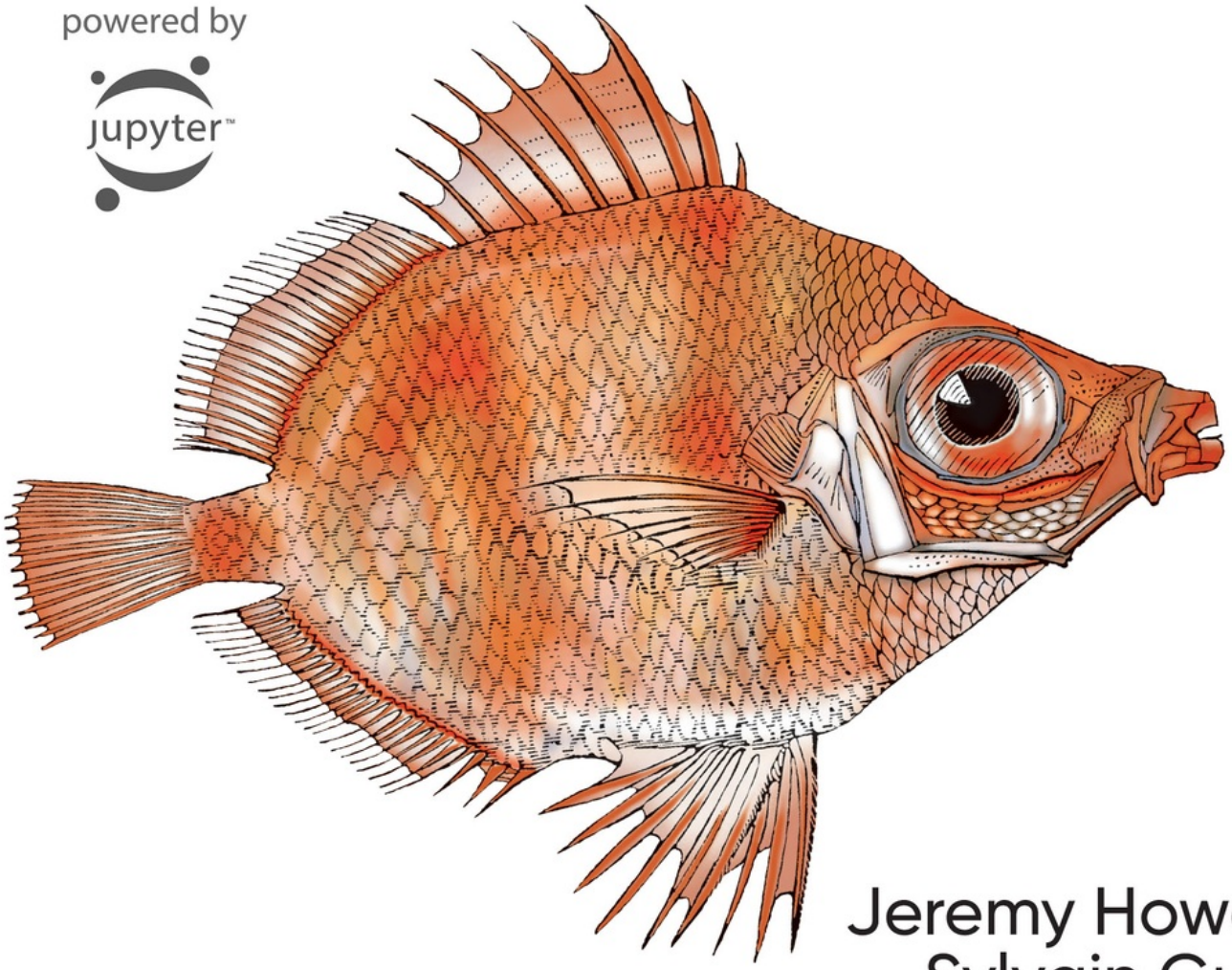
O'REILLY®

Deep Learning for Coders with fastai & PyTorch

AI Applications Without a PhD

powered by

jupyter™



Jeremy Howard &
Sylvain Gugger

Foreword by Soumith Chintala

19. 14. ResNets
    a. Going Back to Imagenette
    b. Building a Modern CNN: ResNet
        i. Skip Connections
        ii. A State-of-the-Art ResNet
        iii. Bottleneck Layers
    c. Conclusion
    d. Questionnaire
        i. Further Research
20. 15. Application Architectures Deep Dive
    a. Computer Vision
        i. cnn_learner
        ii. unet_learner
        iii. A Siamese Network
    b. Natural Language Processing
    c. Tabular
    d. Conclusion
    e. Questionnaire
        i. Further Research
21. 16. The Training Process
    a. Establishing a Baseline
    b. A Generic Optimizer

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 is an incredible resource. The book wastes no time and teaches how to use deep learning effectively in the first few chapters. It then covers the inner workings of ML models and frameworks in a thorough but accessible fashion, which will allow you to understand and build upon them. I wish there was a book like this when I started learning ML, it is an instant classic!

—Emmanuel Ameisen, Author, *Building Machine Learning Powered Applications*

“Deep Learning is for everyone,” as we see in Chapter 1, Section 1 of this book, and while other books may make similar claims, this book delivers on the claim. The authors have extensive knowledge of the field but are able to describe it in a way that is perfectly suited for a reader with experience in programming but not in machine learning. The book shows examples first, and only covers theory in the context of concrete examples. For most people, this is the best way to learn.The book does an impressive job of covering the key applications of deep learning in computer vision, natural language processing, and tabular data processing, but also covers key topics like data ethics that some other books miss. Altogether, this is one of the best sources for a programmer to become proficient in deep learning.

—Peter Norvig, Director of Research, Google

Gugger and Howard have created an ideal resource for anyone who has ever done even a little bit of coding. This book, and the fast.ai courses that go with it, simply and practically demystify deep learning using a hands-on approach, with pre-written code that you can explore and re-use. No more slogging through theorems and proofs about abstract concepts. In Chapter 1 you will build your first deep learning model, and by the end of the book you will know how to read and understand the Methods section of any deep learning paper.

—Curtis Langlotz, Director, Center for Artificial Intelligence in Medicine and Imaging, Stanford University

This book demystifies the blackest of black boxes: deep learning. It enables quick code experimentations with a complete python notebook. It also dives into the ethical implication of artificial intelligence, and shows how to avoid it from becoming dystopian.

