Praise for *Deep Learning for Coders with fastai and PyTorch*

If you are looking for a guide that starts at the ground floor and takes you to the cutting edge of research, this is the book for you. Don't let those PhDs have all the fun—you too can use deep learning to solve practical problems.

Hal Varian, Emeritus Professor, UC Berkeley; Chief Economist, Google

As artificial intelligence has moved into the era of deep learning, it behooves all of us to learn as much as possible about how it works. Deep Learning for Coders provides a terrific way to initiate that, even for the uninitiated, achieving the feat of simplifying what most of us would consider highly complex.

Eric Topol, Author, Deep Medicine; Professor, Scripps Research

Jeremy and Sylvain take you on an interactive—in the most literal sense as each line of code can be run in a notebook—journey through the loss valleys and performance peaks of deep learning. Peppered with thoughtful anecdotes and practical intuitions from years of developing and teaching machine learning, the book strikes the rare balance of communicating deeply technical concepts in a conversational and light-hearted way. In a faithful translation of fast.ai's award-winning online teaching philosophy, the book provides you with state-of-the-art practical tools and the real-world examples to put them to use. Whether you're a beginner or a veteran, this book will fast-track your deep learning journey and take you to new heights—and depths.

Sebastian Ruder, Research Scientist, Deepmind

Jeremy Howard and Sylvain Gugger have authored a bravura of a book that successfully bridges the AI domain with the rest of the world. This work is a singularly substantive and insightful yet absolutely relatable primer on deep learning for anyone who is interested in this domain: a lodestar book amongst many in this genre.

Anthony Chang, Chief Intelligence and Innovation Officer, Children's Hospital of Orange County

How can I "get" deep learning without getting bogged down? How can I quickly learn the concepts, craft, and tricks-of-the-trade using examples and code? Right here. Don't miss the new locus classicus for hands-on deep learning.

Oren Etzioni, Professor, University of Washington; CEO, Allen Institute for AI

This book is a rare gem—the product of carefully crafted and highly effective teaching, iterated and refined over several years resulting in thousands of happy students. I'm one of them. fast.ai changed my life in a wonderful way, and I'm convinced that they can do the same for you.

Jason Antic, Creator, DeOldify

Deep Learning for Coders with fastai and PyTorch

by Jeremy Howard and Sylvain Gugger

Printed in Canada.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jonathan Hassell

Development Editor: Melissa Potter

Production Editor: Christopher Faucher

Copyeditor: Rachel Head

Proofreader: Sharon Wilkey

Indexer: Sue Klefstad

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

July 2020: First Edition

Revision History for the First Edition

- 2020-06-29: First Release
- 2020-09-18: Second Release
- 2020-12-18: Third Release

- 2021-05-07: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492045526 for release details.

978-1-492-04552-6

[TI]

Preface

Deep learning is a powerful new technology, and we believe it should be applied across many disciplines. Domain experts are the most likely to find new applications of it, and we need more people from all backgrounds to get involved and start using it.

That's why Jeremy cofounded fast.ai, to make deep learning easier to use through free online courses and software. Sylvain is a research engineer at Hugging Face. Previously he was a research scientist at fast.ai and a former mathematics and computer science teacher in a program that prepares students for entry into France's elite universities. Together, we wrote this book in the hope of putting deep learning into the hands of as many people as possible.

Who This Book Is For

If you are a complete beginner to deep learning and machine learning, you are most welcome here. Our only expectation is that you already know how to code, preferably in Python.

No Experience? No Problem!

If you don't have any experience coding, that's OK too! The first three chapters have been explicitly written in a way that will allow executives, product managers, etc. to understand the most important things they'll need to know about deep learning. When you see bits of code in the text, try to look them over to get an intuitive sense of what they're doing. We'll explain them line by line. The details of the syntax are not nearly as important as a high-level understanding of what's going on.

If you are already a confident deep learning practitioner, you will also find a lot here. In this book, we will be showing you how to achieve world-class results, including techniques from the latest research. As we will show, this doesn't require advanced mathematical training or years of study. It just requires a bit of common sense and tenacity.

What You Need to Know

As we said before, the only prerequisite is that you know how to code (a year of experience is enough), preferably in Python, and that you have at least followed a high school math course. It doesn't matter if you remember little of it right now; we will brush up on it as needed. Khan Academy has great free resources online that can help.

We are not saying that deep learning doesn't use math beyond high school level, but we will teach you (or direct you to resources that will teach you) the basics you need as we cover the subjects that require them.

The book starts with the big picture and progressively digs beneath the surface, so you may need, from time to time, to put it aside and go learn some additional topic (a way

of coding something or a bit of math). That is completely OK, and it's the way we intend the book to be read. Start browsing it, and consult additional resources only as needed.

Please note that Kindle or other ereader users may need to double-click images to view the full-sized versions.

Online Resources

All the code examples shown in this book are available online in the form of Jupyter notebooks (don't worry; you will learn all about what Jupyter is in Chapter 1). This is an interactive version of the book, where you can actually execute the code and experiment with it. See the book's website for more information. The website also contains up-to-date information on setting up the various tools we present and some additional bonus chapters.

What You Will Learn

After reading this book, you will know the following:

- How to train models that achieve state-of-the-art results in
    - Computer vision, including image classification (e.g., classifying pet photos by breed) and image localization and detection (e.g., finding the animals in an image)
    - Natural language processing (NLP), including document classification (e.g., movie review sentiment analysis) and language modeling
    - Tabular data (e.g., sales prediction) with categorical data, continuous data, and mixed data, including time series
    - Collaborative filtering (e.g., movie recommendation)
- How to turn your models into web applications
- Why and how deep learning models work, and how to use that knowledge to improve the accuracy, speed, and reliability of your models
- The latest deep learning techniques that really matter in practice
- How to read a deep learning research paper
- How to implement deep learning algorithms from scratch
- How to think about the ethical implications of your work, to help ensure that you're making the world a better place and that your work isn't misused for harm

See the table of contents for a complete list, but to give you a taste, here are some of the techniques covered (don't worry if none of these words mean anything to you yet—you'll learn them all soon):

- Affine functions and nonlinearities
- Parameters and activations
- Random initialization and transfer learning
- SGD, Momentum, Adam, and other optimizers
- Convolutions
- Batch normalization
- Dropout
- Data augmentation
- Weight decay
- ResNet and DenseNet architectures
- Image classification and regression
- Embeddings
- Recurrent neural networks (RNNs)
- Segmentation
- U-Net
- And much more!

Chapter Questionnaires

If you look at the end of each chapter, you'll find a questionnaire. That's a great place to see what we cover in each chapter, since (we hope!) by the end of each one, you'll be able to answer all the questions there. In fact, one of our reviewers (thanks, Fred!) said that he likes to read the questionnaire *first*, before reading the chapter, so he knows what to look out for.

O'Reilly Online Learning

Note

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/deep-learning-for-coders*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Foreword

In a very short time, deep learning has become a widely useful technique, solving and automating problems in computer vision, robotics, healthcare, physics, biology, and beyond. One of the delightful things about deep learning is its relative simplicity. Powerful deep learning software has been built to make getting started fast and easy. In a few weeks, you can understand the basics and get comfortable with the techniques.

This opens up a world of creativity. You start applying it to problems that have data at hand, and you feel wonderful seeing a machine solving problems for you. However, you slowly feel yourself getting closer to a giant barrier. You built a deep learning model, but it doesn't work as well as you had hoped. This is when you enter the next stage, finding and reading state-of-the-art research on deep learning.

However, there's a voluminous body of knowledge on deep learning, with three decades of theory, techniques, and tooling behind it. As you read through some of this research, you realize that humans can explain simple things in really complicated ways. Scientists use words and mathematical notation in these papers that appear foreign, and no textbook or blog post seems to cover the necessary background that you need in accessible ways. Engineers and programmers assume you know how GPUs work and have knowledge about obscure tools.

This is when you wish you had a mentor or a friend that you could talk to. Someone who was in your shoes before, who knows the tooling and the math—someone who could guide you through the best research, state-of-the-art techniques, and advanced engineering, and make it comically simple. I was in your shoes a decade ago, when I was breaking into the field of machine learning. For years, I struggled to understand papers that had a little bit of math in them. I had good mentors around me, which helped me greatly, but it took me many years to get comfortable with machine learning and deep learning. That motivated me to coauthor PyTorch, a software framework to make deep learning accessible.

Jeremy Howard and Sylvain Gugger were also in your shoes. They wanted to learn and apply deep learning, without any previous formal training as ML scientists or engineers. Like me, Jeremy and Sylvain learned gradually over the years and eventually became experts and leaders. But unlike me, Jeremy and Sylvain selflessly put a huge amount of energy into making sure others don't have to take the painful path that they took. They built a great course called fast.ai that makes cutting-edge deep learning techniques accessible to people who know basic programming. It has graduated hundreds of thousands of eager learners who have become great practitioners.

In this book, which is another tireless product, Jeremy and Sylvain have constructed a magical journey through deep learning. They use simple words and introduce every concept. They bring cutting-edge deep learning and state-of-the-art research to you, yet make it very accessible.

You are taken through the latest advances in computer vision, dive into natural language processing, and learn some foundational math in a 500-page delightful ride. And the ride doesn't stop at fun, as they take you through shipping your ideas to production. You can treat the fast.ai community, thousands of practitioners online, as your extended family, where individuals like you are available to talk and ideate small and big solutions, whatever the problem may be.

I am very glad you've found this book, and I hope it inspires you to put deep learning to good use, regardless of the nature of the problem.

Soumith Chintala
Cocreator of PyTorch

Part I. Deep Learning in Practice

Chapter 1. Your Deep Learning Journey

Hello, and thank you for letting us join you on your deep learning journey, however far along that you may be! In this chapter, we will tell you a little bit more about what to expect in this book, introduce the key concepts behind deep learning, and train our first models on different tasks. It doesn't matter if you don't come from a technical or a mathematical background (though it's OK if you do too!); we wrote this book to make deep learning accessible to as many people as possible.

Deep Learning Is for Everyone

A lot of people assume that you need all kinds of hard-to-find stuff to get great results with deep learning, but as you'll see in this book, those people are wrong. Table 1-1 lists a few things you *absolutely don't need* for world-class deep learning.

Table 1-1. What you don't need for deep learning

Myth (don't need)	Truth
Lots of math	High school math is sufficient.
Lots of data	We've seen record-breaking results with <50 items of data.
Lots of expensive computers	You can get what you need for state-of-the-art work for free.

Deep learning is a computer technique to extract and transform data—with use cases ranging from human speech recognition to animal imagery classification—by using multiple layers of neural networks. Each of these layers takes its inputs from previous layers and progressively refines them. The layers are trained by algorithms that minimize their errors and improve their accuracy. In this way, the network learns to perform a specified task. We will discuss training algorithms in detail in the next section.

Deep learning has power, flexibility, and simplicity. That's why we believe it should be applied across many disciplines. These include the social and physical sciences, the arts, medicine, finance, scientific research, and many more. To give a personal example, despite having no background in medicine, Jeremy started Enlitic, a company that uses deep learning algorithms to diagnose illness and disease. Within months of starting the company, it was announced that its algorithm could identify malignant tumors more accurately than radiologists.

Here's a list of some of the thousands of tasks in different areas for which deep learning, or methods heavily using deep learning, is now the best in the world:

Natural language processing (NLP)

Answering questions; speech recognition; summarizing documents; classifying documents; finding names, dates, etc. in documents; searching for articles mentioning a concept

Computer vision

Satellite and drone imagery interpretation (e.g., for disaster resilience), face recognition, image captioning, reading traffic signs, locating pedestrians and vehicles in autonomous vehicles

Medicine

Finding anomalies in radiology images, including CT, MRI, and X-ray images; counting features in pathology slides; measuring features in ultrasounds; diagnosing diabetic retinopathy

Biology

Folding proteins; classifying proteins; many genomics tasks, such as tumor-normal sequencing and classifying clinically actionable genetic mutations; cell classification; analyzing protein/protein interactions

Image generation

Colorizing images, increasing image resolution, removing noise from images, converting images to art in the style of famous artists

Recommendation systems

Web search, product recommendations, home page layout

Playing games

Chess, Go, most Atari video games, and many real-time strategy games

Robotics

Handling objects that are challenging to locate (e.g., transparent, shiny, lacking texture) or hard to pick up

Other applications

Financial and logistical forecasting, text to speech, and much, much more...

What is remarkable is that deep learning has such varied applications, yet nearly all of deep learning is based on a single innovative type of model: the neural network.

But neural networks are not, in fact, completely new. In order to have a wider perspective on the field, it is worth starting with a bit of history.

Neural Networks: A Brief History

In 1943 Warren McCulloch, a neurophysiologist, and Walter Pitts, a logician, teamed up to develop a mathematical model of an artificial neuron. In their paper "A Logical Calculus of the Ideas Immanent in Nervous Activity," they declared the following:

> *Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms.*

McCulloch and Pitts realized that a simplified model of a real neuron could be represented using simple addition and thresholding, as shown in Figure 1-1. Pitts was self-taught, and by age 12, had received an offer to study at Cambridge University with the great Bertrand Russell. He did not take up this invitation, and indeed throughout his life did not accept any offers of advanced degrees or positions of authority. Most of his famous work was done while he was homeless. Despite his lack of an officially recognized position and increasing social isolation, his work with McCulloch was influential and was taken up by a psychologist named Frank Rosenblatt.

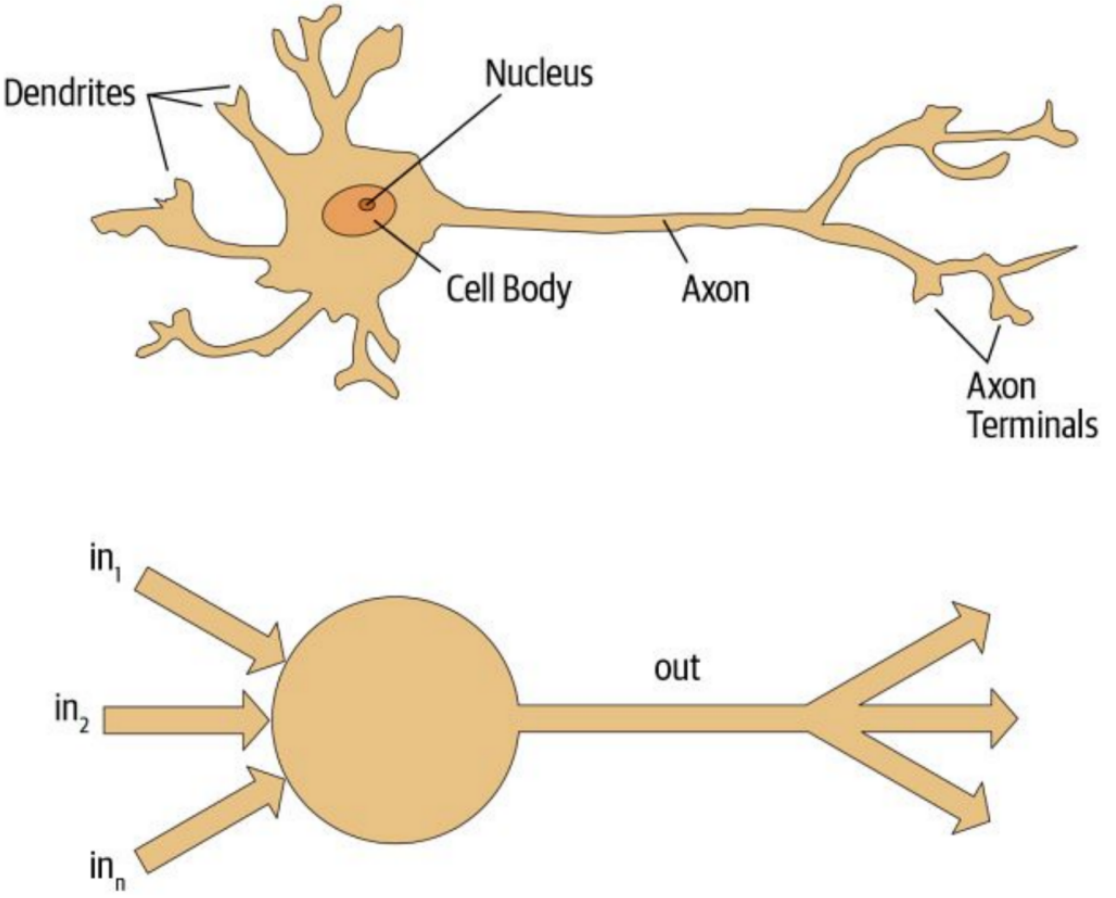


Figure 1-1. Natural and artificial neurons

Rosenblatt further developed the artificial neuron to give it the ability to learn. Even more importantly, he worked on building the first device that used these principles, the Mark I Perceptron. In "The Design of an Intelligent Automaton," Rosenblatt wrote about this work: "We are now about to witness the birth of such a machine—a machine capable of perceiving, recognizing and identifying its surroundings without any human training or control." The perceptron was built and was able to successfully recognize simple shapes.

An MIT professor named Marvin Minsky (who was a grade behind Rosenblatt at the same high school!), along with Seymour Papert, wrote a book called *Perceptrons* (MIT Press) about Rosenblatt's invention. They showed that a single layer of these devices was unable to learn some simple but critical mathematical functions (such as XOR). In the same book, they also showed that using multiple layers of the devices would allow these limitations to be addressed. Unfortunately, only the first of these insights was widely recognized. As a result, the global academic community nearly entirely gave up on neural networks for the next two decades.

Perhaps the most pivotal work in neural networks in the last 50 years was the multi-volume *Parallel Distributed Processing* (PDP) by David Rumelhart, James McClelland, and the PDP Research Group, released in 1986 by MIT Press. Chapter 1 lays out a similar hope to that shown by Rosenblatt:

> *People are smarter than today's computers because the brain employs a basic computational architecture that is more suited to deal with a central aspect of the natural information processing tasks that people are so good at....We will introduce a computational framework for modeling cognitive processes that seems...closer than other frameworks to the style of computation as it might be done by the brain.*

The premise that PDP is using here is that traditional computer programs work very differently from brains, and that might be why computer programs had been (at that point) so bad at doing things that brains find easy (such as recognizing objects in pictures). The authors claimed that the PDP approach was "closer than other frameworks" to how the brain works, and therefore it might be better able to handle these kinds of tasks.

In fact, the approach laid out in PDP is very similar to the approach used in today's neural networks. The book defined parallel distributed processing as requiring the following:

- A set of *processing units*
- A *state of activation*
- An *output function* for each unit
- A *pattern of connectivity* among units
- A *propagation rule* for propagating patterns of activities through the network of connectivities
- An *activation rule* for combining the inputs impinging on a unit with the current state of that unit to produce an output for the unit
- A *learning rule* whereby patterns of connectivity are modified by experience
- An *environment* within which the system must operate

We will see in this book that modern neural networks handle each of these requirements.

In the 1980s, most models were built with a second layer of neurons, thus avoiding the problem that had been identified by Minsky and Papert (this was their "pattern of

connectivity among units," to use the preceding framework). And indeed, neural networks were widely used during the '80s and '90s for real, practical projects. However, again a misunderstanding of the theoretical issues held back the field. In theory, adding just one extra layer of neurons was enough to allow any mathematical function to be approximated with these neural networks, but in practice such networks were often too big and too slow to be useful.

Although researchers showed 30 years ago that to get practical, good performance you need to use even more layers of neurons, it is only in the last decade that this principle has been more widely appreciated and applied. Neural networks are now finally living up to their potential, thanks to the use of more layers, coupled with the capacity to do so because of improvements in computer hardware, increases in data availability, and algorithmic tweaks that allow neural networks to be trained faster and more easily. We now have what Rosenblatt promised: "a machine capable of perceiving, recognizing, and identifying its surroundings without any human training or control."

This is what you will learn how to build in this book. But first, since we are going to be spending a lot of time together, let's get to know each other a bit...

Who We Are

We are Sylvain and Jeremy, your guides on this journey. We hope that you will find us well suited for this position.

Jeremy has been using and teaching machine learning for around 30 years. He started using neural networks 25 years ago. During this time, he has led many companies and projects that have machine learning at their core, including founding the first company to focus on deep learning and medicine, Enlitic, and taking on the role of president and chief scientist at the world's largest machine learning community, Kaggle. He is the cofounder, along with Dr. Rachel Thomas, of fast.ai, the organization that built the course this book is based on.

From time to time, you will hear directly from us in sidebars, like this one from Jeremy:

Jeremy Says

Hi, everybody; I'm Jeremy! You might be interested to know that I do not have any formal technical education. I completed a BA with a major in philosophy, and didn't have great grades. I was much more interested in doing real projects than theoretical studies, so I worked full time at a management consulting firm called McKinsey and Company throughout my university years. If you're somebody who would rather get their hands dirty building stuff than spend years learning abstract concepts, you will understand where I am coming from! Look out for sidebars from me to find information most suited to people with a less mathematical or formal technical

background—that is, people like me...

Sylvain, on the other hand, knows a lot about formal technical education. He has written 10 math textbooks, covering the entire advanced French math curriculum!

Sylvain Says

Unlike Jeremy, I have not spent many years coding and applying machine learning algorithms. Rather, I recently came to the machine learning world by watching Jeremy's fast.ai course videos. So, if you are somebody who has not opened a terminal and written commands at the command line, you will understand where I am coming from! Look out for sidebars from me to find information most suited to people with a more mathematical or formal technical background, but less real-world coding experience—that is, people like me...

The fast.ai course has been studied by hundreds of thousands of students, from all walks of life, from all parts of the world. Sylvain stood out as the most impressive student of the course that Jeremy had ever seen, which led to him joining fast.ai and then becoming the coauthor, along with Jeremy, of the fastai software library.

All this means that between us, you have the best of both worlds: the people who know more about the software than anybody else, because they wrote it; an expert on math, and an expert on coding and machine learning; and also people who understand both what it feels like to be a relative outsider in math, and a relative outsider in coding and machine learning.

Anybody who has watched sports knows that if you have a two-person commentary team, you also need a third person to do "special comments." Our special commentator is Alexis Gallagher. Alexis has a very diverse background: he has been a researcher in mathematical biology, a screenplay writer, an improv performer, a McKinsey consultant (like Jeremy!), a Swift coder, and a CTO.

Alexis Says

I've decided it's time for me to learn about this AI stuff! After all, I've tried pretty much everything else....But I don't really have a background in building machine learning models. Still...how hard can it be? I'm going to be learning throughout this book, just like you are. Look out for my sidebars for learning tips that I found helpful on my journey, and hopefully you will find helpful too.

How to Learn Deep Learning

Harvard professor David Perkins, who wrote *Making Learning Whole* (Jossey-Bass), has much to say about teaching. The basic idea is to teach the *whole game*. That means that if you're teaching baseball, you first take people to a baseball game or get them to play it. You don't teach them how to wind twine to make a baseball from scratch, the

physics of a parabola, or the coefficient of friction of a ball on a bat.

Paul Lockhart, a Columbia math PhD, former Brown professor, and K–12 math teacher, imagines in the influential essay "A Mathematician's Lament" a nightmare world where music and art are taught the way math is taught. Children are not allowed to listen to or play music until they have spent over a decade mastering music notation and theory, spending classes transposing sheet music into a different key. In art class, students study colors and applicators, but aren't allowed to actually paint until college. Sound absurd? This is how math is taught—we require students to spend years doing rote memorization and learning dry, disconnected *fundamentals* that we claim will pay off later, long after most of them quit the subject.

Unfortunately, this is where many teaching resources on deep learning begin—asking learners to follow along with the definition of the Hessian and theorems for the Taylor approximation of your loss functions, without ever giving examples of actual working code. We're not knocking calculus. We love calculus, and Sylvain has even taught it at the college level, but we don't think it's the best place to start when learning deep learning!

In deep learning, it really helps if you have the motivation to fix your model to get it to do better. That's when you start learning the relevant theory. But you need to have the model in the first place. We teach almost everything through real examples. As we build out those examples, we go deeper and deeper, and we'll show you how to make your projects better and better. This means that you'll be gradually learning all the theoretical foundations you need, in context, in such a way that you'll see why it matters and how it works.

So, here's our commitment to you. Throughout this book, we follow these principles:

Teaching the whole game

> We'll start off by showing you how to use a complete, working, usable, state-of-the-art deep learning network to solve real-world problems using simple, expressive tools. And then we'll gradually dig deeper and deeper into understanding how those tools are made, and how the tools that make those tools are made, and so on...

Always teaching through examples

> We'll ensure that there is a context and a purpose that you can understand intuitively, rather than starting with algebraic symbol manipulation.

Simplifying as much as possible

> We've spent years building tools and teaching methods that make previously complex topics simple.

Removing barriers

> Deep learning has, until now, been an exclusive game. We're breaking it open and ensuring that everyone can play.

The hardest part of deep learning is artisanal: how do you know if you've got enough data, whether it is in the right format, if your model is training properly, and, if it's not, what you should do about it? That is why we believe in learning by doing. As with basic data science skills, with deep learning you get better only through practical experience. Trying to spend too much time on the theory can be counterproductive. The key is to just code and try to solve problems: the theory can come later, when you have context and motivation.

There will be times when the journey feels hard. Times when you feel stuck. Don't give up! Rewind through the book to find the last bit where you definitely weren't stuck, and then read slowly through from there to find the first thing that isn't clear. Then try some code experiments yourself, and Google around for more tutorials on whatever the issue you're stuck with is—often you'll find a different angle on the material that might help it to click. Also, it's expected and normal to not understand everything (especially the code) on first reading. Trying to understand the material serially before proceeding can sometimes be hard. Sometimes things click into place after you get more context from parts down the road, from having a bigger picture. So if you do get stuck on a section, try moving on anyway and make a note to come back to it later.

Remember, you don't need any particular academic background to succeed at deep learning. Many important breakthroughs are made in research and industry by folks without a PhD, such as the paper "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks"—one of the most influential papers of the last decade, with over 5,000 citations—which was written by Alec Radford when he was an undergraduate. Even at Tesla, where they're trying to solve the extremely tough challenge of making a self-driving car, CEO Elon Musk says:

> *A PhD is definitely not required. All that matters is a deep understanding of AI & ability to implement NNs in a way that is actually useful (latter point is what's truly hard). Don't care if you even graduated high school.*

What you will need to do to succeed, however, is to apply what you learn in this book to a personal project, and always persevere.

Your Projects and Your Mindset

Whether you're excited to identify if plants are diseased from pictures of their leaves, autogenerate knitting patterns, diagnose TB from X-rays, or determine when a raccoon is using your cat door, we will get you using deep learning on your own problems (via pretrained models from others) as quickly as possible, and then will progressively drill into more details. You'll learn how to use deep learning to solve your own problems at state-of-the-art accuracy within the first 30 minutes of the next chapter! (And feel free to skip straight there now if you're dying to get coding right away.) There is a pernicious myth out there that you need to have computing resources and datasets the size of those at Google to be able to do deep learning, but it's not true.

So, what sorts of tasks make for good test cases? You could train your model to distinguish between Picasso and Monet paintings or to pick out pictures of your daughter instead of pictures of your son. It helps to focus on your hobbies and passions—setting yourself four or five little projects rather than striving to solve a big, grand problem tends to work better when you're getting started. Since it is easy to get stuck, trying to be too ambitious too early can often backfire. Then, once you've got the basics mastered, aim to complete something you're really proud of!

Jeremy Says

Deep learning can be set to work on almost any problem. For instance, my first startup was a company called FastMail, which provided enhanced email services when it launched in 1999 (and still does to this day). In 2002, I set it up to use a primitive form of deep learning, single-layer neural networks, to help categorize emails and stop customers from receiving spam.

Common character traits in the people who do well at deep learning include playfulness and curiosity. The late physicist Richard Feynman is an example of someone we'd expect to be great at deep learning: his development of an understanding of the movement of subatomic particles came from his amusement at how plates wobble when they spin in the air.

Let's now focus on what you will learn, starting with the software.

The Software: PyTorch, fastai, and Jupyter (And Why It Doesn't Matter)

We've completed hundreds of machine learning projects using dozens of packages, and many programming languages. At fast.ai, we have written courses using most of the main deep learning and machine learning packages used today. After PyTorch came out in 2017, we spent over a thousand hours testing it before deciding that we would use it for future courses, software development, and research. Since that time, PyTorch has become the world's fastest-growing deep learning library and is already used for most research papers at top conferences. This is generally a leading indicator of usage in industry, because these are the papers that end up getting used in products and

services commercially. We have found that PyTorch is the most flexible and expressive library for deep learning. It does not trade off speed for simplicity, but provides both.

PyTorch works best as a low-level foundation library, providing the basic operations for higher-level functionality. The fastai library is the most popular library for adding this higher-level functionality on top of PyTorch. It's also particularly well suited to the purposes of this book, because it is unique in providing a deeply layered software architecture (there's even a peer-reviewed academic paper about this layered API). In this book, as we go deeper and deeper into the foundations of deep learning, we will also go deeper and deeper into the layers of fastai. This book covers version 2 of the fastai library, which is a from-scratch rewrite providing many unique features.

However, it doesn't really matter what software you learn, because it takes only a few days to learn to switch from one library to another. What really matters is learning the deep learning foundations and techniques properly. Our focus will be on using code that, as clearly as possible, expresses the concepts that you need to learn. Where we are teaching high-level concepts, we will use high-level fastai code. Where we are teaching low-level concepts, we will use low-level PyTorch or even pure Python code.

Though it may seem like new deep learning libraries are appearing at a rapid pace nowadays, you need to be prepared for a much faster rate of change in the coming months and years. As more people enter the field, they will bring more skills and ideas, and try more things. You should assume that whatever specific libraries and software you learn today will be obsolete in a year or two. Just think about the number of changes in libraries and technology stacks that occur all the time in the world of web programming—a much more mature and slow-growing area than deep learning. We strongly believe that the focus in learning needs to be on understanding the underlying techniques and how to apply them in practice, and how to quickly build expertise in new tools and techniques as they are released.

By the end of the book, you'll understand nearly all the code that's inside fastai (and much of PyTorch too), because in each chapter we'll be digging a level deeper to show you exactly what's going on as we build and train our models. This means that you'll have learned the most important best practices used in modern deep learning—not just how to use them, but how they really work and are implemented. If you want to use those approaches in another framework, you'll have the knowledge you need to do so if needed.

Since the most important thing for learning deep learning is writing code and experimenting, it's important that you have a great platform for experimenting with code. The most popular programming experimentation platform is called Jupyter. This is what we will be using throughout this book. We will show you how you can use

Jupyter to train and experiment with models and introspect every stage of the data preprocessing and model development pipeline. Jupyter is the most popular tool for doing data science in Python, for good reason. It is powerful, flexible, and easy to use. We think you will love it!

Let's see it in practice and train our first model.

Your First Model

As we said before, we will teach you how to do things before we explain why they work. Following this top-down approach, we will begin by actually training an image classifier to recognize dogs and cats with almost 100% accuracy. To train this model and run our experiments, you will need to do some initial setup. Don't worry; it's not as hard as it looks.

Sylvain Says

Do not skip the setup part even if it looks intimidating at first, especially if you have little or no experience using things like a terminal or the command line. Most of that is not necessary, and you will find that the easiest servers can be set up with just your usual web browser. It is crucial that you run your own experiments in parallel with this book in order to learn.

Getting a GPU Deep Learning Server

To do nearly everything in this book, you'll need access to a computer with an NVIDIA GPU (unfortunately, other brands of GPU are not fully supported by the main deep learning libraries). However, we don't recommend you buy one; in fact, even if you already have one, we don't suggest you use it just yet! Setting up a computer takes time and energy, and you want all your energy to focus on deep learning right now. Therefore, we instead suggest you rent access to a computer that already has everything you need preinstalled and ready to go. Costs can be as little as $0.25 per hour while you're using it, and some options are even free.

Jargon: Graphics Processing Unit (GPU)

Also known as a *graphics card*. A special kind of processor in your computer that can handle thousands of single tasks at the same time, especially designed for displaying 3D environments on a computer for playing games. These same basic tasks are very similar to what neural networks do, such that GPUs can run neural networks hundreds of times faster than regular CPUs. All modern computers contain a GPU, but few contain the right kind of GPU necessary for deep learning.

The best choice of GPU servers to use with this book will change over time, as companies come and go and prices change. We maintain a list of our recommended options on the book's website, so go there now and follow the instructions to get connected to a GPU deep learning server. Don't worry; it takes only about two minutes

to get set up on most platforms, and many don't even require any payment or even a credit card to get started.

Alexis Says

My two cents: heed this advice! If you like computers, you will be tempted to set up your own box. Beware! It is feasible but surprisingly involved and distracting. There is a good reason this book is not titled *Everything You Ever Wanted to Know About Ubuntu System Administration, NVIDIA Driver Installation, apt-get, conda, pip, and Jupyter Notebook Configuration*. That would be a book of its own. Having designed and deployed our production machine learning infrastructure at work, I can testify it has its satisfactions, but it is as unrelated to modeling as maintaining an airplane is to flying one.

Each option shown on the website includes a tutorial; after completing the tutorial, you will end up with a screen looking like Figure 1-2.

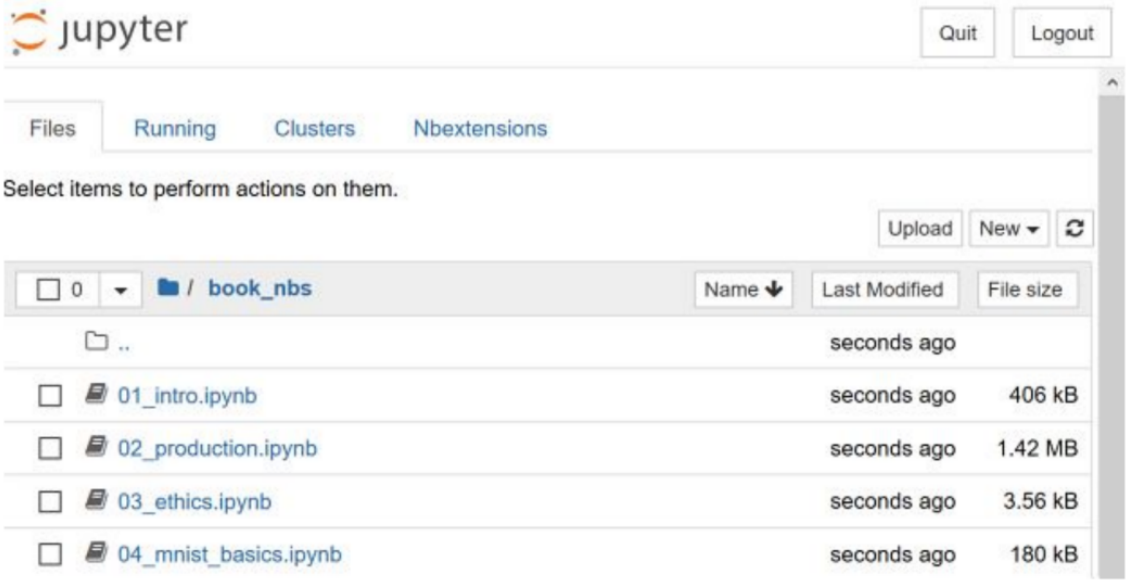


Figure 1-2. Initial view of Jupyter Notebook

You are now ready to run your first Jupyter notebook!

