

HOW AI WORKS

FROM SORCERY TO SCIENCE

RONALD T. KNEUSEL



PRAISE FOR *HOW AI WORKS*

“A must-read for anyone wishing to dig into AI without getting lost in the weeds. Kneusel has succeeded in explaining how AI works to a layperson like myself.”

—KENNETH GASS, HONORARY CURATOR OF GEOLOGY, MILWAUKEE PUBLIC MUSEUM

“*How AI Works* is a friendly and personal peek behind the curtain of modern AI. Ronald T. Kneusel tells the story of how the field grew, and surveys the ideas that are powering the AI revolution. From this book, you’ll learn not only how AI works today, but its limits, its capabilities, and where it might take us tomorrow.”

—ANDREW GLASSNER, AUTHOR OF *DEEP LEARNING: A VISUAL APPROACH*

“*How AI Works* is a tour de force of the rich history of artificial intelligence, from the early perceptions and symbolic systems to large language models such as ChatGPT. For beginners, it demystifies AI and is a perfect resource to get up to date with more than six decades of research and development. For those versed in AI, it serves as an invaluable tool to fill knowledge

gaps. Even AI experts will gain a fresh perspective, enhancing their understanding and ability to articulate complex concepts.”

—BEN DICKSON, SOFTWARE ENGINEER, EDITOR OF TECHTALKS

“After reading this book I have a better understanding of the ML tools I have already used in my work, and a new appreciation and insight to how Large Language Models, and future AI, will likely change the domains in which I work. I recommend this book to anyone who works with software systems, including management, and anyone who just wants to know what AI actually does under the hood.”

—DANIEL KOSEY, CISSP, CYBERSECURITY ENGINEER

HOW AI WORKS

From Sorcery to Science

by Ronald T. Kneusel



San Francisco

HOW AI WORKS. Copyright © 2024 by Ronald T. Kneusel.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

27 26 25 24 23 1 2 3 4 5

ISBN-13: 978-1-7185-0372-4 (print)

ISBN-13: 978-1-7185-0373-1 (ebook)

Publisher: William Pollock

Managing Editor: Jill Franklin

Production Manager: Sabrina Plomitallo-González

Production Editor: Miles Bond

Developmental Editor: Eva Morrow

Cover Illustrator: Gina Redman

Interior Design: Octopod Studios

Technical Reviewer: Alex Kachurin

Copyeditor: Rachel Head

Proofreader: Carl Quesnel

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press® directly at info@nostarch.com or:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900
www.nostarch.com

Library of Congress Control Number: 2023038565

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with

respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

To Frank Rosenblatt—he saw it coming.

About the Author

Ronald T. Kneusel has been working with machine learning in industry since 2003 and completed a PhD in machine learning at the University of Colorado, Boulder, in 2016. Ron has written five other books: *Practical Deep Learning: A Python-Based Introduction* (No Starch Press, 2021), *Math for Deep Learning: What You Need to Know to Understand Neural Networks* (No Starch Press, 2021), *Strange Code: Esoteric Languages That Make Programming Fun Again* (No Starch Press, 2022), *Numbers and Computers* (Springer, 2017), and *Random Numbers and Computers* (Springer, 2018).

About the Technical Reviewer

Alex Kachurin is a data science and machine learning professional with more than 15 years of experience in the field. He earned an MS in computer vision from the University of Central Florida in 2010.

CONTENTS

[Acknowledgments](#)

[Preface](#)

[Chapter 1: And Away We Go: An AI Overview](#)

[Chapter 2: Why Now? A History of AI](#)

[Chapter 3: Classical Models: Old-School Machine Learning](#)

[Chapter 4: Neural Networks: Brain-Like AI](#)

[Chapter 5: Convolutional Neural Networks: AI Learns to See](#)

[Chapter 6: Generative AI: AI Gets Creative](#)

[Chapter 7: Large Language Models: True AI at Last?](#)

[Chapter 8: Musings: The Implications of AI](#)

[Glossary](#)

[Resources](#)

[Index](#)

ACKNOWLEDGMENTS

Thanks, first and foremost, to Eva Morrow for her gentle (and kind) editing. Thanks also to Alex Kachurin, MS, for his insights, thoughtful comments, and suggestions. Finally, I want to thank all the good folks at No Starch Press for believing in the book and helping to make it a reality.

PREFACE



Many books teach you how to do artificial intelligence (AI). Similarly, many popular books tell you about AI. However, what seems to be missing is a book that teaches you how AI works at a conceptual level. AI isn't magic; you can understand what it's doing without burying yourself in complex mathematics.

This book fills that void with a math-free explanation of how AI works. While some books are down in the weeds and others offer a bird's-eye view, this book is at treetop level. It aims to provide you with enough detail to understand the approach without getting bogged down in nitty-gritty mathematics. If that piques your interest, I invite you to read on.

You'll run across places where **** appears throughout the book. These markers highlight a shift in the topic or a transition point. In a textbook, **** would indicate a new section, but this isn't a textbook, nor do I want it to feel like one; so, instead of

sections and subsections, I'll use asterisks to warn you that a change is coming. Like this . . .

I first learned about artificial intelligence in 1987, in an undergraduate course of the same name. What people typically mean by *AI* has changed somewhat over the intervening decades. Still, the goal remains the same: to mimic intelligent behavior in a machine.

Few people in the 1980s had any reason to learn about AI, if they were even aware of it. AI had minimal impact on their daily lives, beyond the occasional renegade computer in science fiction TV shows and movies like *Star Trek* or *WarGames*, to say nothing of the relentless and terrifying *Terminator*.

However, the 1980s are long gone, current retro fashion trends notwithstanding, and AI is everywhere. It affects our lives in numerous ways every day, from phones telling us to drive here and not there, to labeling friends and family in pictures, to the articles and ads fed to us continuously online, like it or not. And this is to say nothing of the recent AI explosion involving large language models, which many interpret as “true AI” at last.

AI is also there behind the scenes in ways we seldom realize: airline flight planning, shipping and logistics, factory automation, satellite imaging of the earth, and helping your doctor decide if that lump is cancer, to name a few.

Why learn about AI now?

This book answers that question by explaining what happened, when it happened, why it happened, and, most importantly, how it happened—all without hype or a single mathematical equation. Frankly, the reality behind the AI revolution is impressive enough; the hype is unnecessary.

At this point, I feel some words about me are in order. After all, I’m asking you to join me on a journey through the world of AI, so it’s reasonable to wonder about your guide. I certainly would.

As mentioned earlier, I was introduced to AI in the late 1980s. I began working in AI, in the subfield known as *machine learning*, in 2003, applying machine learning models to intravascular ultrasound images.

I first heard of deep learning in 2010. *Deep learning* is a subfield of machine learning. I’ll clarify the difference between deep

learning, machine learning, and artificial intelligence in [Chapter 1](#), but for now you can think of them as the same thing.

In 2012, AI burst onto the scene—or at least into the news—with the advent of what came to be called AlexNet and a curious experiment at Google involving computers that learned to identify cats in YouTube videos. I was in the room at the 2012 International Conference on Machine Learning in Edinburgh, Scotland, when Google presented its paper. It was standing room only for the conference's 800 or so attendees.