—Guillaume Chaslot, Fellow, Mozilla

As a pianist turned OpenAI researcher, I'm often asked for advice on getting into Deep Learning, and I always point to fastai. This book manages the seemingly impossible—it's a friendly guide to a complicated subject, and yet it's full of cutting-edge gems that even advanced practitioners will love.

—Christine Payne, Researcher, OpenAI

An extremely hands-on, accessible book to help anyone quickly get started on their deep learning project. It's a very clear, easy to follow and honest guide to practical deep learning. Helpful for beginners to executives/managers alike. The guide I wished I had years ago!

—Carol Reiley, Founding President and Chair, Drive.ai

Jeremy and Sylvain's expertise in deep learning, their practical approach to ML, and their many valuable open-source contributions have made then key figures in the PyTorch community. This book, which continues the work that they and the fast.ai community are doing to make ML more accessible, will greatly benefit the entire field of AI.

—Jerome Pesenti, Vice President of AI, Facebook

Deep Learning is one of the most important technologies now, responsible for many amazing recent advances in AI. It used to be only for PhDs, but no longer! This book, based on a very popular fast.ai course, makes DL accessible to anyone with programming experience. This book teaches the "whole game", with excellent hands-on examples and a companion interactive site. And PhDs will also learn a lot.

—Gregory Piatetsky-Shapiro, President, KDnuggets

An extension of the fast.ai course that I have consistently recommended for years, this book by Jeremy and Sylvain, two of the best deep learning experts today, will take you from beginner to qualified practitioner in a matter of months. Finally, something positive has come out of 2020!

—Louis Monier, Founder, Altavista; former Head of Airbnb AI Lab

We recommend this book! Deep Learning for Coders with fastai and PyTorch uses advanced frameworks to move quickly through concrete, real-world artificial intelligence or automation tasks. This leaves time to cover usually neglected topics, like safely taking models to production and a much-needed chapter on data ethics.

—John Mount and Nina Zumel, Authors, *Practical Data Science with R*

This book is "for Coders" and does not require a PhD. Now, I do have a PhD and I am no coder, so why have I been asked to review this book? Well, to tell you how friggin awesome it really is!

Within a couple of pages from Chapter 1 you'll figure out how to get a state-of-the-art network able to classify cat vs. dogs in 4 lines of code and less than 1 minute of computation. Then you land Chapter 2, which takes you from model to production, showing how you can serve a webapp in no time, without any HTML or JavaScript, without owning a server.

I think of this book as an onion. A complete package that works using the best possible settings. Then, if some alterations are required, you can peel the outer layer. More tweaks? You can keep discarding shells. Even more? You can go as deep as using bare PyTorch. You'll have three independent voices accompanying you around your journey along this 600 page book, providing you guidance and individual perspective.

—Alfredo Canziani, Professor of Computer Science, NYU

Deep Learning for Coders with fastai and PyTorch is an approachable conversationally-driven book that uses the whole game approach to teaching deep learning concepts. The book focuses on getting your hands dirty right out of the gate with real examples and bringing the reader along with reference concepts only as needed. A practitioner may approach the world of deep learning in this book through hands-on examples in the first half, but will find themselves naturally introduced to deeper concepts as they traverse the back half of the book with no pernicious myths left unturned.

—Josh Patterson, Patterson Consulting

Deep Learning for Coders with fastai and PyTorch

AI Applications Without a PhD

Jeremy Howard and Sylvain Gugger

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

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:

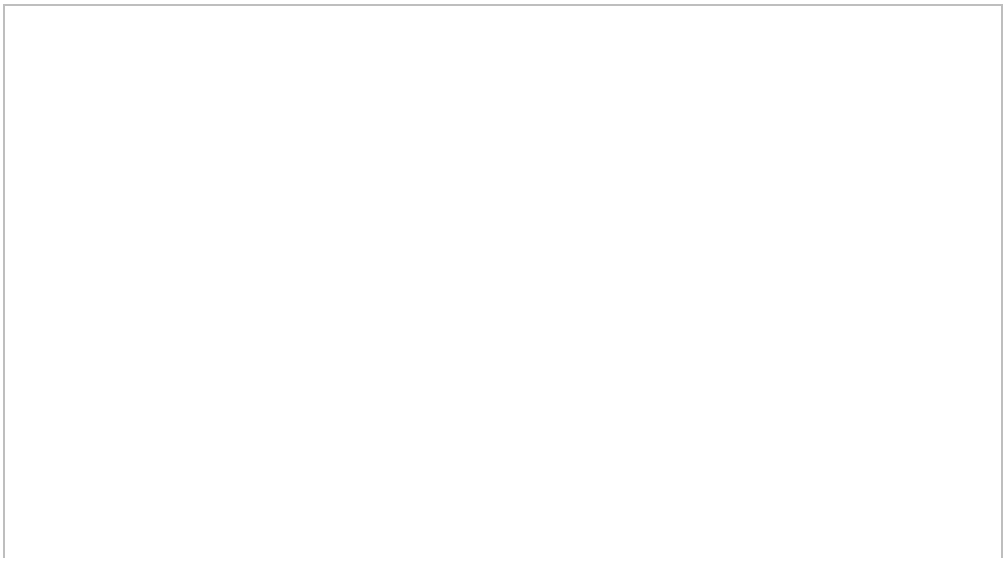
- How to train models that achieve state-of-the-art results in

    - 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.