Jargon: Jupyter Notebook

A piece of software that allows you to include formatted text, code, images, videos, and much more, all within a single interactive document. Jupyter received the highest honor for software, the ACM Software System Award, thanks to its wide use and enormous impact in many academic fields and in industry. Jupyter Notebook is the software most widely used by data scientists for developing and interacting with deep learning models.

Running Your First Notebook

The notebooks are numbered by chapter in the same order as they are presented in this book. So, the very first notebook you will see listed is the notebook that you need to use now. You will be using this notebook to train a model that can recognize dog and cat photos. To do this, you'll be downloading a dataset of dog and cat photos, and using that to *train a model*.

A *dataset* is simply a bunch of data—it could be images, emails, financial indicators, sounds, or anything else. There are many datasets made freely available that are suitable for training models. Many of these datasets are created by academics to help advance research, many are made available for competitions (there are competitions where data scientists can compete to see who has the most accurate model!), and some are byproducts of other processes (such as financial filings).

Full and Stripped Notebooks

There are two versions of the notebooks. The root of the repo contains the exact notebooks used to create the book you're reading now, with all the prose and outputs. The *clean* folder has the same headings and code cells, but all outputs and prose have been removed. After reading a section of the book, we recommend working through the clean notebooks, with the book closed, and seeing if you can figure out what each cell will show before you execute it. Also try to recall what the code is demonstrating.

To open a notebook, just click it. The notebook will open, and it will look something like Figure 1-3 (note that there may be slight differences in details across different platforms; you can ignore those differences).

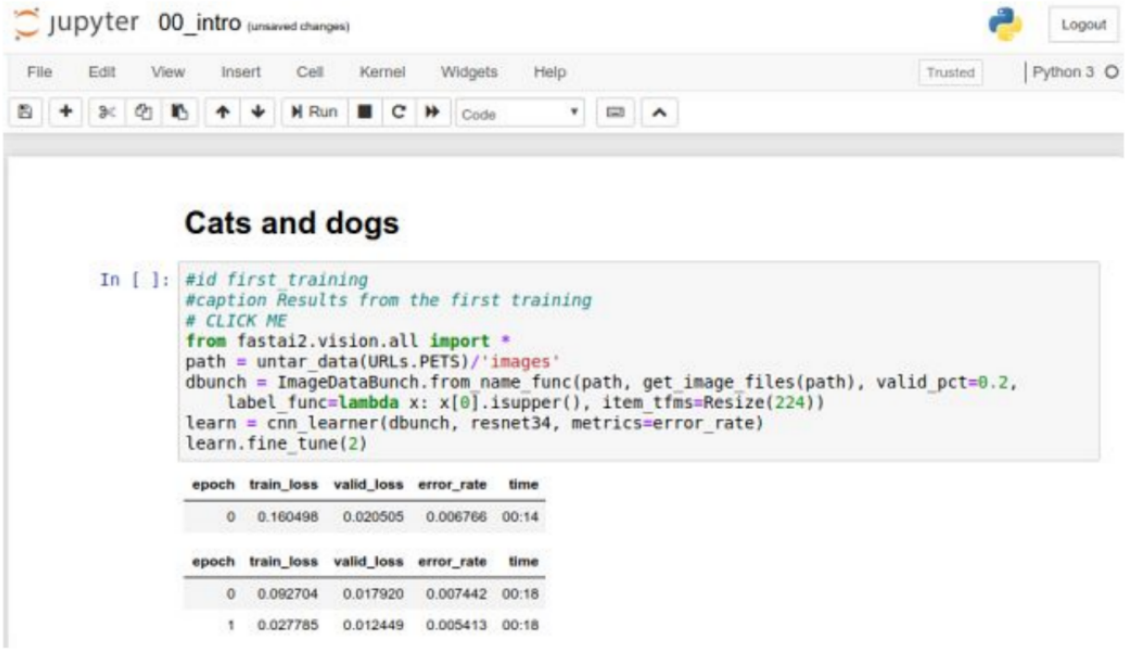


Figure 1-3. A Jupyter notebook

A notebook consists of *cells*. There are two main types of cell:

- Cells containing formatted text, images, and so forth. These use a format called *Markdown*, which you will learn about soon.
- Cells containing code that can be executed, and outputs will appear immediately underneath (which could be plain text, tables, images, animations, sounds, or even interactive applications).

Jupyter notebooks can be in one of two modes: edit mode or command mode. In edit mode, typing on your keyboard enters the letters into the cell in the usual way. However, in command mode, you will not see any flashing cursor, and each key on your keyboard will have a special function.

Before continuing, press the Escape key on your keyboard to switch to command mode (if you are already in command mode, this does nothing, so press it now just in case). To see a complete list of all the functions available, press H; press Escape to remove this help screen. Notice that in command mode, unlike in most programs, commands do not require you to hold down Control, Alt, or similar—you simply press the required letter key.

You can make a copy of a cell by pressing C (the cell needs to be selected first, indicated with an outline around it; if it is not already selected, click it once). Then

press V to paste a copy of it.

Click the cell that begins with the line "# CLICK ME" to select it. The first character in that line indicates that what follows is a comment in Python, so it is ignored when executing the cell. The rest of the cell is, believe it or not, a complete system for creating and training a state-of-the-art model for recognizing cats versus dogs. So, let's train it now! To do so, just press Shift-Enter on your keyboard, or click the Play button on the toolbar. Then wait a few minutes while the following things happen:

1. A dataset called the Oxford-IIIT Pet Dataset that contains 7,349 images of cats and dogs from 37 breeds will be downloaded from the fast.ai datasets collection to the GPU server you are using, and will then be extracted.
2. A *pretrained model* that has already been trained on 1.3 million images using a competition-winning model will be downloaded from the internet.
3. The pretrained model will be *fine-tuned* using the latest advances in transfer learning to create a model that is specially customized for recognizing dogs and cats.

The first two steps need to be run only once on your GPU server. If you run the cell again, it will use the dataset and model that have already been downloaded, rather than downloading them again. Let's take a look at the contents of the cell and the results (Table 1-2):

```
# CLICK ME
from fastai.vision.all import *
path = untar_data(URLs.PETS)/'images'

def is_cat(x): return x[0].isupper()
dls = ImageDataLoaders.from_name_func(
    path, get_image_files(path), valid_pct=0.2, seed=42,
    label_func=is_cat, item_tfms=Resize(224))

learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1)
```

Table 1-2. Results from the first training

epoch	train_loss	valid_loss	error_rate	time
0	0.169390	0.021388	0.005413	00:14

epoch	train_loss	valid_loss	error_rate	time
0	0.058748	0.009240	0.002706	00:19

You will probably not see exactly the same results shown here. A lot of sources of small random variation are involved in training models. We generally see an error rate of well less than 0.02 in this example, however.

Training Time

Depending on your network speed, it might take a few minutes to download the pretrained model and dataset. Running `fine_tune` might take a minute or so. Often models in this book take a few minutes to train, as will your own models, so it's a good idea to come up with good techniques to make the most of this time. For instance, keep reading the next section while your model trains, or open up another notebook and use it for some coding experiments.

This Book Was Written in Jupyter Notebooks

We wrote this book using Jupyter notebooks, so for nearly every chart, table, and calculation in this book, we'll be showing you the exact code required to replicate it yourself. That's why very often in this book, you will see some code immediately followed by a table, a picture, or just some text. If you go on the book's website, you will find all the code, and you can try running and modifying every example yourself.

You just saw how a cell that outputs a table looks in the book. Here is an example of a cell that outputs text:

```
1+1
```

```
2
```

Jupyter will always print or show the result of the last line (if there is one). For instance, here is an example of a cell that outputs an image:

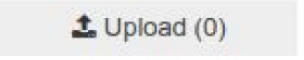
```
img = PILImage.create(image_cat())
img.to_thumb(192)
```

So, how do we know if this model is any good? In the last column of the table, you can see the *error rate*, which is the proportion of images that were incorrectly identified. The error rate serves as our metric—our measure of model quality, chosen to be intuitive and comprehensible. As you can see, the model is nearly perfect, even though the training time was only a few seconds (not including the one-time downloading of the dataset and the pretrained model). In fact, the accuracy you've achieved already is

far better than anybody had ever achieved just 10 years ago!

Finally, let's check that this model actually works. Go and get a photo of a dog or a cat; if you don't have one handy, just search Google Images and download an image that you find there. Now execute the cell with `uploader` defined. It will output a button you can click, so you can select the image you want to classify:

```
uploader = widgets.FileUpload()
uploader
```



Now you can pass the uploaded file to the model. Make sure that it is a clear photo of a single dog or a cat, and not a line drawing, cartoon, or similar. The notebook will tell you whether it thinks it is a dog or a cat, and how confident it is. Hopefully, you'll find that your model did a great job:

```
img = PILImage.create(uploader.data[0])
is_cat,_,probs = learn.predict(img)
print(f"Is this a cat?: {is_cat}.")
print(f"Probability it's a cat: {probs[1].item():.6f}")
```

```
Is this a cat?: True.
Probability it's a cat: 0.999986
```

Congratulations on your first classifier!

But what does this mean? What did you actually do? In order to explain this, let's zoom out again to take in the big picture.

What Is Machine Learning?

Your classifier is a deep learning model. As was already mentioned, deep learning models use neural networks, which originally date from the 1950s and have become powerful very recently thanks to recent advancements.

Another key piece of context is that deep learning is just a modern area in the more general discipline of *machine learning*. To understand the essence of what you did when you trained your own classification model, you don't need to understand deep learning. It is enough to see how your model and your training process are examples of the concepts that apply to machine learning in general.

So in this section, we will describe machine learning. We will explore the key concepts

and see how they can be traced back to the original essay that introduced them.

Machine learning is, like regular programming, a way to get computers to complete a specific task. But how would we use regular programming to do what we just did in the preceding section: recognize dogs versus cats in photos? We would have to write down for the computer the exact steps necessary to complete the task.

Normally, it's easy enough for us to write down the steps to complete a task when we're writing a program. We just think about the steps we'd take if we had to do the task by hand, and then we translate them into code. For instance, we can write a function that sorts a list. In general, we'd write a function that looks something like Figure 1-4 (where *inputs* might be an unsorted list, and *results* a sorted list).

Figure 1-4. A traditional program

But for recognizing objects in a photo, that's a bit tricky; what *are* the steps we take when we recognize an object in a picture? We really don't know, since it all happens in our brain without us being consciously aware of it!

Right back at the dawn of computing, in 1949, an IBM researcher named Arthur Samuel started working on a different way to get computers to complete tasks, which he called *machine learning*. In his classic 1962 essay "Artificial Intelligence: A Frontier of Automation," he wrote:

> *Programming a computer for such computations is, at best, a difficult task, not primarily because of any inherent complexity in the computer itself but, rather, because of the need to spell out every minute step of the process in the most exasperating detail. Computers, as any programmer will tell you, are giant morons, not giant brains.*

His basic idea was this: instead of telling the computer the exact steps required to solve a problem, show it examples of the problem to solve, and let it figure out how to solve it itself. This turned out to be very effective: by 1961, his checkers-playing program had learned so much that it beat the Connecticut state champion! Here's how he described his idea (from the same essay as noted previously):

> *Suppose we arrange for some automatic means of testing the effectiveness of any current weight assignment in terms of actual performance and provide a mechanism for altering the weight assignment so as to maximize the performance. We need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programmed would "learn" from its experience.*

There are a number of powerful concepts embedded in this short statement:

- The idea of a "weight assignment"
- The fact that every weight assignment has some "actual performance"
- The requirement that there be an "automatic means" of testing that performance
- The need for a "mechanism" (i.e., another automatic process) for improving the performance by changing the weight assignments

Let's take these concepts one by one, in order to understand how they fit together in practice. First, we need to understand what Samuel means by a *weight assignment*.

Weights are just variables, and a weight assignment is a particular choice of values for those variables. The program's inputs are values that it processes in order to produce its results—for instance, taking image pixels as inputs, and returning the classification "dog" as a result. The program's weight assignments are other values that define how the program will operate.

Because they will affect the program, they are in a sense another kind of input. We will update our basic picture in Figure 1-4 and replace it with Figure 1-5 in order to take this into account.

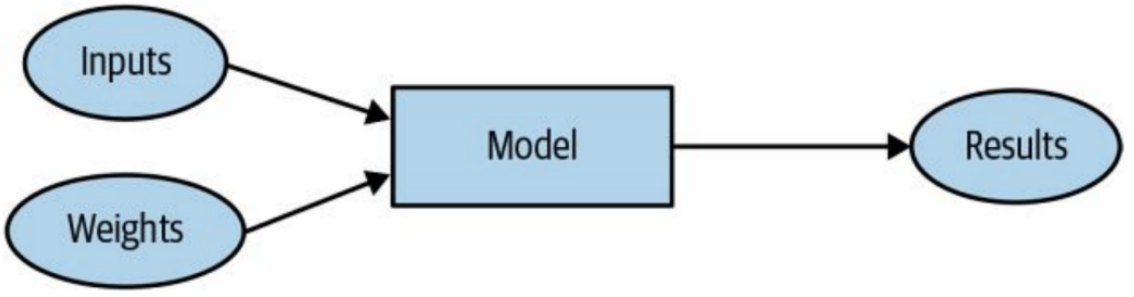


Figure 1-5. A program using weight assignment

We've changed the name of our box from *program* to *model*. This is to follow modern terminology and to reflect that the *model* is a special kind of program: it's one that can do *many different things*, depending on the *weights*. It can be implemented in many different ways. For instance, in Samuel's checkers program, different values of the weights would result in different checkers-playing strategies.

(By the way, what Samuel called "weights" are most generally referred to as model *parameters* these days, in case you have encountered that term. The term *weights* is reserved for a particular type of model parameter.)

Next, Samuel said we need an *automatic means of testing the effectiveness of any current weight assignment in terms of actual performance*. In the case of his checkers program, the "actual performance" of a model would be how well it plays. And you could automatically test the performance of two models by setting them to play against each other, and seeing which one usually wins.

Finally, he says we need *a mechanism for altering the weight assignment so as to maximize the performance*. For instance, we could look at the difference in weights between the winning model and the losing model, and adjust the weights a little further in the winning direction.

We can now see why he said that such a procedure *could be made entirely automatic and... a machine so programmed would "learn" from its experience*. Learning would become entirely automatic when the adjustment of the weights was also automatic—when instead of us improving a model by adjusting its weights manually, we relied on an automated mechanism that produced adjustments based on performance.

Figure 1-6 shows the full picture of Samuel's idea of training a machine learning model.

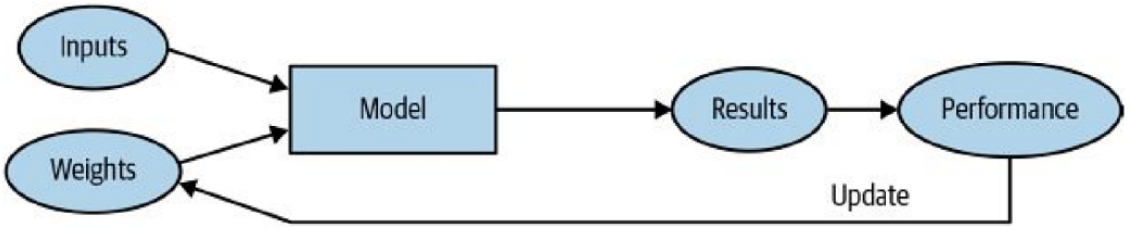


Figure 1-6. Training a machine learning model

Notice the distinction between the model's *results* (e.g., the moves in a checkers game) and its *performance* (e.g., whether it wins the game, or how quickly it wins).

Also note that once the model is trained—that is, once we've chosen our final, best, favorite weight assignment—then we can think of the weights as being *part of the model*, since we're not varying them anymore.

Therefore, actually *using* a model after it's trained looks like Figure 1-7.

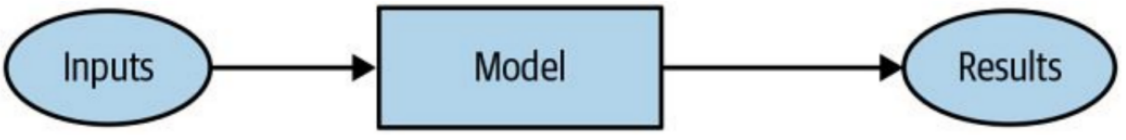


Figure 1-7. Using a trained model as a program

This looks identical to our original diagram in Figure 1-4, just with the word *program* replaced with *model*. This is an important insight: *a trained model can be treated just like a*

regular computer program.

Jargon: Machine Learning

The training of programs developed by allowing a computer to learn from its experience, rather than through manually coding the individual steps.

What Is a Neural Network?

It's not too hard to imagine what the model might look like for a checkers program. There might be a range of checkers strategies encoded, and some kind of search mechanism, and then the weights could vary how strategies are selected, what parts of the board are focused on during a search, and so forth. But it's not at all obvious what the model might look like for an image recognition program, or for understanding text, or for many other interesting problems we might imagine.

What we would like is some kind of function that is so flexible that it could be used to solve any given problem, just by varying its weights. Amazingly enough, this function actually exists! It's the neural network, which we already discussed. That is, if you regard a neural network as a mathematical function, it turns out to be a function that is extremely flexible depending on its weights. A mathematical proof called the *universal approximation theorem* shows that this function can solve any problem to any level of accuracy, in theory. The fact that neural networks are so flexible means that, in practice, they are often a suitable kind of model, and you can focus your effort on the process of training them—that is, of finding good weight assignments.

But what about that process? One could imagine that you might need to find a new "mechanism" for automatically updating weights for every problem. This would be laborious. What we'd like here as well is a completely general way to update the weights of a neural network, to make it improve at any given task. Conveniently, this also exists!

This is called *stochastic gradient descent* (SGD). We'll see how neural networks and SGD work in detail in Chapter 4, as well as explaining the universal approximation theorem. For now, however, we will instead use Samuel's own words: *We need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programmed would "learn" from its experience.*

Jeremy Says

Don't worry; neither SGD nor neural nets are mathematically complex. Both nearly entirely rely on addition and multiplication to do their work (but they do a *lot* of addition and multiplication!). The main reaction we hear from students when they see the details is: "Is that all it is?"

In other words, to recap, a neural network is a particular kind of machine learning

model, which fits right in to Samuel's original conception. Neural networks are special because they are highly flexible, which means they can solve an unusually wide range of problems just by finding the right weights. This is powerful, because stochastic gradient descent provides us a way to find those weight values automatically.

Having zoomed out, let's now zoom back in and revisit our image classification problem using Samuel's framework.

Our inputs are the images. Our weights are the weights in the neural net. Our model is a neural net. Our results are the values that are calculated by the neural net, like "dog" or "cat."

What about the next piece, an *automatic means of testing the effectiveness of any current weight assignment in terms of actual performance*? Determining "actual performance" is easy enough: we can simply define our model's performance as its accuracy at predicting the correct answers.

Putting this all together, and assuming that SGD is our mechanism for updating the weight assignments, we can see how our image classifier is a machine learning model, much like Samuel envisioned.

A Bit of Deep Learning Jargon

Samuel was working in the 1960s, and since then terminology has changed. Here is the modern deep learning terminology for all the pieces we have discussed:

- The functional form of the *model* is called its *architecture* (but be careful—sometimes people use *model* as a synonym of *architecture*, so this can get confusing).
- The *weights* are called *parameters*.
- The *predictions* are calculated from the *independent variable*, which is the *data* not including the *labels*.
- The *results* of the model are called *predictions*.
- The measure of *performance* is called the *loss*.
- The loss depends not only on the predictions, but also on the correct *labels* (also known as *targets* or the *dependent variable*); e.g., "dog" or "cat."

After making these changes, our diagram in Figure 1-6 looks like Figure 1-8.

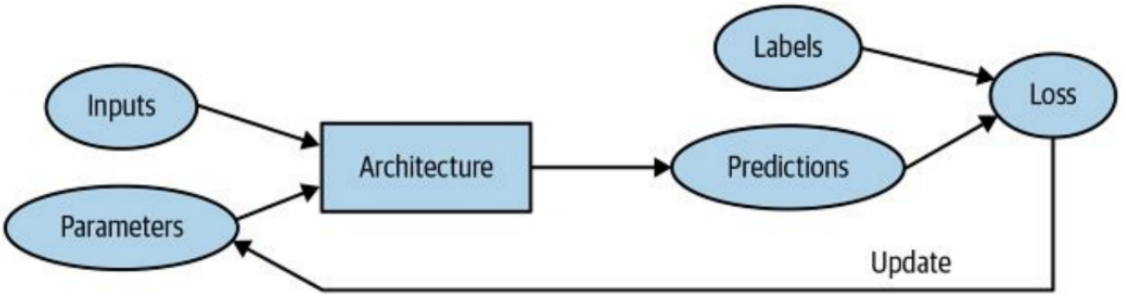


Figure 1-8. Detailed training loop

Limitations Inherent to Machine Learning

From this picture, we can now see some fundamental things about training a deep learning model:

- A model cannot be created without data.
- A model can learn to operate on only the patterns seen in the input data used to train it.
- This learning approach creates only *predictions*, not recommended *actions*.
- It's not enough to just have examples of input data; we need *labels* for that data too (e.g., pictures of dogs and cats aren't enough to train a model; we need a label for each one, saying which ones are dogs and which are cats).

Generally speaking, we've seen that most organizations that say they don't have enough data actually mean they don't have enough *labeled* data. If any organization is interested in doing something in practice with a model, then presumably they have some inputs they plan to run their model against. And presumably they've been doing that some other way for a while (e.g., manually, or with some heuristic program), so they have data from those processes! For instance, a radiology practice will almost certainly have an archive of medical scans (since they need to be able to check how their patients are progressing over time), but those scans may not have structured labels containing a list of diagnoses or interventions (since radiologists generally create free-text natural language reports, not structured data). We'll be discussing labeling approaches a lot in this book, because it's such an important issue in practice.

Since these kinds of machine learning models can only make *predictions* (i.e., attempt to replicate labels), this can result in a significant gap between organizational goals and model capabilities. For instance, in this book you'll learn how to create a *recommendation system* that can predict what products a user might purchase. This is often used in ecommerce, such as to customize products shown on a home page by

showing the highest-ranked items. But such a model is generally created by looking at a user and their buying history (*inputs*) and what they went on to buy or look at (*labels*), which means that the model is likely to tell you about products the user already has, or already knows about, rather than new products that they are most likely to be interested in hearing about. That's very different from what, say, an expert at your local bookseller might do, where they ask questions to figure out your taste, and then tell you about authors or series that you've never heard of before.

Another critical insight comes from considering how a model interacts with its environment. This can create *feedback loops*, as described here:

1. A *predictive policing* model is created based on where arrests have been made in the past. In practice, this is not actually predicting crime, but rather predicting arrests, and is therefore partially simply reflecting biases in existing policing processes.
2. Law enforcement officers then might use that model to decide where to focus their policing activity, resulting in increased arrests in those areas.
3. Data on these additional arrests would then be fed back in to retrain future versions of the model.

This is a *positive feedback loop*: the more the model is used, the more biased the data becomes, making the model even more biased, and so forth.

Feedback loops can also create problems in commercial settings. For instance, a video recommendation system might be biased toward recommending content consumed by the biggest watchers of video (e.g., conspiracy theorists and extremists tend to watch more online video content than the average), resulting in those users increasing their video consumption, resulting in more of those kinds of videos being recommended. We'll consider this topic in more detail in Chapter 3.

Now that you have seen the base of the theory, let's go back to our code example and see in detail how the code corresponds to the process we just described.

How Our Image Recognizer Works

Let's see just how our image recognizer code maps to these ideas. We'll put each line into a separate cell, and look at what each one is doing (we won't explain every detail of every parameter yet, but will give a description of the important bits; full details will come later in the book). The first line imports all of the fastai.vision library:

```
from fastai.vision.all import *
```

This gives us all of the functions and classes we will need to create a wide variety of computer vision models.

Jeremy Says

A lot of Python coders recommend avoiding importing a whole library like this (using the `import *` syntax) because in large software projects it can cause problems. However, for interactive work such as in a Jupyter notebook, it works great. The fastai library is specially designed to support this kind of interactive use, and it will import only the necessary pieces into your environment.

The second line downloads a standard dataset from the fast.ai datasets collection (if not previously downloaded) to your server, extracts it (if not previously extracted), and returns a `Path` object with the extracted location:

```
path = untar_data(URLs.PETS)/'images'
```

Sylvain Says

Throughout my time studying at fast.ai, and even still today, I've learned a lot about productive coding practices. The fastai library and fast.ai notebooks are full of great little tips that have helped make me a better programmer. For instance, notice that the fastai library doesn't just return a string containing the path to the dataset, but a `Path` object. This is a really useful class from the Python 3 standard library that makes accessing files and directories much easier. If you haven't come across it before, be sure to check out its documentation or a tutorial and try it out. Note that the book's website contains links to recommended tutorials for each chapter. I'll keep letting you know about little coding tips I've found useful as we come across them.

In the third line, we define a function, `is_cat`, that labels cats based on a filename rule provided by the dataset's creators:

```
def is_cat(x): return x[0].isupper()
```

We use that function in the fourth line, which tells fastai what kind of dataset we have and how it is structured:

```
dls = ImageDataLoaders.from_name_func(
    path, get_image_files(path), valid_pct=0.2, seed=42,
    label_func=is_cat, item_tfms=Resize(224))
```

There are various classes for different kinds of deep learning datasets and problems—

here we're using `ImageDataLoaders`. The first part of the class name will generally be the type of data you have, such as image or text.

The other important piece of information that we have to tell fastai is how to get the labels from the dataset. Computer vision datasets are normally structured in such a way that the label for an image is part of the filename or path—most commonly the parent folder name. fastai comes with a number of standardized labeling methods, and ways to write your own. Here we're telling fastai to use the `is_cat` function we just defined.

Finally, we define the `Transforms` that we need. A `Transform` contains code that is applied automatically during training; fastai includes many predefined `Transforms`, and adding new ones is as simple as creating a Python function. There are two kinds: `item_tfms` are applied to each item (in this case, each item is resized to a 224-pixel square), while `batch_tfms` are applied to a *batch* of items at a time using the GPU, so they're particularly fast (we'll see many examples of these throughout this book).

Why 224 pixels? This is the standard size for historical reasons (old pretrained models require this size exactly), but you can pass pretty much anything. If you increase the size, you'll often get a model with better results (since it will be able to focus on more details), but at the price of speed and memory consumption; the opposite is true if you decrease the size.

Jargon: Classification and Regression

Classification and *regression* have very specific meanings in machine learning. These are the two main types of model that we will be investigating in this book. A *classification model* is one that attempts to predict a class, or category. That is, it's predicting from a number of discrete possibilities, such as "dog" or "cat." A *regression model* is one that attempts to predict one or more numeric quantities, such as a temperature or a location. Sometimes people use the word *regression* to refer to a particular kind of model called a *linear regression model*; this is a bad practice, and we won't be using that terminology in this book!

The Pet dataset contains 7,390 pictures of dogs and cats, consisting of 37 breeds. Each image is labeled using its filename: for instance, the file *great_pyrenees_173.jpg* is the 173rd example of an image of a Great Pyrenees breed dog in the dataset. The filenames start with an uppercase letter if the image is a cat, and a lowercase letter otherwise. We have to tell fastai how to get labels from the filenames, which we do by calling `from_name_func` (which means that labels can be extracted using a function applied to the filename) and passing `is_cat`, which returns `x[0].isupper()`, which evaluates to `True` if the first letter is uppercase (i.e., it's a cat).

The most important parameter to mention here is `valid_pct=0.2`. This tells fastai to hold out 20% of the data and *not use it for training the model at all*. This 20% of the data is called the *validation set*; the remaining 80% is called the *training set*. The validation set is used to measure the accuracy of the model. By default, the 20% that is held out is selected randomly. The parameter `seed=42` sets the *random seed* to the same value every time we run this code, which means we get the same validation set every time we run it—this way, if we change our model and retrain it, we know that any differences are due to the changes to the model, not due to having a different random validation set.

fastai will *always* show you your model's accuracy using *only* the validation set, *never* the training set. This is absolutely critical, because if you train a large enough model for a long enough time, it will eventually memorize the label of every item in your dataset! The result will not be a useful model, because what we care about is how well our model works on *previously unseen images*. That is always our goal when creating a model: for it to be useful on data that the model sees only in the future, after it has been trained.

Even when your model has not fully memorized all your data, earlier on in training it may have memorized certain parts of it. As a result, the longer you train for, the better your accuracy will get on the training set; the validation set accuracy will also improve for a while, but eventually it will start getting worse as the model starts to memorize the training set rather than finding generalizable underlying patterns in the data. When this happens, we say that the model is *overfitting*.

Figure 1-9 shows what happens when you overfit, using a simplified example where we have just one parameter and some randomly generated data based on the function `x**2`. As you see, although the predictions in the overfit model are accurate for data near the observed data points, they are way off when outside of that range.

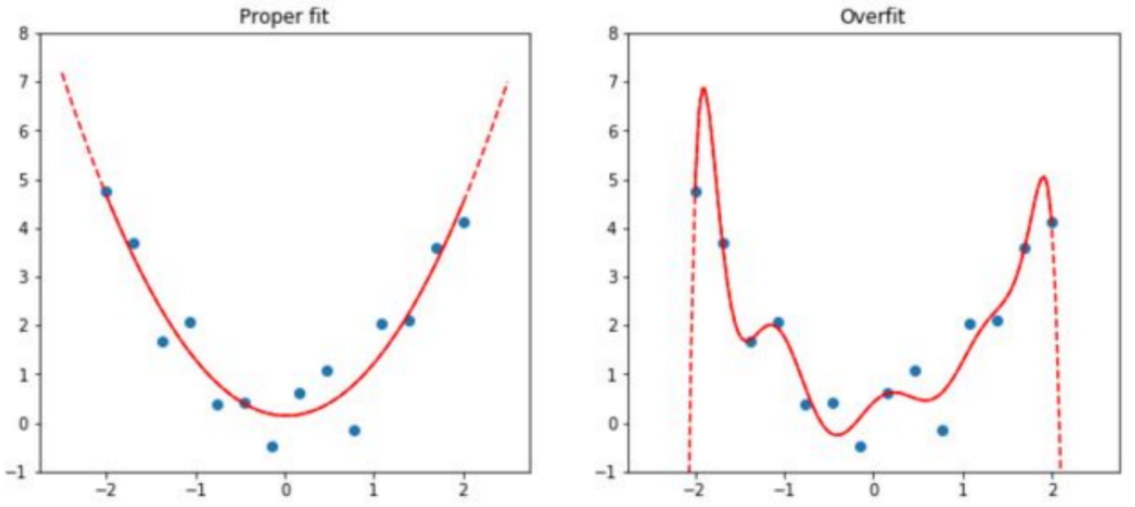


Figure 1-9. Example of overfitting

Overfitting is the single most important and challenging issue when training for all machine learning practitioners, and all algorithms. As you will see, it is easy to create a model that does a great job at making predictions on the exact data it has been trained on, but it is much harder to make accurate predictions on data the model has never seen before. And of course, this is the data that will matter in practice. For instance, if you create a handwritten digit classifier (as we will soon!) and use it to recognize numbers written on checks, then you are never going to see any of the numbers that the model was trained on—every check will have slightly different variations of writing to deal with.

You will learn many methods to avoid overfitting in this book. However, you should use those methods only after you have confirmed that overfitting is occurring (i.e., if you have observed the validation accuracy getting worse during training). We often see practitioners using overfitting avoidance techniques even when they have enough data that they didn't need to do so, ending up with a model that may be less accurate than what they could have achieved.

Validation Set

When you train a model, you must *always* have both a training set and a validation set, and you must measure the accuracy of your model only on the validation set. If you train for too long, with not enough data, you will see the accuracy of your model start to get worse; this is called *overfitting*. fastai defaults `valid_pct` to `0.2`, so even if you forget, fastai will create a validation set for you!