In 2016, I completed a PhD in computer science specializing in AI at the University of Colorado, Boulder, under the direction of Michael Mozer. I've worked in AI daily since then, primarily in the defense industry, with a short break in 2016 to help co-found a medical AI startup.

After AlexNet, things changed quickly, as seemingly monthly some new AI-related “miracle” appeared in the academic literature, if not on the evening news. The only way to keep up was to attend conferences multiple times per year; waiting for results to appear in an academic journal was pointless, as the field was progressing too rapidly for the typically slow pace of academic publishing.

I'm writing this preface in November 2022 at the NeurIPS conference. NeurIPS is arguably the premier AI conference (no hate emails, please!), and this is the first time it's been held in person since the COVID-19 pandemic. Attendance is high, though perhaps not as high as at the 2019 conference, for which a lottery was held to determine which 13,500 people could attend. The fact that conference attendance has blossomed from a few hundred to over 10,000 in a decade tells us how important AI research has become.

The names of the tech industry leaders who support these conferences, which are prime hunting grounds for graduate students, also reveal the significance of AI. You'll find expo booths for Google, DeepMind (also Google), Meta (read: Facebook), Amazon, Apple, and others. AI drives much of what these companies do. AI is big bucks. AI runs on data, and these companies gobble up all the data we freely give them in exchange for their services.

By the end of the book, you'll understand what AI is doing under the hood (or bonnet, if you prefer). Ultimately, it isn't all that difficult to comprehend, though the devil is definitely in the details.

The book proceeds as follows:

Chapter 1, And Away We Go: An AI Overview We dive in with a quick overview of AI essentials and a basic example.

Chapter 2, Why Now? A History of AI AI didn't just fall from the sky. This chapter gives you AI's backstory and clarifies why the revolution is happening now.

Chapter 3, Classical Models: Old-School Machine Learning Modern AI is all neural networks, but to understand what neural networks are doing, it helps to understand the models that came before.

Chapter 4, Neural Networks: Brain-Like AI If you want to know what a neural network is, how it's trained, and how it's used, then this chapter is for you.

Chapter 5, Convolutional Neural Networks: AI Learns to See Much of the power of modern AI comes from learning new ways to represent data. If that sentence has no meaning for you, this chapter will help.

Chapter 6, Generative AI: AI Gets Creative Traditional supervised machine learning models attach labels to inputs. Generative AI produces novel output, including text, images, and even video. This chapter explores two popular approaches: generative adversarial networks (GANs) and diffusion models. GANs provide the intuition we need to explore diffusion models and, in [Chapter 7](#), large language models (LLMs). Diffusion models are adept at producing detailed, photorealistic images and videos from text prompts.

Chapter 7, Large Language Models: True AI at Last? OpenAI's fall 2022 release of its large language model, ChatGPT, might very well have ushered in the era of true AI. This chapter explores LLMs: what they are, how they work, and the claim that they are something new and disruptive.

Chapter 8, Musings: The Implications of AI The advent of large language models has altered the AI landscape. This chapter muses on the implications.

At the end of the book, you'll find a collection of additional resources to explore, should the AI bug bite and you want to learn more. Personally, and admittedly with bias, I recommend my books *Practical Deep Learning: A Python-Based Introduction* (2021) and *Math for Deep Learning: What You Need to Know to Understand Neural Networks* (2021), both available from No Starch Press. They will give you what you need to go from reading about how AI works conceptually to “doing” AI.

Finally, as you read, you'll notice that specific phrases in the text are *emphasized*. Definitions for many of these emphasized words and phrases are found in the glossary at the end of the book. Like every field, AI has its jargon. Keeping all the terms in your head is burdensome, hence the glossary to help you remember them.

I'm a real person. I know because I can successfully identify and click images of trains and traffic lights. If you have comments or questions about the material in this book, I want to hear from you. Please email me at rkneusebooks@gmail.com.

Now, if you're ready, away we go.

1

AND AWAY WE GO: AN AI OVERVIEW



Artificial intelligence attempts to coax a machine, typically a computer, to behave in ways humans judge to be intelligent. The phrase was coined in the 1950s by prominent computer scientist John McCarthy (1927–2011).

This chapter aims to clarify what AI is and its relationship to *machine learning* and *deep learning*, two terms you may have heard in recent years. We'll dive in with an example of machine learning in action. Think of this chapter as an overview of AI as a whole. Later chapters will build on and review the concepts introduced here.

Computers are *programmed* to carry out a particular task by giving them a sequence of instructions, a *program*, which

embodies an *algorithm*, or the recipe that the program causes the computer to execute.

The word *algorithm* is cast about often these days, though it isn't new; it's a corruption of *Al-Khwarizmi*, referring to ninth-century Persian mathematician Muhammad ibn Musa al-Khwarizmi, whose primary gift to the world was the mathematics we call *algebra*.

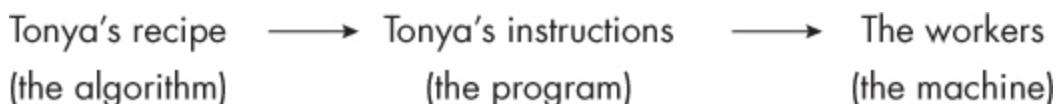
Let's begin with a story.

Tonya owns a successful hot sauce factory. The hot sauce recipe is Tonya's own, and she guards it carefully. It's literally her secret sauce, and only she understands the process of making it.

Tonya employs one worker for each step of the hot sauce-making process. These are human workers, but Tonya treats them as if they were machines because she's worried they'll steal her hot sauce recipe—and because Tonya is a bit of a monster. In truth, the workers don't mind much because she pays them well, and they laugh at her behind her back.

Tonya's recipe is an algorithm; it's the set of steps that must be followed to create the hot sauce. The collection of instructions

Tonya uses to tell her workers how to make the hot sauce is a program. The program embodies the algorithm in a way that the workers (the machine) can follow step by step. Tonya has programmed her workers to implement her algorithm to create hot sauce. The sequence looks something like this:



There are a few things to note about this scenario. First, Tonya is definitely a monster for treating human beings as machines. Second, at no point in the process of making hot sauce does any worker need to understand why they do what they do. Third, the programmer (Tonya) knows why the machine (the workers) does what it does, even if the machine doesn't.

* * * *

What I've just described is how we've controlled virtually all computers, going back to the first conceptual machines envisioned by Alan Turing in the 1930s and even earlier to the 19th-century Analytical Engine of Charles Babbage. A human conceives an algorithm, then translates that algorithm into a sequence of steps (a program). The machine executes the program, thereby implementing the algorithm. The machine doesn't understand what it's doing; it's simply performing a series of primitive instructions.

The genius of Babbage and Turing lay in the realization that there could be a general-purpose machine capable of executing arbitrary algorithms via programs. However, I would argue that it was Ada Lovelace, a friend of Babbage's often regarded as the world's first programmer, who initially understood the far-reaching possibilities of what we now call a computer. We'll talk more about Turing, Babbage, and Lovelace in [Chapter 2](#).

NOTE

In Lovelace's day, a “computer” was not a machine but a human being who calculated by hand. Hence, Babbage's Engine was a mechanical computer.

Let's take a moment to explore the relationship between the terms *AI*, [*machine learning*](#), and [*deep learning*](#). On the one hand, all three have become synonymous as referring to modern AI. This is wrong, but convenient. [Figure 1-1](#) shows the proper relationship between the terms.