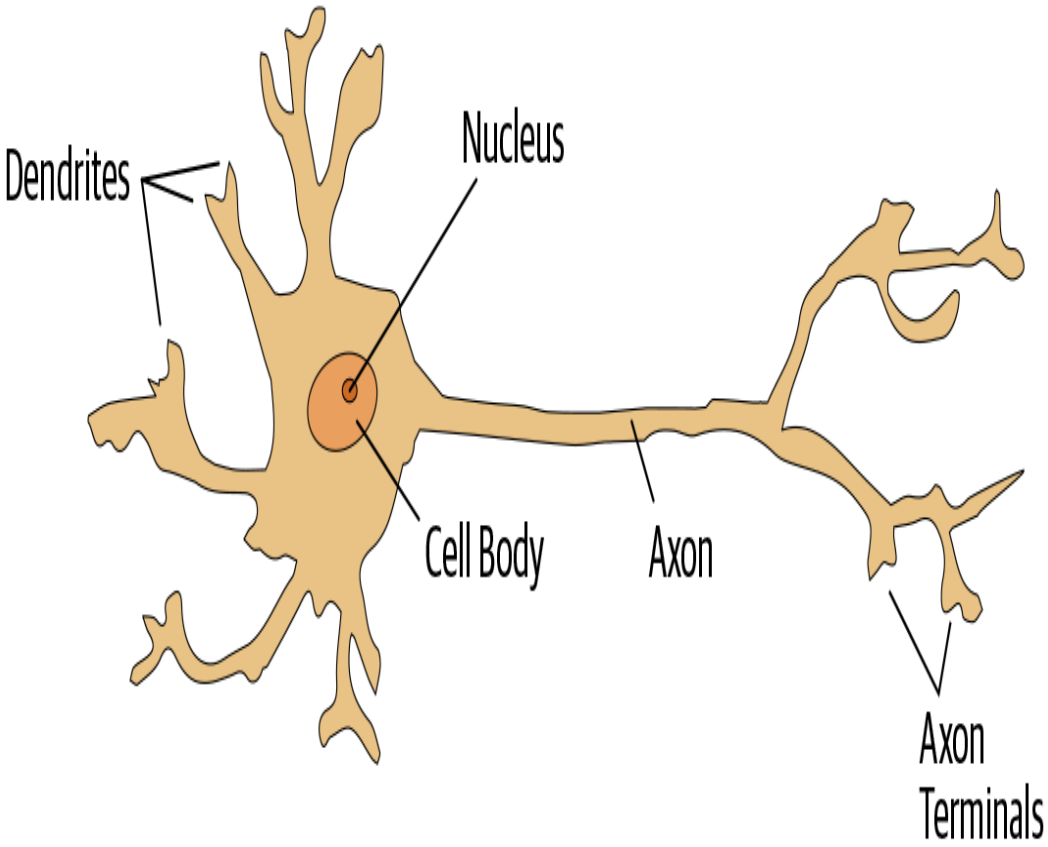
But neural networks are not, in fact, completely new. In order to have a wider perspective on the field, it is worth starting with a bit of history.

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.


Dendrites
Nucleus
Cell Body
Axon
Axon
Terminals

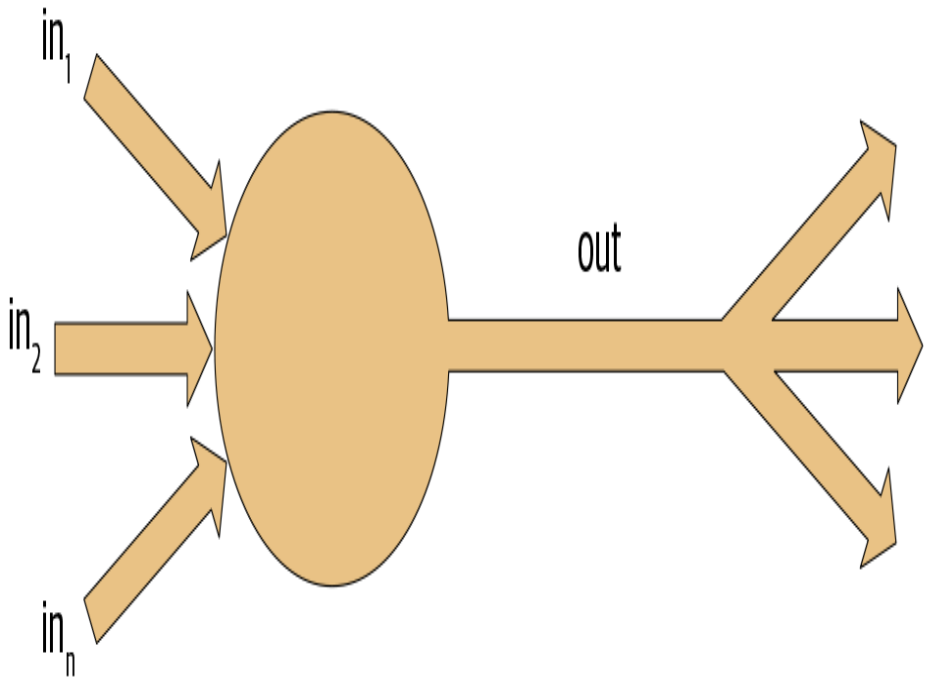

in1
in2
inn
out

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.