The fifth line of the code training our image recognizer tells fastai to create a *convolutional neural network* (CNN) and specifies what *architecture* to use (i.e., what kind

of model to create), what data we want to train it on, and what *metric* to use:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
```

Why a CNN? It's the current state-of-the-art approach to creating computer vision models. We'll be learning all about how CNNs work in this book. Their structure is inspired by how the human vision system works.

There are many architectures in fastai, which we will introduce in this book (as well as discussing how to create your own). Most of the time, however, picking an architecture isn't a very important part of the deep learning process. It's something that academics love to talk about, but in practice it is unlikely to be something you need to spend much time on. There are some standard architectures that work most of the time, and in this case we're using one called *ResNet* that we'll be talking a lot about in the book; it is both fast and accurate for many datasets and problems. The `34` in `resnet34` refers to the number of layers in this variant of the architecture (other options are `18`, `50`, `101`, and `152`). Models using architectures with more layers take longer to train and are more prone to overfitting (i.e., you can't train them for as many epochs before the accuracy on the validation set starts getting worse). On the other hand, when using more data, they can be quite a bit more accurate.

What is a metric? A *metric* is a function that measures the quality of the model's predictions using the validation set, and will be printed at the end of each epoch. In this case, we're using `error_rate`, which is a function provided by fastai that does just what it says: tells you what percentage of images in the validation set are being classified incorrectly. Another common metric for classification is `accuracy` (which is just `1.0 - error_rate`). fastai provides many more, which will be discussed throughout this book.

The concept of a metric may remind you of *loss*, but there is an important distinction. The entire purpose of loss is to define a "measure of performance" that the training system can use to update weights automatically. In other words, a good choice for loss is a choice that is easy for stochastic gradient descent to use. But a metric is defined for human consumption, so a good metric is one that is easy for you to understand, and that hews as closely as possible to what you want the model to do. At times, you might decide that the loss function is a suitable metric, but that is not necessarily the case.

`cnn_learner` also has a parameter `pretrained`, which defaults to `True` (so it's used in this case, even though we haven't specified it), which sets the weights in your model to values that have already been trained by experts to recognize a thousand different

categories across 1.3 million photos (using the famous *ImageNet* dataset). A model that has weights that have already been trained on another dataset is called a *pretrained model*. You should nearly always use a pretrained model, because it means that your model, before you've even shown it any of your data, is already very capable. And as you'll see, in a deep learning model, many of these capabilities are things you'll need, almost regardless of the details of your project. For instance, parts of pretrained models will handle edge, gradient, and color detection, which are needed for many tasks.

When using a pretrained model, `cnn_learner` will remove the last layer, since that is always specifically customized to the original training task (i.e., ImageNet dataset classification), and replace it with one or more new layers with randomized weights, of an appropriate size for the dataset you are working with. This last part of the model is known as the *head*.

Using pretrained models is the *most* important method we have to allow us to train more accurate models, more quickly, with less data and less time and money. You might think that would mean that using pretrained models would be the most studied area in academic deep learning...but you'd be very, very wrong! The importance of pretrained models is generally not recognized or discussed in most courses, books, or software library features, and is rarely considered in academic papers. As we write this at the start of 2020, things are just starting to change, but it's likely to take a while. So be careful: most people you speak to will probably greatly underestimate what you can do in deep learning with few resources, because they probably won't deeply understand how to use pretrained models.

Using a pretrained model for a task different from what it was originally trained for is known as *transfer learning*. Unfortunately, because transfer learning is so under-studied, few domains have pretrained models available. For instance, few pretrained models are currently available in medicine, making transfer learning challenging to use in that domain. In addition, it is not yet well understood how to use transfer learning for tasks such as time series analysis.

Jargon: Transfer Learning

Using a pretrained model for a task different from what it was originally trained for.

The sixth line of our code tells fastai how to *fit* the model:

```
learn.fine_tune(1)
```

As we've discussed, the architecture only describes a *template* for a mathematical function; it doesn't actually do anything until we provide values for the millions of

parameters it contains.

This is the key to deep learning—determining how to fit the parameters of a model to get it to solve your problem. To fit a model, we have to provide at least one piece of information: how many times to look at each image (known as number of *epochs*). The number of epochs you select will largely depend on how much time you have available, and how long you find it takes in practice to fit your model. If you select a number that is too small, you can always train for more epochs later.

But why is the method called `fine_tune`, and not `fit`? fastai *does* have a method called `fit`, which does indeed fit a model (i.e., look at images in the training set multiple times, each time updating the parameters to make the predictions closer and closer to the target labels). But in this case, we've started with a pretrained model, and we don't want to throw away all those capabilities that it already has. As you'll learn in this book, there are some important tricks to adapt a pretrained model for a new dataset—a process called *fine-tuning*.

Jargon: Fine-Tuning

A transfer learning technique that updates the parameters of a pretrained model by training for additional epochs using a different task from that used for pretraining.

When you use the `fine_tune` method, fastai will use these tricks for you. There are a few parameters you can set (which we'll discuss later), but in the default form shown here, it does two steps:

1. Use one epoch to fit just those parts of the model necessary to get the new random head to work correctly with your dataset.
2. Use the number of epochs requested when calling the method to fit the entire model, updating the weights of the later layers (especially the head) faster than the earlier layers (which, as we'll see, generally don't require many changes from the pretrained weights).

The *head* of a model is the part that is newly added to be specific to the new dataset. An *epoch* is one complete pass through the dataset. After calling `fit`, the results after each epoch are printed, showing the epoch number, the training and validation set losses (the "measure of performance" used for training the model), and any *metrics* you've requested (error rate, in this case).

So, with all this code, our model learned to recognize cats and dogs just from labeled examples. But how did it do it?

What Our Image Recognizer Learned

At this stage, we have an image recognizer that is working well, but we have no idea what it is doing! Although many people complain that deep learning results in impenetrable "black box" models (that is, something that gives predictions but that no one can understand), this really couldn't be further from the truth. There is a vast body of research showing how to deeply inspect deep learning models and get rich insights from them. Having said that, all kinds of machine learning models (including deep learning and traditional statistical models) can be challenging to fully understand, especially when considering how they will behave when coming across data that is very different from the data used to train them. We'll be discussing this issue throughout this book.

In 2013, PhD student Matt Zeiler and his supervisor, Rob Fergus, published "Visualizing and Understanding Convolutional Networks", which showed how to visualize the neural network weights learned in each layer of a model. They carefully analyzed the model that won the 2012 ImageNet competition, and used this analysis to greatly improve the model, such that they were able to go on to win the 2013 competition! Figure 1-10 is the picture that they published of the first layer's weights.

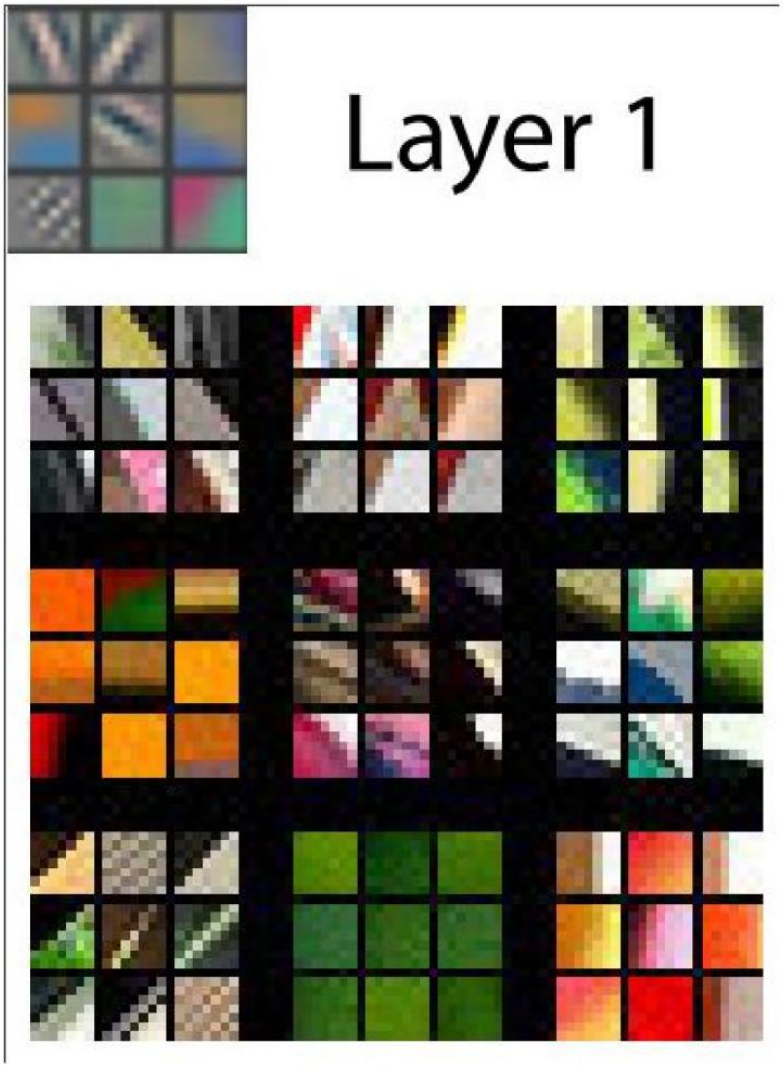


Figure 1-10. Activations of the first layer of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

This picture requires some explanation. For each layer, the image part with the light gray background shows the reconstructed weights, and the larger section at the bottom shows the parts of the training images that most strongly matched each set of weights. For layer 1, what we can see is that the model has discovered weights that represent diagonal, horizontal, and vertical edges, as well as various gradients. (Note that for each layer, only a subset of the features is shown; in practice there are thousands across all of the layers.)

These are the basic building blocks that the model has learned for computer vision. They have been widely analyzed by neuroscientists and computer vision researchers, and it turns out that these learned building blocks are very similar to the basic visual machinery in the human eye, as well as the handcrafted computer vision features that

were developed prior to the days of deep learning. The next layer is represented in Figure 1-11.

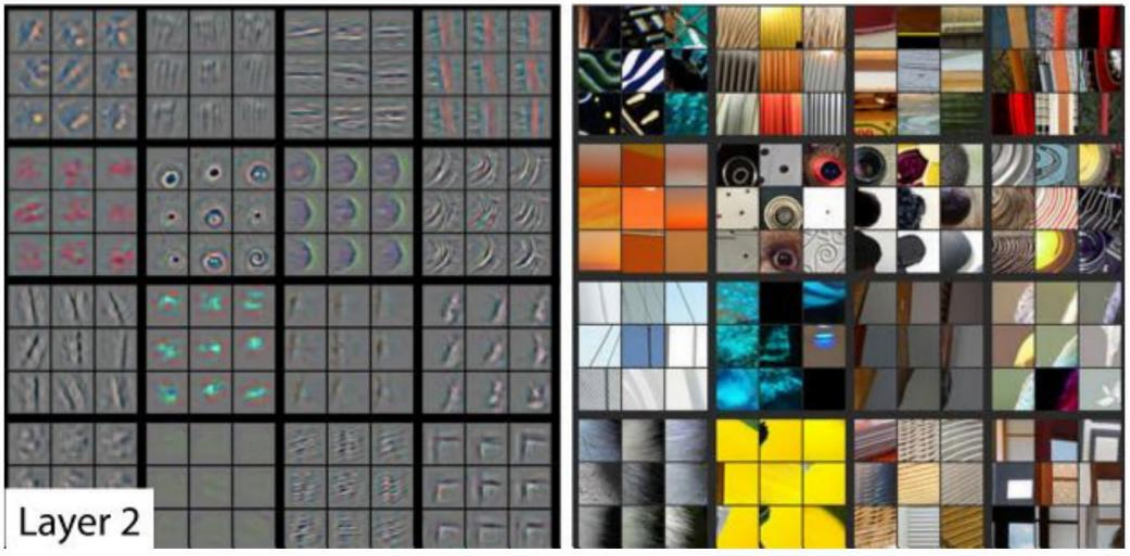


Figure 1-11. Activations of the second layer of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

For layer 2, there are nine examples of weight reconstructions for each of the features found by the model. We can see that the model has learned to create feature detectors that look for corners, repeating lines, circles, and other simple patterns. These are built from the basic building blocks developed in the first layer. For each of these, the righthand side of the picture shows small patches from actual images that these features most closely match. For instance, the particular pattern in row 2, column 1 matches the gradients and textures associated with sunsets.

Figure 1-12 shows the image from the paper showing the results of reconstructing the features of layer 3.

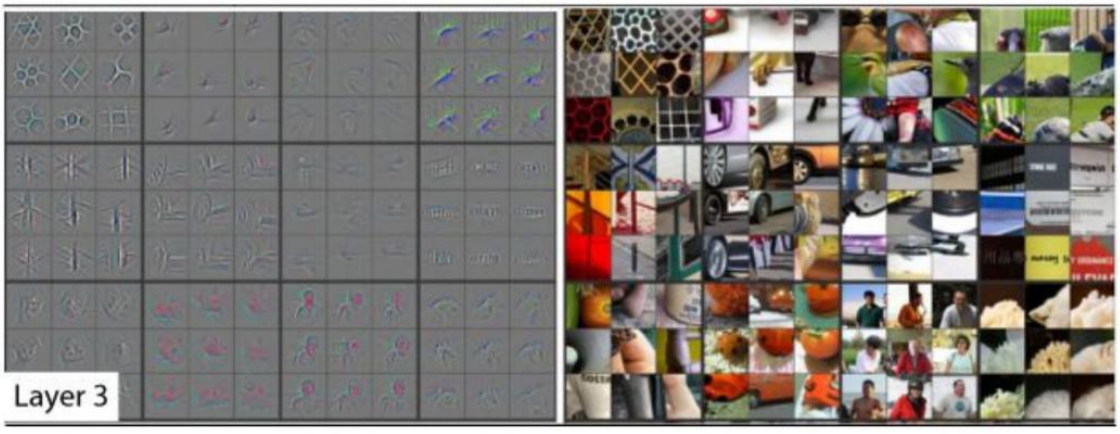


Figure 1-12. Activations of the third layer of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

As you can see by looking at the righthand side of this picture, the features are now able to identify and match with higher-level semantic components, such as car wheels, text, and flower petals. Using these components, layers 4 and 5 can identify even higher-level concepts, as shown in Figure 1-13.

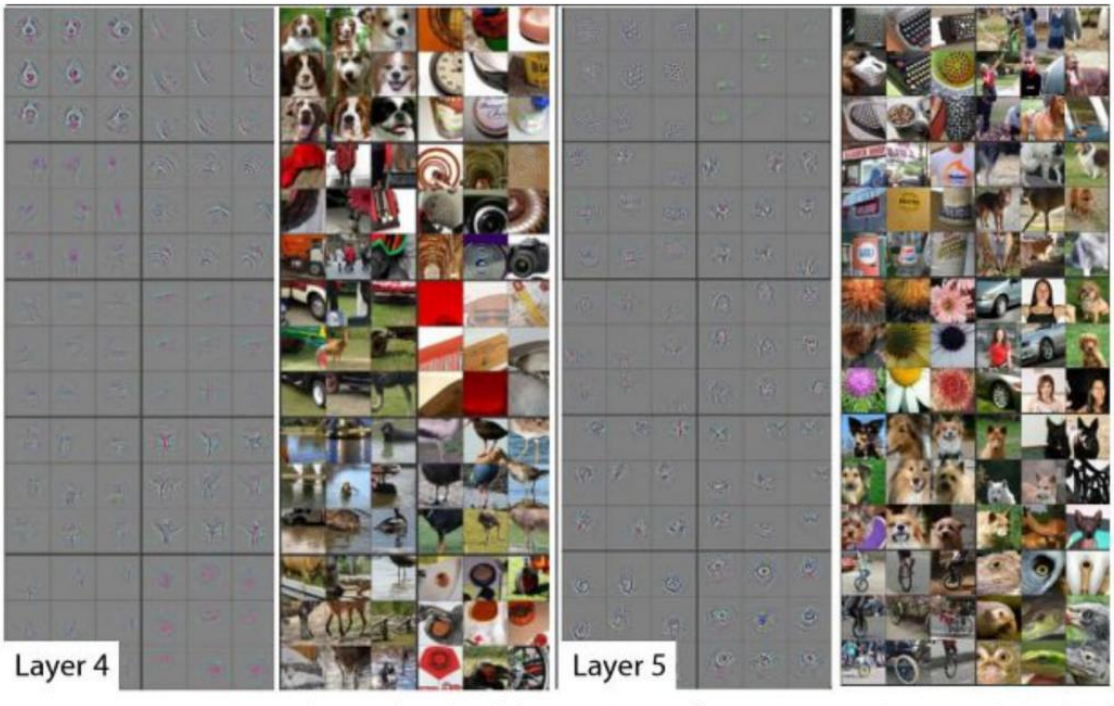


Figure 1-13. Activations of the fourth and fifth layers of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

This article was studying an older model called *AlexNet* that contained only five layers. Networks developed since then can have hundreds of layers—so you can imagine how

rich the features developed by these models can be!

When we fine-tuned our pretrained model earlier, we adapted what those last layers focus on (flowers, humans, animals) to specialize on the cats versus dogs problem. More generally, we could specialize such a pretrained model on many different tasks. Let's have a look at some examples.

Image Recognizers Can Tackle Non-Image Tasks

An image recognizer can, as its name suggests, only recognize images. But a lot of things can be represented as images, which means that an image recognizer can learn to complete many tasks.

For instance, a sound can be converted to a spectrogram, which is a chart that shows the amount of each frequency at each time in an audio file. Fast.ai student Ethan Sutin used this approach to easily beat the published accuracy of a state-of-the-art environmental sound detection model using a dataset of 8,732 urban sounds. fastai's `show_batch` clearly shows how each sound has a quite distinctive spectrogram, as you can see in Figure 1-14.

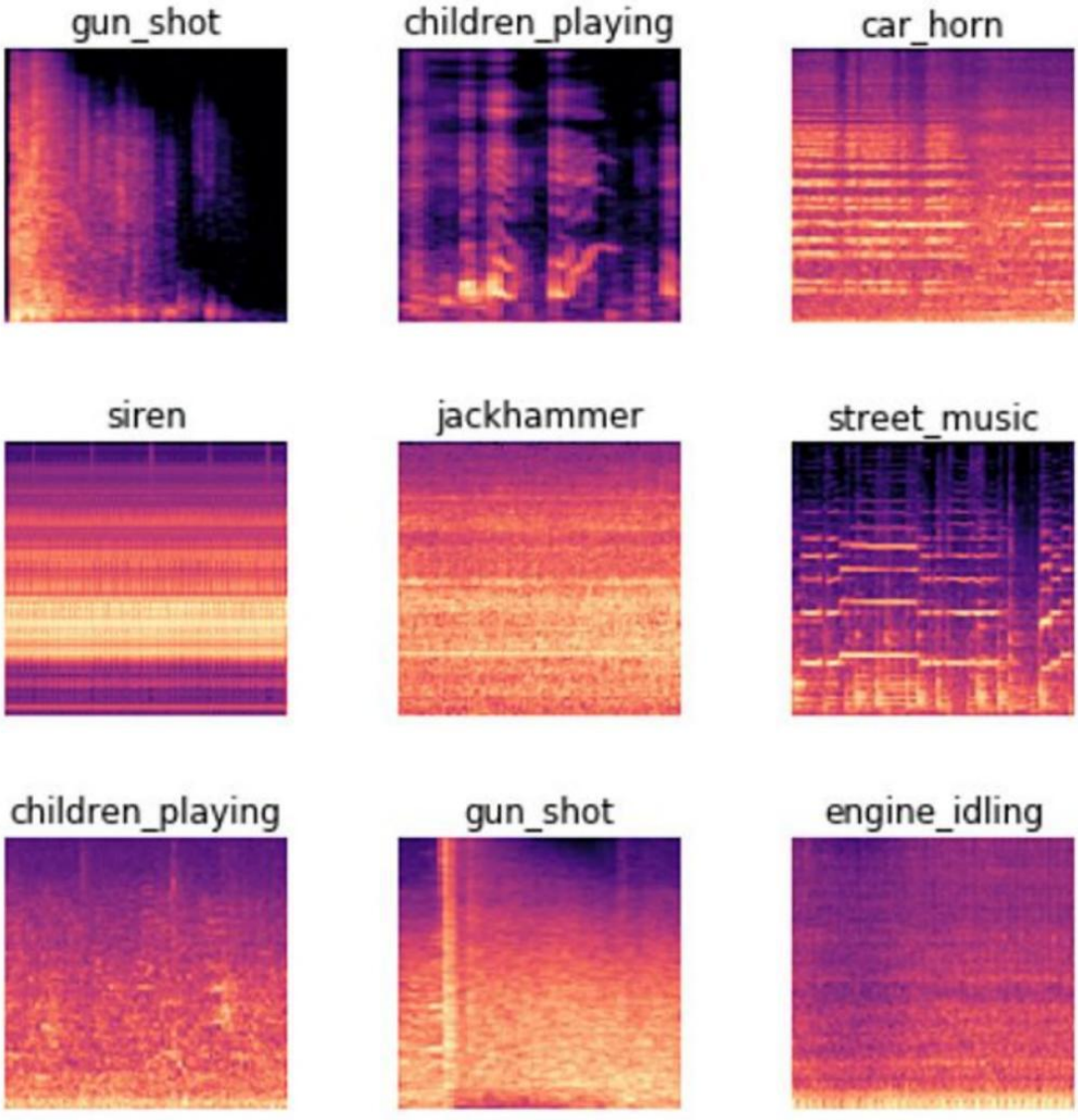


Figure 1-14. show_batch with spectrograms of sounds

A time series can easily be converted into an image by simply plotting the time series on a graph. However, it is often a good idea to try to represent your data in a way that makes it as easy as possible to pull out the most important components. In a time series, things like seasonality and anomalies are most likely to be of interest.

Various transformations are available for time series data. For instance, fast.ai student Ignacio Oguiza created images from a time series dataset for olive oil classification, using a technique called Gramian Angular Difference Field (GADF); you can see the result in Figure 1-15. He then fed those images to an image classification model just

like the one you see in this chapter. His results, despite having only 30 training set images, were well over 90% accurate, and close to the state of the art.

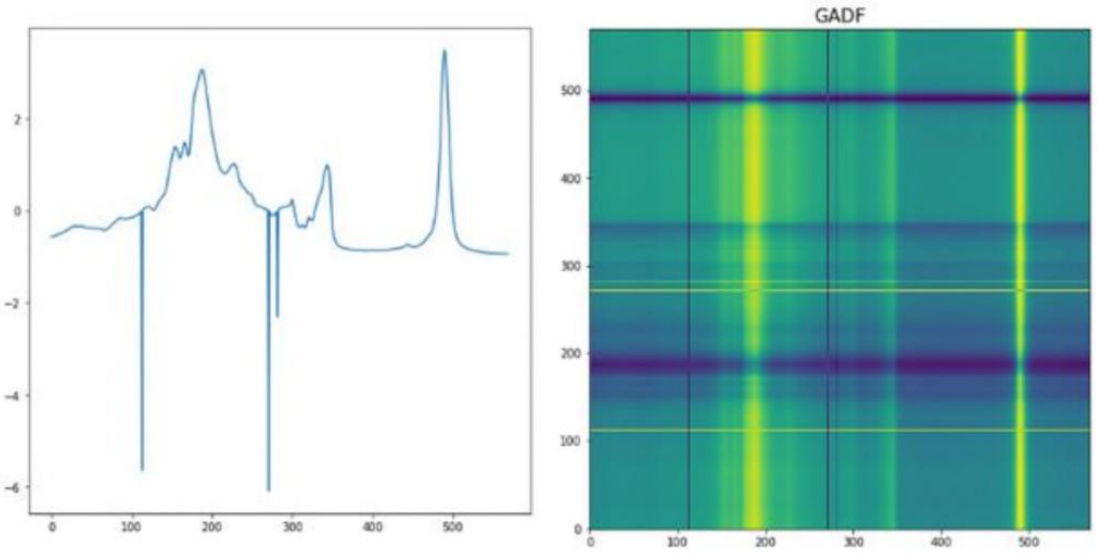


Figure 1-15. Converting a time series into an image

Another interesting fast.ai student project example comes from Gleb Esman. He was working on fraud detection at Splunk, using a dataset of users' mouse movements and mouse clicks. He turned these into pictures by drawing an image displaying the position, speed, and acceleration of the mouse pointer by using colored lines, and the clicks were displayed using small colored circles, as shown in Figure 1-16. He fed this into an image recognition model just like the one we've used in this chapter, and it worked so well that it led to a patent for this approach to fraud analytics!

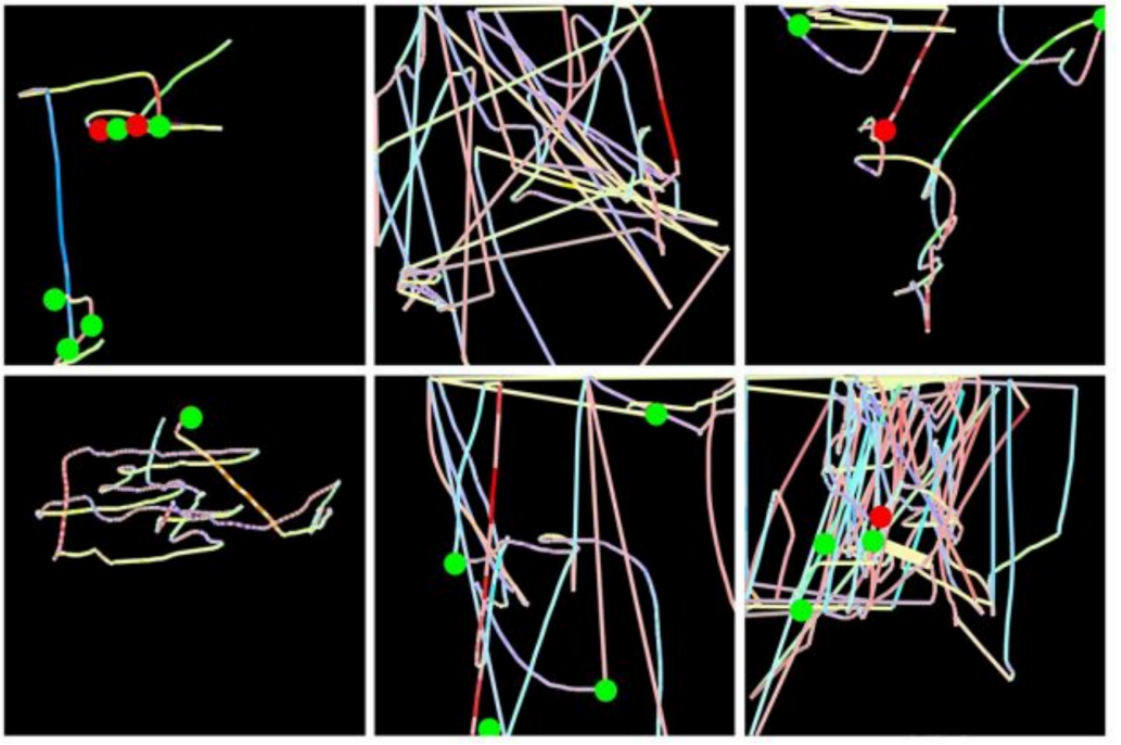


Figure 1-16. Converting computer mouse behavior to an image

Another example comes from the paper "Malware Classification with Deep Convolutional Neural Networks" by Mahmoud Kalash et al., which explains that "the malware binary file is divided into 8-bit sequences which are then converted to equivalent decimal values. This decimal vector is reshaped and [a] gray-scale image is generated that represent[s] the malware sample," in Figure 1-17.

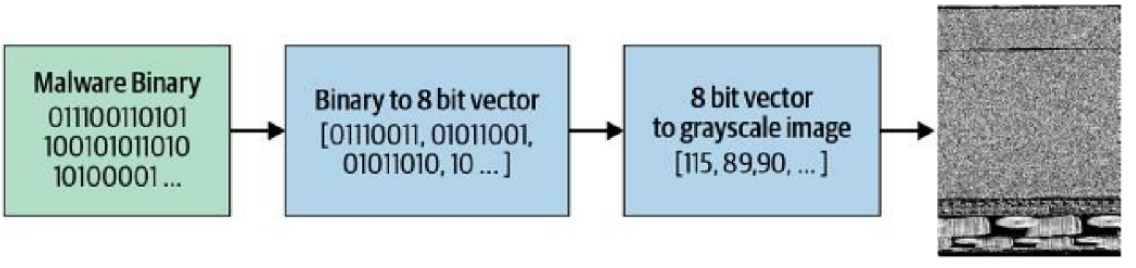


Figure 1-17. Malware classification process

The authors then show "pictures" generated through this process of malware in different categories, as shown in Figure 1-18.

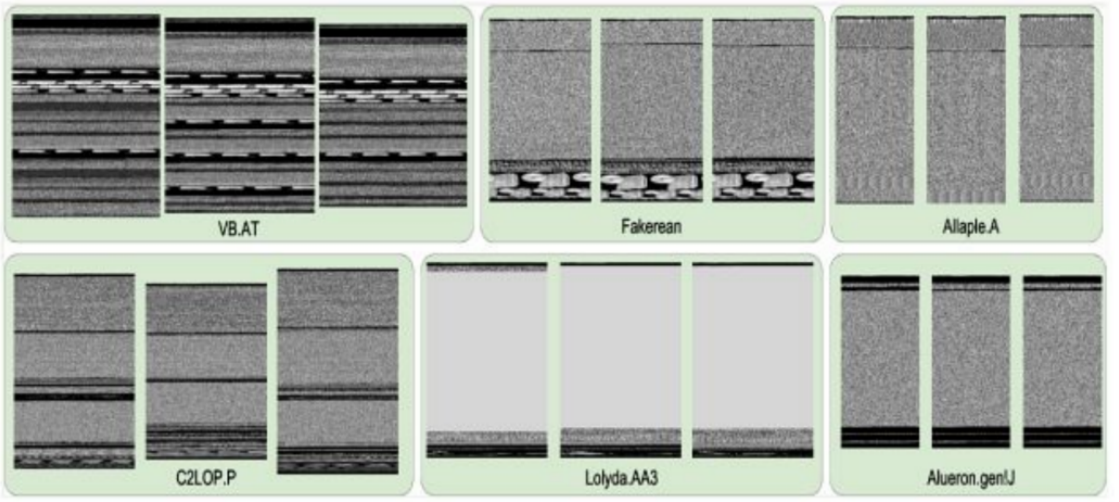


Figure 1-18. Malware examples

As you can see, the different types of malware look very distinctive to the human eye. The model the researchers trained based on this image representation was more accurate at malware classification than any previous approach shown in the academic literature. This suggests a good rule of thumb for converting a dataset into an image representation: if the human eye can recognize categories from the images, then a deep learning model should be able to do so too.

In general, you'll find that a small number of general approaches in deep learning can go a long way, if you're a bit creative in how you represent your data! You shouldn't think of approaches like the ones described here as "hacky workarounds," because they often (as here) beat previously state-of-the-art results. These really are the right ways to think about these problem domains.

Jargon Recap

We just covered a lot of information, so let's recap briefly. Table 1-3 provides a handy vocabulary list.

Table 1-3. Deep learning vocabulary

Term	Meaning
Label	The data that we're trying to predict, such as "dog" or "cat"
Architecture	The *template* of the model that we're trying to fit; i.e., the actual mathematical function that we're passing the input data and parameters to
Model	The combination of the architecture with a particular set of parameters
Parameters	The values in the model that change what task it can do and that are updated through model training
Fit	Update the parameters of the model such that the predictions of the model using the input data match the target labels
Train	A synonym for *fit*
Pretrained model	A model that has already been trained, generally using a large dataset, and will be fine-tuned
Fine-tune	Update a pretrained model for a different task
Epoch	One complete pass through the input data
Loss	A measure of how good the model is, chosen to drive training via SGD
Metric	A measurement of how good the model is using the validation set, chosen for human consumption
Validation set	A set of data held out from training, used only for measuring how good the model is
Training set	The data used for fitting the model; does not include any data from the validation set
Overfitting	Training a model in such a way that it *remembers* specific features of the input data, rather than generalizing well to data not seen during training
CNN	Convolutional neural network; a type of neural network that works particularly well for computer vision tasks

With this vocabulary in hand, we are now in a position to bring together all the key concepts introduced so far. Take a moment to review those definitions and read the following summary. If you can follow the explanation, you're well equipped to understand the discussions to come.

Machine learning is a discipline in which we define a program not by writing it entirely ourselves, but by learning from data. *Deep learning* is a specialty within machine learning that uses *neural networks* with multiple *layers*. *Image classification* is a representative example (also known as *image recognition*). We start with *labeled data*—a set of images for which we have assigned a *label* to each image, indicating what it represents. Our goal is to produce a program, called a *model*, that, given a new image, will make an accurate *prediction* regarding what that new image represents.

Every model starts with a choice of *architecture*, a general template for how that kind of model works internally. The process of *training* (or *fitting*) the model is the process of finding a set of *parameter values* (or *weights*) that specialize that general architecture into a model that works well for our particular kind of data. To define how well a model does on a single prediction, we need to define a *loss function*, which determines how we score a prediction as good or bad.

To make the training process go faster, we might start with a *pretrained model*—a model that has already been trained on someone else's data. We can then adapt it to our data by training it a bit more on our data, a process called *fine-tuning*.

When we train a model, a key concern is to ensure that our model *generalizes*: it learns general lessons from our data that also apply to new items it will encounter, so it can make good predictions on those items. The risk is that if we train our model badly, instead of learning general lessons, it effectively memorizes what it has already seen, and then it will make poor predictions about new images. Such a failure is called *overfitting*.

To avoid this, we always divide our data into two parts, the *training set* and the *validation set*. We train the model by showing it only the training set, and then we evaluate how well the model is doing by seeing how well it performs on items from the validation set. In this way, we check if the lessons the model learns from the training set are lessons that generalize to the validation set. In order for a person to assess how well the model is doing on the validation set overall, we define a *metric*. During the training process, when the model has seen every item in the training set, we call that an *epoch*.

All these concepts apply to machine learning in general. They apply to all sorts of schemes for defining a model by training it with data. What makes deep learning distinctive is a particular class of architectures: the architectures based on *neural networks*. In particular, tasks like image classification rely heavily on *convolutional neural networks*, which we will discuss shortly.

Deep Learning Is Not Just for Image Classification

Deep learning's effectiveness for classifying images has been widely discussed in recent years, even showing *superhuman* results on complex tasks like recognizing malignant tumors in CT scans. But it can do a lot more than this, as we will show here.

For instance, let's talk about something that is critically important for autonomous vehicles: localizing objects in a picture. If a self-driving car doesn't know where a pedestrian is, then it doesn't know how to avoid one! Creating a model that can recognize the content of every individual pixel in an image is called *segmentation*. Here is how we can train a segmentation model with fastai, using a subset of the *CamVid* dataset from the paper "Semantic Object Classes in Video: A High-Definition Ground Truth Database" by Gabriel J. Brostow et al.:

```
path = untar_data(URLs.CAMVID_TINY)
dls = SegmentationDataLoaders.from_label_func(
    path, bs=8, fnames = get_image_files(path/"images"),
    label_func = lambda o: path/'labels'/f'{o.stem}_P{o.suffix}',
    codes = np.loadtxt(path/'codes.txt', dtype=str)
)

learn = unet_learner(dls, resnet34)
learn.fine_tune(8)
```

epoch	train_loss	valid_loss	time
0	2.906601	2.347491	00:02

epoch	train_loss	valid_loss	time
0	1.988776	1.765969	00:02
1	1.703356	1.265247	00:02
2	1.591550	1.309860	00:02
3	1.459745	1.102660	00:02
4	1.324229	0.948472	00:02
5	1.205859	0.894631	00:02
6	1.102528	0.809563	00:02
7	1.020853	0.805135	00:02

We are not even going to walk through this code line by line, because it is nearly identical to our previous example! (We will be doing a deep dive into segmentation models in Chapter 15, along with all of the other models that we are briefly introducing in this chapter and many, many more.)

We can visualize how well it achieved its task by asking the model to color-code each

pixel of an image. As you can see, it nearly perfectly classifies every pixel in every object. For instance, notice that all of the cars are overlaid with the same color, and all of the trees are overlaid with the same color (in each pair of images, the lefthand image is the ground truth label, and the right is the prediction from the model):

```
learn.show_results(max_n=6, figsize=(7,8))
```

One other area where deep learning has dramatically improved in the last couple of years is natural language processing (NLP). Computers can now generate text, translate automatically from one language to another, analyze comments, label words in sentences, and much more. Here is all of the code necessary to train a model that can classify the sentiment of a movie review better than anything that existed in the world just five years ago:

```
from fastai.text.all import *

dls = TextDataLoaders.from_folder(untar_data(URLs.IMDB), valid='test')
learn = text_classifier_learner(dls, AWD_LSTM, drop_mult=0.5, metrics=accuracy)
learn.fine_tune(4, 1e-2)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.594912	0.407416	0.823640	01:35

epoch	train_loss	valid_loss	accuracy	time
0	0.268259	0.316242	0.876000	03:03
1	0.184861	0.246242	0.898080	03:10
2	0.136392	0.220086	0.918200	03:16
3	0.106423	0.191092	0.931360	03:15

This model is using the IMDb Large Movie Review dataset from "Learning Word Vectors for Sentiment Analysis" by Andrew Maas et al. It works well with movie reviews of many thousands of words, but let's test it on a short one to see how it works:

```
learn.predict("I really liked that movie!")
```

```
('pos', tensor(1), tensor([0.0041, 0.9959]))
```

Here we can see the model has considered the review to be positive. The second part of the result is the index of "pos" in our data vocabulary, and the last part is the probabilities attributed to each class (99.6% for "pos" and 0.4% for "neg").