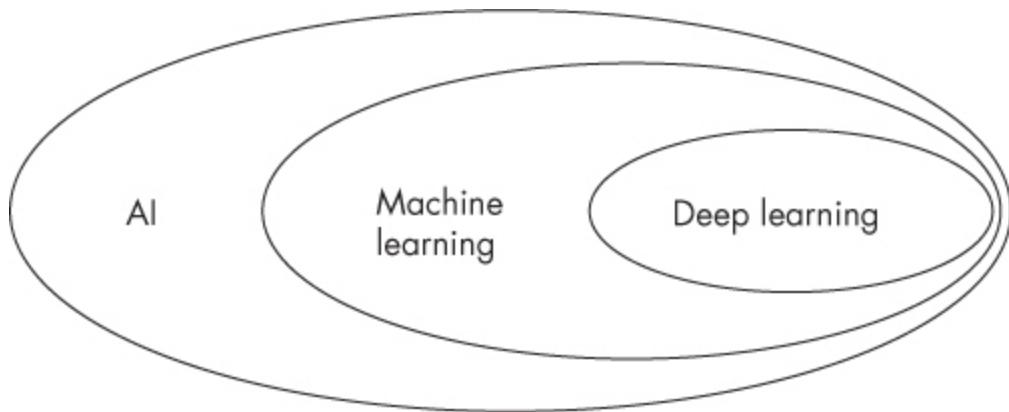


Figure 1-1: The relationship between artificial intelligence, machine learning, and deep learning

Deep learning is a subfield of machine learning, which is a subfield of artificial intelligence. This relationship implies that AI involves concepts that are neither machine learning nor deep learning. We'll call those concepts *old-school AI*, which includes the algorithms and approaches developed from the 1950s onward. Old-school AI is not what people currently mean when discussing AI. Going forward, we'll entirely (and unfairly) ignore this portion of the AI universe.

Machine learning builds models from data. For us, a ***model*** is an abstract notion of something that accepts inputs and generates outputs, where the inputs and outputs are related in some meaningful way. The primary goal of machine learning is to condition a model using *known* data so that the model produces meaningful output when given *unknown* data. That's about as

clear as muddy water, but bear with me; the mud will settle in time.

Deep learning uses large models of the kind previously too big to make useful. More muddy water, but I'm going to argue that there's no strict definition of deep learning other than that it involves neural networks with many layers. [Chapter 4](#) will clarify.

In this book, we'll be sloppy but in accord with popular usage, even by experts, and take "deep learning" to mean large neural networks (yet to be formally defined), "machine learning" to mean models conditioned by data, and "AI" to be a catchall for both machine learning and deep learning—remembering that there is more to AI than what we discuss here.

Data is everything in AI. I can't emphasize this enough. Models are blank slates that data must condition to make them suitable for a task. If the data is bad, the model is bad. Throughout the book, we'll return to this notion of "good" and "bad" data.

For now, let's focus on what a model is, how it's made useful by conditioning, and how it's used after conditioning. All this talk of conditioning and using sounds dark and sinister, if not

altogether evil, but, I assure you, it's not, even though we have ways of making the model talk.

A machine learning model is a black box that accepts an input, usually a collection of numbers, and produces an output, typically a label like “dog” or “cat,” or a continuous value like the probability of being a “dog” or the value of a house with the characteristics given to the model (size, number of bathrooms, ZIP code, and so on).

The model has *parameters*, which control the model’s output. Conditioning a model, known as *training*, seeks to set the model’s parameters in such a way that they produce the correct output for a given input.

Training implies that we have a collection of inputs, and the outputs the model should produce when given those inputs. At first blush, this seems a bit silly; why do we want the model to give us an output we already have? The answer is that we will, at some future point, have inputs for which we don’t already have the output. This is the entire point of making the model: to use it with unknown inputs and to believe the model when it gives us an output.

Training uses the collection of known inputs and outputs to adjust the model's parameters to minimize mistakes. If we can do that, we begin to believe the model's outputs when given new, unknown inputs.

Training a model is fundamentally different from programming. In programming, we implement the algorithm we want by instructing the computer step by step. In training, we use data to teach the model to adjust its parameters to produce correct output. There is no programming because, most of the time, we have no idea what the algorithm should be. We only know or believe a relationship exists between the inputs and the desired outputs. We hope a model can approximate that relationship well enough to be useful.

It's worth remembering the sage words of British statistician George Box, who said that all models are wrong, but some are useful. At the time, he was referring to other kinds of mathematical models, but the wisdom applies to machine learning.

Now we understand why the field is called machine learning: we teach the machine (model) by giving it data. We don't program the machine; we instruct it.

Here, then, is the machine learning algorithm:

1. Gather a training dataset consisting of a collection of inputs to the model and the outputs we expect from the model for those inputs.
2. Select the type of model we want to train.
3. Train the model by presenting the training inputs and adjusting the model's parameters when it gets the outputs wrong.
4. Repeat step 3 until we are satisfied with the model's performance.
5. Use the now-trained model to produce outputs for new, unknown inputs.

Most of machine learning follows this algorithm. Since we're using known *labeled data* to train the model, this approach is called *supervised learning*: we supervise the model while it learns to produce correct output. In a sense, we punish the model until it gets it right. This is a dark enterprise, after all.

We're ready for an example, but let's first summarize the story so far. We want a system where, for an unknown input, we get a meaningful output. To make the system, we train a machine learning model using a collection of inputs and their known outputs. Training conditions the model by modifying its

parameters to minimize the mistakes it makes on the training data. When we're satisfied with the model's performance, we use the model with unknown inputs because we now believe the model when it gives us an output (at least, most of the time).

Our first example comes from a famous dataset consisting of measurements of the parts of iris flowers. This dataset is from the 1930s, indicating how long people have contemplated what we now call machine learning.

The goal is a model that, for an input collection of measurements, outputs the specific species of iris flower. The full dataset has four measurements for three iris species. We'll keep it simple and use two measurements and two species: petal length and width in centimeters (cm) for *I. setosa* versus *I. versicolor*. Therefore, we want the model to accept two measurements as input and give us an output we can interpret as *I. setosa* or *I. versicolor*. *Binary models* like this decide between two possible outputs and are common in AI. If the model decides between more than two categories, it's a *multiclass* model.

We have 100 samples in our dataset: 100 pairs of petal measurements, and the corresponding iris flower types. We'll

call *I. setosa* class 0 and *I. versicolor* class 1, where *class* labels the input categories.

Models often want numeric class labels, which tells us that models don't know what their inputs and outputs mean; they only make associations between sets of inputs and outputs.

Models don't "think" using any commonly accepted definition of the word. (The models of [Chapter 7](#) might beg to differ, but more on that then.)

Here we must pause to introduce some critical terminology. I know, not what you want to read, but it's essential to all that follows. Artificial intelligence makes frequent use of vectors and matrices (singular "matrix"). A *vector* is a string of numbers treated as a single entity. For example, the four measurements of each iris flower mean we can represent the flower as a string of four numbers, say, (4.5, 2.3, 1.3, 0.3). The flower described by this vector has a sepal length of 4.5 cm, sepal width of 2.3 cm, petal length of 1.3 cm, and petal width of 0.3 cm. By grouping these measurements together, we can refer to them as a single entity.