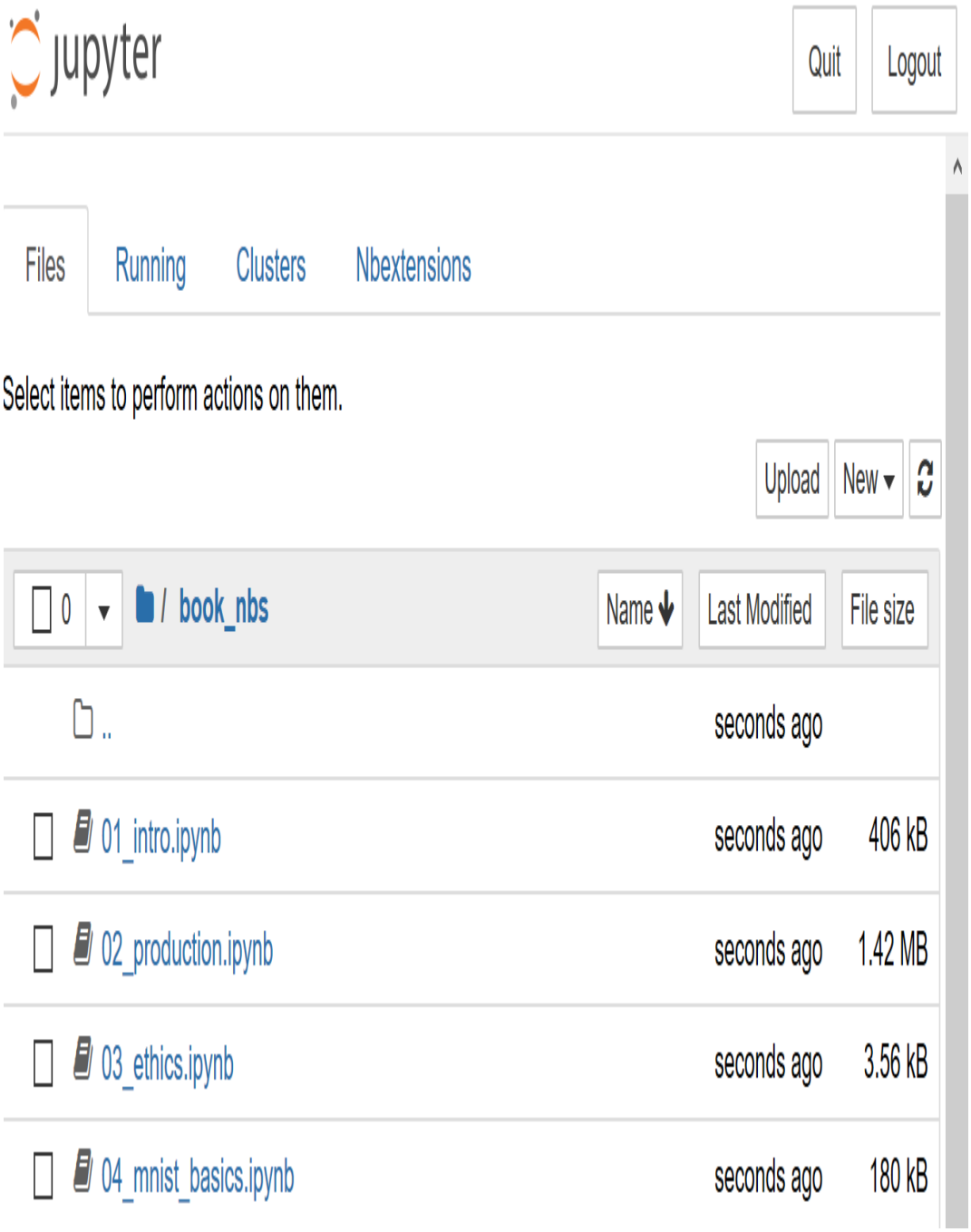


Figure 1-2. Initial view of Jupyter Notebook

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 folders containing different versions of the notebooks. The *full* folder contains the exact notebooks used to create the book you're reading now, with all the prose and outputs. The *stripped* version 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 stripped 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).

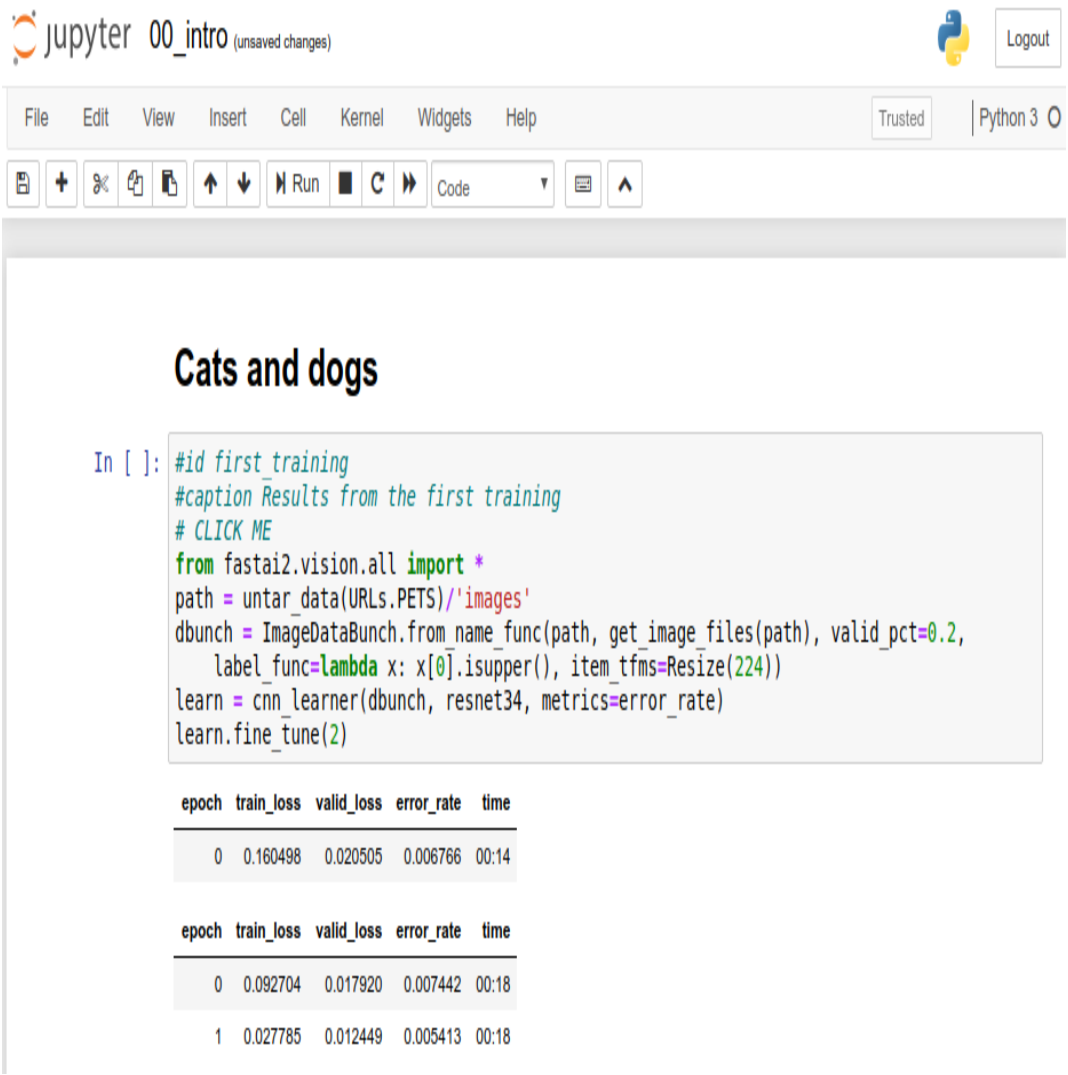


Figure 1-3. A Jupyter notebook

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:

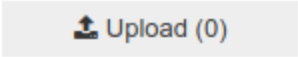
```
img = PILImage.create('images/chapter1_cat_example.jpg')
img.to_thumb(192)
```

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).

