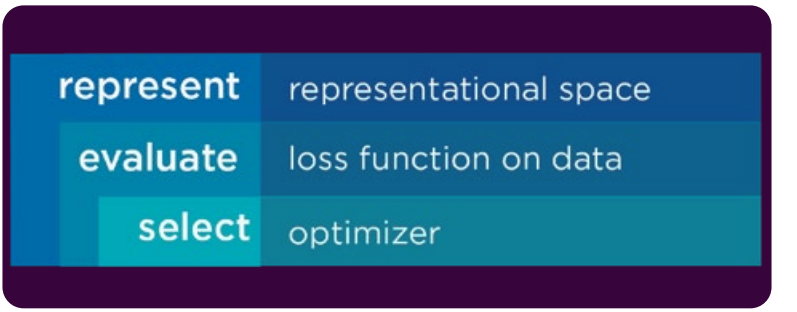
The code for solving this optimization problem runs quickly. Instead of checking all 500^{1024} grids, it just checks each of 500 images 1,024 times.

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

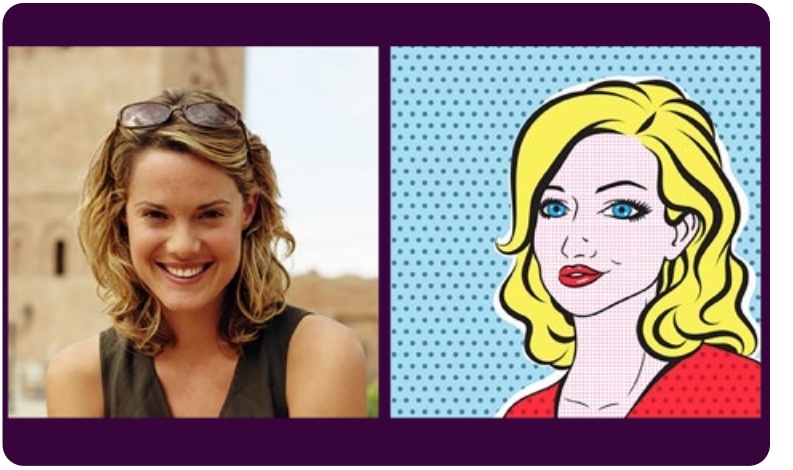
Instead of a pixel-based discriminator that prefers images that match a specific target image pixel by pixel, we can consider a discriminator that employs an image classifier. A classifier discriminator, as the name suggests, prefers images it recognizes as being in a specified class.

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.



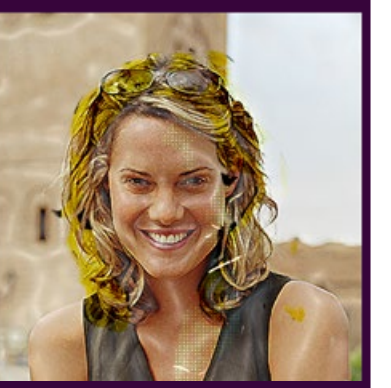
We'd like an image of the same face, only with some of the features that make Lichtenstein's style so recognizable. Maybe the hair could be made more solid, with more use of strong primary colors. Maybe the face and other skin could be rendered with dots.

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 190

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.

What makes GANs so powerful and revolutionary is that they dynamically construct their own discriminators as part of the training process.

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.

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.

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

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.

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 191

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.

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.

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.

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.

Hidden Markov models capture the probabilities of sequences.

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.

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 page 191

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.

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.

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.

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 191

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.

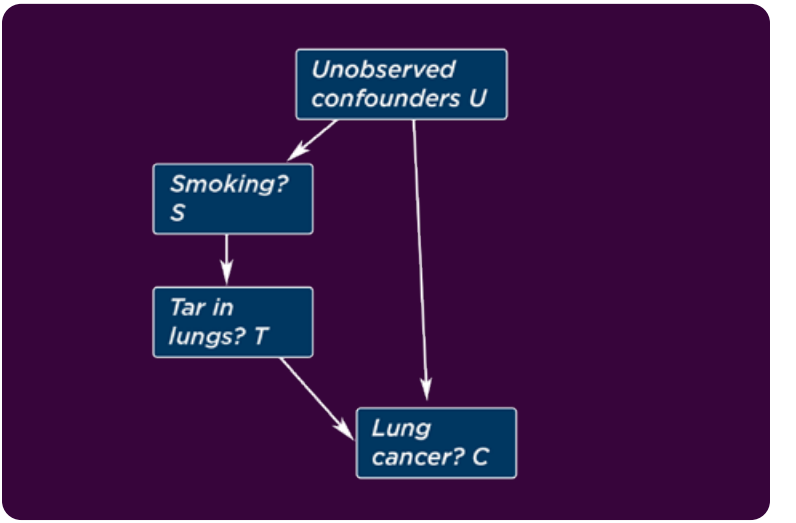
The causal graph includes variables that could be relevant: Does a given person smoke (S)? Does the person develop lung cancer (C)? On an x-ray, can you see tar in the person's lungs* (T)?