The number of elements in a vector determines its dimensionality; for example, the iris dataset uses four-dimensional vectors, the four measurements of the flower. AI often works with inputs that have hundreds or even thousands of dimensions. If the input is an image, every pixel of that image is one dimension, meaning a small 28-pixel-square image becomes an input vector of 28×28 , or 784 dimensions. The concept is the same in 3 dimensions or 33,000 dimensions: it remains a string of numbers treated as a single entity. But an image has rows and columns, making it a two-dimensional array of numbers, not a string. Two-dimensional arrays of numbers are *matrices*. In machine learning, we often represent datasets as matrices, where the rows are vectors representing the elements of the dataset, like an iris flower, and the columns are the measurements. For example, the first five flowers in the iris dataset form the following matrix:

$$\begin{bmatrix} 4.5 & 2.3 & 1.3 & 0.3 \\ 5.6 & 2.9 & 3.6 & 1.3 \\ 5.7 & 4.4 & 1.5 & 0.4 \\ 6.7 & 3.1 & 4.4 & 1.4 \\ 4.6 & 3.1 & 1.5 & 0.2 \end{bmatrix}$$

Each row is a flower. Notice that the first row matches the vector example. The remaining rows list the measurements for other flowers.

While you're reading, keep these thoughts in the back of your mind:

- Vectors are strings of numbers often representing measurements in a dataset.
- Matrices are two-dimensional arrays of numbers often representing datasets (stacks of vectors).

As we continue our exploration of AI, the differences between vectors and matrices will come into focus. Now, let's return to our story.

The inputs to a model are its *features*. Our iris flower dataset has two features, the petal's length and width, which are grouped into *feature vectors* (or *samples*). A single feature vector serves as the model's input. A binary model's output is typically a number relating to the model's belief that the input belongs to class 1. For our example, we'll give the model a feature vector consisting of two features and expect an output that lets us decide whether we should call the input *I. versicolor*. If not, we declare the input to be *I. setosa* because we *assume* that inputs will always be one or the other.

Machine learning etiquette states that we should test our model; otherwise, how will we know it's working? You might think it's working when it gets all the training samples right, but experience has taught practitioners this isn't always the case. The proper way to test a model is to keep some of the labeled training data to use after training. The model's performance on this held-out test dataset better indicates how well the model has learned. We'll use 80 labeled samples for training and keep 20 of them for testing, making sure that both the training and test sets contain an approximately even mix of both classes (flower types). This is also essential in practice, as far as possible. If we never show the model examples of a particular class of input, how can it learn to distinguish that class from others?

Using a held-out test set to judge the performance of a model isn't just etiquette. It addresses a foundational issue in machine learning: generalization. Some machine learning models follow a process quite similar to a widely used approach known as *optimization*. Scientists and engineers use optimization to fit measured data to known functions; machine learning models also use optimization to condition their parameters, but the goal is different. Fitting data to a function, like a line, seeks to create the best possible *fit*, or the line that best explains the measured data. In machine learning, we instead want a model

that learns the general characteristics of the training data to *generalize* to new data. That's why we evaluate the model with the held-out test set. To the model, the test set contains new, unseen data it didn't use to modify its parameters. The model's performance on the test set is a clue to its generalization abilities.

Our example has two input features, meaning the feature vectors are two-dimensional. Since we have two dimensions, we can opt to make a plot of the training dataset. (If we have two or three features in a feature vector, we can plot the feature vectors. However, most feature vectors have hundreds to thousands of features. I don't know about you, but I can't visualize a thousand-dimensional space.)

[Figure 1-2](#) displays the two-dimensional iris training data; the x -axis is petal length, and the y -axis is petal width. The circles correspond to instances of *I. setosa* and the squares *I. versicolor*. Each circle or square represents a single training sample, the petal length and width for a specific flower. To place each point, find the petal length on the x -axis and the petal width on the y -axis. Then, move up from the x -axis and to the right from the y -axis. Where your fingers meet is the point representing that flower. If the flower is *I. setosa*, make the point a circle; otherwise, make it a square.

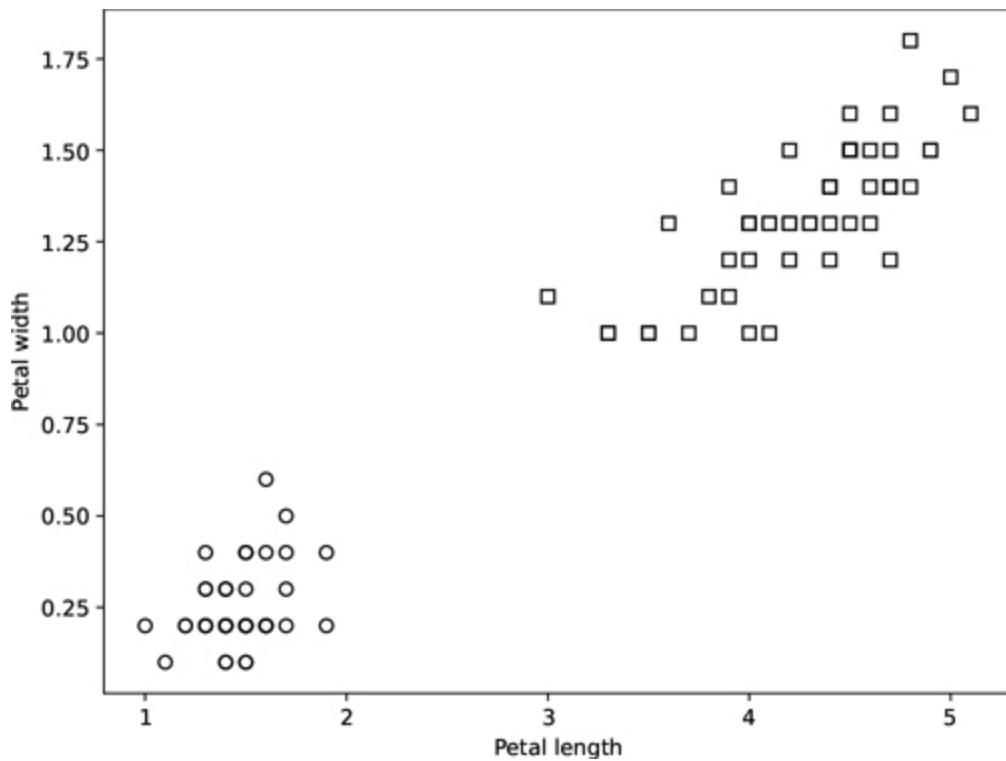


Figure 1-2: The iris training data

The plot in Figure 1-2 shows the *feature space* of the training set. In this case, we can visualize the training set directly, because we only have two features. When that's not possible, all is not lost. Advanced algorithms exist that allow us to make plots like Figure 1-2 where the points in two or three dimensions reflect the distribution of the samples in the much higher-dimensional space. Here, the word *space* means much the same as it does in everyday parlance.

Look carefully at Figure 1-2. Does anything jump out at you? Are the different classes mixed or well separated? Every circle inhabits the lower-left corner of the plot, while all of the

squares are in the upper right. There is no overlap between the classes, meaning they are entirely separate in the feature space.

How can we use this fact to make a *classifier*, a model that classifies iris flowers? (While *model* is the more general term, as not all models place their inputs into categories, when they do, use the term *classifier*.)