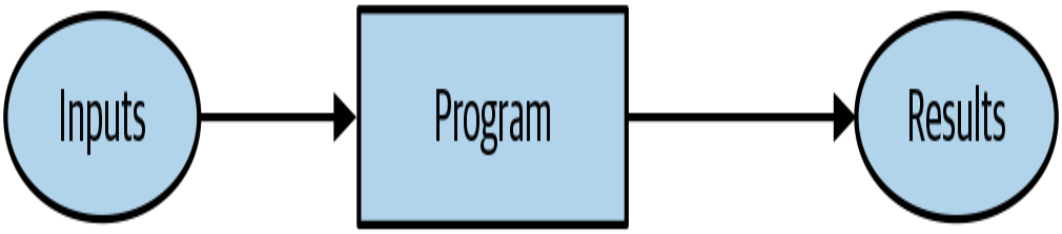


Figure 1-4. A traditional program

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.

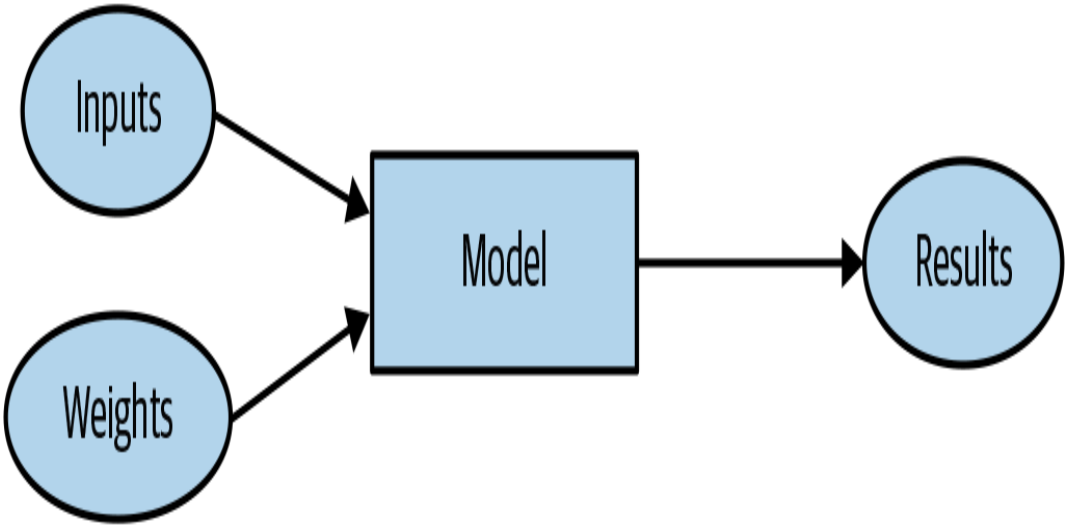


Figure 1-5. A program using weight assignment

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.

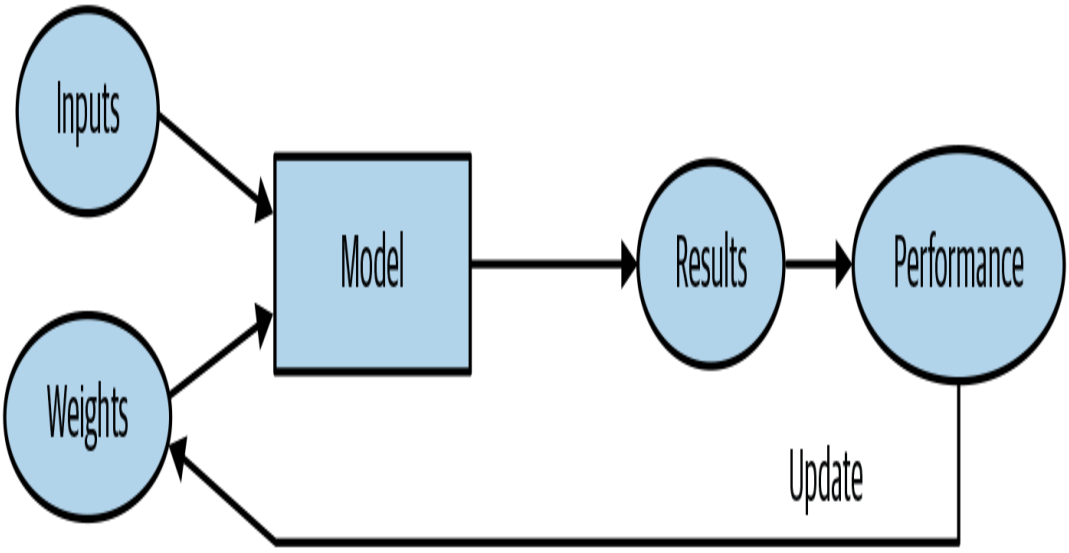


Figure 1-6. Training a machine learning model

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.

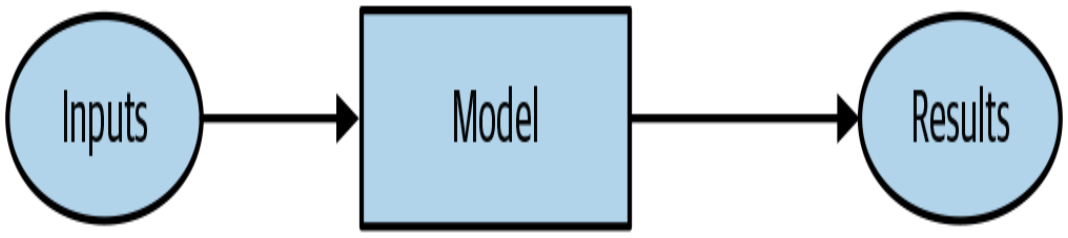


Figure 1-7. Using a trained model as a program

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 weight 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.