* 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.

$$P(C|do(S)) = \sum_{UT} P(U)\, P(T|S)\, P(C|T,U)$$

$$\begin{aligned}
&\sum_{S'T} P(C|S',T)\, P(Z|S)\, P(S') \\
&= \sum_{U} \sum_{S'T} P(U)\, P(C|S',T,U)\, P(T|S,U)\, P(S'|U) \\
&= \sum_{U} \sum_{S'T} P(U)\, P(C|T,U)\, P(T|S)\, P(S'|U) \\
&= \sum_{U} \sum_{T} P(U)\, P(C|T,U)\, P(T|S) \sum_{S'} P(S'|U) \\
&= \sum_{U} \sum_{T} P(U)\, P(C|T,U)\, P(T|S)
\end{aligned}$$

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 192

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.

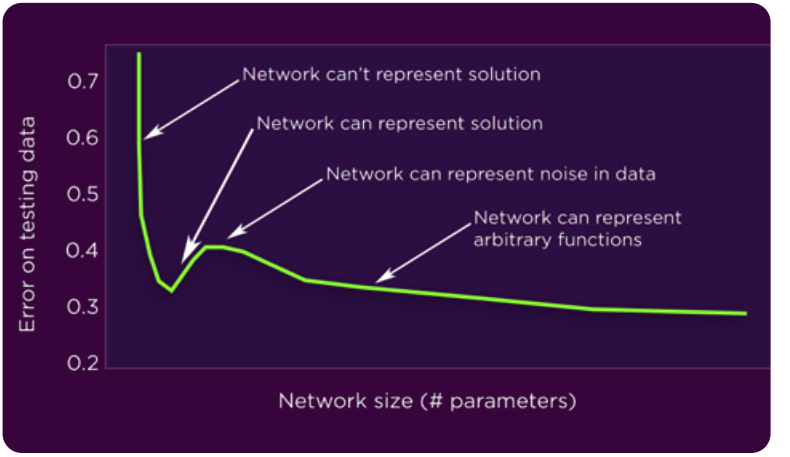
The Double Descent Phenomenon

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.

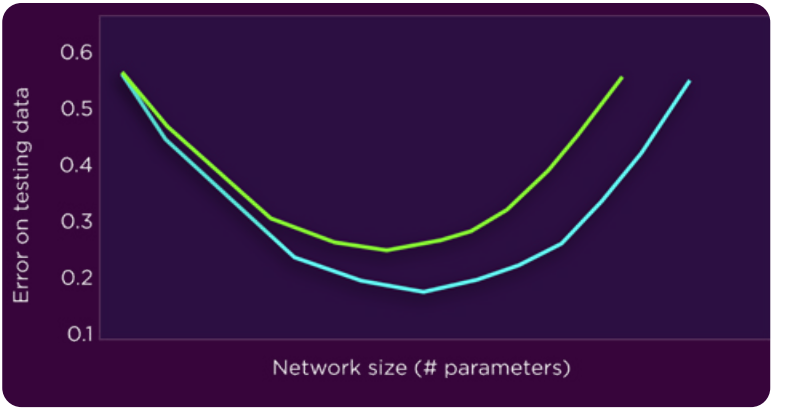
In the second descent, the network captures the solution and interpolates around the noise. Now the network is like a dog in an open yard that's had many, many times to chase cars. It is briefly distracted by cars, but it comes back to the yard right away.

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.