We have many model types to choose from for our classifier, including *decision trees*, which generate a series of yes/no questions related to the features used to decide the class label to output for a given input. When the questions are laid out visually, they form a structure reminiscent of an upside-down tree. Think of a decision tree as a computer-generated version of the game *20 Questions*.

Even though we have two features, petal length and petal width, we can classify new iris flowers by asking a single question: is the petal length less than 2.5 cm? If the answer is “yes,” then return class 0, *I. setosa*; otherwise, return class 1, *I. versicolor*. To classify the training data correctly, we need only the answer to this simple question.

Did you catch what I did just now? I said that the question correctly classifies all the *training* data. What about the 20 test

samples we didn't use? Is our single-question classifier sufficient to give each of them the correct label? In practice, that's what we want to know, and that is what we would report as the classifier's performance.

[Figure 1-3](#) shows the training data again, along with the test data we didn't use to make our single-question classifier. The solid circles and squares represent the test data.

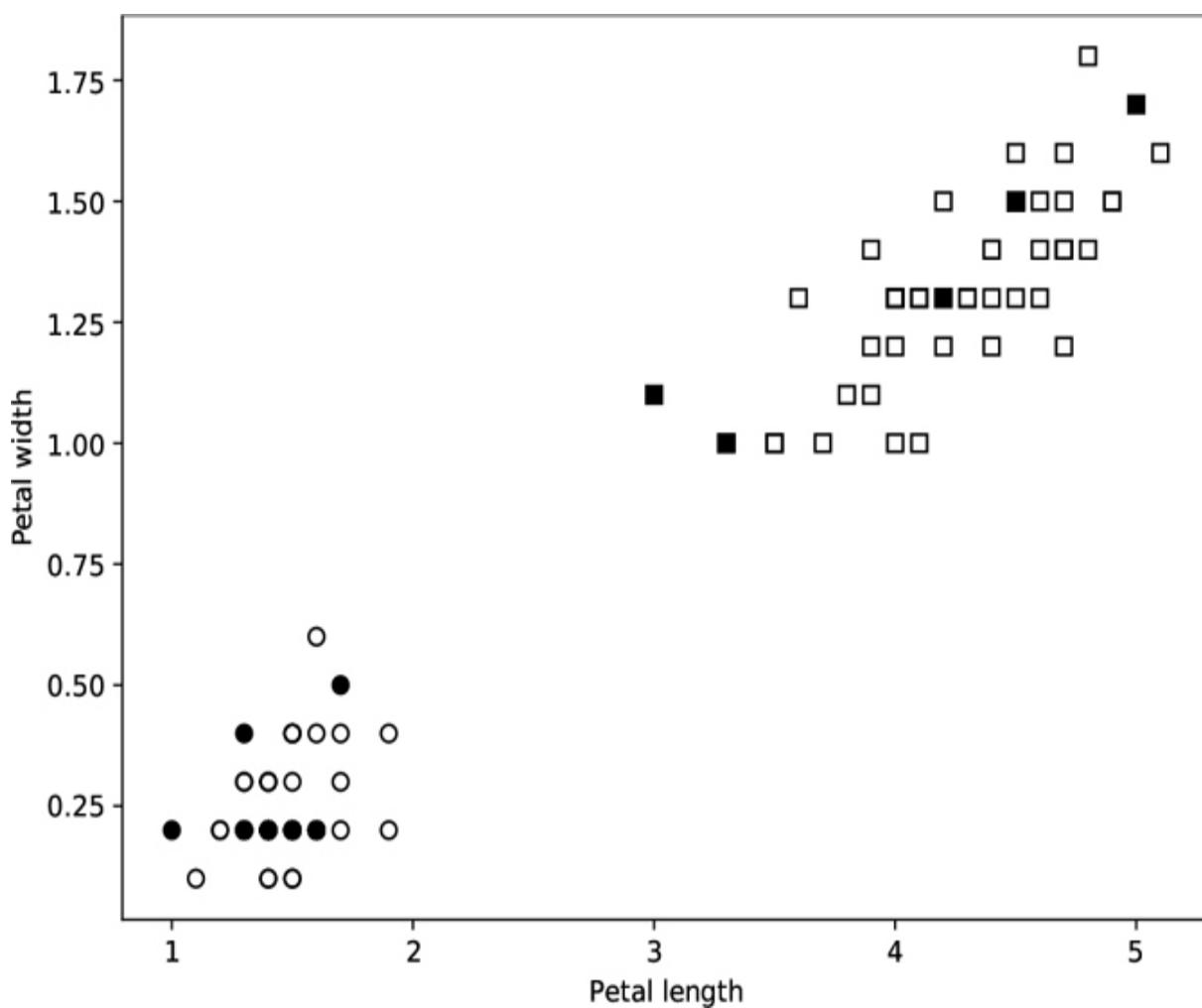


Figure 1-3: The iris training data with the held-out test data (solid)

None of the test data violates our rule; we still get correct class labels by asking if the petal length is less than 2.5 cm. Therefore, our model is perfect; it makes no mistakes. Congratulations, you just created your first machine learning model!

We should be happy, but not too happy. Let's repeat this exercise, replacing *I. setosa* with the remaining iris species, *I. virginica*. This leads to [Figure 1-4](#), where the triangles are instances of *I. virginica*.

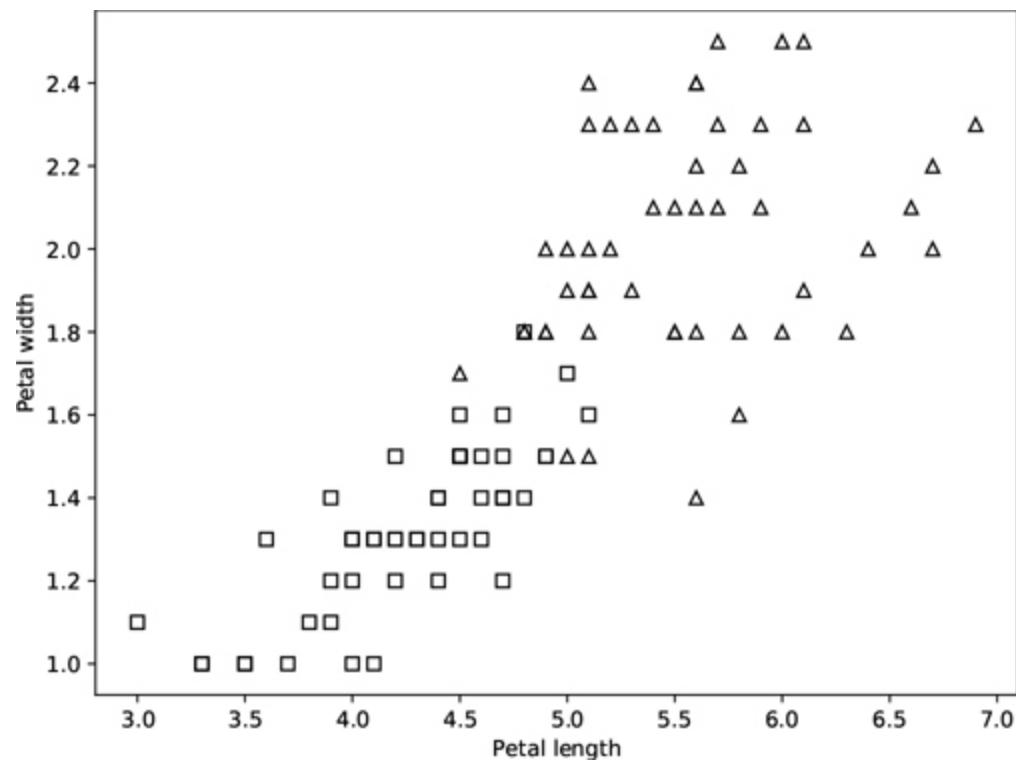


Figure 1-4: The new iris training data

Hmm, things are not as clear-cut now. The obvious gap between the classes is gone, and they overlap.