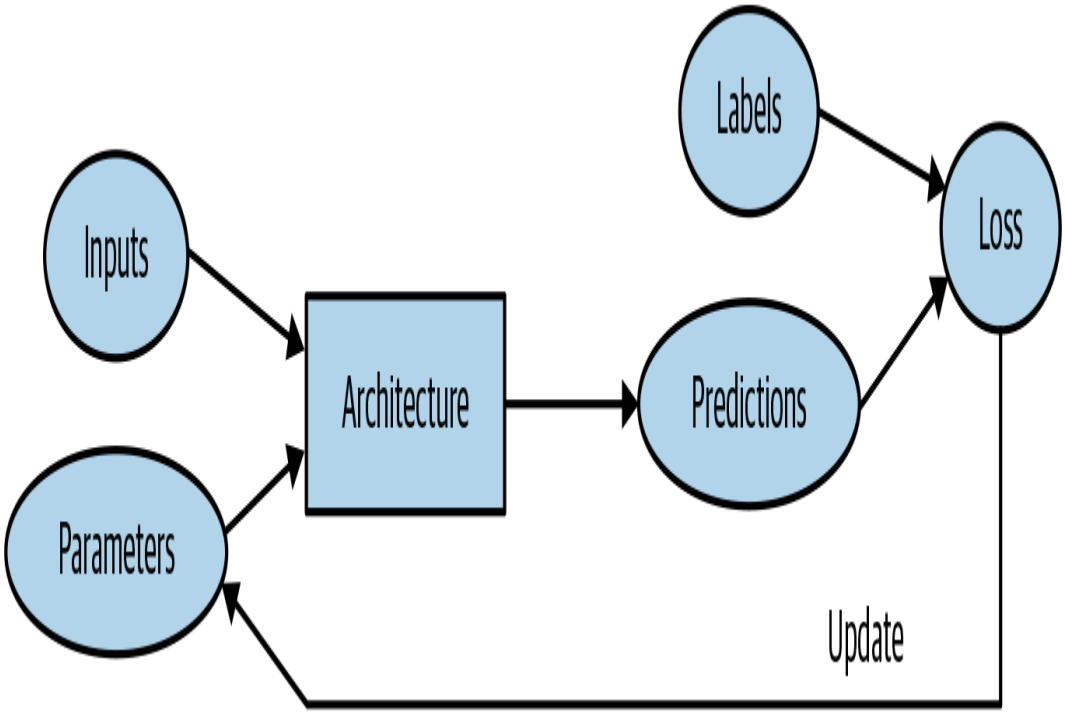


Figure 1-8. Detailed training loop

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 `Transform`s that we need. A `Transform` contains code that is applied automatically during training; fastai includes many predefined `Transform`s, 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 filenames can be extracted using a function applied to the filename) and passing `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.

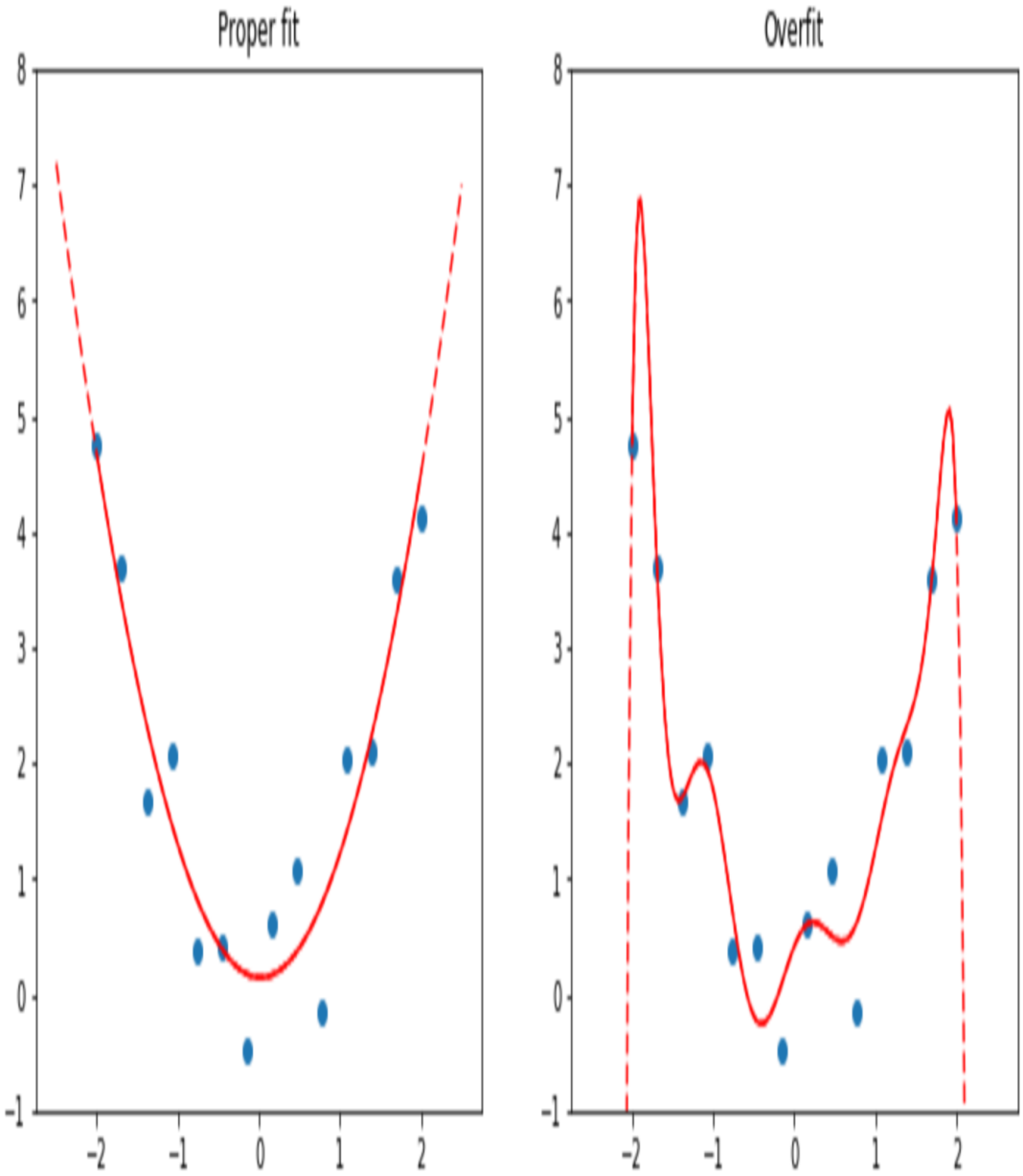


Figure 1-9. Example of overfitting

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.