Now it's your turn! Write your own mini movie review, or copy one from the internet, and you can see what this model thinks about it.

The Order Matters

In a Jupyter notebook, the order you execute each cell is important. It's not like Excel, where everything gets updated as soon as you type something anywhere—it has an inner state that gets updated each time you execute a cell. For instance, when you run the first cell of the notebook (with the "CLICK ME" comment), you create an object called `learn` that contains a model and data for an image classification problem.

If we were to run the cell just shown in the text (the one that predicts whether a review is good) straight after, we would get an error as this `learn` object does not contain a text classification model. This cell needs to be run after the one containing this:

```
from fastai.text.all import *

dls = TextDataLoaders.from_folder(untar_data(URLs.IMDB), valid='test')
learn = text_classifier_learner(dls, AWD_LSTM, drop_mult=0.5,
                                metrics=accuracy)
learn.fine_tune(4, 1e-2)
```

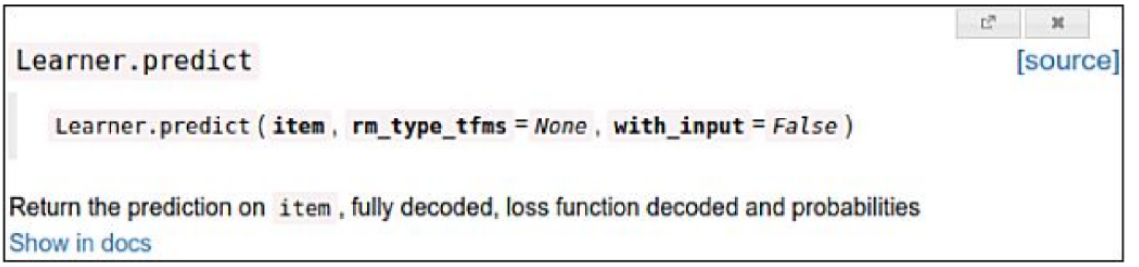
The outputs themselves can be deceiving, because they include the results of the last time the cell was executed; if you change the code inside a cell without executing it, the old (misleading) results will remain.

Except when we mention it explicitly, the notebooks provided on the book's website are meant to be run in order, from top to bottom. In general, when experimenting, you will find yourself executing cells in any order to go fast (which is a super neat feature of Jupyter Notebook), but once you have explored and arrived at the final version of your code, make sure you can run the cells of your notebooks in order (your future self won't necessarily remember the convoluted path you took otherwise!).

In command mode, typing `0` twice will restart the *kernel* (which is the engine powering your notebook). This will wipe your state clean and make it as if you had just started in the notebook. Choose Run All Above from the Cell menu to run all cells above the point where you are. We have found this to be useful when developing the fastai library.

If you ever have any questions about a fastai method, you should use the function `doc`, passing it the method name:

```
doc(learn.predict)
```



A window pops up containing a brief one-line explanation. The "Show in docs" link takes you to the full documentation, where you'll find all the details and lots of examples. Also, most of fastai's methods are just a handful of lines, so you can click the "source" link to see exactly what's going on behind the scenes.

Let's move on to something much less sexy, but perhaps significantly more widely commercially useful: building models from plain *tabular* data.

Jargon: Tabular

Data that is in the form of a table, such as from a spreadsheet, database, or a comma-separated values (CSV) file. A tabular model is a model that tries to predict one column of a table based on information in other columns of the table.

It turns out that looks very similar too. Here is the code necessary to train a model that will predict whether a person is a high-income earner, based on their socioeconomic background:

```
from fastai.tabular.all import *
path = untar_data(URLs.ADULT_SAMPLE)

dls = TabularDataLoaders.from_csv(path/'adult.csv', path=path, y_names="salary",
    cat_names = ['workclass', 'education', 'marital-status', 'occupation',
                 'relationship', 'race'],
    cont_names = ['age', 'fnlwgt', 'education-num'],
    procs = [Categorify, FillMissing, Normalize])

learn = tabular_learner(dls, metrics=accuracy)
```

As you see, we had to tell fastai which columns are *categorical* (contain values that are one of a discrete set of choices, such as `occupation`) versus *continuous* (contain a number that represents a quantity, such as `age`).

There is no pretrained model available for this task (in general, pretrained models are not widely available for any tabular modeling tasks, although some organizations have

created them for internal use), so we don't use `fine_tune` in this case. Instead, we use `fit_one_cycle`, the most commonly used method for training fastai models *from scratch* (i.e., without transfer learning):

```
learn.fit_one_cycle(3)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.359960	0.357917	0.831388	00:11
1	0.353458	0.349657	0.837991	00:10
2	0.338368	0.346997	0.843213	00:10

This model is using the *Adult* dataset from the paper "Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid" by Ron Kohavi, which contains some demographic data about individuals (like their education, marital status, race, sex and whether they have an annual income greater than $50k). The model is over 80% accurate and took around 30 seconds to train.

Let's look at one more. Recommendation systems are important, particularly in ecommerce. Companies like Amazon and Netflix try hard to recommend products or movies that users might like. Here's how to train a model that will predict movies people might like based on their previous viewing habits, using the MovieLens dataset:

```
from fastai.collab import *
path = untar_data(URLs.ML_SAMPLE)
dls = CollabDataLoaders.from_csv(path/'ratings.csv')
learn = collab_learner(dls, y_range=(0.5,5.5))
learn.fine_tune(10)
```

epoch	train_loss	valid_loss	time
0	1.554056	1.428071	00:01

epoch	train_loss	valid_loss	time
0	1.393103	1.361342	00:01
1	1.297930	1.159169	00:00
2	1.052705	0.827934	00:01
3	0.810124	0.668735	00:01
4	0.711552	0.627836	00:01
5	0.657402	0.611715	00:01
6	0.633079	0.605733	00:01
7	0.622399	0.602674	00:01
8	0.629075	0.601671	00:00
9	0.619955	0.601550	00:01

This model is predicting movie ratings on a scale of 0.5 to 5.0 to within around 0.6 average error. Since we're predicting a continuous number, rather than a category, we have to tell fastai what range our target has, using the `y_range` parameter.

Although we're not actually using a pretrained model (for the same reason that we didn't for the tabular model), this example shows that fastai lets us use `fine_tune` anyway in this case (you'll learn how and why this works in Chapter 5). Sometimes it's best to experiment with `fine_tune` versus `fit_one_cycle` to see which works best for your dataset.

We can use the same `show_results` call we saw earlier to view a few examples of user and movie IDs, actual ratings, and predictions:

```
learn.show_results()
```

	userId	movieId	rating	rating_pred
0	157	1200	4.0	3.558502
1	23	344	2.0	2.700709
2	19	1221	5.0	4.390801
3	430	592	3.5	3.944848
4	547	858	4.0	4.076881
5	292	39	4.5	3.753513
6	529	1265	4.0	3.349463
7	19	231	3.0	2.881087
8	475	4963	4.0	4.023387
9	130	260	4.5	3.979703

Datasets: Food for Models

You've already seen quite a few models in this section, each one trained using a different dataset to do a different task. In machine learning and deep learning, we can't do anything without data. So, the people who create datasets for us to train our models on are the (often underappreciated) heroes. Some of the most useful and important datasets are those that become important *academic baselines*—datasets that are widely studied by researchers and used to compare algorithmic changes. Some of these become household names (at least, among households that train models!), such as MNIST, CIFAR-10, and ImageNet.

The datasets used in this book have been selected because they provide great examples of the kinds of data that you are likely to encounter, and the academic literature has many examples of model results using these datasets to which you can compare your work.

Most datasets used in this book took the creators a lot of work to build. For instance, later in the book we'll be showing you how to create a model that can translate between French and English. The key input to this is a French/English parallel text corpus prepared in 2009 by Professor Chris Callison-Burch of the University of Pennsylvania. This dataset contains over 20 million sentence pairs in French and English. He built the dataset in a really clever way: by crawling millions of Canadian web pages (which are often multilingual) and then using a set of simple heuristics to transform URLs of French content to URLs pointing to the same content in English.

As you look at datasets throughout this book, think about where they might have come from and how they might have been curated. Then think about what kinds of interesting datasets you could create for your own projects. (We'll even take you step by step through the process of creating your own image dataset soon.)

fast.ai has spent a lot of time creating cut-down versions of popular datasets that are specially designed to support rapid prototyping and experimentation, and to be easier to learn with. In this book, we will often start by using one of the cut-down versions and later scale up to the full-size version (just as we're doing in this chapter!). This is how the world's top practitioners do their modeling in practice; they do most of their experimentation and prototyping with subsets of their data, and use the full dataset only when they have a good understanding of what they have to do.

Each of the models we trained showed a training and validation loss. A good validation

set is one of the most important pieces of the training process. Let's see why and learn how to create one.

Validation Sets and Test Sets

As we've discussed, the goal of a model is to make predictions about data. But the model training process is fundamentally dumb. If we trained a model with all our data and then evaluated the model using that same data, we would not be able to tell how well our model can perform on data it hasn't seen. Without this very valuable piece of information to guide us in training our model, there is a very good chance it would become good at making predictions about that data but would perform poorly on new data.

To avoid this, our first step was to split our dataset into two sets: the *training set* (which our model sees in training) and the *validation set*, also known as the *development set* (which is used only for evaluation). This lets us test that the model learns lessons from the training data that generalize to new data, the validation data.

One way to understand this situation is that, in a sense, we don't want our model to get good results by "cheating." If it makes an accurate prediction for a data item, that should be because it has learned characteristics of that kind of item, and not because the model has been shaped by *actually having seen that particular item*.

Splitting off our validation data means our model never sees it in training and so is completely untainted by it, and is not cheating in any way. Right?

In fact, not necessarily. The situation is more subtle. This is because in realistic scenarios we rarely build a model just by training its parameters once. Instead, we are likely to explore many versions of a model through various modeling choices regarding network architecture, learning rates, data augmentation strategies, and other factors we will discuss in upcoming chapters. Many of these choices can be described as choices of *hyperparameters*. The word reflects that they are parameters about parameters, since they are the higher-level choices that govern the meaning of the weight parameters.

The problem is that even though the ordinary training process is looking at only predictions on the training data when it learns values for the weight parameters, the same is not true of us. We, as modelers, are evaluating the model by looking at predictions on the validation data when we decide to explore new hyperparameter values! So subsequent versions of the model are, indirectly, shaped by us having seen the validation data. Just as the automatic training process is in danger of overfitting the training data, we are in danger of overfitting the validation data through human trial and error and exploration.

The solution to this conundrum is to introduce another level of even more highly reserved data: the *test set*. Just as we hold back the validation data from the training process, we must hold back the test set data even from ourselves. It cannot be used to improve the model; it can be used only to evaluate the model at the very end of our efforts. In effect, we define a hierarchy of cuts of our data, based on how fully we want to hide it from training and modeling processes: training data is fully exposed, the validation data is less exposed, and test data is totally hidden. This hierarchy parallels the different kinds of modeling and evaluation processes themselves—the automatic training process with backpropagation, the more manual process of trying different hyperparameters between training sessions, and the assessment of our final result.

The test and validation sets should have enough data to ensure that you get a good estimate of your accuracy. If you're creating a cat detector, for instance, you generally want at least 30 cats in your validation set. That means that if you have a dataset with thousands of items, using the default 20% validation set size may be more than you need. On the other hand, if you have lots of data, using some of it for validation probably doesn't have any downsides.

Having two levels of "reserved data"—a validation set and a test set, with one level representing data that you are virtually hiding from yourself—may seem a bit extreme. But it is often necessary because models tend to gravitate toward the simplest way to do good predictions (memorization), and we as fallible humans tend to gravitate toward fooling ourselves about how well our models are performing. The discipline of the test set helps us keep ourselves intellectually honest. That doesn't mean we *always* need a separate test set—if you have very little data, you may need just a validation set—but generally it's best to use one if at all possible.

This same discipline can be critical if you intend to hire a third party to perform modeling work on your behalf. A third party might not understand your requirements accurately, or their incentives might even encourage them to misunderstand them. A good test set can greatly mitigate these risks and let you evaluate whether their work solves your actual problem.

To put it bluntly, if you're a senior decision maker in your organization (or you're advising senior decision makers), the most important takeaway is this: if you ensure that you really understand what test and validation sets are and why they're important, you'll avoid the single biggest source of failures we've seen when organizations decide to use AI. For instance, if you're considering bringing in an external vendor or service, make sure that you hold out some test data that the vendor *never gets to see*. Then *you* check their model on your test data, using a metric that *you* choose based on what actually matters to you in practice, and *you* decide what level of

performance is adequate. (It's also a good idea for you to try out simple baseline yourself, so you know what a really simple model can achieve. Often it'll turn out that your simple model performs just as well as one produced by an external "expert"!)

Use Judgment in Defining Test Sets

To do a good job of defining a validation set (and possibly a test set), you will sometimes want to do more than just randomly grab a fraction of your original dataset. Remember: a key property of the validation and test sets is that they must be representative of the new data you will see in the future. This may sound like an impossible order! By definition, you haven't seen this data yet. But you usually still do know some things.

It's instructive to look at a few example cases. Many of these examples come from predictive modeling competitions on the *Kaggle* platform, which is a good representation of problems and methods you might see in practice.

One case might be if you are looking at time series data. For a time series, choosing a random subset of the data will be both too easy (you can look at the data both before and after the dates you are trying to predict) and not representative of most business use cases (where you are using historical data to build a model for use in the future). If your data includes the date and you are building a model to use in the future, you will want to choose a continuous section with the latest dates as your validation set (for instance, the last two weeks or last month of available data).

Suppose you want to split the time series data in Figure 1-19 into training and validation sets.

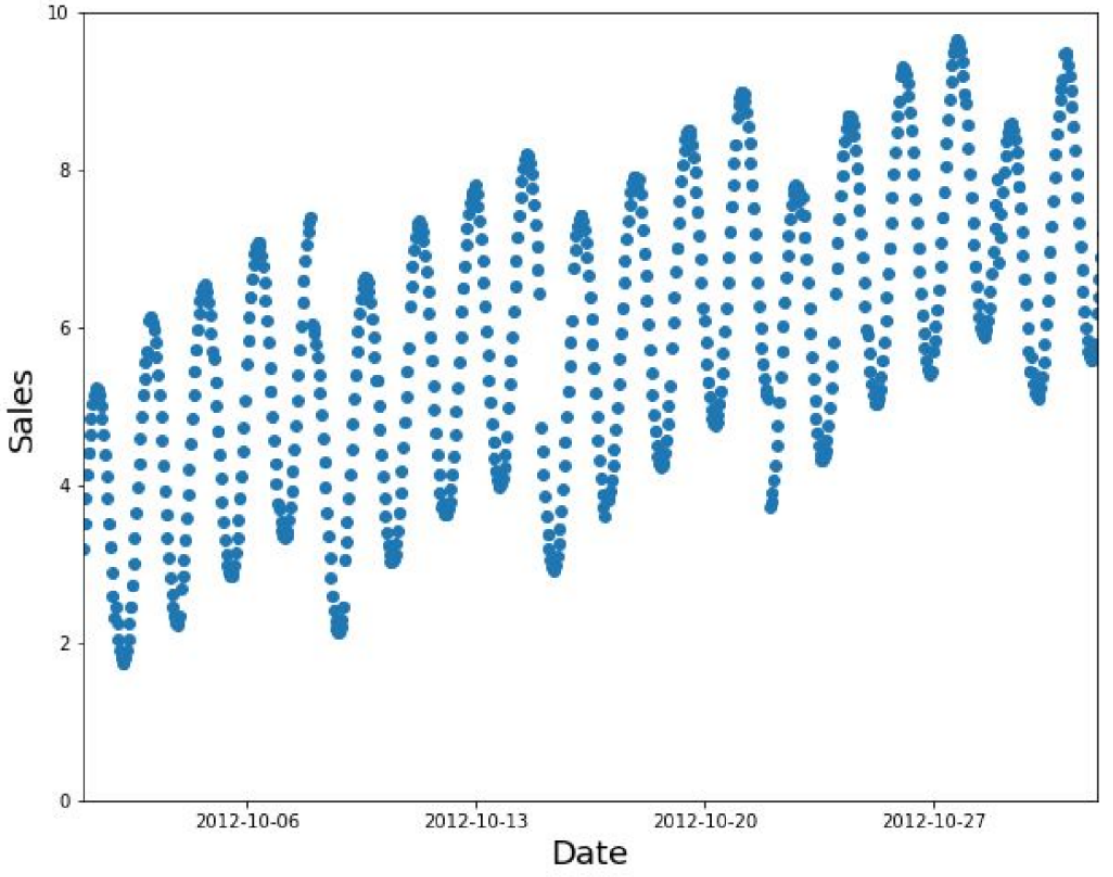


Figure 1-19. A time series

A random subset is a poor choice (too easy to fill in the gaps, and not indicative of what you'll need in production), as we can see in Figure 1-20.

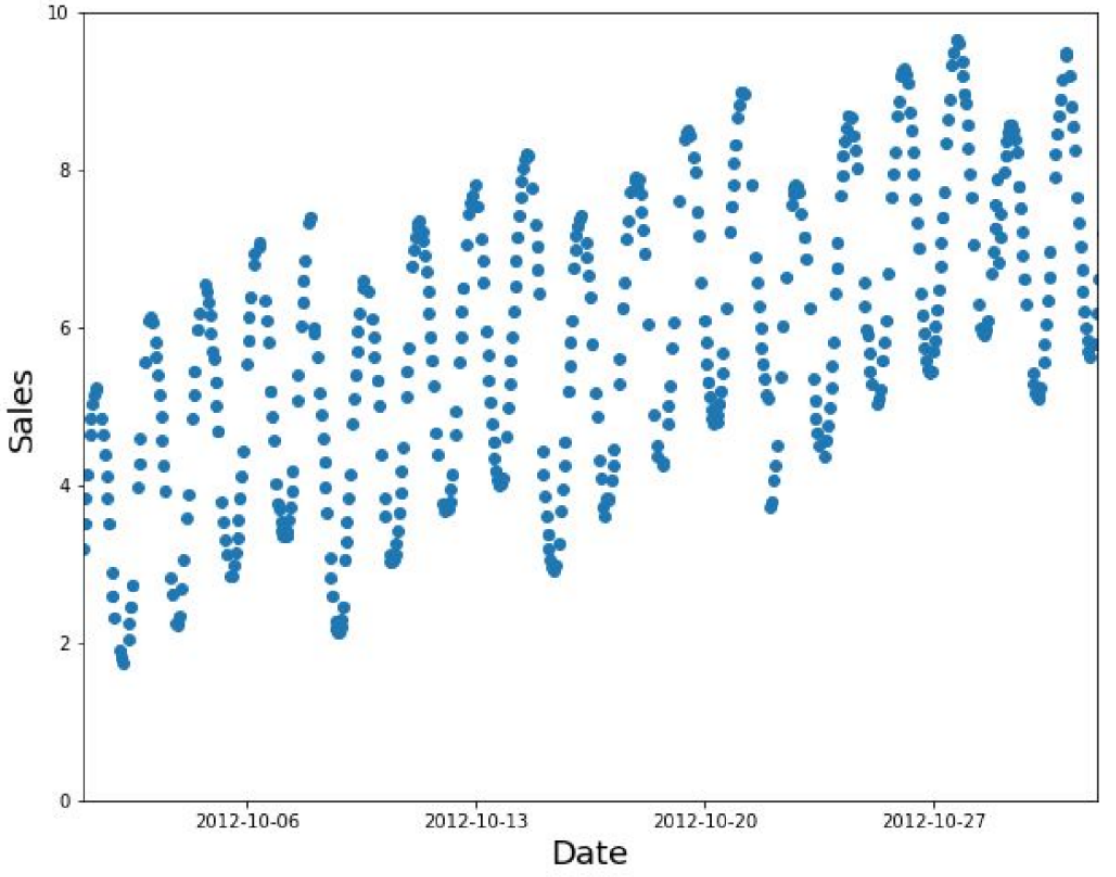


Figure 1-20. A poor training subset

Instead, use the earlier data as your training set (and the later data for the validation set), as shown in Figure 1-21.

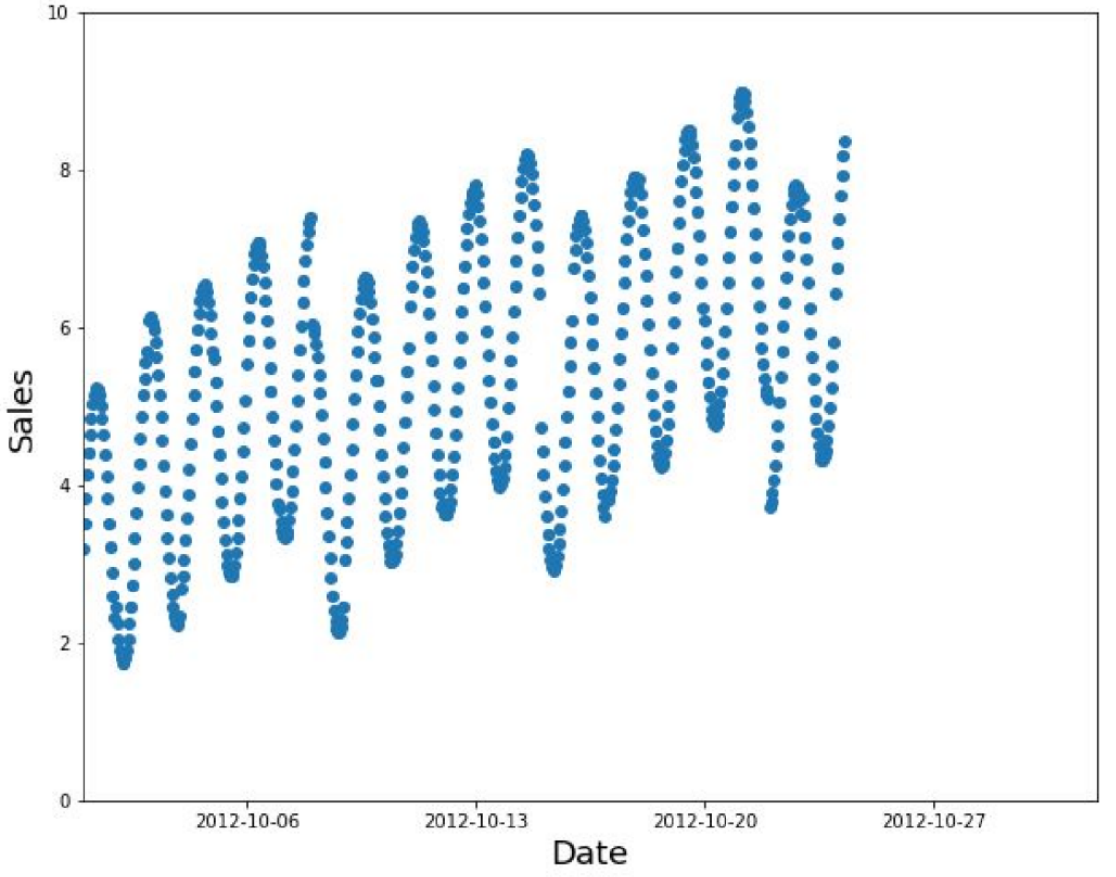


Figure 1-21. A good training subset

For example, Kaggle had a competition to predict the sales in a chain of Ecuadorian grocery stores. Kaggle's training data ran from Jan 1, 2013 to Aug 15, 2017, and the test data spanned from Aug 16, 2017 to Aug 31, 2017. That way, the competition organizer ensured that entrants were making predictions for a time period that was *in the future*, from the perspective of their model. This is similar to the way quantitative hedge fund traders do *backtesting* to check whether their models are predictive of future periods, based on past data.

A second common case occurs when you can easily anticipate ways the data you will be making predictions for in production may be *qualitatively different* from the data you have to train your model with.

In the Kaggle distracted driver competition, the independent variables are pictures of drivers at the wheel of a car, and the dependent variables are categories such as texting, eating, or safely looking ahead. Lots of pictures are of the same drivers in different positions, as we can see in Figure 1-22. If you were an insurance company

building a model from this data, note that you would be most interested in how the model performs on drivers it hasn't seen before (since you would likely have training data for only a small group of people). In recognition of this, the test data for the competition consists of images of people that don't appear in the training set.

Figure 1-22. Two pictures from the training data

If you put one of the images in Figure 1-22 in your training set and one in the validation set, your model will have an easy time making a prediction for the one in the validation set, so it will seem to be performing better than it would on new people. Another perspective is that if you used all the people in training your model, your model might be overfitting to particularities of those specific people and not just learning the states (texting, eating, etc.).

A similar dynamic was at work in the Kaggle fisheries competition to identify the species of fish caught by fishing boats in order to reduce illegal fishing of endangered populations. The test set consisted of images from boats that didn't appear in the training data, so in this case you'd want your validation set to also include boats that are not in the training set.

Sometimes it may not be clear how your validation data will differ. For instance, for a problem using satellite imagery, you'd need to gather more information on whether the training set contained just certain geographic locations or came from geographically scattered data.

Now that you have gotten a taste of how to build a model, you can decide what you want to dig into next.

A *Choose Your Own Adventure* Moment

If you would like to learn more about how to use deep learning models in practice, including how to identify and fix errors, create a real working web application, and avoid your model causing unexpected harm to your organization or society more

generally, then keep reading the next two chapters. If you would like to start learning the foundations of how deep learning works under the hood, skip to Chapter 4. (Did you ever read *Choose Your Own Adventure* books as a kid? Well, this is kind of like that... except with more deep learning than that book series contained.)

You will need to read all these chapters to progress further in the book, but the order in which you read them is totally up to you. They don't depend on each other. If you skip ahead to Chapter 4, we will remind you at the end to come back and read the chapters you skipped over before you go any further.

Questionnaire

After reading pages and pages of prose, it can be hard to know which key things you really need to focus on and remember. So, we've prepared a list of questions and suggested steps to complete at the end of each chapter. All the answers are in the text of the chapter, so if you're not sure about anything here, reread that part of the text and make sure you understand it. Answers to all these questions are also available on the book's website. You can also visit the forums if you get stuck to get help from other folks studying this material.

For more questions, including detailed answers and links to the video timeline, have a look at Radek Osmulski's aiquizzes.

1. Do you need these for deep learning?
    - Lots of math T/F
    - Lots of data T/F
    - Lots of expensive computers T/F
    - A PhD T/F
2. Name five areas where deep learning is now the best tool in the world.
3. What was the name of the first device that was based on the principle of the artificial neuron?
4. Based on the book of the same name, what are the requirements for parallel distributed processing (PDP)?
5. What were the two theoretical misunderstandings that held back the field of neural networks?
6. What is a GPU?

7. Open a notebook and execute a cell containing: `1+1`. What happens?
8. Follow through each cell of the stripped version of the notebook for this chapter. Before executing each cell, guess what will happen.
9. Complete the Jupyter Notebook online appendix.
10. Why is it hard to use a traditional computer program to recognize images in a photo?
11. What did Samuel mean by "weight assignment"?
12. What term do we normally use in deep learning for what Samuel called "weights"?
13. Draw a picture that summarizes Samuel's view of a machine learning model.
14. Why is it hard to understand why a deep learning model makes a particular prediction?
15. What is the name of the theorem that shows that a neural network can solve any mathematical problem to any level of accuracy?
16. What do you need in order to train a model?
17. How could a feedback loop impact the rollout of a predictive policing model?
18. Do we always have to use 224×224-pixel images with the cat recognition model?
19. What is the difference between classification and regression?
20. What is a validation set? What is a test set? Why do we need them?
21. What will fastai do if you don't provide a validation set?
22. Can we always use a random sample for a validation set? Why or why not?
23. What is overfitting? Provide an example.
24. What is a metric? How does it differ from loss?
25. How can pretrained models help?
26. What is the "head" of a model?
27. What kinds of features do the early layers of a CNN find? How about the later layers?

28. Are image models useful only for photos?
29. What is an architecture?
30. What is segmentation?
31. What is `y_range` used for? When do we need it?
32. What are hyperparameters?
33. What's the best way to avoid failures when using AI in an organization?

Further Research

Each chapter also has a "Further Research" section that poses some questions that aren't fully answered in the text, or gives more advanced assignments. Answers to these questions aren't on the book's website; you'll need to do your own research!

1. Why is a GPU useful for deep learning? How is a CPU different, and why is it less effective for deep learning?
2. Try to think of three areas where feedback loops might impact the use of machine learning. See if you can find documented examples of that happening in practice.

Chapter 2. From Model to Production

The six lines of code we saw in Chapter 1 are just one small part of the process of using deep learning in practice. In this chapter, we're going to use a computer vision example to look at the end-to-end process of creating a deep learning application. More specifically, we're going to build a bear classifier! In the process, we'll discuss the capabilities and constraints of deep learning, explore how to create datasets, look at possible gotchas when using deep learning in practice, and more. Many of the key points will apply equally well to other deep learning problems, such as those in Chapter 1. If you work through a problem similar in key respects to our example problems, we expect you to get excellent results with little code, quickly.

Let's start with how you should frame your problem.

The Practice of Deep Learning

We've seen that deep learning can solve a lot of challenging problems quickly and with little code. As a beginner, there's a sweet spot of problems that are similar enough to our example problems that you can very quickly get extremely useful results. However, deep learning isn't magic! The same six lines of code won't work for every problem anyone can think of today.

Underestimating the constraints and overestimating the capabilities of deep learning may lead to frustratingly poor results, at least until you gain some experience and can solve the problems that arise. Conversely, overestimating the constraints and underestimating the capabilities of deep learning may mean you do not attempt a solvable problem because you talk yourself out of it.

We often talk to people who underestimate both the constraints and the capabilities of deep learning. Both of these can be problems: underestimating the capabilities means that you might not even try things that could be very beneficial, and underestimating the constraints might mean that you fail to consider and react to important issues.

The best thing to do is to keep an open mind. If you remain open to the possibility that deep learning might solve part of your problem with less data or complexity than you expect, you can design a process through which you can find the specific capabilities and constraints related to your particular problem. This doesn't mean making any risky bets—we will show you how you can gradually roll out models so that they don't create significant risks, and can even backtest them prior to putting them in production.

Starting Your Project

So where should you start your deep learning journey? The most important thing is to ensure that you have a project to work on—it is only through working on your own

projects that you will get real experience building and using models. When selecting a project, the most important consideration is data availability.

Regardless of whether you are doing a project just for your own learning or for practical application in your organization, you want to be able to start quickly. We have seen many students, researchers, and industry practitioners waste months or years while they attempt to find their perfect dataset. The goal is not to find the "perfect" dataset or project, but just to get started and iterate from there. If you take this approach, you will be on your third iteration of learning and improving while the perfectionists are still in the planning stages!

We also suggest that you iterate from end to end in your project; don't spend months fine-tuning your model, or polishing the perfect GUI, or labeling the perfect dataset.... Instead, complete every step as well as you can in a reasonable amount of time, all the way to the end. For instance, if your final goal is an application that runs on a mobile phone, that should be what you have after each iteration. But perhaps in the early iterations you take shortcuts; for instance, by doing all of the processing on a remote server and using a simple responsive web application. By completing the project end to end, you will see where the trickiest bits are, and which bits make the biggest difference to the final result.

As you work through this book, we suggest that you complete lots of small experiments, by running and adjusting the notebooks we provide, at the same time that you gradually develop your own projects. That way, you will be getting experience with all of the tools and techniques that we're explaining as we discuss them.

Sylvain Says

To make the most of this book, take the time to experiment between each chapter, whether on your own project or by exploring the notebooks we provide. Then try rewriting those notebooks from scratch on a new dataset. It's only by practicing (and failing) a lot that you will develop intuition of how to train a model.

By using the end-to-end iteration approach, you will also get a better understanding of how much data you really need. For instance, you may find you can easily get only 200 labeled data items, and you can't really know until you try whether that's enough to get the performance you need for your application to work well in practice.

In an organizational context, you will be able to show your colleagues that your idea can work by showing them a real working prototype. We have repeatedly observed that this is the secret to getting good organizational buy-in for a project.

Since it is easiest to get started on a project for which you already have data available, that means it's probably easiest to get started on a project related to something you

are already doing, because you already have data about things that you are doing. For instance, if you work in the music business, you may have access to many recordings. If you work as a radiologist, you probably have access to lots of medical images. If you are interested in wildlife preservation, you may have access to lots of images of wildlife.

Sometimes you have to get a bit creative. Maybe you can find a previous machine learning project, such as a Kaggle competition, that is related to your field of interest. Sometimes you have to compromise. Maybe you can't find the exact data you need for the precise project you have in mind; but you might be able to find something from a similar domain, or measured in a different way, tackling a slightly different problem. Working on these kinds of similar projects will still give you a good understanding of the overall process, and may help you identify other shortcuts, data sources, and so forth.

Especially when you are just starting out with deep learning, it's not a good idea to branch out into very different areas, to places that deep learning has not been applied to before. That's because if your model does not work at first, you will not know whether it is because you have made a mistake, or if the very problem you are trying to solve is simply not solvable with deep learning. And you won't know where to look to get help. Therefore, it is best at first to start by finding an example online of something that somebody has had good results with and that is at least somewhat similar to what you are trying to achieve, by converting your data into a format similar to what someone else has used before (such as creating an image from your data). Let's have a look at the state of deep learning, just so you know what kinds of things deep learning is good at right now.

The State of Deep Learning

Let's start by considering whether deep learning can be any good at the problem you are looking to work on. This section provides a summary of the state of deep learning at the start of 2020. However, things move very fast, and by the time you read this, some of these constraints may no longer exist. We will try to keep the book's website up-to-date; in addition, a Google search for "what can AI do now" is likely to provide current information.

Computer vision

There are many domains in which deep learning has not been used to analyze images yet, but those where it has been tried have nearly universally shown that computers can recognize items in an image at least as well as people can—even specially trained people, such as radiologists. This is known as *object recognition*. Deep learning is also good at recognizing where objects in an image are, and can highlight their locations and name each found object. This is known as *object detection* (in a variant of this that

we saw in Chapter 1, every pixel is categorized based on the kind of object it is part of—this is called *segmentation*).

Deep learning algorithms are generally not good at recognizing images that are significantly different in structure or style from those used to train the model. For instance, if there were no black-and-white images in the training data, the model may do poorly on black-and-white images. Similarly, if the training data did not contain hand-drawn images, the model will probably do poorly on hand-drawn images. There is no general way to check which types of images are missing in your training set, but we will show in this chapter some ways to try to recognize when unexpected image types arise in the data when the model is being used in production (this is known as checking for *out-of-domain* data).

One major challenge for object detection systems is that image labeling can be slow and expensive. There is a lot of work at the moment going into tools to try to make this labeling faster and easier, and to require fewer handcrafted labels to train accurate object detection models. One approach that is particularly helpful is to synthetically generate variations of input images, such as by rotating them or changing their brightness and contrast; this is called *data augmentation* and also works well for text and other types of models. We will be discussing it in detail in this chapter.

Another point to consider is that although your problem might not look like a computer vision problem, it might be possible with a little imagination to turn it into one. For instance, if what you are trying to classify are sounds, you might try converting the sounds into images of their acoustic waveforms and then training a model on those images.

Text (natural language processing)

Computers are good at classifying both short and long documents based on categories such as spam or not spam, sentiment (e.g., is the review positive or negative), author, source website, and so forth. We are not aware of any rigorous work done in this area to compare computers to humans, but anecdotally it seems to us that deep learning performance is similar to human performance on these tasks.

Deep learning is also good at generating context-appropriate text, such as replies to social media posts, and imitating a particular author's style. It's good at making this content compelling to humans too—in fact, even more compelling than human-generated text. However, deep learning is not good at generating *correct* responses! We don't have a reliable way to, for instance, combine a knowledge base of medical information with a deep learning model for generating medically correct natural language responses. This is dangerous, because it is so easy to create content that appears to a layman to be compelling, but actually is entirely incorrect.

Another concern is that context-appropriate, highly compelling responses on social media could be used at massive scale—thousands of times greater than any troll farm previously seen—to spread disinformation, create unrest, and encourage conflict. As a rule of thumb, text generation models will always be technologically a bit ahead of models for recognizing automatically generated text. For instance, it is possible to use a model that can recognize artificially generated content to actually improve the generator that creates that content, until the classification model is no longer able to complete its task.

Despite these issues, deep learning has many applications in NLP: it can be used to translate text from one language to another, summarize long documents into something that can be digested more quickly, find all mentions of a concept of interest, and more. Unfortunately, the translation or summary could well include completely incorrect information! However, the performance is already good enough that many people are using these systems—for instance, Google's online translation system (and every other online service we are aware of) is based on deep learning.

Combining text and images

The ability of deep learning to combine text and images into a single model is, generally, far better than most people intuitively expect. For example, a deep learning model can be trained on input images with output captions written in English, and can learn to generate surprisingly appropriate captions automatically for new images! But again, we have the same warning that we discussed in the previous section: there is no guarantee that these captions will be correct.

Because of this serious issue, we generally recommend that deep learning be used not as an entirely automated process, but as part of a process in which the model and a human user interact closely. This can potentially make humans orders of magnitude more productive than they would be with entirely manual methods, and result in more accurate processes than using a human alone.

For instance, an automatic system can be used to identify potential stroke victims directly from CT scans, and send a high-priority alert to have those scans looked at quickly. There is only a three-hour window to treat strokes, so this fast feedback loop could save lives. At the same time, however, all scans could continue to be sent to radiologists in the usual way, so there would be no reduction in human input. Other deep learning models could automatically measure items seen on the scans and insert those measurements into reports, warning the radiologists about findings that they may have missed and telling them about other cases that might be relevant.

Tabular data

For analyzing time series and tabular data, deep learning has recently been making

great strides. However, deep learning is generally used as part of an ensemble of multiple types of model. If you already have a system that is using random forests or gradient boosting machines (popular tabular modeling tools that you will learn about soon), then switching to or adding deep learning may not result in any dramatic improvement.

Deep learning does greatly increase the variety of columns that you can include—for example, columns containing natural language (book titles, reviews, etc.) and high-cardinality categorical columns (i.e., something that contains a large number of discrete choices, such as zip code or product ID). On the down side, deep learning models generally take longer to train than random forests or gradient boosting machines, although this is changing thanks to libraries such as RAPIDS, which provides GPU acceleration for the whole modeling pipeline. We cover the pros and cons of all these methods in detail in Chapter 9.

Recommendation systems

Recommendation systems are really just a special type of tabular data. In particular, they generally have a high-cardinality categorical variable representing users, and another one representing products (or something similar). A company like Amazon represents every purchase that has ever been made by its customers as a giant sparse matrix, with customers as the rows and products as the columns. Once they have the data in this format, data scientists apply some form of collaborative filtering to *fill in the matrix*. For example, if customer A buys products 1 and 10, and customer B buys products 1, 2, 4, and 10, the engine will recommend that A buy 2 and 4.

Because deep learning models are good at handling high-cardinality categorical variables, they are quite good at handling recommendation systems. They particularly come into their own, just like for tabular data, when combining these variables with other kinds of data, such as natural language or images. They can also do a good job of combining all of these types of information with additional metadata represented as tables, such as user information, previous transactions, and so forth.

However, nearly all machine learning approaches have the downside that they tell you only which products a particular user might like, rather than what recommendations would be helpful for a user. Many kinds of recommendations for products a user might like may not be at all helpful—for instance, if the user is already familiar with the products, or if they are simply different packagings of products they have already purchased (such as a boxed set of novels, when they already have each of the items in that set). Jeremy likes reading books by Terry Pratchett, and for a while Amazon was recommending nothing but Terry Pratchett books to him (see Figure 2-1), which really wasn't helpful because he was already aware of these books!

Figure 2-1. A not-so-useful recommendation

Other data types

Often you will find that domain-specific data types fit very nicely into existing categories. For instance, protein chains look a lot like natural language documents, in that they are long sequences of discrete tokens with complex relationships and meaning throughout the sequence. And indeed, it does turn out that using NLP deep learning methods is the current state-of-the-art approach for many types of protein analysis. As another example, sounds can be represented as spectrograms, which can be treated as images; standard deep learning approaches for images turn out to work really well on spectrograms.

The Drivetrain Approach

Many accurate models are of no use to anyone, and many inaccurate models are highly useful. To ensure that your modeling work is useful in practice, you need to consider how your work will be used. In 2012, Jeremy, along with Margit Zwemer and Mike Loukides, introduced a method called *the Drivetrain Approach* for thinking about this issue.

The Drivetrain Approach, illustrated in Figure 2-2, was described in detail in "Designing Great Data Products". The basic idea is to start with considering your objective, then think about what actions you can take to meet that objective and what data you have (or can acquire) that can help, and then build a model that you can use to determine the best actions to take to get the best results in terms of your objective.

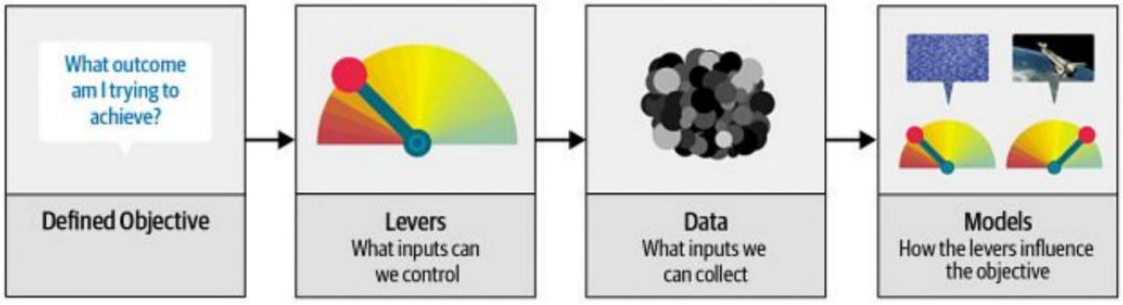


Figure 2-2. The Drivetrain Approach

Consider a model in an autonomous vehicle: you want to help a car drive safely from point A to point B without human intervention. Great predictive modeling is an important part of the solution, but it doesn't stand on its own; as products become more sophisticated, it disappears into the plumbing. Someone using a self-driving car is completely unaware of the hundreds (if not thousands) of models and the petabytes of data that make it work. But as data scientists build increasingly sophisticated products, they need a systematic design approach.

We use data not just to generate more data (in the form of predictions), but to produce *actionable outcomes*. That is the goal of the Drivetrain Approach. Start by defining a clear *objective*. For instance, Google, when creating its first search engine, considered "What is the user's main objective in typing in a search query?" This led to Google's objective, which was to "show the most relevant search result." The next step is to consider what *levers* you can pull (i.e., what actions you can take) to better achieve that objective. In Google's case, that was the ranking of the search results. The third step was to consider what new *data* they would need to produce such a ranking; they realized that the implicit information regarding which pages linked to which other pages could be used for this purpose.

Only after these first three steps do we begin thinking about building the predictive *models*. Our objective and available levers, what data we already have and what additional data we will need to collect, determine the models we can build. The models will take both the levers and any uncontrollable variables as their inputs; the outputs from the models can be combined to predict the final state for our objective.

Let's consider another example: recommendation systems. The *objective* of a recommendation engine is to drive additional sales by surprising and delighting the customer with recommendations of items they would not have purchased without the recommendation. The *lever* is the ranking of the recommendations. New *data* must be collected to generate recommendations that will *cause new sales*. This will require conducting many randomized experiments in order to collect data about a wide range

of recommendations for a wide range of customers. This is a step that few organizations take; but without it, you don't have the information you need to optimize recommendations based on your true objective (more sales!).

Finally, you could build two *models* for purchase probabilities, conditional on seeing or not seeing a recommendation. The difference between these two probabilities is a utility function for a given recommendation to a customer. It will be low in cases where the algorithm recommends a familiar book that the customer has already rejected (both components are small) or a book that they would have bought even without the recommendation (both components are large and cancel each other out).

As you can see, in practice often the practical implementation of your models will require a lot more than just training a model! You'll often need to run experiments to collect more data, and consider how to incorporate your models into the overall system you're developing. Speaking of data, let's now focus on how to find data for your project.

Gathering Data

For many types of projects, you may be able to find all the data you need online. The project we'll be completing in this chapter is a *bear detector*. It will discriminate between three types of bear: grizzly, black, and teddy bears. There are many images on the internet of each type of bear that we can use. We just need a way to find them and download them.

We've provided a tool you can use for this purpose, so you can follow along with this chapter and create your own image recognition application for whatever kinds of objects you're interested in. In the fast.ai course, thousands of students have presented their work in the course forums, displaying everything from hummingbird varieties in Trinidad to bus types in Panama—one student even created an application that would help his fiancée recognize his 16 cousins during Christmas vacation!

At the time of writing, Bing Image Search is the best option we know of for finding and downloading images. It's free for up to 1,000 queries per month, and each query can download up to 150 images. However, something better might have come along between when we wrote this and when you're reading the book, so be sure to check out this book's website for our current recommendation.

Keeping in Touch with the Latest Services

Services that can be used for creating datasets come and go all the time, and their features, interfaces, and pricing change regularly too. In this section, we'll show how to use the Bing Image Search API available as part of Azure Cognitive Services at the time this book was written.

To download images with Bing Image Search, sign up at Microsoft Azure for a free account. You will be given a key, which you can copy and enter in a cell as follows (replacing *XXX* with your key and executing it):