I trained a decision tree using this new iris dataset. As before, there were 80 samples for training and 20 held back for testing. This time, the model wasn't perfect. It correctly labeled 18 of the 20 samples, for an accuracy of 9 out of 10, or 90 percent. This roughly means that when this model assigns a flower to a particular class, there is a 90 percent chance it's correct. The previous sentence, to be rigorous, needs careful clarification, but for now, you get the idea—machine learning models are not always perfect; they (quite frequently) make mistakes.

[Figure 1-5](#) shows the learned decision tree. Begin at the top, which is the root, and answer the question in that box. If the answer is “yes,” move to the box on the left; otherwise, move to the right. Keep answering and moving in this way until you arrive at a leaf: a box with no arrows. The label in this box is assigned to the input.

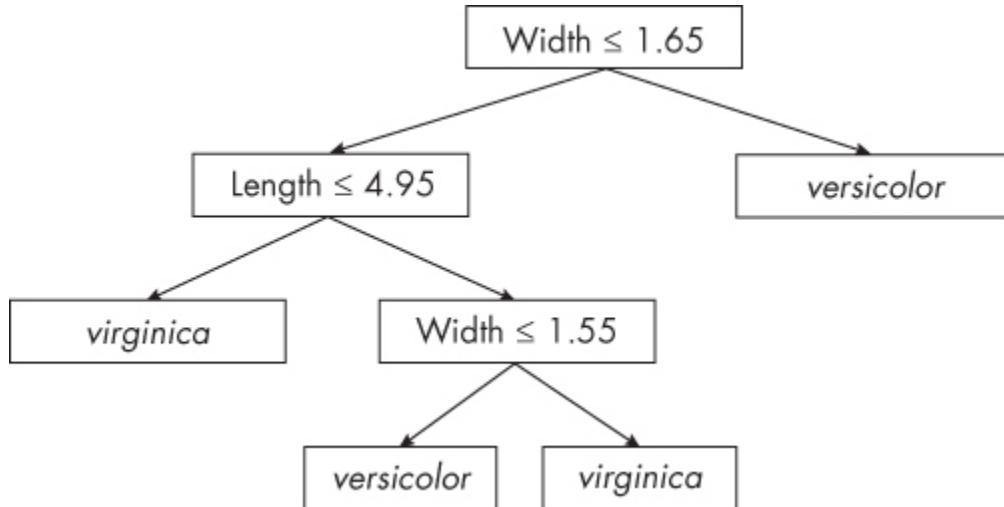


Figure 1-5: The decision tree for I. virginica versus I. versicolor

The first decision tree classifier was trivial, as the answer to a single question was sufficient to decide class membership. The second decision tree classifier is more common. Most machine learning models are not particularly simple. Though their operation is comprehensible, understanding why they act as they do is an entirely different matter. Decision trees are among the few model types that readily explain themselves. For any input, the path traversed from root to leaf in [Figure 1-5](#) explains in detail why the input received a particular label. The neural networks behind modern AI are not so transparent.

For a model to perform well “in the wild,” meaning when used in the real world, the data used to train the model must cover the entire range of inputs that the model might encounter. For

example, say we want a model to identify pictures of dogs, and our training set contains images of only dogs and parrots. While the model performs well on our held-out test set, which also includes pictures of dogs and parrots, what might happen when we deploy the model and it comes across a picture of a wolf? Intuitively, we might expect the model to say “it’s a dog,” just as a small child might before they learn what a wolf is. This is precisely what most machine learning models would do.

To illustrate this, let’s try an experiment. A popular dataset used by all AI researchers consists of tens of thousands of small images containing handwritten digits, 0 through 9. It goes by the uninspiring name of MNIST (Modified NIST) because it was derived in the late 1990s from a dataset constructed by the National Institute of Standards and Technology (NIST), the division of the United States Department of Commerce tasked with implementing all manner of standards for just about everything in the commercial and industrial realm.

[Figure 1-6](#) presents some typical MNIST digit samples. Our goal is to build a neural network that learns to identify the digits 0, 1, 3, and 9. We can train neural networks without knowing how they work because of powerful, open source toolkits like scikit-learn that are available to everyone. On the one hand, this democratizes AI; on the other, a little knowledge is often a

dangerous thing. Models may appear to be good when they're flawed in reality, and lack of knowledge about how the models work might prevent us from realizing that fact before it's too late.

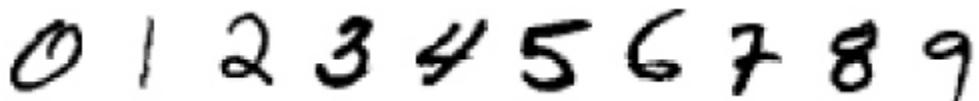


Figure 1-6: Sample MNIST digits

After the classifier is trained, we'll throw it a few curveballs by handing it images of fours and sevens—inputs the AI never saw during training. What might the model do with such inputs?

I trained the digits model using an open source toolkit. For now, all we need to know about the dataset is that the input feature vectors are unraveled digit images; the first row of pixels is followed by the second row, then the third row, and so on, until the entire image is unraveled into one long vector, a string of numbers. The digit images are 28×28 pixels, making the feature vector 784 numbers long. We're asking the neural network to learn about things in a 784-dimensional space, rather than the simple 2-dimensional space we used previously, but machine learning is up to the challenge.

The training set used to condition the neural network model contained 24,745 samples, roughly 6,000 of each digit type (0, 1,

3, and 9). This is likely enough to fairly represent the types of digits the model might encounter when used, but we need to try it to know. AI is a largely empirical science.

The held-out test set, also containing the digits 0, 1, 3, and 9, had 4,134 samples (about 1,000 for each digit).

We'll use a *confusion matrix*, a two-dimensional table of numbers, to evaluate the model. Confusion matrices are the most common way to evaluate a model because they show how it behaves on the test data.

In this case, the confusion matrix for our digit classifier is shown in [Table 1-1](#).

Table 1-1: The Digit Classifier Confusion Matrix

		0	1	3	9
0	978	0	1	1	
1	2	1,128	3	2	
3	5	0	997	8	

0	1	3	9
9	5	1	8

995

The matrix rows represent the true labels for the samples given to the model. The columns are the model's responses. The values in the table are counts, the number of times each possible combination of input class and model-assigned label happened.

For example, the first row represents the zeros in the test set. Of those 980 inputs, the model returned a label of zero for 978 of them, but it said the input was a three once and a nine another time. Therefore, when zero was the input, the model's output was correct 978 out of 980 times. That's encouraging.