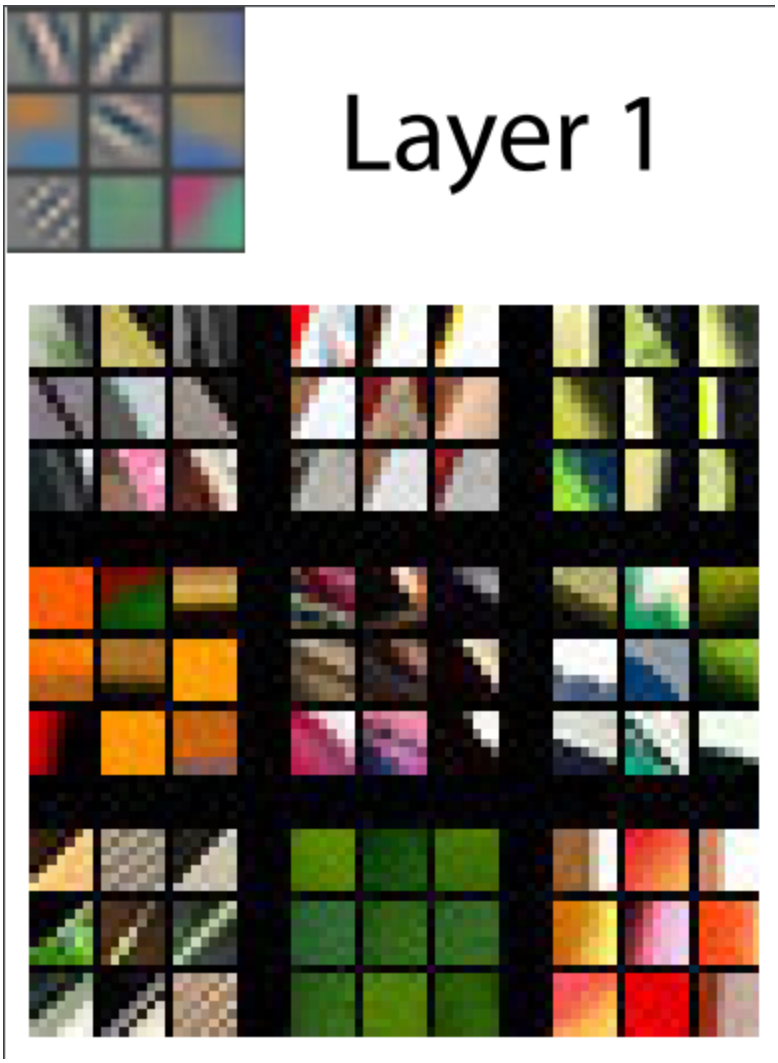


Figure 1-10. Activations of the first layer of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

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.

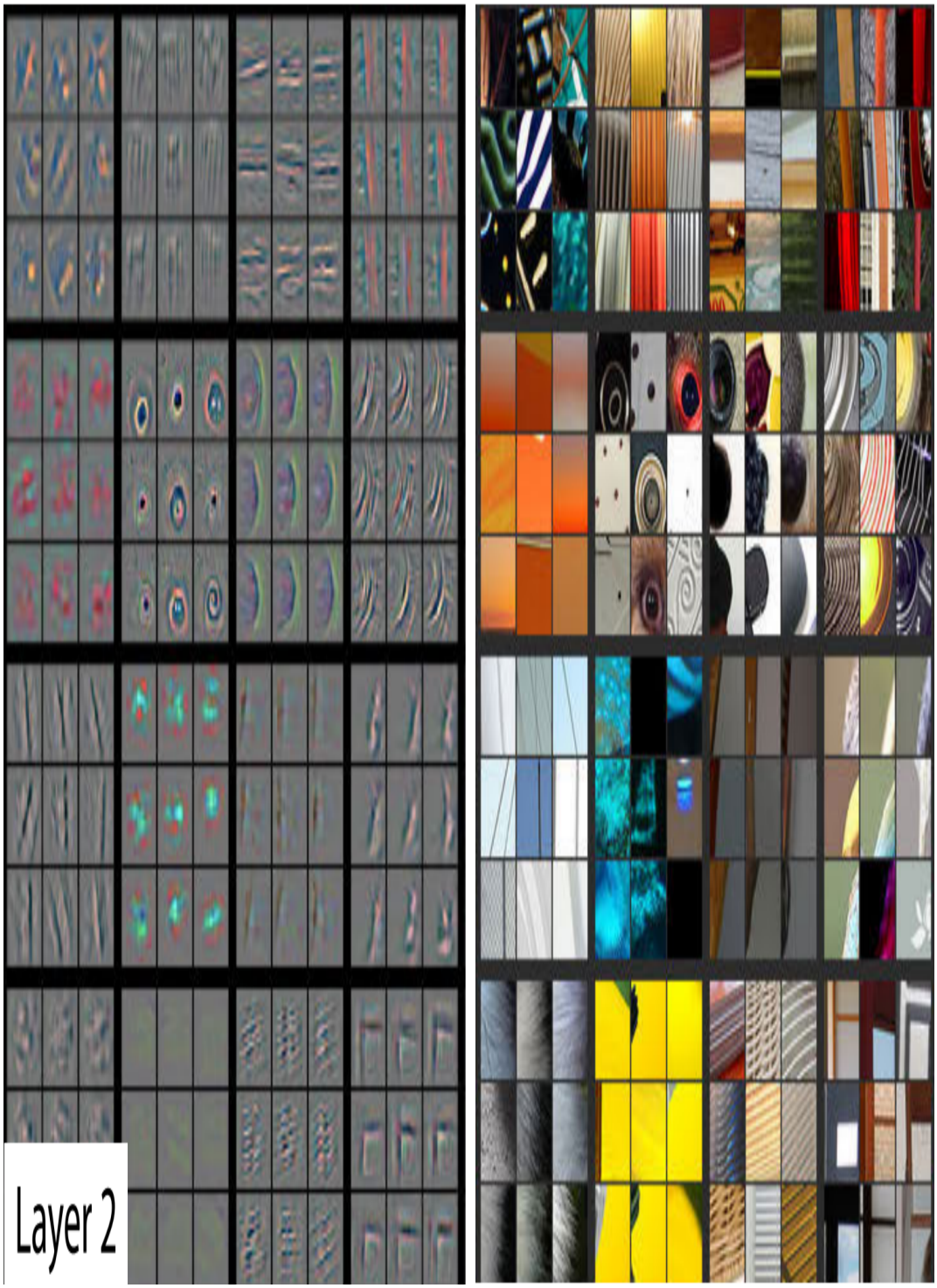


Figure 1-11. Activations of the second layer of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

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.