```
key = os.environ.get('AZURE_SEARCH_KEY', 'XXX')
```

Or, if you're comfortable at the command line, you can set it in your terminal with

```
export AZURE_SEARCH_KEY=your_key_here
```

and then restart Jupyter Notebook, and use the above line without editing it.

Once you've set `key`, you can use `search_images_bing`. This function is provided by the small `utils` class included with the notebooks online (if you're not sure where a function is defined, you can just type it in your notebook to find out, as shown here):

```
search_images_bing
```

```
<function utils.search_images_bing(key, term, min_sz=128)>
```

Let's try this function out:

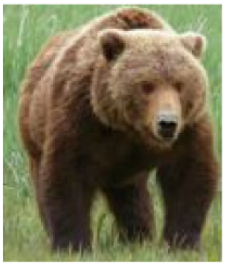
```
results = search_images_bing(key, 'grizzly bear')
ims = results.attrgot('content_url')
len(ims)
```

```
150
```

We've successfully downloaded the URLs of 150 grizzly bears (or, at least, images that Bing Image Search finds for that search term). Let's look at one:

```
dest = 'images/grizzly.jpg'
download_url(ims[0], dest)
```

```
im = Image.open(dest)
im.to_thumb(128,128)
```



This seems to have worked nicely, so let's use fastai's `download_images` to download all the URLs for each of our search terms. We'll put each in a separate folder:

```
bear_types = 'grizzly','black','teddy'
path = Path('bears')
```

```
if not path.exists():
    path.mkdir()
    for o in bear_types:
        dest = (path/o)
        dest.mkdir(exist_ok=True)
        results = search_images_bing(key, f'{o} bear')
        download_images(dest, urls=results.attrgot('contentUrl'))
```

Our folder has image files, as we'd expect:

```
fns = get_image_files(path)
fns
```

```
(#421) [Path('bears/black/00000095.jpg'),Path('bears/black/00000133.jpg'),Path('
 > bears/black/00000062.jpg'),Path('bears/black/00000023.jpg'),Path('bears/black
 > /00000029.jpg'),Path('bears/black/00000094.jpg'),Path('bears/black/00000124.j
 > pg'),Path('bears/black/00000056.jpeg'),Path('bears/black/00000046.jpg'),Path(
 > 'bears/black/00000045.jpg')...]
```

Jeremy Says

I just love this about working in Jupyter notebooks! It's so easy to gradually build what I want, and check my work every step of the way. I make a *lot* of mistakes, so this is really helpful to me.

Often when we download files from the internet, a few are corrupt. Let's check:

```
failed = verify_images(fns)
failed
```

```
(#0) []
```

To remove all the failed images, you can use `unlink`. Like most fastai functions that return a collection, `verify_images` returns an object of type `L`, which includes the `map` method. This calls the passed function on each element of the collection:

```
failed.map(Path.unlink);
```

Getting Help in Jupyter Notebooks

Jupyter notebooks are great for experimenting and immediately seeing the results of each function, but there is also a lot of functionality to help you figure out how to use different functions, or even directly look at their source code. For instance, say you type this in a cell:

```
??verify_images
```

A window will pop up with this:

```
Signature: verify_images(fns)
Source:
def verify_images(fns):
    "Find images in `fns` that can't be opened"
    return L(fns[i] for i,o in
             enumerate(parallel(verify_image, fns)) if not o)
File:      ~/git/fastai/fastai/vision/utils.py
Type:      function
```

This tells us what argument the function accepts (`fns`), and then shows us the source code and the file it comes from. Looking at that source code, we can see it applies the function `verify_image` in parallel and keeps only the image files for which the result of that function is `False`, which is consistent with the doc string: it finds the images in `fns` that can't be opened.

Here are some other features that are very useful in Jupyter notebooks:

- At any point, if you don't remember the exact spelling of a function or argument name, you can press Tab to get autocompletion suggestions.
- When inside the parentheses of a function, pressing Shift and Tab simultaneously will display a window with the signature of the function

and a short description. Pressing these keys twice will expand the documentation, and pressing them three times will open a full window with the same information at the bottom of your screen.

- In a cell, typing `?func_name` and executing will open a window with the signature of the function and a short description.
- In a cell, typing `??func_name` and executing will open a window with the signature of the function, a short description, and the source code.
- If you are using the fastai library, we added a `doc` function for you: executing `doc(func_name)` in a cell will open a window with the signature of the function, a short description, and links to the source code on GitHub and the full documentation of the function in the library docs.
- Unrelated to the documentation but still very useful: to get help at any point if you get an error, type `%debug` in the next cell and execute to open the Python debugger, which will let you inspect the content of every variable.

One thing to be aware of in this process: as we discussed in Chapter 1, models can reflect only the data used to train them. And the world is full of biased data, which ends up reflected in, for example, Bing Image Search (which we used to create our dataset). For instance, let's say you were interested in creating an app that could help users figure out whether they had healthy skin, so you trained a model on the results of searches for (say) "healthy skin." Figure 2-3 shows you kind of the results you would get.

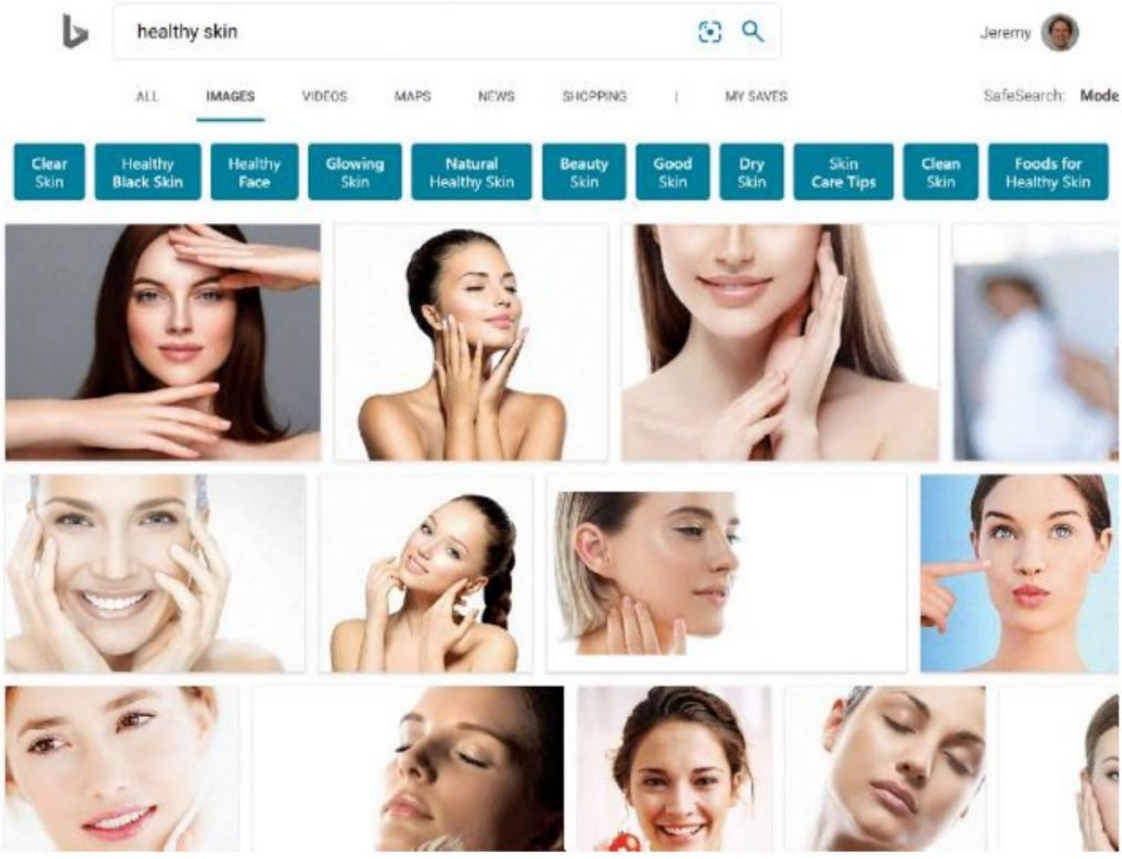


Figure 2-3. Data for a healthy skin detector?

With this as your training data, you would end up not with a healthy skin detector, but a *young white woman touching her face* detector! Be sure to think carefully about the types of data that you might expect to see in practice in your application, and check carefully to ensure that all these types are reflected in your model's source data. (Thanks to Deb Raji, who came up with the healthy skin example. See her paper "Actionable Auditing: Investigating the Impact of Publicly Naming Biased Performance Results of Commercial AI Products" for more fascinating insights into model bias.)

Now that we have downloaded some data, we need to assemble it in a format suitable for model training. In fastai, that means creating an object called `DataLoaders`.

From Data to DataLoaders

`DataLoaders` is a thin class that just stores whatever `DataLoader` objects you pass to it and makes them available as `train` and `valid`. Although it's a simple class, it's important in fastai: it provides the data for your model. The key functionality in `DataLoaders` is provided with just these four lines of code (it has some other minor functionality we'll skip over for now):

```
class DataLoaders(GetAttr):
    def __init__(self, *loaders): self.loaders = loaders
    def __getitem__(self, i): return self.loaders[i]
    train,valid = add_props(lambda i,self: self[i])
```

Jargon: DataLoaders

A fastai class that stores multiple `DataLoader` objects you pass to it—normally a `train` and a `valid`, although it's possible to have as many as you like. The first two are made available as properties.

Later in the book, you'll also learn about the `Dataset` and `Datasets` classes, which have the same relationship. To turn our downloaded data into a `DataLoaders` object, we need to tell fastai at least four things:

- What kinds of data we are working with
- How to get the list of items
- How to label these items
- How to create the validation set

So far we have seen a number of *factory methods* for particular combinations of these things, which are convenient when you have an application and data structure that happen to fit into those predefined methods. For when you don't, fastai has an extremely flexible system called the *data block API*. With this API, you can fully customize every stage of the creation of your `DataLoaders`. Here is what we need to create a `DataLoaders` for the dataset that we just downloaded:

```
bears = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(128))
```

Let's look at each of these arguments in turn. First we provide a tuple specifying the types we want for the independent and dependent variables:

```
blocks=(ImageBlock, CategoryBlock)
```

The *independent variable* is the thing we are using to make predictions from, and the

dependent variable is our target. In this case, our independent variable is a set of images, and our dependent variables are the categories (type of bear) for each image. We will see many other types of block in the rest of this book.

For this `DataLoaders`, our underlying items will be file paths. We have to tell fastai how to get a list of those files. The `get_image_files` function takes a path, and returns a list of all of the images in that path (recursively, by default):

```
get_items=get_image_files
```

Often, datasets that you download will already have a validation set defined. Sometimes this is done by placing the images for the training and validation sets into different folders. Sometimes it is done by providing a CSV file in which each filename is listed along with which dataset it should be in. There are many ways that this can be done, and fastai provides a general approach that allows you to use one of its predefined classes for this or to write your own.

In this case, we want to split our training and validation sets randomly. However, we would like to have the same training/validation split each time we run this notebook, so we fix the random seed (computers don't really know how to create random numbers at all, but simply create lists of numbers that look random; if you provide the same starting point for that list each time—called the *seed*—then you will get the exact same list each time).

```
splitter=RandomSplitter(valid_pct=0.2, seed=42)
```

The independent variable is often referred to as `x`, and the dependent variable is often referred to as `y`. Here, we are telling fastai what function to call to create the labels in our dataset:

```
get_y=parent_label
```

`parent_label` is a function provided by fastai that simply gets the name of the folder a file is in. Because we put each of our bear images into folders based on the type of bear, this is going to give us the labels that we need.

Our images are all different sizes, and this is a problem for deep learning: we don't feed the model one image at a time but several of them (what we call a *mini-batch*). To group them in a big array (usually called a *tensor*) that is going to go through our model, they

all need to be of the same size. So, we need to add a transform that will resize these images to the same size. *Item transforms* are pieces of code that run on each individual item, whether it be an image, category, or so forth. fastai includes many predefined transforms; we use the `Resize` transform here and specify a size of 128 pixels:

```
item_tfms=Resize(128)
```

This command has given us a `DataBlock` object. This is like a *template* for creating a `DataLoaders`. We still need to tell fastai the actual source of our data—in this case, the path where the images can be found:

```
dls = bears.dataloaders(path)
```

A `DataLoaders` includes validation and training `DataLoaders`. A `DataLoader` is a class that provides batches of a few items at a time to the GPU. We'll be learning a lot more about this class in the next chapter. When you loop through a `DataLoader`, fastai will give you 64 (by default) items at a time, all stacked up into a single tensor. We can take a look at a few of those items by calling the `show_batch` method on a `DataLoader`:

```
dls.valid.show_batch(max_n=4, nrows=1)
```

By default, `Resize` *crops* the images to fit a square shape of the size requested, using the full width or height. This can result in losing some important details. Alternatively, you can ask fastai to pad the images with zeros (black), or squish/stretch them:

```
bears = bears.new(item_tfms=Resize(128, ResizeMethod.Squish))
dls = bears.dataloaders(path)
dls.valid.show_batch(max_n=4, nrows=1)
```


```
bears = bears.new(item_tfms=Resize(128, ResizeMethod.Pad, pad_mode='zeros'))
dls = bears.dataloaders(path)
dls.valid.show_batch(max_n=4, nrows=1)
```

All of these approaches seem somewhat wasteful or problematic. If we squish or stretch the images, they end up as unrealistic shapes, leading to a model that learns that things look different from how they actually are, which we would expect to result in lower accuracy. If we crop the images, we remove some of the features that allow us to perform recognition. For instance, if we were trying to recognize breeds of dog or cat, we might end up cropping out a key part of the body or the face necessary to distinguish between similar breeds. If we pad the images, we have a whole lot of empty space, which is just wasted computation for our model and results in a lower effective resolution for the part of the image we actually use.

Instead, what we normally do in practice is to randomly select part of the image and then crop to just that part. On each epoch (which is one complete pass through all of our images in the dataset), we randomly select a different part of each image. This means that our model can learn to focus on, and recognize, different features in our images. It also reflects how images work in the real world: different photos of the same thing may be framed in slightly different ways.

In fact, an entirely untrained neural network knows nothing whatsoever about how images behave. It doesn't even recognize that when an object is rotated by one degree, it still is a picture of the same thing! So training the neural network with examples of

images in which the objects are in slightly different places and are slightly different sizes helps it to understand the basic concept of what an object is, and how it can be represented in an image.

Here is another example where we replace `Resize` with `RandomResizedCrop`, which is the transform that provides the behavior just described. The most important parameter to pass in is `min_scale`, which determines how much of the image to select at minimum each time:

```
bears = bears.new(item_tfms=RandomResizedCrop(128, min_scale=0.3))
dls = bears.dataloaders(path)
dls.train.show_batch(max_n=4, nrows=1, unique=True)
```

Here, we used `unique=True` to have the same image repeated with different versions of this `RandomResizedCrop` transform.

`RandomResizedCrop` is a specific example of a more general technique, called data augmentation.

Data Augmentation

Data augmentation refers to creating random variations of our input data, such that they appear different but do not change the meaning of the data. Examples of common data augmentation techniques for images are rotation, flipping, perspective warping, brightness changes, and contrast changes. For natural photo images such as the ones we are using here, a standard set of augmentations that we have found work pretty well are provided with the `aug_transforms` function.

Because our images are now all the same size, we can apply these augmentations to an entire batch of them using the GPU, which will save a lot of time. To tell fastai we want to use these transforms on a batch, we use the `batch_tfms` parameter (note that we're not using `RandomResizedCrop` in this example, so you can see the differences more clearly; we're also using double the amount of augmentation compared to the default,

for the same reason):

```
bears = bears.new(item_tfms=Resize(128), batch_tfms=aug_transforms(mult=2))
dls = bears.dataloaders(path)
dls.train.show_batch(max_n=8, nrows=2, unique=True)
```

Now that we have assembled our data in a format fit for model training, let's train an image classifier using it.

Training Your Model, and Using It to Clean Your Data

Time to use the same lines of code as in Chapter 1 to train our bear classifier. We don't have a lot of data for our problem (150 pictures of each sort of bear at most), so to train our model, we'll use `RandomResizedCrop`, an image size of 224 pixels, which is fairly standard for image classification, and the default `aug_transforms`:

```
bears = bears.new(
    item_tfms=RandomResizedCrop(224, min_scale=0.5),
    batch_tfms=aug_transforms())
dls = bears.dataloaders(path)
```

We can now create our `Learner` and fine-tune it in the usual way:

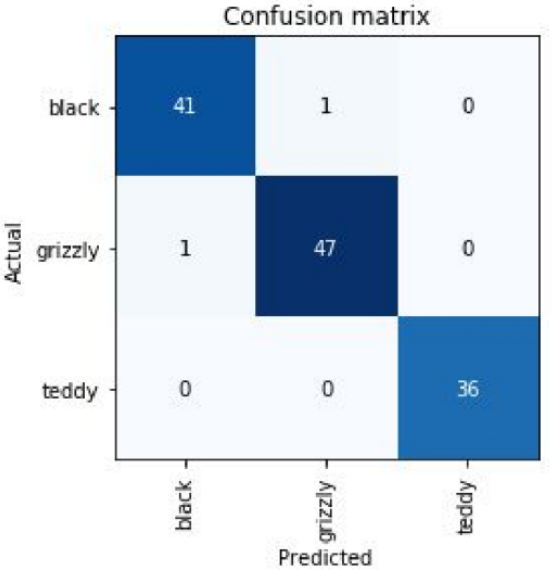
```
learn = cnn_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(4)
```

epoch	train_loss	valid_loss	error_rate	time
0	1.235733	0.212541	0.087302	00:05

epoch	train_loss	valid_loss	error_rate	time
0	0.213371	0.112450	0.023810	00:05
1	0.173855	0.072306	0.023810	00:06
2	0.147096	0.039068	0.015873	00:06
3	0.123984	0.026801	0.015873	00:06

Now let's see whether the mistakes the model is making are mainly thinking that grizzlies are teddies (that would be bad for safety!), or that grizzlies are black bears, or something else. To visualize this, we can create a *confusion matrix*:

```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```



The rows represent all the black, grizzly, and teddy bears in our dataset, respectively. The columns represent the images that the model predicted as black, grizzly, and teddy bears, respectively. Therefore, the diagonal of the matrix shows the images that were classified correctly, and the off-diagonal cells represent those that were classified incorrectly. This is one of the many ways that fastai allows you to view the results of your model. It is (of course!) calculated using the validation set. With the color-coding,

the goal is to have white everywhere except the diagonal, where we want dark blue. Our bear classifier isn't making many mistakes!

It's helpful to see where exactly our errors are occurring, to see whether they're due to a dataset problem (e.g., images that aren't bears at all, or are labeled incorrectly) or a model problem (perhaps it isn't handling images taken with unusual lighting, or from a different angle, etc.). To do this, we can sort our images by their loss.

The *loss* is a number that is higher if the model is incorrect (especially if it's also confident of its incorrect answer), or if it's correct but not confident of its correct answer. In the beginning of Part II, we'll learn in depth how loss is calculated and used in the training process. For now, `plot_top_losses` shows us the images with the highest loss in our dataset. As the title of the output says, each image is labeled with four things: prediction, actual (target label), loss, and probability. The *probability* here is the confidence level, from zero to one, that the model has assigned to its prediction:

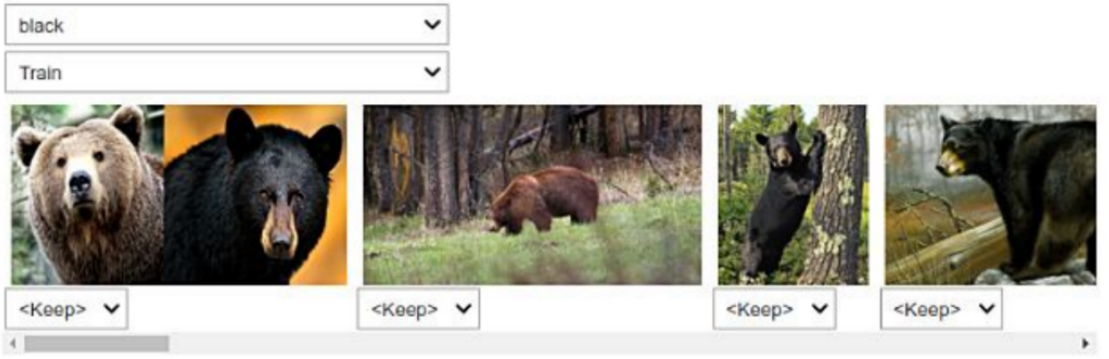
```
interp.plot_top_losses(5, nrows=1)
```

This output shows that the image with the highest loss is one that has been predicted as "grizzly" with high confidence. However, it's labeled (based on our Bing image search) as "black." We're not bear experts, but it sure looks to us like this label is incorrect! We should probably change its label to "grizzly."