Similarly, when the input was a one, the model returned the correct label 1,128 times. It was right 997 times for threes and 995 times for nines. When a classifier is good, the numbers along the diagonal of the confusion matrix from upper left to lower right are high, and there are almost no numbers off that diagonal. Off-diagonal numbers are errors made by the model.

Overall, the digits model is 99 percent accurate. We have a solid, well-performing model—that is, if we can ensure that all inputs

to the model are indeed a 0, 1, 3, or 9. But what if they aren't?

I handed the model 982 fours. The model replied like this:

0	1	3	9
---	---	---	---

48	9	8	917
----	---	---	-----

In other words, the model returned a label of 9 for 917 of the 982 fours, a label of 1 for 48 fours, and labels of 1 or 3 for the rest. How about sevens?

0	1	3	9
---	---	---	---

19	20	227	762
----	----	-----	-----

The model still favored calling sevens nines, but it often called them threes as well. Neural networks are loath to give up their secrets when explaining their actions, but in this case, of the 227 sevens labeled as threes, 47 of them were European-style sevens with a slash. A random sample of 227 sevens from the entire dataset turned up only 24 European-style sevens. The comparison isn't rigorous mathematically, but it hints that

sevens with a slash are often close enough to a three to fool the model.

The model was never taught to recognize fours or sevens, so it did the next best thing and placed them in a nearby category. Depending on how they're written, people might sometimes confuse fours and sevens for nines, for example. The model is making the kind of mistakes people make, which is interesting—but, more significantly, the model is poor because it wasn't trained on the full range of inputs it might encounter. It has no way of saying "I don't know," and getting a model to reliably say this can be tricky.

This is a simple exercise, but the implications are profound. Instead of digits, what if the model was looking for cancer in medical images but was never trained to recognize an important category of lesion or all the forms that lesion might take? A properly constructed and comprehensive dataset might mean the difference between life and death.

We can also think about the digits example in terms of interpolation and extrapolation. *Interpolation* approximates

within the range of known data, and *extrapolation* goes beyond known data.

For the digits example, interpolation might refer to encountering a tilted zero in the wild when none of the zeros in the training set were particularly tilted. The model must interpolate, in a sense, to respond correctly. Extrapolation is more like classifying a zero with a slash through it, which is something unseen during training time. To better understand these terms, let's model the world population from 1950 through 2020.

First, we'll fit a line to the data from 1950 through 1970. Fitting a line is a form of curve fitting; think of it as machine learning's less sophisticated cousin. To fit a line, find two numbers: the slope and the intercept. The slope tells us how steep the line is. If the slope is positive, the line is increasing as we move from left to right along the x -axis of a graph. A negative slope means the line decreases as we move along the x -axis. The intercept is where the line intersects the y -axis; that is, the value when the input is zero.

To fit a line, we use an algorithm to find the slope and intercept that best characterize the data (here, world population from 1950 through 1970). [Figure 1-7](#) shows a plot of the line and the

actual populations by year, denoted by plus signs. The line passes through or near to most of the plus signs, so the fit is reasonable. Notice that the population is in billions.

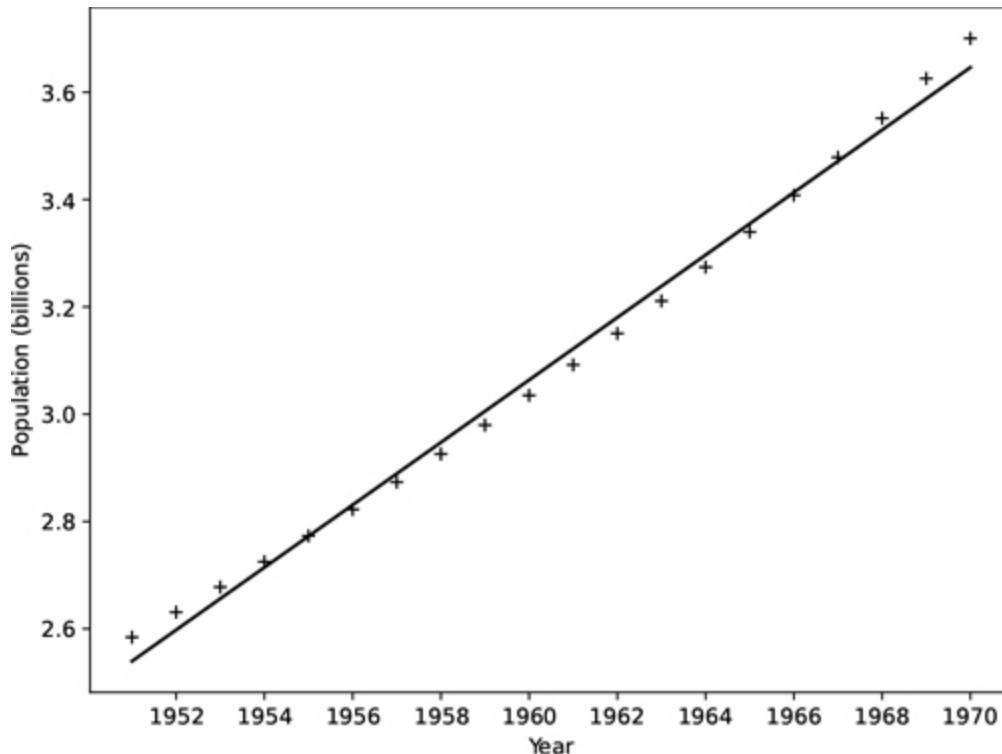


Figure 1-7: World population from 1950 through 1970

Once we have the line, we can use the slope and intercept to estimate the population for any year. Estimating for years between 1950 and 1970 is interpolating, because we used data from that range of years to create the line. If we estimate populations for years before 1950 or after 1970, we are extrapolating. [Table 1-2](#) shows our results when interpolating.

Table 1-2: Interpolating the Population Between 1950 and 1970

Year	Interpolated	Actual
1954	2.71	2.72
1960	3.06	3.03
1966	3.41	3.41

The interpolated population values are quite close to the actual population values, meaning the model (here the line fit to the data) is doing well. Now, let's extrapolate to dates outside the fit range, as shown in [Table 1-3](#).

Table 1-3: Extrapolating the Population After 1970

Year	Extrapolated	Actual
1995	5.10	5.74
2010	5.98	6.96
2020	6.56	7.79

The difference between the extrapolated population values and the actual population is increasing with each year. The model isn't doing well. Plotting the full range from 1950 through 2020 reveals the problem; see [Figure 1-8](#).