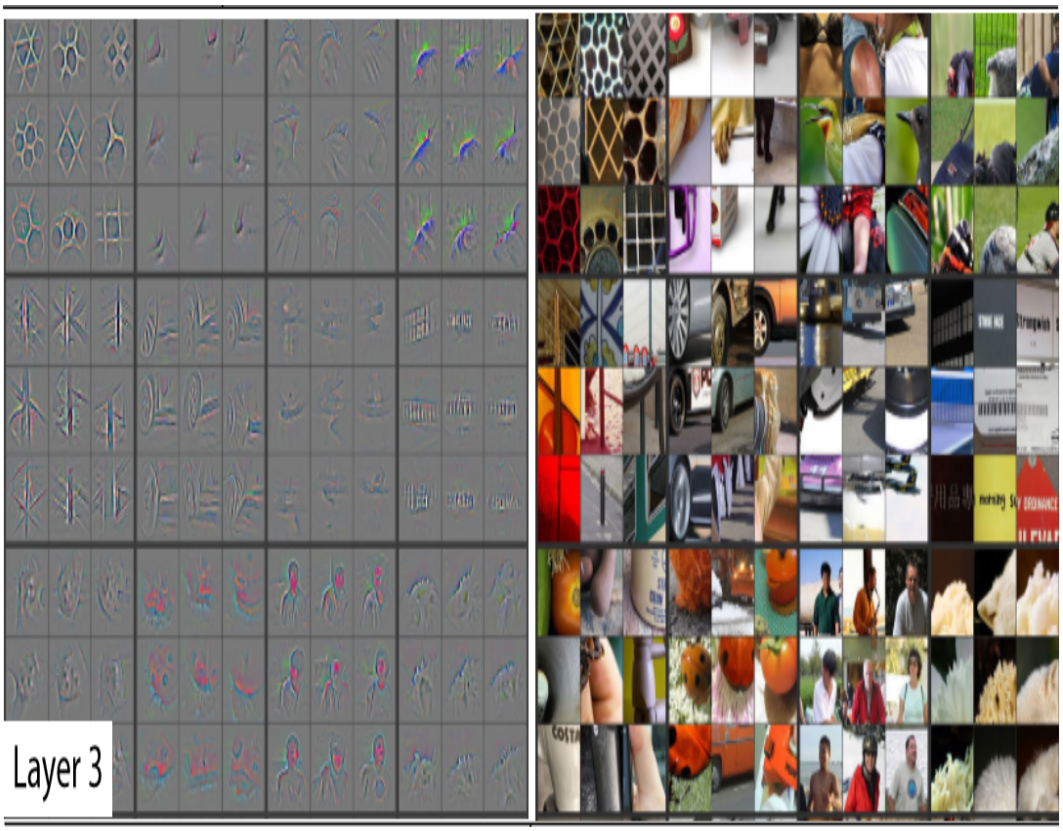


Figure 1-12. Activations of the third layer of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)

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.

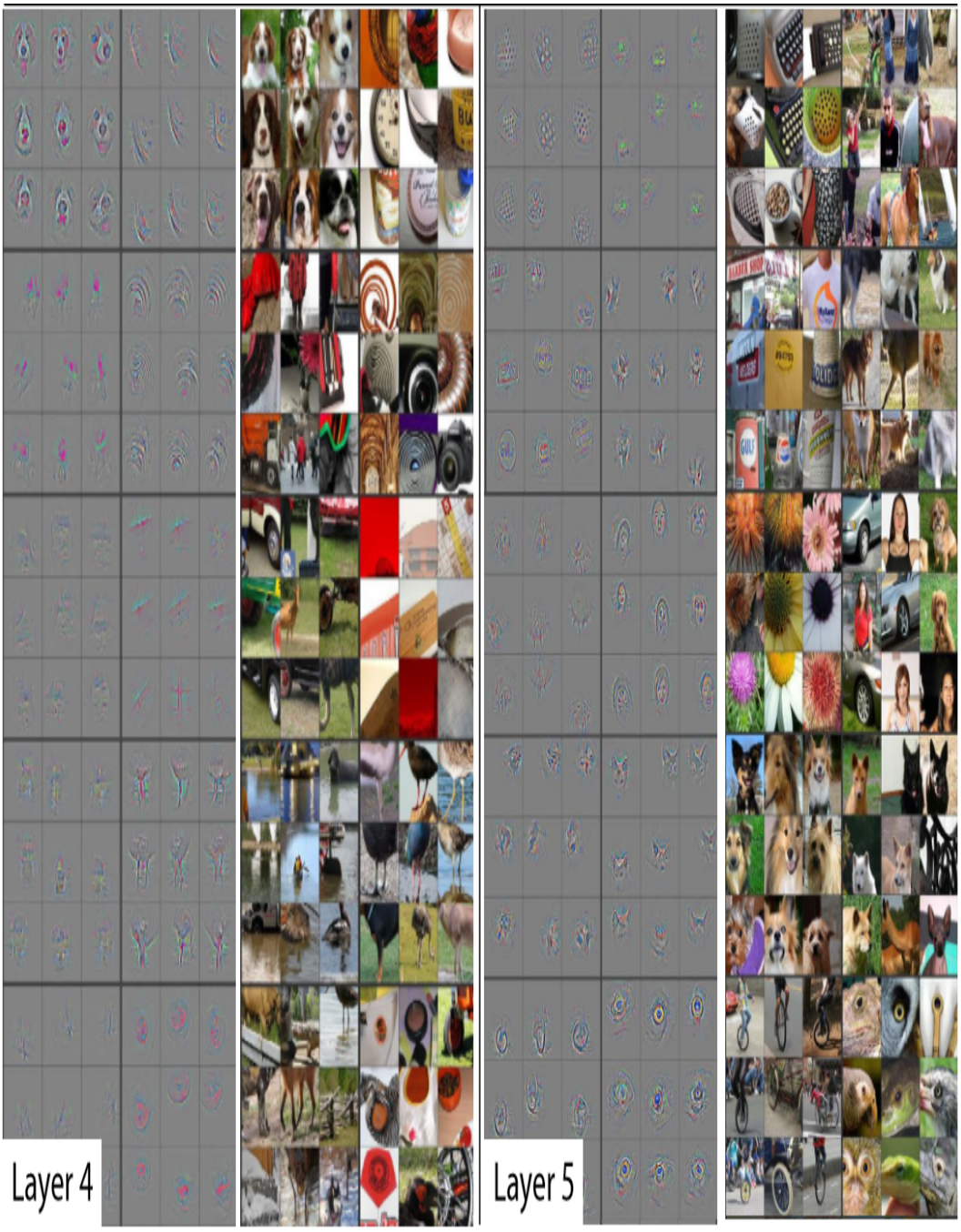


Figure 1-13. Activations of the fourth and fifth layers of a CNN (courtesy of Matthew D. Zeiler and Rob Fergus)