The intuitive approach to doing data cleaning is to do it *before* you train a model. But as you've seen in this case, a model can help you find data issues more quickly and easily. So, we normally prefer to train a quick and simple model first, and then use it to help us with data cleaning.

fastai includes a handy GUI for data cleaning called `ImageClassifierCleaner` that allows you to choose a category and the training versus validation set and view the highest-loss images (in order), along with menus to allow images to be selected for removal or relabeling:

```
cleaner = ImageClassifierCleaner(learn)
cleaner
```



We can see that among our "black bears" is an image that contains two bears: one grizzly, one black. So, we should choose `<Delete>` in the menu under this image. `ImageClassifierCleaner` doesn't do the deleting or changing of labels for you; it just returns the indices of items to change. So, for instance, to delete (`unlink`) all images selected for deletion, we would run this:

```
for idx in cleaner.delete(): cleaner.fns[idx].unlink()
```

To move images for which we've selected a different category, we would run this:

```
for idx,cat in cleaner.change(): shutil.move(str(cleaner.fns[idx]), path/cat)
```

Sylvain Says

Cleaning the data and getting it ready for your model are two of the biggest challenges for data scientists; they say it takes 90% of their time. The fastai library aims to provide tools that make it as easy as possible.

We'll be seeing more examples of model-driven data cleaning throughout this book. Once we've cleaned up our data, we can retrain our model. Try it yourself, and see if your accuracy improves!

No Need for Big Data

After cleaning the dataset using these steps, we generally are seeing 100% accuracy on this task. We even see that result when we download a lot fewer images than the 150 per class we're using here. As you can see, the common complaint that *you need massive amounts of data to do deep learning* can be a very long way from the truth!

Now that we have trained our model, let's see how we can deploy it to be used in practice.

Turning Your Model into an Online Application

We are now going to look at what it takes to turn this model into a working online application. We will just go as far as creating a basic working prototype; we do not have the scope in this book to teach you all the details of web application development generally.

Using the Model for Inference

Once you've got a model you're happy with, you need to save it so you can then copy it over to a server where you'll use it in production. Remember that a model consists of two parts: the *architecture* and the trained *parameters*. The easiest way to save a model is to save both of these, because that way, when you load the model, you can be sure that you have the matching architecture and parameters. To save both parts, use the `export` method.

This method even saves the definition of how to create your `DataLoaders`. This is important, because otherwise you would have to redefine how to transform your data in order to use your model in production. fastai automatically uses your validation set `DataLoader` for inference by default, so your data augmentation will not be applied, which is generally what you want.

When you call `export`, fastai will save a file called *export.pkl*:

```
learn.export()
```

Let's check that the file exists, by using the `ls` method that fastai adds to Python's `Path` class:

```
path = Path()
path.ls(file_exts='.pkl')
```

```
(#1) [Path('export.pkl')]
```

You'll need this file wherever you deploy your app to. For now, let's try to create a simple app within our notebook.

When we use a model for getting predictions, instead of training, we call it *inference*. To create our inference learner from the exported file, we use `load_learner` (in this case, this isn't really necessary, since we already have a working `Learner` in our notebook;

we're doing it here so you can see the whole process end to end):

```
learn_inf = load_learner(path/'export.pkl')
```

When we're doing inference, we're generally getting predictions for just one image at a time. To do this, pass a filename to `predict`:

```
learn_inf.predict('images/grizzly.jpg')
```

```
('grizzly', tensor(1), tensor([9.0767e-06, 9.9999e-01, 1.5748e-07]))
```

This has returned three things: the predicted category in the same format you originally provided (in this case, that's a string), the index of the predicted category, and the probabilities of each category. The last two are based on the order of categories in the *vocab* of the `DataLoaders`; that is, the stored list of all possible categories. At inference time, you can access the `DataLoaders` as an attribute of the `Learner`:

```
learn_inf.dls.vocab
```

```
(#3) ['black','grizzly','teddy']
```

We can see here that if we index into the vocab with the integer returned by `predict`, we get back "grizzly," as expected. Also, note that if we index into the list of probabilities, we see a nearly 1.00 probability that this is a grizzly.

We know how to make predictions from our saved model, so we have everything we need to start building our app. We can do it directly in a Jupyter notebook.

Creating a Notebook App from the Model

To use our model in an application, we can simply treat the `predict` method as a regular function. Therefore, creating an app from the model can be done using any of the myriad of frameworks and techniques available to application developers.

However, most data scientists are not familiar with the world of web application development. So let's try using something that you do, at this point, know: it turns out that we can create a complete working web application using nothing but Jupyter notebooks! The two things we need to make this happen are as follows:

- IPython widgets (ipywidgets)

- Voilà

IPython widgets are GUI components that bring together JavaScript and Python functionality in a web browser, and can be created and used within a Jupyter notebook. For instance, the image cleaner that we saw earlier in this chapter is entirely written with IPython widgets. However, we don't want to require users of our application to run Jupyter themselves.

That is why *Voilà* exists. It is a system for making applications consisting of IPython widgets available to end users, without them having to use Jupyter at all. Voilà is taking advantage of the fact that a notebook *already is* a kind of web application, just a rather complex one that depends on another web application: Jupyter itself. Essentially, it helps us automatically convert the complex web application we've already implicitly made (the notebook) into a simpler, easier-to-deploy web application, which functions like a normal web application rather than like a notebook.

But we still have the advantage of developing in a notebook, so with ipywidgets, we can build up our GUI step by step. We will use this approach to create a simple image classifier. First, we need a file upload widget:

```
btn_upload = widgets.FileUpload()
btn_upload
```

Now we can grab the image:

```
img = PILImage.create(btn_upload.data[-1])
```

We can use an `Output` widget to display it:

```
out_pl = widgets.Output()
out_pl.clear_output()
with out_pl: display(img.to_thumb(128,128))
out_pl
```

Then we can get our predictions:

```
pred,pred_idx,probs = learn_inf.predict(img)
```

And use a `Label` to display them:

```
lbl_pred = widgets.Label()
lbl_pred.value = f'Prediction: {pred}; Probability: {probs[pred_idx]:.04f}'
lbl_pred
```

```
Prediction: grizzly; Probability: 1.0000
```

We'll need a button to do the classification. It looks exactly like the Upload button:

```
btn_run = widgets.Button(description='Classify')
btn_run
```

We'll also need a *click event handler*; that is, a function that will be called when it's pressed. We can just copy over the previous lines of code:

```
def on_click_classify(change):
    img = PILImage.create(btn_upload.data[-1])
    out_pl.clear_output()
    with out_pl: display(img.to_thumb(128,128))
    pred,pred_idx,probs = learn_inf.predict(img)
    lbl_pred.value = f'Prediction: {pred}; Probability: {probs[pred_idx]:.04f}'

btn_run.on_click(on_click_classify)
```

You can test the button now by clicking it, and you should see the image and

predictions update automatically!

We can now put them all in a vertical box (`VBox`) to complete our GUI:

```
VBox([widgets.Label('Select your bear!'),
      btn_upload, btn_run, out_pl, lbl_pred])
```

We have written all the code necessary for our app. The next step is to convert it into something we can deploy.

Turning Your Notebook into a Real App

Now that we have everything working in this Jupyter notebook, we can create our application. To do this, start a new notebook and add to it only the code needed to create and show the widgets that you need, and Markdown for any text that you want to appear. Have a look at the *bear_classifier* notebook in the book's repo to see the simple notebook application we created.

Next, install Voilà if you haven't already by copying these lines into a notebook cell and executing it:

```
!pip install voila
!jupyter serverextension enable --sys-prefix voila
```

Cells that begin with a `!` do not contain Python code, but instead contain code that is passed to your shell (bash, Windows PowerShell, etc.). If you are comfortable using the command line, which we'll discuss more in this book, you can of course simply type

these two lines (without the `!` prefix) directly into your terminal. In this case, the first line installs the `voila` library and application, and the second connects it to your existing Jupyter notebook.

Voilà runs Jupyter notebooks just like the Jupyter notebook server you are using now does, but it also does something very important: it removes all of the cell inputs, and shows only output (including ipywidgets), along with your Markdown cells. So what's left is a web application! To view your notebook as a Voilà web application, replace the word "notebooks" in your browser's URL with "voila/render". You will see the same content as your notebook, but without any of the code cells.

Of course, you don't need to use Voilà or ipywidgets. Your model is just a function you can call (`pred,pred_idx,probs = learn.predict(img)`), so you can use it with any framework, hosted on any platform. And you can take something you've prototyped in ipywidgets and Voilà and later convert it into a regular web application. We're showing you this approach in the book because we think it's a great way for data scientists and other folks who aren't web development experts to create applications from their models.

We have our app; now let's deploy it!

Deploying Your App

As you now know, you need a GPU to train nearly any useful deep learning model. So, do you need a GPU to use that model in production? No! You almost certainly *do not need a GPU to serve your model in production*. There are a few reasons for this:

- As we've seen, GPUs are useful only when they do lots of identical work in parallel. If you're doing (say) image classification, you'll normally be classifying just one user's image at a time, and there isn't normally enough work to do in a single image to keep a GPU busy for long enough for it to be very efficient. So, a CPU will often be more cost-effective.
- An alternative could be to wait for a few users to submit their images, and then batch them up and process them all at once on a GPU. But then you're asking your users to wait, rather than getting answers straight away! And you need a high-volume site for this to be workable. If you do need this functionality, you can use a tool such as Microsoft's ONNX Runtime or AWS SageMaker.
- The complexities of dealing with GPU inference are significant. In particular, the GPU's memory will need careful manual management, and you'll need a careful queueing system to ensure you process only one batch at a time.

- There's a lot more market competition in CPU than GPU servers, and as a result, there are much cheaper options available for CPU servers.

Because of the complexity of GPU serving, many systems have sprung up to try to automate this. However, managing and running these systems is also complex, and generally requires compiling your model into a different form that's specialized for that system. It's typically preferable to avoid dealing with this complexity until/unless your app gets popular enough that it makes clear financial sense for you to do so.

For at least the initial prototype of your application, and for any hobby projects that you want to show off, you can easily host them for free. The best place and the best way to do this will vary over time, so check the book's website for the most up-to-date recommendations. As we're writing this book in early 2020, the simplest (and free!) approach is to use Binder. To publish your web app on Binder, you follow these steps:

1. Add your notebook to a GitHub repository.
2. Paste the URL of that repo into Binder's URL field, as shown in Figure 2-4.
3. Change the File drop-down to instead select URL.
4. In the "URL to open" field, enter `/voila/render/`*`name`*`.ipynb` (replacing *name* with the name of your notebook).
5. Click the clipboard button at the bottom right to copy the URL and paste it somewhere safe.
6. Click Launch.

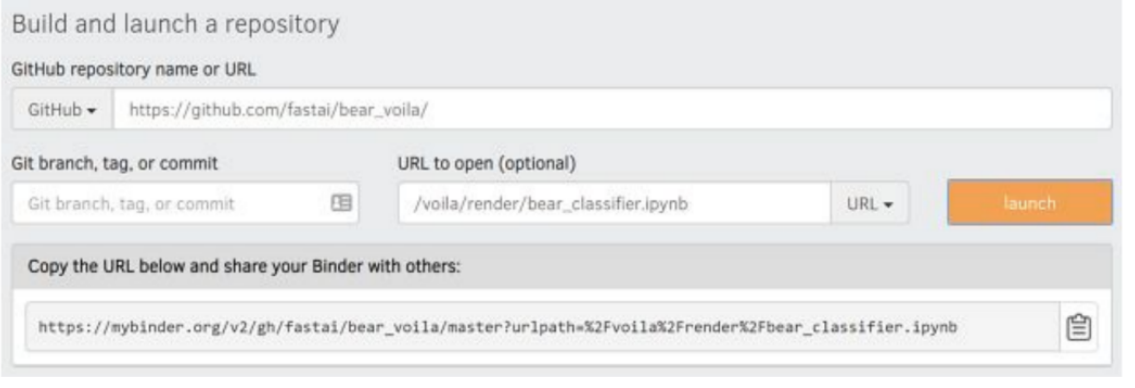


Figure 2-4. Deploying to Binder

The first time you do this, Binder will take around 5 minutes to build your site. Behind the scenes, it is finding a virtual machine that can run your app, allocating storage, and

collecting the files needed for Jupyter, for your notebook, and for presenting your notebook as a web application.

Finally, once it has started the app running, it will navigate your browser to your new web app. You can share the URL you copied to allow others to access your app as well.

For other (both free and paid) options for deploying your web app, be sure to take a look at the book's website.

You may well want to deploy your application onto mobile devices, or edge devices such as a Raspberry Pi. There are a lot of libraries and frameworks that allow you to integrate a model directly into a mobile application. However, these approaches tend to require a lot of extra steps and boilerplate, and do not always support all the PyTorch and fastai layers that your model might use. In addition, the work you do will depend on the kinds of mobile devices you are targeting for deployment—you might need to do some work to run on iOS devices, different work to run on newer Android devices, different work for older Android devices, etc. Instead, we recommend wherever possible that you deploy the model itself to a server, and have your mobile or edge application connect to it as a web service.

There are quite a few upsides to this approach. The initial installation is easier, because you have to deploy only a small GUI application, which connects to the server to do all the heavy lifting. More importantly perhaps, upgrades of that core logic can happen on your server, rather than needing to be distributed to all of your users. Your server will have a lot more memory and processing capacity than most edge devices, and it is far easier to scale those resources if your model becomes more demanding. The hardware that you will have on a server is also going to be more standard and more easily supported by fastai and PyTorch, so you don't have to compile your model into a different form.

There are downsides too, of course. Your application will require a network connection, and there will be some latency each time the model is called. (It takes a while for a neural network model to run anyway, so this additional network latency may not make a big difference to your users in practice. In fact, since you can use better hardware on the server, the overall latency may even be less than if it were running locally!) Also, if your application uses sensitive data, your users may be concerned about an approach that sends that data to a remote server, so sometimes privacy considerations will mean that you need to run the model on the edge device (it may be possible to avoid this by having an *on-premise* server, such as inside a company's firewall). Managing the complexity and scaling the server can create additional overhead too, whereas if your model runs on the edge devices, each user is bringing their own compute resources, which leads to easier scaling with an increasing

number of users (also known as *horizontal scaling*).

Alexis Says

I've had a chance to see up close how the mobile ML landscape is changing in my work. We offer an iPhone app that depends on computer vision, and for years we ran our own computer vision models in the cloud. This was the only way to do it then since those models needed significant memory and compute resources and took minutes to process inputs. This approach required building not only the models (fun!), but also the infrastructure to ensure a certain number of "compute worker machines" were absolutely always running (scary), that more machines would automatically come online if traffic increased, that there was stable storage for large inputs and outputs, that the iOS app could know and tell the user how their job was doing, etc. Nowadays Apple provides APIs for converting models to run efficiently on devices, and most iOS devices have dedicated ML hardware, so that's the strategy we use for our newer models. It's still not easy, but in our case it's worth it for a faster user experience and to worry less about servers. What works for you will depend, realistically, on the user experience you're trying to create and what you personally find is easy to do. If you really know how to run servers, do it. If you really know how to build native mobile apps, do that. There are many roads up the hill.

Overall, we'd recommend using a simple CPU-based server approach where possible, for as long as you can get away with it. If you're lucky enough to have a very successful application, you'll be able to justify the investment in more complex deployment approaches at that time.

Congratulations—you have successfully built a deep learning model and deployed it! Now is a good time to take a pause and think about what could go wrong.

How to Avoid Disaster

In practice, a deep learning model will be just one piece of a much bigger system. As we discussed at the start of this chapter, building a data product requires thinking about the entire end-to-end process, from conception to use in production. In this book, we can't hope to cover all the complexity of managing deployed data products, such as managing multiple versions of models, A/B testing, canarying, refreshing the data (should we just grow and grow our datasets all the time, or should we regularly remove some of the old data?), handling data labeling, monitoring all this, detecting model rot, and so forth.

In this section, we will give an overview of some of the most important issues to consider; for a more detailed discussion of deployment issues, we refer you to the excellent *Building Machine Learning Powered Applications* by Emmanuel Ameisin (O'Reilly).

One of the biggest issues to consider is that understanding and testing the behavior of a deep learning model is much more difficult than with most other code you write. With normal software development, you can analyze the exact steps that the software is taking, and carefully study which of these steps match the desired behavior that you are trying to create. But with a neural network, the behavior emerges from the model's attempt to match the training data, rather than being exactly defined.

This can result in disaster! For instance, let's say we really were rolling out a bear detection system that will be attached to video cameras around campsites in national parks and will warn campers of incoming bears. If we used a model trained with the dataset we downloaded, there would be all kinds of problems in practice, such as these:

- Working with video data instead of images
- Handling nighttime images, which may not appear in this dataset
- Dealing with low-resolution camera images
- Ensuring results are returned fast enough to be useful in practice
- Recognizing bears in positions that are rarely seen in photos that people post online (for example from behind, partially covered by bushes, or a long way away from the camera)

A big part of the issue is that the kinds of photos that people are most likely to upload to the internet are the kinds of photos that do a good job of clearly and artistically displaying their subject matter—which isn't the kind of input this system is going to be getting. So, we may need to do a lot of our own data collection and labeling to create a useful system.

This is just one example of the more general problem of *out-of-domain* data. That is to say, there may be data that our model sees in production that is very different from what it saw during training. There isn't a complete technical solution to this problem; instead, we have to be careful about our approach to rolling out the technology.

There are other reasons we need to be careful too. One very common problem is *domain shift*, whereby the type of data that our model sees changes over time. For instance, an insurance company may use a deep learning model as part of its pricing and risk algorithm, but over time the types of customers the company attracts and the types of risks it represents may change so much that the original training data is no longer relevant.

Out-of-domain data and domain shift are examples of a larger problem: that you can never fully understand all the possible behaviors of a neural network, because they

have far too many parameters. This is the natural downside of their best feature—their flexibility, which enables them to solve complex problems where we may not even be able to fully specify our preferred solution approaches. The good news, however, is that there are ways to mitigate these risks using a carefully thought-out process. The details of this will vary depending on the details of the problem you are solving, but we will attempt to lay out a high-level approach, summarized in Figure 2-5, which we hope will provide useful guidance.

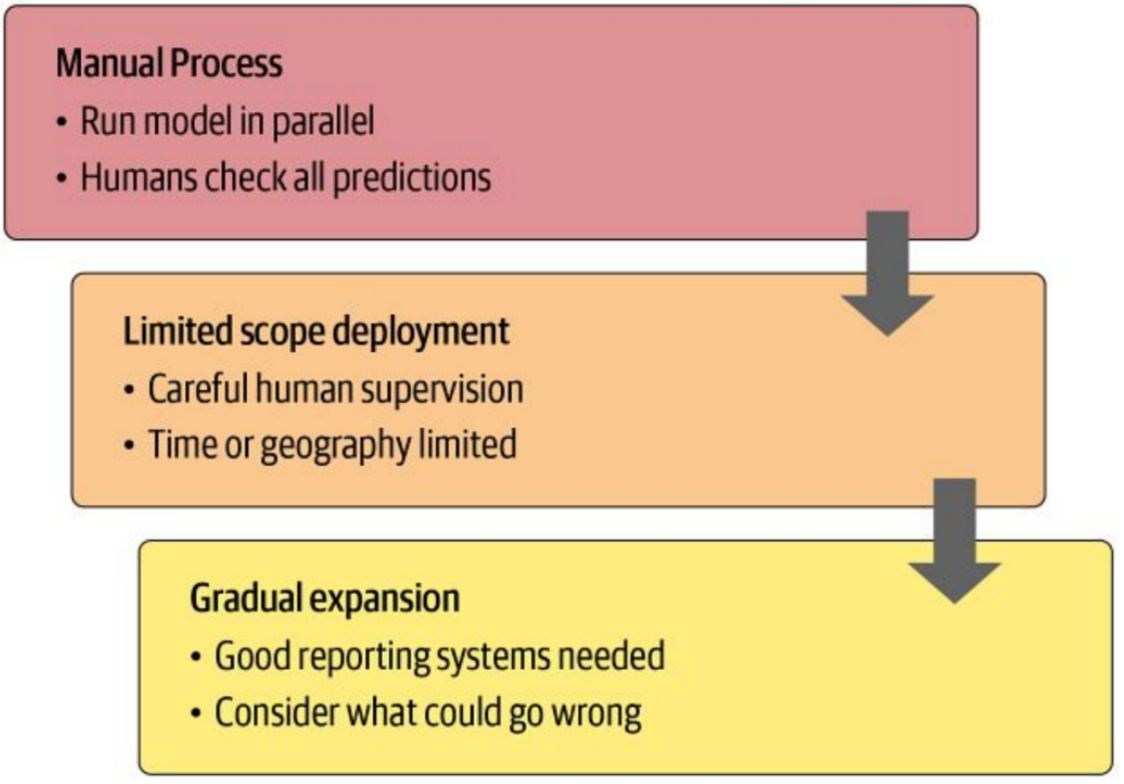


Figure 2-5. Deployment process

Where possible, the first step is to use an entirely manual process, with your deep learning model approach running in parallel but not being used directly to drive any actions. The humans involved in the manual process should look at the deep learning outputs and check whether they make sense. For instance, with our bear classifier, a park ranger could have a screen displaying video feeds from all the cameras, with any possible bear sightings simply highlighted in red. The park ranger would still be expected to be just as alert as before the model was deployed; the model is simply helping to check for problems at this point.

The second step is to try to limit the scope of the model, and have it carefully supervised by people. For instance, do a small geographically and time-constrained

trial of the model-driven approach. Rather than rolling out our bear classifier in every national park throughout the country, we could pick a single observation post, for a one-week period, and have a park ranger check each alert before it goes out.

Then, gradually increase the scope of your rollout. As you do so, ensure that you have really good reporting systems in place, to make sure that you are aware of any significant changes to the actions being taken compared to your manual process. For instance, if the number of bear alerts doubles or halves after rollout of the new system in some location, you should be very concerned. Try to think about all the ways in which your system could go wrong, and then think about what measure or report or picture could reflect that problem, and ensure that your regular reporting includes that information.

Jeremy Says

I started a company 20 years ago called Optimal Decisions that used machine learning and optimization to help giant insurance companies set their pricing, impacting tens of billions of dollars of risks. We used the approaches described here to manage the potential downsides of something going wrong. Also, before we worked with our clients to put anything in production, we tried to simulate the impact by testing the end-to-end system on their previous year's data. It was always quite a nerve-wracking process putting these new algorithms into production, but every rollout was successful.

Unforeseen Consequences and Feedback Loops

One of the biggest challenges in rolling out a model is that your model may change the behavior of the system it is a part of. For instance, consider a "predictive policing" algorithm that predicts more crime in certain neighborhoods, causing more police officers to be sent to those neighborhoods, which can result in more crimes being recorded in those neighborhoods, and so on. In the Royal Statistical Society paper "To Predict and Serve?" Kristian Lum and William Isaac observe that "predictive policing is aptly named: it is predicting future policing, not future crime."

Part of the issue in this case is that in the presence of bias (which we'll discuss in depth in the next chapter), *feedback loops* can result in negative implications of that bias getting worse and worse. For instance, there are concerns that this is already happening in the US, where there is significant bias in arrest rates on racial grounds. According to the ACLU, "despite roughly equal usage rates, Blacks are 3.73 times more likely than whites to be arrested for marijuana." The impact of this bias, along with the rollout of predictive policing algorithms in many parts of the United States, led Bärí Williams to write in the *New York Times*: "The same technology that's the source of so much excitement in my career is being used in law enforcement in ways that could mean that in the coming years, my son, who is 7 now, is more likely to be profiled or

arrested—or worse—for no reason other than his race and where we live."

A helpful exercise prior to rolling out a significant machine learning system is to consider this question: "What would happen if it went really, really well?" In other words, what if the predictive power was extremely high, and its ability to influence behavior was extremely significant? In that case, who would be most impacted? What would the most extreme results potentially look like? How would you know what was really going on?

Such a thought exercise might help you to construct a more careful rollout plan, with ongoing monitoring systems and human oversight. Of course, human oversight isn't useful if it isn't listened to, so make sure that reliable and resilient communication channels exist so that the right people will be aware of issues and will have the power to fix them.

Get Writing!

One of the things our students have found most helpful to solidify their understanding of this material is to write it down. There is no better test of your understanding of a topic than attempting to teach it to somebody else. This is helpful even if you never show your writing to anybody—but it's even better if you share it! So we recommend that, if you haven't already, you start a blog. Now that you've completed this chapter and have learned how to train and deploy models, you're well placed to write your first blog post about your deep learning journey. What's surprised you? What opportunities do you see for deep learning in your field? What obstacles do you see?

Rachel Thomas, cofounder of fast.ai, wrote in the article "Why You (Yes, You) Should Blog":

The top advice I would give my younger self would be to start blogging sooner. Here are some reasons to blog:

- *It's like a resume, only better. I know of a few people who have had blog posts lead to job offers!*
- *Helps you learn. Organizing knowledge always helps me synthesize my own ideas. One of the tests of whether you understand something is whether you can explain it to someone else. A blog post is a great way to do that.*
- *I've gotten invitations to conferences and invitations to speak from my blog posts. I was invited to the TensorFlow Dev Summit (which was awesome!) for writing a blog post about how I don't like TensorFlow.*
- *Meet new people. I've met several people who have responded to blog posts I wrote.*
- *Saves time. Any time you answer a question multiple times through email, you should turn it into a blog post, which makes it easier for you to share the next time someone asks.*

Perhaps her most important tip is this:

You are best positioned to help people one step behind you. The material is still fresh in your mind. Many experts have forgotten what it was like to be a beginner (or an intermediate) and have forgotten why the topic is hard to understand when you first hear it. The context of your particular background, your particular style, and your knowledge level will give a different twist to what you're writing about.

We've provided full details on how to set up a blog in Appendix A. If you don't have a blog already, take a look at that now, because we've got a really great approach for you to start blogging for free, with no ads—and you can even use Jupyter Notebook!

Questionnaire

1. Where do text models currently have a major deficiency?
2. What are possible negative societal implications of text generation models?
3. In situations where a model might make mistakes, and those mistakes could be harmful, what is a good alternative to automating a process?
4. What kind of tabular data is deep learning particularly good at?
5. What's a key downside of directly using a deep learning model for recommendation systems?
6. What are the steps of the Drivetrain Approach?

7. How do the steps of the Drivetrain Approach map to a recommendation system?
8. Create an image recognition model using data you curate, and deploy it on the web.
9. What is `DataLoaders`?
10. What four things do we need to tell fastai to create `DataLoaders`?
11. What does the `splitter` parameter to `DataBlock` do?
12. How do we ensure a random split always gives the same validation set?
13. What letters are often used to signify the independent and dependent variables?
14. What's the difference between the crop, pad, and squish resize approaches? When might you choose one over the others?
15. What is data augmentation? Why is it needed?
16. Provide an example of where the bear classification model might work poorly in production, due to structural or style differences in the training data.
17. What is the difference between `item_tfms` and `batch_tfms`?
18. What is a confusion matrix?
19. What does `export` save?
20. What is it called when we use a model for making predictions, instead of training?
21. What are IPython widgets?
22. When would you use a CPU for deployment? When might a GPU be better?
23. What are the downsides of deploying your app to a server, instead of to a client (or edge) device such as a phone or PC?
24. What are three examples of problems that could occur when rolling out a bear warning system in practice?
25. What is out-of-domain data?
26. What is domain shift?

27. What are the three steps in the deployment process?

Further Research

1. Consider how the Drivetrain Approach maps to a project or problem you're interested in.
2. When might it be best to avoid certain types of data augmentation?
3. For a project you're interested in applying deep learning to, consider the thought experiment, "What would happen if it went really, really well?"
4. Start a blog and write your first blog post. For instance, write about what you think deep learning might be useful for in a domain you're interested in.

Chapter 3. Data Ethics

Acknowledgment: Dr. Rachel Thomas

This chapter was coauthored by Dr. Rachel Thomas, the cofounder of fast.ai and founding director of the Center for Applied Data Ethics at the University of San Francisco. It largely follows a subset of the syllabus she developed for the Introduction to Data Ethics course.

As we discussed in Chapters 1 and 2, sometimes machine learning models can go wrong. They can have bugs. They can be presented with data that they haven't seen before and behave in ways we don't expect. Or they could work exactly as designed, but be used for something that we would much prefer they were never, ever used for.

Because deep learning is such a powerful tool and can be used for so many things, it becomes particularly important that we consider the consequences of our choices. The philosophical study of *ethics* is the study of right and wrong, including how we can define those terms, recognize right and wrong actions, and understand the connection between actions and consequences. The field of *data ethics* has been around for a long time, and many academics are focused on this field. It is being used to help define policy in many jurisdictions; it is being used in companies big and small to consider how best to ensure good societal outcomes from product development; and it is being used by researchers who want to make sure that the work they are doing is used for good, and not for bad.

As a deep learning practitioner, therefore, you will likely at some point be put in a situation requiring you to consider data ethics. So what is data ethics? It's a subfield of ethics, so let's start there.

Jeremy Says

At university, philosophy of ethics was my main thing (it would have been the topic of my thesis, if I'd finished it, instead of dropping out to join the real world). Based on the years I spent studying ethics, I can tell you this: no one really agrees on what right and wrong are, whether they exist, how to spot them, which people are good and which bad, or pretty much anything else. So don't expect too much from the theory! We're going to focus on examples and thought starters here, not theory.

In answering the question "What Is Ethics?" the Markkula Center for Applied Ethics says that the term refers to the following:

- Well-founded standards of right and wrong that prescribe what humans should do

- The study and development of one's ethical standards

There is no list of right answers. There is no list of dos and don'ts. Ethics is complicated and context-dependent. It involves the perspectives of many stakeholders. Ethics is a muscle that you have to develop and practice. In this chapter, our goal is to provide some signposts to help you on that journey.

Spotting ethical issues is best to do as part of a collaborative team. This is the only way you can really incorporate different perspectives. Different people's backgrounds will help them to see things that may not be obvious to you. Working with a team is helpful for many "muscle-building" activities, including this one.

This chapter is certainly not the only part of the book where we talk about data ethics, but it's good to have a place where we focus on it for a while. To get oriented, it's perhaps easiest to look at a few examples. So, we picked out three that we think illustrate effectively some of the key topics.

Key Examples for Data Ethics

We are going to start with three specific examples that illustrate three common ethical issues in tech (we'll study these issues in more depth later in the chapter):

Recourse processes

Arkansas's buggy healthcare algorithms left patients stranded.

Feedback loops

YouTube's recommendation system helped unleash a conspiracy theory boom.

Bias

When a traditionally African-American name is searched for on Google, it displays ads for criminal background checks.

In fact, for every concept that we introduce in this chapter, we are going to provide at least one specific example. For each one, think about what you could have done in this situation, and what kinds of obstructions there might have been to you getting that done. How would you deal with them? What would you look out for?

Bugs and Recourse: Buggy Algorithm Used for Healthcare Benefits

The Verge investigated software used in over half of the US states to determine how much healthcare people receive, and documented its findings in the article "What Happens When an Algorithm Cuts Your Healthcare". After implementation of the algorithm in Arkansas, hundreds of people (many with severe disabilities) had their

healthcare drastically cut.

For instance, Tammy Dobbs, a woman with cerebral palsy who needs an aide to help her to get out of bed, to go to the bathroom, to get food, and more, had her hours of help suddenly reduced by 20 hours a week. She couldn't get any explanation for why her healthcare was cut. Eventually, a court case revealed that there were mistakes in the software implementation of the algorithm, negatively impacting people with diabetes or cerebral palsy. However, Dobbs and many other people reliant on these health-care benefits live in fear that their benefits could again be cut suddenly and inexplicably.

Feedback Loops: YouTube's Recommendation System

Feedback loops can occur when your model is controlling the next round of data you get. The data that is returned quickly becomes flawed by the software itself.

For instance, YouTube has 1.9 billion users, who watch over 1 billion hours of YouTube videos a day. Its recommendation algorithm (built by Google), which was designed to optimize watch time, is responsible for around 70% of the content that is watched. But there was a problem: it led to out-of-control feedback loops, leading the *New York Times* to run the headline "YouTube Unleashed a Conspiracy Theory Boom. Can It Be Contained?" in February 2019. Ostensibly, recommendation systems are predicting what content people will like, but they also have a lot of power in determining what content people even see.

Bias: Professor Latanya Sweeney "Arrested"

Dr. Latanya Sweeney is a professor at Harvard and director of the university's data privacy lab. In the paper "Discrimination in Online Ad Delivery" (see Figure 3-1), she describes her discovery that Googling her name resulted in advertisements saying "Latanya Sweeney, Arrested?" even though she is the only known Latanya Sweeney and has never been arrested. However, when she Googled other names, such as "Kirsten Lindquist," she got more neutral ads, even though Kirsten Lindquist has been arrested three times.

Figure 3-1. Google search showing ads about Professor Latanya Sweeney's (nonexistent) arrest record

Being a computer scientist, she studied this systematically and looked at over 2,000 names. She found a clear pattern: historically Black names received advertisements suggesting that the person had a criminal record, whereas traditionally white names had more neutral advertisements.

This is an example of bias. It can make a big difference to people's lives—for instance, if a job applicant is Googled, it may appear that they have a criminal record when they do not.

Why Does This Matter?

One very natural reaction to considering these issues is: "So what? What's that got to do with me? I'm a data scientist, not a politician. I'm not one of the senior executives at my company who make the decisions about what we do. I'm just trying to build the most predictive model I can."

These are very reasonable questions. But we're going to try to convince you that the answer is that everybody who is training models absolutely needs to consider how

their models will be used, and consider how to best ensure that they are used as positively as possible. There are things you can do. And if you don't do them, things can go pretty badly.

One particularly hideous example of what happens when technologists focus on technology at all costs is the story of IBM and Nazi Germany. In 2001, a Swiss judge ruled that it was not unreasonable "to deduce that IBM's technical assistance facilitated the tasks of the Nazis in the commission of their crimes against humanity, acts also involving accountancy and classification by IBM machines and utilized in the concentration camps themselves."