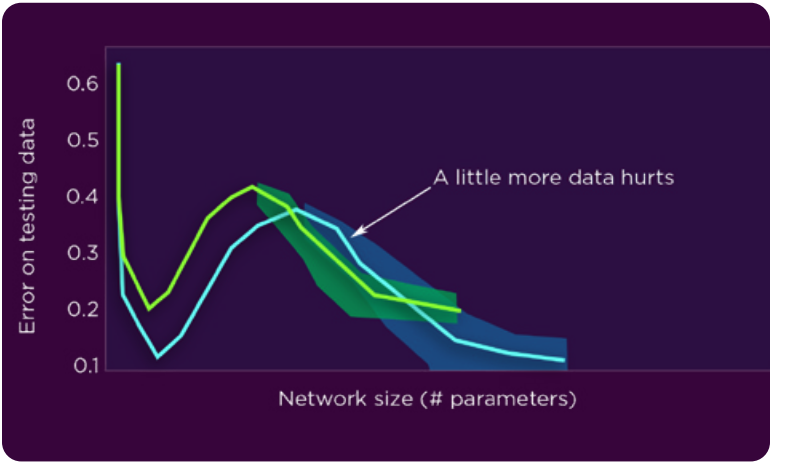
But consider what happens in the new over-parameterized double-descent deep learning regime. Because the curve has a hump, stretching it down and to the right does not necessarily result in error improvements at all network sizes on the curve. Here, there's a range of network sizes where additional data 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 192

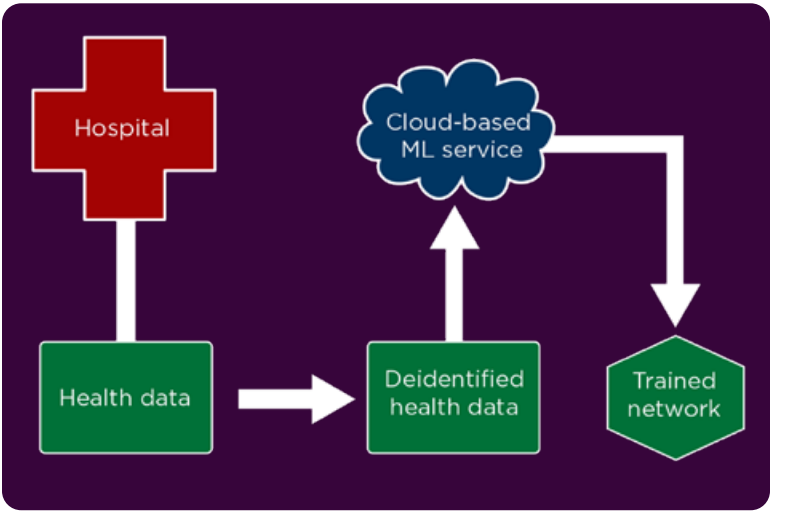
LESSON 24

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.

Every phase of the machine learning pipeline—from storage, to deidentification, to model deployment—has privacy threats.

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.*

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 193

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 193

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.

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.

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.

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 23

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 26

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 31

LESSON 05

1.
```
import numpy as np

# p = an image array

l = np.reshape(p,(-1,3))

sum(l[:,1] > l[:,0])
```

2. 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.

3. 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 35

LESSON 06

1. 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.

2. The parameters are chosen to maximize the likelihood of the observed data, so the negative log likelihood is the loss function it minimizes.

3. 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 41

LESSON 07

1. 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.

2. 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.

3. 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 46

LESSON 08

1. The number of nearby neighbors grows linearly with the dimension, but the number of non-neighbors grows exponentially—much faster.

2. 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.

3. 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 51

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 56

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 63

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 71

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 78

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 85

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 90

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.

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 96

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 103

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 111

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 121

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 127

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 135

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 142

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 149

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.

3. 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 158

LESSON 24

1. 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.
2. Differentially private.
3. In this case, there was no decrease in accuracy and no measurable increase in running time.

   Code: L24qs.ipynb

The questions on page 166

LESSON 25

1. 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.
2. Meta-learning!
3. 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 175

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
113	taviphoto/iStock/Getty Images.
113	bergamont/Getty Images.
113	wuestenigel/flickr/CC BY 2.0.
113	stevepb/Pixabay.com.
113	pagerniki/Pixabay.com.
117	Digital Vision/Getty Images.
117	StockSanta/Getty Images.
125	lucidrai/ns/stylegan2-pytorchGitHub/GPL 3.0.