IBM, you see, supplied the Nazis with data tabulation products necessary to track the extermination of Jews and other groups on a massive scale. This was driven from the top of the company, with marketing to Hitler and his leadership team. Company President Thomas Watson personally approved the 1939 release of special IBM alphabetizing machines to help organize the deportation of Polish Jews. Pictured in Figure 3-2 is Adolf Hitler (far left) meeting with IBM CEO Tom Watson Sr. (second from left), shortly before Hitler awarded Watson a special "Service to the Reich" medal in 1937.

Figure 3-2. IBM CEO Tom Watson Sr. meeting with Adolf Hitler

But this was not an isolated incident—the organization's involvement was extensive. IBM and its subsidiaries provided regular training and maintenance onsite at the concentration camps: printing off cards, configuring machines, and repairing them as they broke frequently. IBM set up categorizations on its punch card system for the way that each person was killed, which group they were assigned to, and the logistical information necessary to track them through the vast Holocaust system (see Figure 3-3). IBM's code for Jews in the concentration camps was 8: some 6,000,000 were killed. Its code for Romanis was 12 (they were labeled by the Nazis as "asocials," with over 300,000 killed in the *Zigeunerlager*, or "Gypsy camp"). General executions were coded as 4, death in the gas chambers as 6.

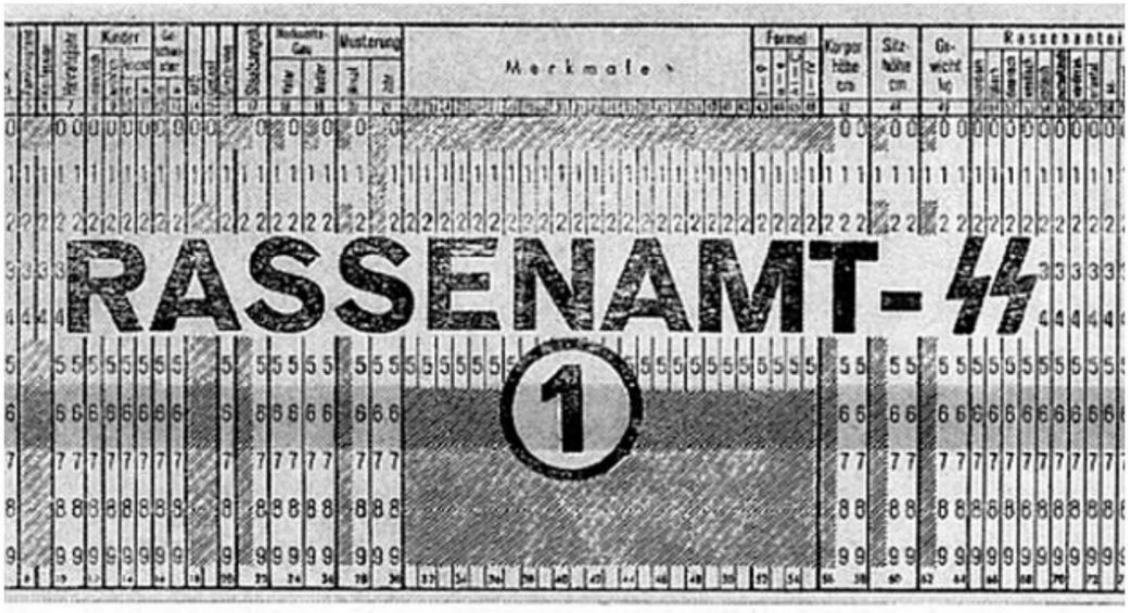


Figure 3-3. A punch card used by IBM in concentration camps

Of course, the project managers and engineers and technicians involved were just living their ordinary lives. Caring for their families, going to the church on Sunday, doing their jobs the best they could. Following orders. The marketers were just doing what they could to meet their business development goals. As Edwin Black, author of *IBM and the Holocaust* (Dialog Press) observed: "To the blind technocrat, the means were more important than the ends. The destruction of the Jewish people became even less important because the invigorating nature of IBM's technical achievement was only heightened by the fantastical profits to be made at a time when bread lines stretched across the world."

Step back for a moment and consider: How would you feel if you discovered that you had been part of a system that ended up hurting society? Would you be open to finding out? How can you help make sure this doesn't happen? We have described the most extreme situation here, but there are many negative societal consequences linked to AI and machine learning being observed today, some of which we'll describe in this chapter.

It's not just a moral burden, either. Sometimes technologists pay very directly for their actions. For instance, the first person who was jailed as a result of the Volkswagen scandal, in which the car company was revealed to have cheated on its diesel emissions tests, was not the manager who oversaw the project, or an executive at the helm of the company. It was one of the engineers, James Liang, who just did what he was told.

Of course, it's not all bad—if a project you are involved in turns out to make a huge positive impact on even one person, this is going to make you feel pretty great!

OK, so hopefully we have convinced you that you ought to care. But what should you do? As data scientists, we're naturally inclined to focus on making our models better by optimizing some metric or other. But optimizing that metric may not lead to better outcomes. And even if it *does* help create better outcomes, it almost certainly won't be the only thing that matters. Consider the pipeline of steps that occurs between the development of a model or an algorithm by a researcher or practitioner, and the point at which this work is used to make a decision. This entire pipeline needs to be considered *as a whole* if we're to have a hope of getting the kinds of outcomes we want.

Normally, there is a very long chain from one end to the other. This is especially true if you are a researcher who might not even know if your research will ever get used for anything, or if you're involved in data collection, which is even earlier in the pipeline. But no one is better placed to inform everyone involved in this chain about the capabilities, constraints, and details of your work than you are. Although there's no "silver bullet" that can ensure your work is used the right way, by getting involved in the process, and asking the right questions, you can at the very least ensure that the right issues are being considered.

Sometimes, the right response to being asked to do a piece of work is to just say "no." Often, however, the response we hear is, "If I don't do it, someone else will." But consider this: if you've been picked for the job, you're the best person they've found to do it—so if you don't do it, the best person isn't working on that project. If the first five people they ask all say no too, so much the better!

Integrating Machine Learning with Product Design

Presumably, the reason you're doing this work is that you hope it will be used for something. Otherwise, you're just wasting your time. So, let's start with the assumption that your work will end up somewhere. Now, as you are collecting your data and developing your model, you are making lots of decisions. What level of aggregation will you store your data at? What loss function should you use? What validation and training sets should you use? Should you focus on simplicity of implementation, speed of inference, or accuracy of the model? How will your model handle out-of-domain data items? Can it be fine-tuned, or must it be retrained from scratch over time?

These are not just algorithm questions. They are data product design questions. But the product managers, executives, judges, journalists, doctors—whoever ends up developing and using the system of which your model is a part—will not be well-placed to understand the decisions that you made, let alone change them.

For instance, two studies found that Amazon's facial recognition software produced inaccurate and racially biased results. Amazon claimed that the researchers should have changed the default parameters, without explaining how this would have changed the biased results. Furthermore, it turned out that Amazon was not instructing police departments that used its software to do this either. There was, presumably, a big distance between the researchers who developed these algorithms and the Amazon documentation staff who wrote the guidelines provided to the police.

A lack of tight integration led to serious problems for society at large, the police, and Amazon. It turned out that its system erroneously matched 28 members of Congress to criminal mugshots! (And the Congresspeople wrongly matched to criminal mugshots were disproportionately people of color, as seen in Figure 3-4.)

Figure 3-4. Congresspeople matched to criminal mugshots by Amazon software

Data scientists need to be part of a cross-disciplinary team. And researchers need to work closely with the kinds of people who will end up using their research. Better still, domain experts themselves could learn enough to be able to train and debug some models themselves—hopefully, a few of you are reading this book right now!

The modern workplace is a very specialized place. Everybody tends to have well-defined jobs to perform. Especially in large companies, it can be hard to know all the pieces of the puzzle. Sometimes companies even intentionally obscure the overall project goals being worked on, if they know that employees are not going to like the answers. This is sometimes done by compartmentalizing pieces as much as possible.

In other words, we're not saying that any of this is easy. It's hard. It's really hard. We

all have to do our best. And we have often seen that the people who do get involved in the higher-level context of these projects, and attempt to develop cross-disciplinary capabilities and teams, become some of the most important and well rewarded members of their organizations. It's the kind of work that tends to be highly appreciated by senior executives, even if it is sometimes considered rather uncomfortable by middle management.

Topics in Data Ethics

Data ethics is a big field, and we can't cover everything. Instead, we're going to pick a few topics that we think are particularly relevant:

- The need for recourse and accountability
- Feedback loops
- Bias
- Disinformation

Let's look at each in turn.

Recourse and Accountability

In a complex system, it is easy for no one person to feel responsible for outcomes. While this is understandable, it does not lead to good results. In the earlier example of the Arkansas healthcare system in which a bug led to people with cerebral palsy losing access to needed care, the creator of the algorithm blamed government officials, and government officials blamed those who implemented the software. NYU professor Danah Boyd described this phenomenon: "Bureaucracy has often been used to shift or evade responsibility....Today's algorithmic systems are extending bureaucracy."

An additional reason why recourse is so necessary is that data often contains errors. Mechanisms for audits and error correction are crucial. A database of suspected gang members maintained by California law enforcement officials was found to be full of errors, including 42 babies who had been added to the database when they were less than 1 year old (28 of whom were marked as "admitting to being gang members"). In this case, there was no process in place for correcting mistakes or removing people after they'd been added. Another example is the US credit report system: a large-scale study of credit reports by the Federal Trade Commission (FTC) in 2012 found that 26% of consumers had at least one mistake in their files, and 5% had errors that could be devastating.

Yet, the process of getting such errors corrected is incredibly slow and opaque. When public radio reporter Bobby Allyn discovered that he was erroneously listed as having a firearms conviction, it took him "more than a dozen phone calls, the handiwork of a

county court clerk and six weeks to solve the problem. And that was only after I contacted the company's communications department as a journalist."

As machine learning practitioners, we do not always think of it as our responsibility to understand how our algorithms end up being implemented in practice. But we need to.

Feedback Loops

We explained in Chapter 1 how an algorithm can interact with its environment to create a feedback loop, making predictions that reinforce actions taken in the real world, which lead to predictions even more pronounced in the same direction. As an example, let's again consider YouTube's recommendation system. A couple of years ago, the Google team talked about how they had introduced reinforcement learning (closely related to deep learning, but your loss function represents a result potentially a long time after an action occurs) to improve YouTube's recommendation system. They described how they used an algorithm that made recommendations such that watch time would be optimized.

However, human beings tend to be drawn to controversial content. This meant that videos about things like conspiracy theories started to get recommended more and more by the recommendation system. Furthermore, it turns out that the kinds of people who are interested in conspiracy theories are also people who watch a lot of online videos! So, they started to get drawn more and more toward YouTube. The increasing number of conspiracy theorists watching videos on YouTube resulted in the algorithm recommending more and more conspiracy theory and other extremist content, which resulted in more extremists watching videos on YouTube, and more people watching YouTube developing extremist views, which led to the algorithm recommending more extremist content. The system was spiraling out of control.

And this phenomenon was not contained to this particular type of content. In June 2019, the *New York Times* published an article on YouTube's recommendation system titled "On YouTube's Digital Playground, an Open Gate for Pedophiles". The article started with this chilling story:

> *Christiane C. didn't think anything of it when her 10-year-old daughter and a friend uploaded a video of themselves playing in a backyard pool...A few days later...the video had thousands of views. Before long, it had ticked up to 400,000..."I saw the video again and I got scared by the number of views," Christiane said. She had reason to be. YouTube's automated recommendation system...had begun showing the video to users who watched other videos of prepubescent, partially clothed children, a team of researchers has found.*
>
> *On its own, each video might be perfectly innocent, a home movie, say, made by a child. Any revealing frames are fleeting and appear accidental. But, grouped together, their shared features become unmistakable.*

YouTube's recommendation algorithm had begun curating playlists for pedophiles, picking out innocent home videos that happened to contain prepubescent, partially clothed children.

No one at Google planned to create a system that turned family videos into porn for pedophiles. So what happened?

Part of the problem here is the centrality of metrics in driving a financially important system. When an algorithm has a metric to optimize, as you have seen, it will do everything it can to optimize that number. This tends to lead to all kinds of edge cases, and humans interacting with a system will search for, find, and exploit these edge cases and feedback loops for their advantage.

There are signs that this is exactly what has happened with YouTube's recommendation system in 2018. *The Guardian* ran an article called "How an Ex-YouTube Insider Investigated Its Secret Algorithm" about Guillaume Chaslot, an ex-YouTube engineer who created a website that tracks these issues. Chaslot published the chart in Figure 3-5 following the release of Robert Mueller's "Report on the Investigation Into Russian Interference in the 2016 Presidential Election."

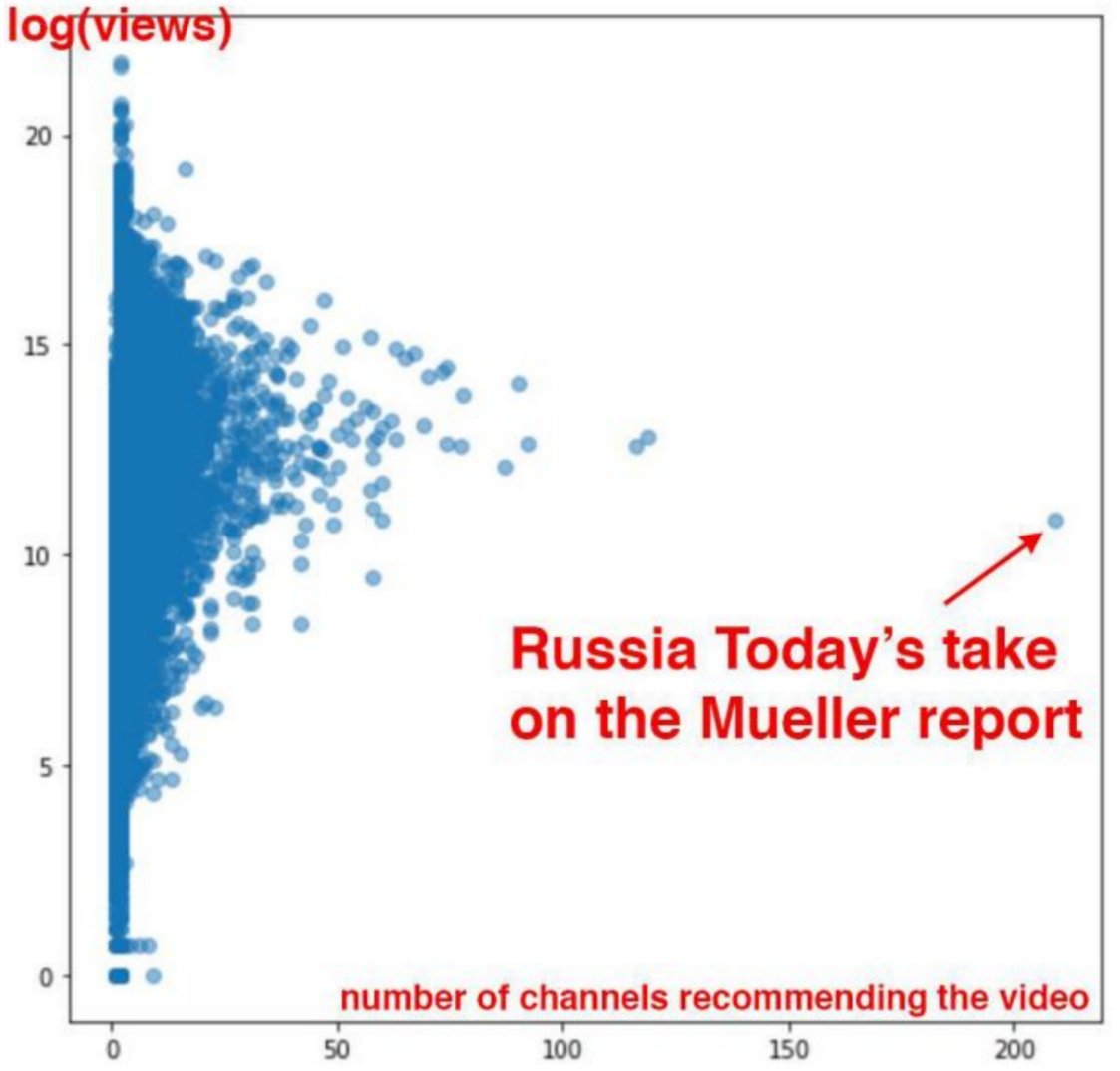


Figure 3-5. Coverage of the Mueller report

Russia Today's coverage of the Mueller report was an extreme outlier in terms of how many channels were recommending it. This suggests the possibility that Russia Today, a state-owned Russia media outlet, has been successful in gaming YouTube's recommendation algorithm. Unfortunately, the lack of transparency of systems like this makes it hard to uncover the kinds of problems that we're discussing.

One of our reviewers for this book, Aurélien Géron, led YouTube's video classification team from 2013 to 2016 (well before the events discussed here). He pointed out that it's not just feedback loops involving humans that are a problem. There can also be feedback loops without humans! He told us about an example from YouTube:

One important signal to classify the main topic of a video is the channel it comes from. For example, a video uploaded to a cooking channel is very likely to be a cooking video. But how do we know what topic a channel is about? Well...in part by looking at the topics of the videos it contains! Do you see the loop? For example, many videos have a description which indicates what camera was used to shoot the video. As a result, some of these videos might get classified as videos about "photography." If a channel has such a misclassified video, it might be classified as a "photography" channel, making it even more likely for future videos on this channel to be wrongly classified as "photography." This could even lead to runaway virus-like classifications! One way to break this feedback loop is to classify videos with and without the channel signal. Then when classifying the channels, you can only use the classes obtained without the channel signal. This way, the feedback loop is broken.

There are positive examples of people and organizations attempting to combat these problems. Evan Estola, lead machine learning engineer at Meetup, discussed the example of men expressing more interest than women in tech meetups. Taking gender into account could therefore cause Meetup's algorithm to recommend fewer tech meetups to women, and as a result, fewer women would find out about and attend tech meetups, which could cause the algorithm to suggest even fewer tech meetups to women, and so on in a self-reinforcing feedback loop. So, Evan and his team made the ethical decision for their recommendation algorithm to not create such a feedback loop, by explicitly not using gender for that part of their model. It is encouraging to see a company not just unthinkingly optimize a metric, but consider its impact. According to Evan, "You need to decide which feature not to use in your algorithm... the most optimal algorithm is perhaps not the best one to launch into production."

While Meetup chose to avoid such an outcome, Facebook provides an example of allowing a runaway feedback loop to run wild. Like YouTube, it tends to radicalize users interested in one conspiracy theory by introducing them to more. As Renee DiResta, a researcher on proliferation of disinformation, writes:

Once people join a single conspiracy-minded [Facebook] group, they are algorithmically routed to a plethora of others. Join an anti-vaccine group, and your suggestions will include anti-GMO, chemtrail watch, flat Earther (yes, really), and "curing cancer naturally" groups. Rather than pulling a user out of the rabbit hole, the recommendation engine pushes them further in.

It is extremely important to keep in mind that this kind of behavior can happen, and to either anticipate a feedback loop or take positive action to break it when you see the first signs of it in your own projects. Another thing to keep in mind is *bias*, which, as we discussed briefly in the previous chapter, can interact with feedback loops in very troublesome ways.

Bias

Discussions of bias online tend to get pretty confusing pretty fast. The word "bias"

means so many different things. Statisticians often think when data ethicists are talking about bias that they're talking about the statistical definition of the term bias—but they're not. And they're certainly not talking about the biases that appear in the weights and biases that are the parameters of your model!

What they're talking about is the social science concept of bias. In "A Framework for Understanding Unintended Consequences of Machine Learning" MIT's Harini Suresh and John Guttag describe six types of bias in machine learning, summarized in Figure 3-6.

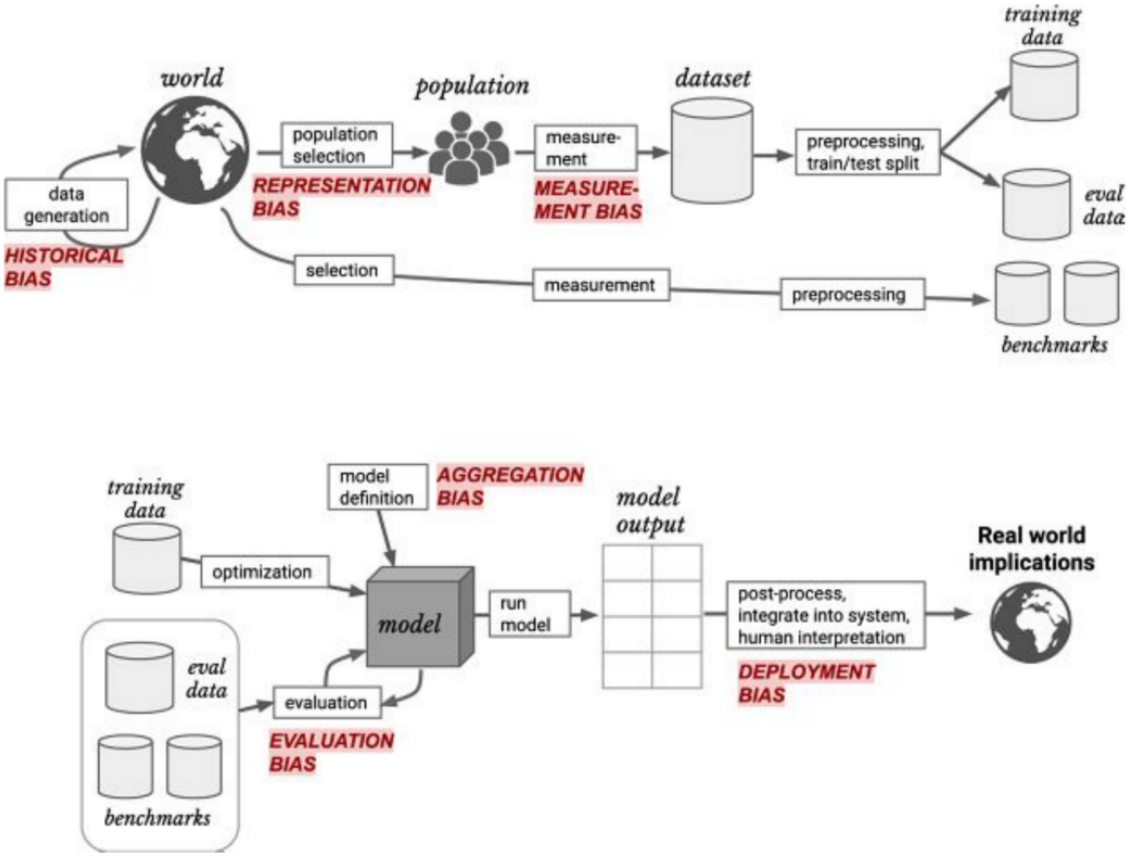


Figure 3-6. Bias in machine learning can come from multiple sources (courtesy of Harini Suresh and John V. Guttag)

We'll discuss four of these types of bias, those that we've found most helpful in our own work (see the paper for details on the others).

Historical bias

Historical bias comes from the fact that people are biased, processes are biased, and society is biased. Suresh and Guttag say: "Historical bias is a fundamental, structural issue with the first step of the data generation process and can exist even given perfect

sampling and feature selection."

For instance, here are a few examples of historical *race bias* in the US, from the *New York Times* article "Racial Bias, Even When We Have Good Intentions" by the University of Chicago's Sendhil Mullainathan:

- When doctors were shown identical files, they were much less likely to recommend cardiac catheterization (a helpful procedure) to Black patients.
- When bargaining for a used car, Black people were offered initial prices $700 higher and received far smaller concessions.
- Responding to apartment rental ads on Craigslist with a Black name elicited fewer responses than with a white name.
- An all-white jury was 16 percentage points more likely to convict a Black defendant than a white one, but when a jury had one Black member, it convicted both at the same rate.

The COMPAS algorithm, widely used for sentencing and bail decisions in the US, is an example of an important algorithm that, when tested by ProPublica, showed clear racial bias in practice (Figure 3-7).

Prediction Fails Differently for Black Defendants

	WHITE	AFRICAN AMERICAN
Labeled Higher Risk, But Didn't Re-Offend	23.5%	44.9%
Labeled Lower Risk, Yet Did Re-Offend	47.7%	28.0%

Figure 3-7. Results of the COMPAS algorithm

Any dataset involving humans can have this kind of bias: medical data, sales data, housing data, political data, and so on. Because underlying bias is so pervasive, bias in datasets is very pervasive. Racial bias even turns up in computer vision, as shown in the example of autocategorized photos shared on Twitter by a Google Photos user shown in Figure 3-8.

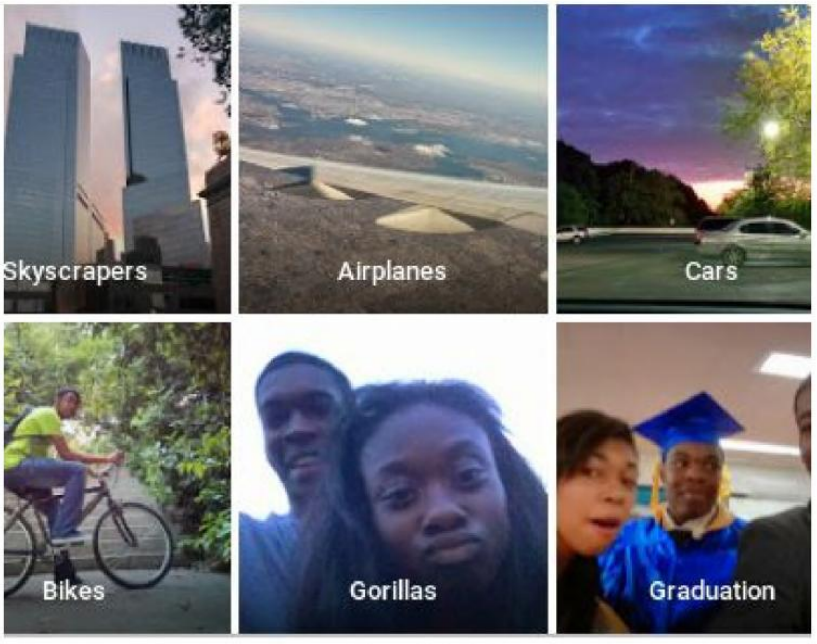


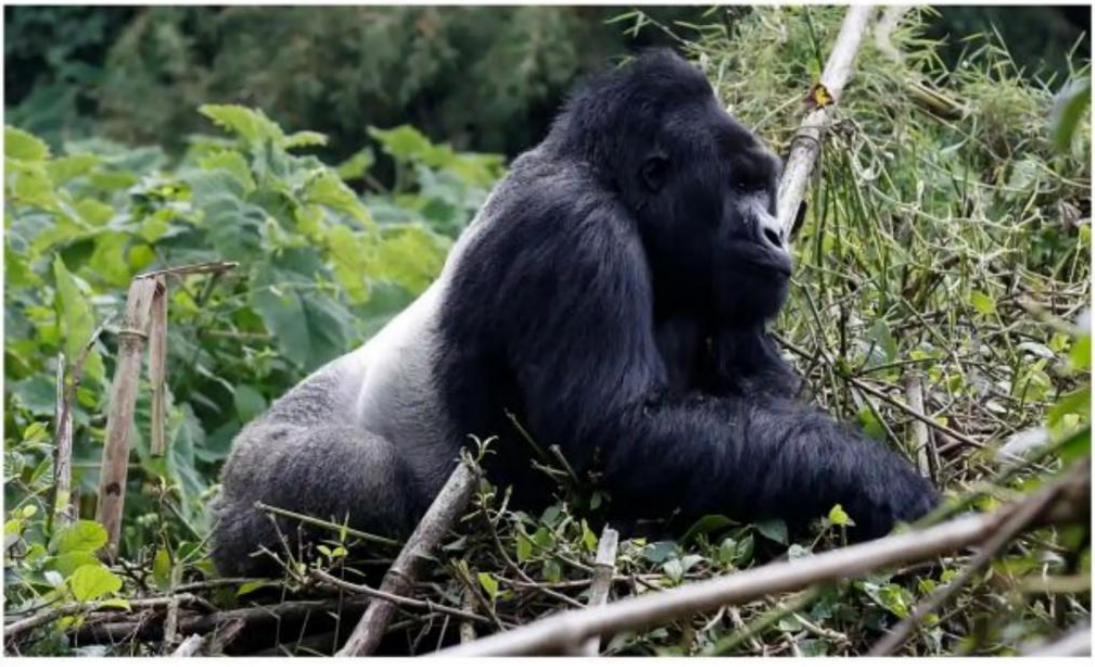
Figure 3-8. One of these labels is very wrong...

Yes, that is showing what you think it is: Google Photos classified a Black user's photo with their friend as "gorillas"! This algorithmic misstep got a lot of attention in the media. "We're appalled and genuinely sorry that this happened," a company spokeswoman said. "There is still clearly a lot of work to do with automatic image labeling, and we're looking at how we can prevent these types of mistakes from happening in the future."

Unfortunately, fixing problems in machine learning systems when the input data has problems is hard. Google's first attempt didn't inspire confidence, as coverage by *The Guardian* suggested (Figure 3-9).

Google's solution to accidental algorithmic racism: ban gorillas

Google's 'immediate action' over AI labelling of black people as gorillas was simply to block the word, along with chimpanzee and monkey, reports suggest



▲ A silverback high mountain gorilla, which you'll no longer be able to label satisfactorily on Google Photos. Photograph: Thomas Mukoya/Reuters

Figure 3-9. Google's first response to the problem

These kinds of problems are certainly not limited to Google. MIT researchers studied the most popular online computer vision APIs to see how accurate they were. But they didn't just calculate a single accuracy number—instead, they looked at the accuracy across four groups, as illustrated in Figure 3-10.

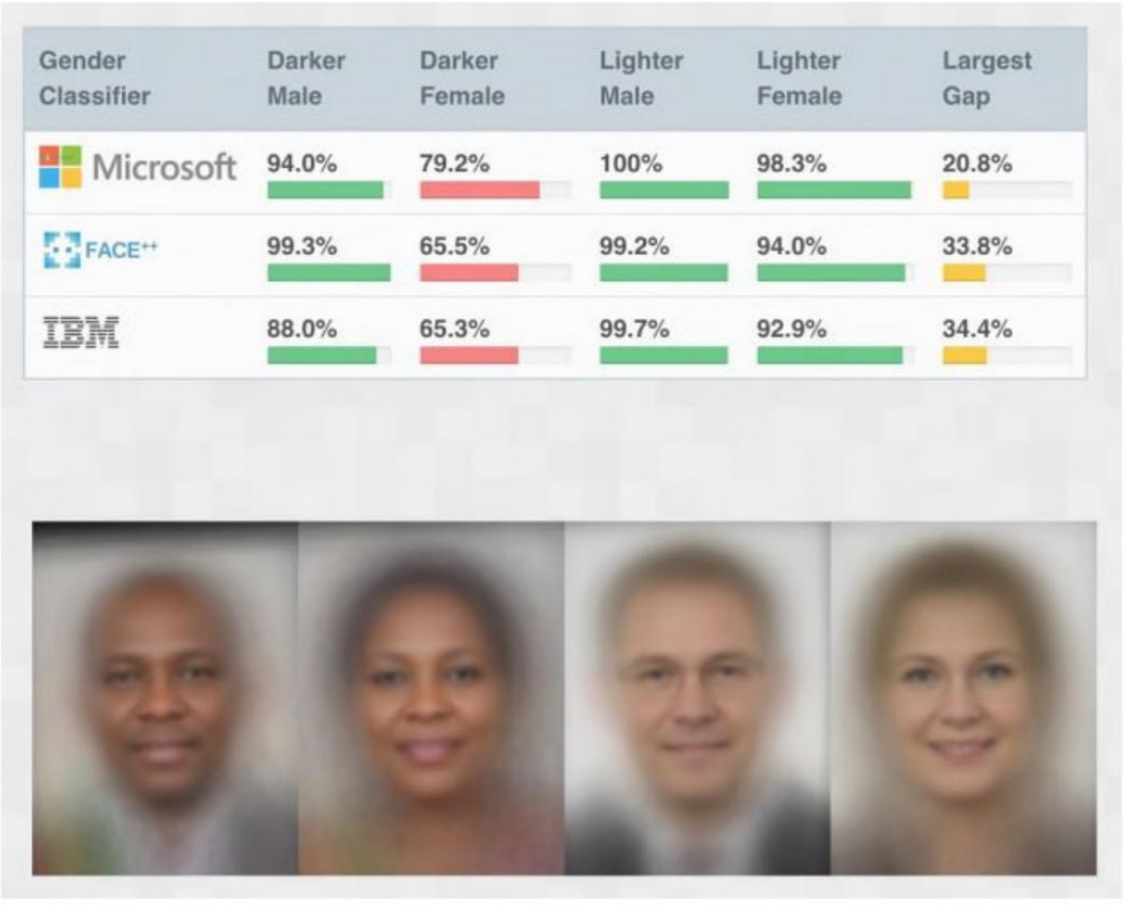


Figure 3-10. Error rate per gender and race for various facial recognition systems

IBM's system, for instance, had a 34.7% error rate for darker females, versus 0.3% for lighter males—over 100 times more errors! Some people incorrectly reacted to these experiments by claiming that the difference was simply because darker skin is harder for computers to recognize. However, what happened was that, after the negative publicity that this result created, all of the companies in question dramatically improved their models for darker skin, such that one year later, they were nearly as good as for lighter skin. So what this showed is that the developers failed to utilize datasets containing enough darker faces, or test their product with darker faces.

One of the MIT researchers, Joy Buolamwini, warned: "We have entered the age of automation overconfident yet underprepared. If we fail to make ethical and inclusive artificial intelligence, we risk losing gains made in civil rights and gender equity under the guise of machine neutrality."

Part of the issue appears to be a systematic imbalance in the makeup of popular datasets used for training models. The abstract of the paper "No Classification Without

Representation: Assessing Geodiversity Issues in Open Data Sets for the Developing World" by Shreya Shankar et al. states, "We analyze two large, publicly available image data sets to assess geo-diversity and find that these data sets appear to exhibit an observable amerocentric and eurocentric representation bias. Further, we analyze classifiers trained on these data sets to assess the impact of these training distributions and find strong differences in the relative performance on images from different locales." Figure 3-11 shows one of the charts from the paper, showing the geographic makeup of what were at the time (and still are, as this book is being written) the two most important image datasets for training models.

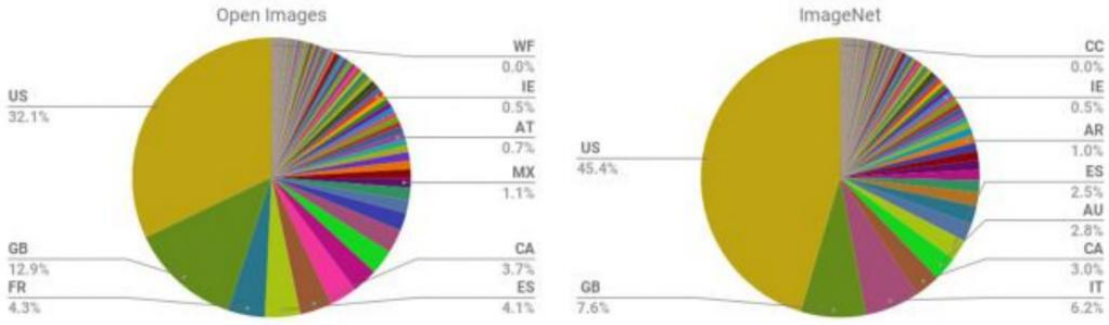


Figure 3-11. Image provenance in popular training sets

The vast majority of the images are from the US and other Western countries, leading to models trained on ImageNet performing worse on scenes from other countries and cultures. For instance, research found that such models are worse at identifying household items (such as soap, spices, sofas, or beds) from lower-income countries. Figure 3-12 shows an image from the paper "Does Object Recognition Work for Everyone?" by Terrance DeVries et al. of Facebook AI Research that illustrates this point.

Ground truth: Soap **Nepal, 288 $/month**

Azure: food, cheese, bread, cake, sandwich
Clarifai: food, wood, cooking, delicious, healthy
Google: food, dish, cuisine, comfort food, spam
Amazon: food, confectionary, sweets, burger
Watson: food, food product, turmeric, seasoning
Tencent: food, dish, matter, fast food, nutriment

Ground truth: Soap **UK, 1890 $/month**

Azure: toilet, design, art, sink
Clarifai: people, faucet, healthcare, lavatory, wash closet
Google: product, liquid, water, fluid, bathroom accessory
Amazon: sink, indoors, bottle, sink faucet
Watson: gas tank, storage tank, toiletry, dispenser, soap dispenser
Tencent: lotion, toiletry, soap dispenser, dispenser, after shave

Ground truth: Spices **Phillipines, 262 $/month**

Azure: bottle, beer, counter, drink, open
Clarifai: container, food, bottle, drink, stock
Google: product, yellow, drink, bottle, plastic bottle
Amazon: beverage, beer, alcohol, drink, bottle
Watson: food, larder food supply, pantry, condiment, food seasoning
Tencent: condiment, sauce, flavorer, catsup, hot sauce

Ground truth: Spices **USA, 4559 $/month**

Azure: bottle, wall, counter, food
Clarifai: container, food, can, medicine, stock
Google: seasoning, seasoned salt, ingredient, spice, spice rack
Amazon: shelf, tin, pantry, furniture, aluminium
Watson: tin, food, pantry, paint, can
Tencent: spice rack, chili sauce, condiment, canned food, rack

Figure 3-12. Object detection in action

In this example, we can see that the lower-income soap example is a very long way

away from being accurate, with every commercial image recognition service predicting "food" as the most likely answer!

As we will discuss shortly, in addition, the vast majority of AI researchers and developers are young white men. Most projects that we have seen do most user testing using friends and families of the immediate product development group. Given this, the kinds of problems we just discussed should not be surprising.

Similar historical bias is found in the texts used as data for natural language processing models. This crops up in downstream machine learning tasks in many ways. For instance, it was widely reported that until last year, Google Translate showed systematic bias in how it translated the Turkish gender-neutral pronoun "o" into English: when applied to jobs that are often associated with males, it used "he," and when applied to jobs that are often associated with females, it used "she" (Figure 3-13).

Figure 3-13. Gender bias in text datasets

We also see this kind of bias in online advertisements. For instance, a study in 2019 by Muhammad Ali et al. found that even when the person placing the ad does not intentionally discriminate, Facebook will show ads to very different audiences based on race and gender. Housing ads with the same text but picturing either a white or a Black family were shown to racially different audiences.

Measurement bias

In "Does Machine Learning Automate Moral Hazard and Error" in *American Economic Review*, Sendhil Mullainathan and Ziad Obermeyer look at a model that tries to answer

this question: using historical electronic health record (EHR) data, what factors are most predictive of stroke? These are the top predictors from the model:

- Prior stroke
- Cardiovascular disease
- Accidental injury
- Benign breast lump
- Colonoscopy
- Sinusitis

However, only the top two have anything to do with a stroke! Based on what we've studied so far, you can probably guess why. We haven't really measured *stroke*, which occurs when a region of the brain is denied oxygen due to an interruption in the blood supply. What we've measured is who had symptoms, went to a doctor, got the appropriate tests, *and* received a diagnosis of stroke. Actually having a stroke is not the only thing correlated with this complete list—it's also correlated with being the kind of person who goes to the doctor (which is influenced by who has access to healthcare, can afford their co-pay, doesn't experience racial or gender-based medical discrimination, and more)! If you are likely to go to the doctor for an *accidental injury*, you are likely to also go the doctor when you are having a stroke.

This is an example of *measurement bias*. It occurs when our models make mistakes because we are measuring the wrong thing, or measuring it in the wrong way, or incorporating that measurement into the model inappropriately.

Aggregation bias

Aggregation bias occurs when models do not aggregate data in a way that incorporates all of the appropriate factors, or when a model does not include the necessary interaction terms, nonlinearities, or so forth. This can particularly occur in medical settings. For instance, the way diabetes is treated is often based on simple univariate statistics and studies involving small groups of heterogeneous people. Analysis of results is often done in a way that does not take into account different ethnicities or genders. However, it turns out that diabetes patients have different complications across ethnicities, and HbA1c levels (widely used to diagnose and monitor diabetes) differ in complex ways across ethnicities and genders. This can result in people being misdiagnosed or incorrectly treated because medical decisions are based on a model that does not include these important variables and interactions.

Representation bias

The abstract of the paper "Bias in Bios: A Case Study of Semantic Representation Bias in a High-Stakes Setting" by Maria De-Arteaga et al. notes that there is gender imbalance in occupations (e.g., females are more likely to be nurses, and males are more likely to be pastors), and says that "differences in true positive rates between genders are correlated with existing gender imbalances in occupations, which may compound these imbalances."

In other words, the researchers noticed that models predicting occupation did not only *reflect* the actual gender imbalance in the underlying population, but *amplified* it! This type of *representation bias* is quite common, particularly for simple models. When there is a clear, easy-to-see underlying relationship, a simple model will often assume that this relationship holds all the time. As Figure 3-14 from the paper shows, for occupations that had a higher percentage of females, the model tended to overestimate the prevalence of that occupation.

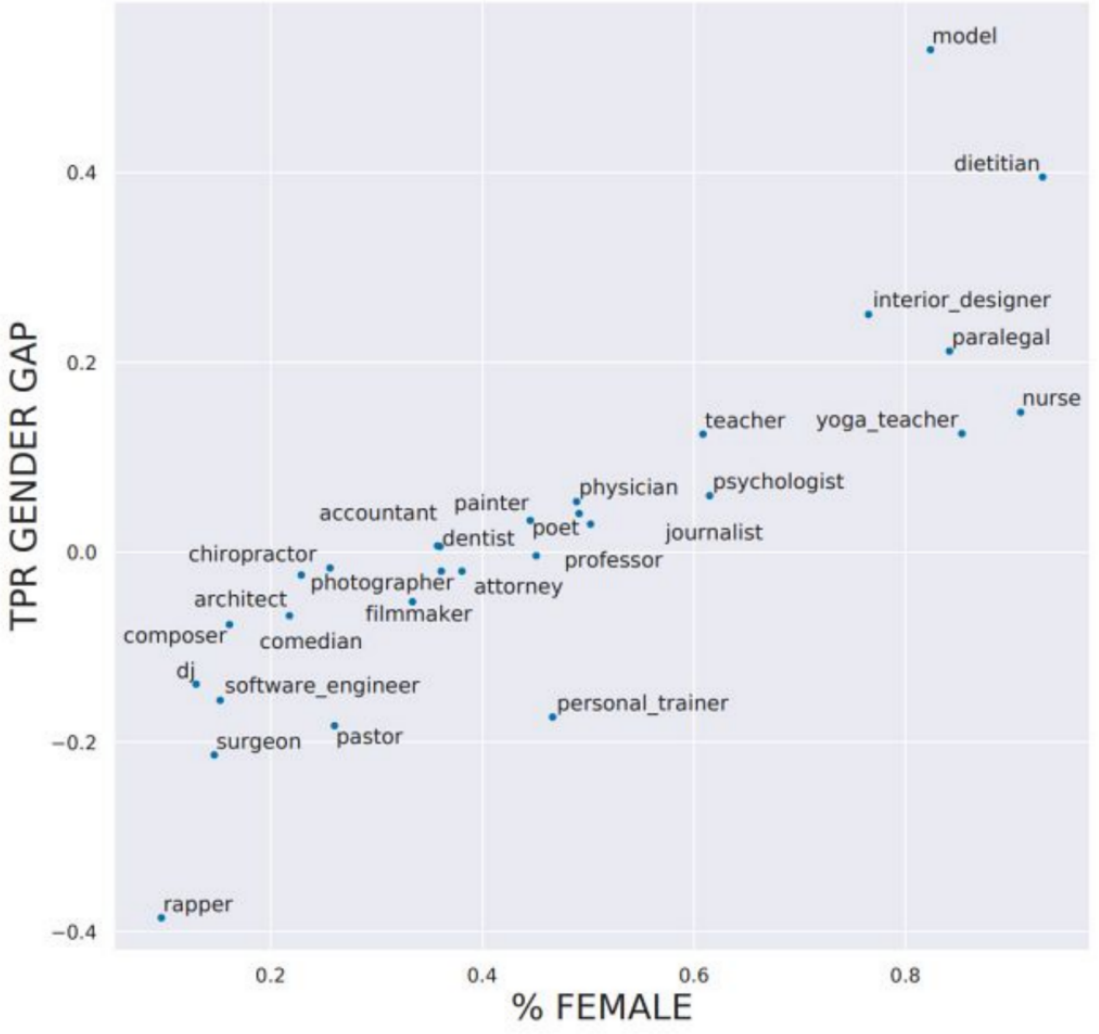


Figure 3-14. Model error in predicting occupation plotted against percentage of women in said occupation

For example, in the training dataset 14.6% of surgeons were women, yet in the model predictions only 11.6% of the true positives were women. The model is thus amplifying the bias existing in the training set.

Now that we've seen that those biases exist, what can we do to mitigate them?

Addressing different types of bias

Different types of bias require different approaches for mitigation. While gathering a more diverse dataset can address representation bias, this would not help with historical bias or measurement bias. All datasets contain bias. There is no such thing as a completely debiased dataset. Many researchers in the field have been converging on a set of proposals to enable better documentation of the decisions, context, and

specifics about how and why a particular dataset was created, what scenarios it is appropriate to use in, and what the limitations are. This way, those using a particular dataset will not be caught off guard by its biases and limitations.

We often hear the question, "Humans are biased, so does algorithmic bias even matter?" This comes up so often, there must be some reasoning that makes sense to the people who ask it, but it doesn't seem very logically sound to us! Independently of whether this is logically sound, it's important to realize that algorithms (particularly machine learning algorithms!) and people are different. Consider these points about machine learning algorithms:

Machine learning can create feedback loops

Small amounts of bias can rapidly increase exponentially because of feedback loops.

Machine learning can amplify bias

Human bias can lead to larger amounts of machine learning bias.

Algorithms and humans are used differently

Human decision makers and algorithmic decision makers are not used in a plug-and-play interchangeable way in practice. These examples are given in the list on the next page.

Technology is power

And with that comes responsibility.

As the Arkansas healthcare example showed, machine learning is often implemented in practice not because it leads to better outcomes, but because it is cheaper and more efficient. Cathy O'Neill, in her book *Weapons of Math Destruction* (Crown), described a pattern in which the privileged are processed by people, whereas the poor are processed by algorithms. This is just one of a number of ways that algorithms are used differently than human decision makers. Others include the following:

- People are more likely to assume algorithms are objective or error-free (even if they're given the option of a human override).
- Algorithms are more likely to be implemented with no appeals process in place.

- Algorithms are often used at scale.
- Algorithmic systems are cheap.

Even in the absence of bias, algorithms (and deep learning especially, since it is such an effective and scalable algorithm) can lead to negative societal problems, such as when used for *disinformation*.

Disinformation

Disinformation has a history stretching back hundreds or even thousands of years. It is not necessarily about getting someone to believe something false, but rather often used to sow disharmony and uncertainty, and to get people to give up on seeking the truth. Receiving conflicting accounts can lead people to assume that they can never know whom or what to trust.

Some people think disinformation is primarily about false information or *fake news*, but in reality, disinformation can often contain seeds of truth, or half-truths taken out of context. Ladislav Bittman was an intelligence officer in the USSR who later defected to the US and wrote some books in the 1970s and 1980s on the role of disinformation in Soviet propaganda operations. In *The KGB and Soviet Disinformation* (Pergamon), he wrote "Most campaigns are a carefully designed mixture of facts, half-truths, exaggerations, and deliberate lies."

In the US, this has hit close to home in recent years, with the FBI detailing a massive disinformation campaign linked to Russia in the 2016 election. Understanding the disinformation that was used in this campaign is very educational. For instance, the FBI found that the Russian disinformation campaign often organized two separate fake "grass roots" protests, one for each side of an issue, and got them to protest at the same time! The Houston Chronicle reported on one of these odd events (Figure 3-15):

> *A group that called itself the "Heart of Texas" had organized it on social media—a protest, they said, against the "Islamization" of Texas. On one side of Travis Street, I found about 10 protesters. On the other side, I found around 50 counterprotesters. But I couldn't find the rally organizers. No "Heart of Texas." I thought that was odd, and mentioned it in the article: What kind of group is a no-show at its own event? Now I know why. Apparently, the rally's organizers were in Saint Petersburg, Russia, at the time. "Heart of Texas" is one of the internet troll groups cited in Special Prosecutor Robert Mueller's recent indictment of Russians attempting to tamper with the US presidential election.*

Figure 3-15. Event organized by the group Heart of Texas

Disinformation often involves coordinated campaigns of inauthentic behavior. For instance, fraudulent accounts may try to make it seem like many people hold a particular viewpoint. While most of us like to think of ourselves as independent-minded, in reality we evolved to be influenced by others in our in-group, and in opposition to those in our out-group. Online discussions can influence our viewpoints, or alter the range of what we consider acceptable viewpoints. Humans are social animals, and as social animals, we are extremely influenced by the people around us. Increasingly, radicalization occurs in online environments; so influence is coming from people in the virtual space of online forums and social networks.

Disinformation through autogenerated text is a particularly significant issue, due to the greatly increased capability provided by deep learning. We discuss this issue in

depth when we delve into creating language models in Chapter 10.

One proposed approach is to develop some form of digital signature, to implement it in a seamless way, and to create norms that we should trust only content that has been verified. The head of the Allen Institute on AI, Oren Etzioni, wrote such a proposal in an article titled "How Will We Prevent AI-Based Forgery?": "AI is poised to make high-fidelity forgery inexpensive and automated, leading to potentially disastrous consequences for democracy, security, and society. The specter of AI forgery means that we need to act to make digital signatures de rigueur as a means of authentication of digital content."

While we can't hope to discuss all the ethical issues that deep learning, and algorithms more generally, bring up, hopefully this brief introduction has been a useful starting point you can build on. We'll now move on to the questions of how to identify ethical issues and what to do about them.

Identifying and Addressing Ethical Issues

Mistakes happen. Finding out about them, and dealing with them, needs to be part of the design of any system that includes machine learning (and many other systems too). The issues raised within data ethics are often complex and interdisciplinary, but it is crucial that we work to address them.

So what can we do? This is a big topic, but here are a few steps toward addressing ethical issues:

- Analyze a project you are working on.
- Implement processes at your company to find and address ethical risks.
- Support good policy.
- Increase diversity.

Let's walk through each step, starting with analyzing a project you are working on.

Analyze a Project You Are Working On

It's easy to miss important issues when considering ethical implications of your work. One thing that helps enormously is simply asking the right questions. Rachel Thomas recommends considering the following questions throughout the development of a data project:

- Should we even be doing this?
- What bias is in the data?

- Can the code and data be audited?
- What are the error rates for different subgroups?
- What is the accuracy of a simple rule-based alternative?
- What processes are in place to handle appeals or mistakes?
- How diverse is the team that built it?

These questions may be able to help you identify outstanding issues, and possible alternatives that are easier to understand and control. In addition to asking the right questions, it's also important to consider practices and processes to implement.

One thing to consider at this stage is what data you are collecting and storing. Data often ends up being used for different purposes the original intent. For instance, IBM began selling to Nazi Germany well before the Holocaust, including helping with Germany's 1933 census conducted by Adolf Hitler, which was effective at identifying far more Jewish people than had previously been recognized in Germany. Similarly, US census data was used to round up Japanese-Americans (who were US citizens) for internment during World War II. It is important to recognize how data and images collected can be weaponized later. Columbia professor Tim Wu wrote "You must assume that any personal data that Facebook or Android keeps are data that governments around the world will try to get or that thieves will try to steal."

Processes to Implement

The Markkula Center has released An Ethical Toolkit for Engineering/Design Practice that includes concrete practices to implement at your company, including regularly scheduled sweeps to proactively search for ethical risks (in a manner similar to cybersecurity penetration testing), expanding the ethical circle to include the perspectives of a variety of stakeholders, and considering the terrible people (how could bad actors abuse, steal, misinterpret, hack, destroy, or weaponize what you are building?).

Even if you don't have a diverse team, you can still try to proactively include the perspectives of a wider group, considering questions such as these (provided by the Markkula Center):

- Whose interests, desires, skills, experiences, and values have we simply assumed, rather than actually consulted?
- Who are all the stakeholders who will be directly affected by our product? How have their interests been protected? How do we know what their interests really are—have we asked?

- Who/which groups and individuals will be indirectly affected in significant ways?
- Who might use this product that we didn't expect to use it, or for purposes we didn't initially intend?

Ethical lenses

Another useful resource from the Markkula Center is its Conceptual Frameworks in Technology and Engineering Practice. This considers how different foundational ethical lenses can help identify concrete issues, and lays out the following approaches and key questions:

The rights approach

Which option best respects the rights of all who have a stake?

The justice approach

Which option treats people equally or proportionately?

The utilitarian approach

Which option will produce the most good and do the least harm?

The common good approach

Which option best serves the community as a whole, not just some members?

The virtue approach

Which option leads me to act as the sort of person I want to be?

Markkula's recommendations include a deeper dive into each of these perspectives, including looking at a project through the lens of its *consequences*:

- Who will be directly affected by this project? Who will be indirectly affected?
- Will the effects in aggregate likely create more good than harm, and what types of good and harm?
- Are we thinking about all relevant types of harm/benefit (psychological, political, environmental, moral, cognitive, emotional, institutional, cultural)?
- How might future generations be affected by this project?

- Do the risks of harm from this project fall disproportionately on the least powerful in society? Will the benefits go disproportionately to the well-off?
- Have we adequately considered "dual-use" and unintended downstream effects?

The alternative lens to this is the *deontological* perspective, which focuses on basic concepts of *right* and *wrong*:

- What rights of others and duties to others must we respect?
- How might the dignity and autonomy of each stakeholder be impacted by this project?
- What considerations of trust and of justice are relevant to this design/project?
- Does this project involve any conflicting moral duties to others, or conflicting stakeholder rights? How can we prioritize these?

One of the best ways to help come up with complete and thoughtful answers to questions like these is to ensure that the people asking the questions are *diverse*.

The Power of Diversity

Currently, less than 12% of AI researchers are women, according to a study from Element AI. The statistics are similarly dire when it comes to race and age. When everybody on a team has similar backgrounds, they are likely to have similar blind spots around ethical risks. The *Harvard Business Review* (HBR) has published a number of studies showing many benefits of diverse teams, including the following:

- "How Diversity Can Drive Innovation"
- "Teams Solve Problems Faster When They're More Cognitively Diverse"
- "Why Diverse Teams Are Smarter"
- "Defend Your Research: What Makes a Team Smarter? More Women"

Diversity can lead to problems being identified earlier, and a wider range of solutions being considered. For instance, Tracy Chou was an early engineer at Quora. She wrote of her experiences, describing how she advocated internally for adding a feature that would allow trolls and other bad actors to be blocked. Chou recounts, "I was eager to work on the feature because I personally felt antagonized and abused on the site (gender isn't an unlikely reason as to why)...But if I hadn't had that personal perspective, it's possible that the Quora team wouldn't have prioritized building a block button so early in its existence." Harassment often drives people from

marginalized groups off online platforms, so this functionality has been important for maintaining the health of Quora's community.

A crucial aspect to understand is that women leave the tech industry at over twice the rate that men do. According to the Harvard Business Review, 41% of women working in tech leave, compared to 17% of men. An analysis of over 200 books, whitepapers, and articles found that the reason they leave is that "they're treated unfairly; underpaid, less likely to be fast-tracked than their male colleagues, and unable to advance."

Studies have confirmed a number of the factors that make it harder for women to advance in the workplace. Women receive more vague feedback and personality criticism in performance evaluations, whereas men receive actionable advice tied to business outcomes (which is more useful). Women frequently experience being excluded from more creative and innovative roles, and not receiving high-visibility "stretch" assignments that are helpful in getting promoted. One study found that men's voices are perceived as more persuasive, fact-based, and logical than women's voices, even when reading identical scripts.

Receiving mentorship has been statistically shown to help men advance, but not women. The reason behind this is that when women receive mentorship, it's advice on how they should change and gain more self-knowledge. When men receive mentorship, it's public endorsement of their authority. Guess which is more useful in getting promoted?

As long as qualified women keep dropping out of tech, teaching more girls to code will not solve the diversity issues plaguing the field. Diversity initiatives often end up focusing primarily on white women, even though women of color face many additional barriers. In interviews with 60 women of color who work in STEM research, 100% had experienced discrimination.

The hiring process is particularly broken in tech. One study indicative of the disfunction comes from Triplebyte, a company that helps place software engineers in companies, conducting a standardized technical interview as part of this process. The company has a fascinating dataset: the results of how over 300 engineers did on their exam, coupled with the results of how those engineers did during the interview process for a variety of companies. The number one finding from Triplebyte's research is that "the types of programmers that each company looks for often have little to do with what the company needs or does. Rather, they reflect company culture and the backgrounds of the founders."

This is a challenge for those trying to break into the world of deep learning, since most companies' deep learning groups today were founded by academics. These groups tend

to look for people "like them"—that is, people who can solve complex math problems and understand dense jargon. They don't always know how to spot people who are actually good at solving real problems using deep learning.

This leaves a big opportunity for companies that are ready to look beyond status and pedigree, and focus on results!

Fairness, Accountability, and Transparency

The professional society for computer scientists, the ACM, runs a data ethics conference called the Conference on Fairness, Accountability, and Transparency (ACM FAccT), which used to go under the acronym FAT but now uses the less objectionable FAccT. Microsoft also has a group focused on Fairness, Accountability, Transparency, and Ethics in AI (FATE). In this section, we'll use the acronym FAccT to refer to the concepts of fairness, accountability, and transparency.

FAccT is a lens some people have used for considering ethical issues. One helpful resource for this is the free online book *Fairness and Machine Learning: Limitations and Opportunities* by Solon Barocas et al., which "gives a perspective on machine learning that treats fairness as a central concern rather than an afterthought." It also warns, however, that it "is intentionally narrow in scope...A narrow framing of machine learning ethics might be tempting to technologists and businesses as a way to focus on technical interventions while sidestepping deeper questions about power and accountability. We caution against this temptation." Rather than provide an overview of the FAccT approach to ethics (which is better done in books such as that one), our focus here will be on the limitations of this kind of narrow framing.

One great way to consider whether an ethical lens is complete is to try to come up with an example in which the lens and our own ethical intuitions give diverging results. Os Keyes et al. explored this in a graphic way in their paper "A Mulching Proposal: Analysing and Improving an Algorithmic System for Turning the Elderly into High-Nutrient Slurry". The paper's abstract says:

> *The ethical implications of algorithmic systems have been much discussed in both HCI and the broader community of those interested in technology design, development, and policy. In this paper, we explore the application of one prominent ethical framework—Fairness, Accountability, and Transparency—to a proposed algorithm that resolves various societal issues around food security and population aging. Using various standardised forms of algorithmic audit and evaluation, we drastically increase the algorithm's adherence to the FAT framework, resulting in a more ethical and beneficent system. We discuss how this might serve as a guide to other researchers or practitioners looking to ensure better ethical outcomes from algorithmic systems in their line of work.*

In this paper, the rather controversial proposal ("Turning the Elderly into High-

Nutrient Slurry") and the results ("drastically increase the algorithm's adherence to the FAT framework, resulting in a more ethical and beneficent system") are at odds... to say the least!

In philosophy, and especially philosophy of ethics, this is one of the most effective tools: first, come up with a process, definition, set of questions, etc., which is designed to resolve a problem. Then try to come up with an example in which that apparent solution results in a proposal that no one would consider acceptable. This can then lead to a further refinement of the solution.

So far, we've focused on things that you and your organization can do. But sometimes individual or organizational action is not enough. Sometimes governments also need to consider policy implications.

Role of Policy

We often talk to people who are eager for technical or design fixes to be a full solution to the kinds of problems that we've been discussing; for instance, a technical approach to debias data, or design guidelines for making technology less addictive. While such measures can be useful, they will not be sufficient to address the underlying problems that have led to our current state. For example, as long as it is profitable to create addictive technology, companies will continue to do so, regardless of whether this has the side effect of promoting conspiracy theories and polluting our information ecosystem. While individual designers may try to tweak product designs, we will not see substantial changes until the underlying profit incentives change.

The Effectiveness of Regulation

To look at what can cause companies to take concrete action, consider the following two examples of how Facebook has behaved. In 2018, a UN investigation found that Facebook had played a "determining role" in the ongoing genocide of the Rohingya, an ethnic minority in Mynamar described by UN Secretary-General Antonio Guterres as "one of, if not the, most discriminated people in the world." Local activists had been warning Facebook executives that their platform was being used to spread hate speech and incite violence since as early as 2013. In 2015, they were warned that Facebook could play the same role in Myanmar that the radio broadcasts played during the Rwandan genocide (where a million people were killed). Yet, by the end of 2015, Facebook employed only four contractors who spoke Burmese. As one person close to the matter said, "That's not 20/20 hindsight. The scale of this problem was significant and it was already apparent." Zuckerberg promised during the congressional hearings to hire "dozens" to address the genocide in Myanmar (in 2018, years after the genocide had begun, including the destruction by fire of at least 288 villages in northern Rakhine state after August 2017).

This stands in stark contrast to Facebook quickly hiring 1,200 people in Germany to try to avoid expensive penalties (of up to 50 million euros) under a new German law against hate speech. Clearly, in this case, Facebook was more reactive to the threat of a financial penalty than to the systematic destruction of an ethnic minority.

In an article on privacy issues, Maciej Ceglowski draws parallels with the environmental movement:

> *This regulatory project has been so successful in the First World that we risk forgetting what life was like before it. Choking smog of the kind that today kills thousands in Jakarta and Delhi was once emblematic of London. The Cuyahoga River in Ohio used to reliably catch fire. In a particularly horrific example of unforeseen consequences, tetraethyl lead added to gasoline raised violent crime rates worldwide for fifty years. None of these harms could have been fixed by telling people to vote with their wallet, or carefully review the environmental policies of every company they gave their business to, or to stop using the technologies in question. It took coordinated, and sometimes highly technical, regulation across jurisdictional boundaries to fix them. In some cases, like the ban on commercial refrigerants that depleted the ozone layer, that regulation required a worldwide consensus. We're at the point where we need a similar shift in perspective in our privacy law.*

Rights and Policy

Clean air and clean drinking water are public goods that are nearly impossible to protect through individual market decisions, but rather require coordinated regulatory action. Similarly, many of the harms resulting from unintended consequences of misuses of technology involve public goods, such as a polluted information environment or deteriorated ambient privacy. Too often privacy is framed as an individual right, yet there are societal impacts to widespread surveillance (which would still be the case even if it was possible for a few individuals to opt out).

Many of the issues we are seeing in tech are human rights issues, such as when a biased algorithm recommends that Black defendants have longer prison sentences, when particular job ads are shown only to young people, or when police use facial recognition to identify protesters. The appropriate venue to address human rights issues is typically through the law.

We need both regulatory and legal changes, *and* the ethical behavior of individuals. Individual behavior change can't address misaligned profit incentives, externalities (where corporations reap large profits while offloading their costs and harms to the broader society), or systemic failures. However, the law will never cover all edge cases, and it is important that individual software developers and data scientists are equipped to make ethical decisions in practice.

Cars: A Historical Precedent

The problems we are facing are complex, and there are no simple solutions. This can be discouraging, but we find hope in considering other large challenges that people have tackled throughout history. One example is the movement to increase car safety, covered as a case study in "Datasheets for Datasets" by Timnit Gebru et al. and in the design podcast 99% Invisible. Early cars had no seatbelts, metal knobs on the dashboard that could lodge in people's skulls during a crash, regular plate glass windows that shattered in dangerous ways, and noncollapsible steering columns that impaled drivers. However, car companies were resistant to even discussing safety as something they could help address, and the widespread belief was that cars are just the way they are, and that it was the people using them who caused problems.

It took consumer safety activists and advocates decades of work to change the national conversation to consider that perhaps car companies had some responsibility that should be addressed through regulation. When the collapsible steering column was invented, it was not implemented for several years as there was no financial incentive to do so. Major car company General Motors hired private detectives to try to dig up dirt on consumer safety advocate Ralph Nader. The requirement of seatbelts, crash test dummies, and collapsible steering columns were major victories. It was only in 2011 that car companies were required to start using crash test dummies that would represent the average woman, and not just average men's bodies; prior to this, women were 40% more likely to be injured in a car crash of the same impact compared to a man. This is a vivid example of the ways that bias, policy, and technology have important consequences.

Conclusion

Coming from a background of working with binary logic, the lack of clear answers in ethics can be frustrating at first. Yet, the implications of how our work impacts the world, including unintended consequences and the work becoming weaponized by bad actors, are some of the most important questions we can (and should!) consider. Even though there aren't any easy answers, there are definite pitfalls to avoid and practices to follow to move toward more ethical behavior.

Many people (including us!) are looking for more satisfying, solid answers about how to address harmful impacts of technology. However, given the complex, far-reaching, and interdisciplinary nature of the problems we are facing, there are no simple solutions. Julia Angwin, former senior reporter at ProPublica who focuses on issues of algorithmic bias and surveillance (and one of the 2016 investigators of the COMPAS recidivism algorithm that helped spark the field of FAccT) said in a 2019 interview:

I strongly believe that in order to solve a problem, you have to diagnose it, and that we're still in the diagnosis phase of this. If you think about the turn of the century and industrialization, we had, I don't know, 30 years of child labor, unlimited work hours, terrible working conditions, and it took a lot of journalist muckraking and advocacy to diagnose the problem and have some understanding of what it was, and then the activism to get laws changed. I feel like we're in a second industrialization of data information... I see my role as trying to make as clear as possible what the downsides are, and diagnosing them really accurately so that they can be solvable. That's hard work, and lots more people need to be doing it.

It's reassuring that Angwin thinks we are largely still in the diagnosis phase: if your understanding of these problems feels incomplete, that is normal and natural. Nobody has a "cure" yet, although it is vital that we continue working to better understand and address the problems we are facing.

One of our reviewers for this book, Fred Monroe, used to work in hedge fund trading. He told us, after reading this chapter, that many of the issues discussed here (distribution of data being dramatically different from what a model was trained on, the impact of feedback loops on a model once deployed and at scale, and so forth) were also key issues for building profitable trading models. The kinds of things you need to do to consider societal consequences are going to have a lot of overlap with things you need to do to consider organizational, market, and customer consequences—so thinking carefully about ethics can also help you think carefully about how to make your data product successful more generally!

Questionnaire

1. Does ethics provide a list of "right answers"?
2. How can working with people of different backgrounds help when considering ethical questions?
3. What was the role of IBM in Nazi Germany? Why did the company participate as it did? Why did the workers participate?
4. What was the role of the first person jailed in the Volkswagen diesel scandal?
5. What was the problem with a database of suspected gang members maintained by California law enforcement officials?
6. Why did YouTube's recommendation algorithm recommend videos of partially clothed children to pedophiles, even though no employee at Google had programmed this feature?
7. What are the problems with the centrality of metrics?

8. Why did Meetup.com not include gender in its recommendation system for tech meetups?
9. What are the six types of bias in machine learning, according to Suresh and Guttag?
10. Give two examples of historical race bias in the US.
11. Where are most images in ImageNet from?
12. In the paper "Does Machine Learning Automate Moral Hazard and Error?" why is sinusitis found to be predictive of a stroke?
13. What is representation bias?
14. How are machines and people different, in terms of their use for making decisions?
15. Is disinformation the same as "fake news"?
16. Why is disinformation through autogenerated text a particularly significant issue?
17. What are the five ethical lenses described by the Markkula Center?
18. Where is policy an appropriate tool for addressing data ethics issues?

Further Research

1. Read the article "What Happens When an Algorithm Cuts Your Healthcare". How could problems like this be avoided in the future?
2. Research to find out more about YouTube's recommendation system and its societal impacts. Do you think recommendation systems must always have feedback loops with negative results? What approaches could Google take to avoid them? What about the government?
3. Read the paper "Discrimination in Online Ad Delivery". Do you think Google should be considered responsible for what happened to Dr. Sweeney? What would be an appropriate response?
4. How can a cross-disciplinary team help avoid negative consequences?
5. Read the paper "Does Machine Learning Automate Moral Hazard and Error?" What actions do you think should be taken to deal with the issues identified in this paper?