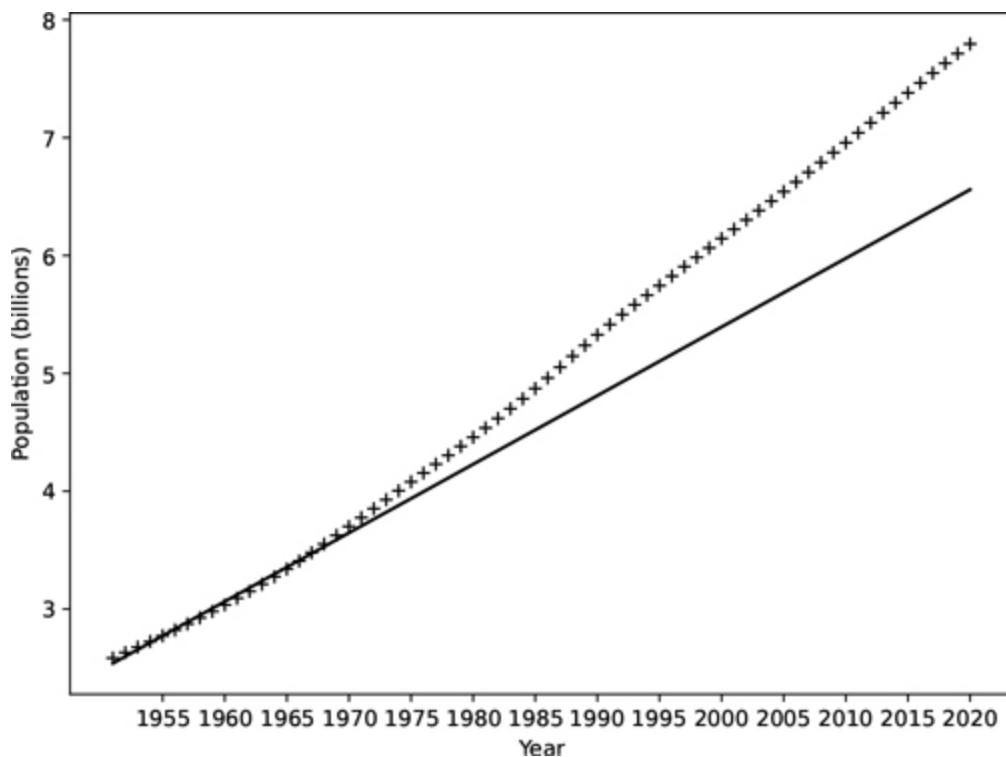


Figure 1-8: World population from 1950 through 2020

As time goes by, the fit line becomes increasingly wrong because the data is not linear after all. That is, the rate of growth is not constant and doesn't follow a straight line.

When extrapolating, we might have reason to believe that the data will continue to fit the line; if that's a valid assumption, then the line will continue to be a good fit. However, in the real

world, we usually have no such assurance. So, as a slogan, we might say interpolation good, extrapolation bad.

Fitting a line to some data is an example of *curve fitting*. What is true for curve fitting is also true for AI. The handwritten digits model did well when given inputs close to the data it was trained to recognize. The digits in the test data were all instances of 0, 1, 3, and 9, so the test data was like the training data. The two datasets are from the same *distribution*, and the same data-generating process created both. We can therefore claim that the model was, in a way, interpolating in those cases. However, when we forced the model to make decisions about fours and sevens, we were extrapolating by having the model make decisions about data it never saw during training.

It bears repeating: interpolation good, extrapolation bad. Bad datasets lead to bad models; good datasets lead to good models, which behave badly when forced to extrapolate. And, for good measure: all models are wrong, but some are useful.

Along the same lines of Hilaire Belloc's 1907 book *Cautionary Tales for Children*—an amusing and somewhat horrifying look at foolish things children do that could lead to an unfortunate

end—let's examine some cautionary tales that AI practitioners should be aware of when training, testing, and, most of all, deploying models.

In 2016, I attended a conference talk where the presenter demonstrated research into understanding why a neural network chooses the way it does. This is not yet a solved problem, but progress has been made. In this case, the research marked parts of the input images that influenced the model's decision.

The speaker displayed pictures of huskies and wolves and discussed his classifier for differentiating between the two. He showed how well it performed on a test set and asked the audience of machine learning researchers if this was a good model. Many people said yes, but with hesitation because they expected a trap. They were right to be hesitant. The speaker then marked the images to show the parts that the neural network focused on when making its decisions. The model wasn't paying attention to the dogs or the wolves. Instead, the model noticed that all the wolf training images had snow in the background, while none of the dog images contained snow. The model learned nothing about dogs and wolves but only about snow and no snow. Careless acceptance of the model's behavior

wouldn't have revealed that fact, and the model might have been deployed only to fail in the wild.

A similar tale is told of a very early machine learning system from the 1950s or 1960s. This one is likely apocryphal, though I have read a paper from that period that might be the origin of the urban legend. In this case, the images were bird's-eye views of forests. Some images contained a tank, while others did not.

A model trained to detect tanks seemed to work well on the training data but failed miserably when set loose in the wild. It was eventually realized that one set of training images had been taken on a sunny day and the other on a cloudy day. The model had learned nothing that its creators assumed it had.

More recent examples of this phenomenon exist with more advanced machine learning models. Some have even fooled experts into believing the model had learned something fundamental about language or the like when, instead, it had learned extremely subtle correlations in the training data that no human could (easily) detect.

The word *correlation* has a strict mathematical meaning, but we capture its essence with the phrase “correlation does not imply causation.” Correlation is when two things are linked so that the

occurrence of one implies the occurrence of the other, often in a particular order. More concretely, correlation measures how strongly a change in one thing is associated with a change in another. If both increase, they are positively correlated. If one increases while the other decreases, they are negatively correlated.

For example, a rooster crows, and the sun comes up. The two events are time-dependent: the rooster first, then the sun. This correlation does not imply causation, as the rooster crowing doesn't cause the sun to rise, but if such a correlation is observed often enough, the human mind begins to see one as causing the other, even when there is no real evidence of this. Why humans act this way isn't hard to understand. Evolution favored early humans who made such associations because, sometimes, the associations led to behavior beneficial for survival.

“Correlation does not imply causation” also applies to AI. The aforementioned models learned to detect things in the training data that correlated with the intended targets (dogs, wolves, tanks) but didn't learn about the targets themselves. Savvy machine learning practitioners are always on the lookout for such spurious correlations. Using a large and highly diverse

dataset for training and testing can defend against this effect, though this isn't always possible in practice.

We must ask whether our models have learned what we assume they have. And, as we saw with the MNIST digits, we must ensure that our models have seen all the kinds of inputs they will encounter in the wild—they should interpolate, not extrapolate.

This matters more than it might initially appear. Google learned this lesson in 2015 when it deployed a feature for Google Photos, wherein the model was insufficiently trained on human faces and made incorrect and inappropriate associations. Bias, in both the generic and social senses, is a real issue in AI.

Let's perform another experiment with MNIST digits. This time, the model has a seemingly simple decision to make: is the input digit a nine? The model is the same neural network used previously. If trained on a dataset where every image is either a nine or any other digit except four or seven (that is, no fours or sevens are in the training data), then the model is 99 percent accurate, as the confusion matrix shows:

Not 9	9
Not 9	9

Not 9	9	
Not 9	9,754	23
9	38	1,362

The confusion matrix tells us that the model correctly labeled 9,754 out of 9,777 test images that were not a nine. The model's label was also correct for 1,362 of the 1,400 nines. While the model performs well on the test set, the set does not contain fours or sevens.

In this case, the confusion matrix is small because the model has only two classes: nine or not nine. In other words, this is a binary model.

The 23 in the upper-right corner of the matrix represents 23 times when the input wasn't a nine, but the model said it was. For a binary model, class 1 is usually considered the class of interest, or the positive class. Therefore, these 23 inputs represent *false positives*, because the model said “it's a nine” when it wasn't. Similarly, the 38 samples at the lower left are *false negatives* because the model said “it's not a nine” when the input actually was. We want models with no false positives or

negatives, but sometimes it's more important to minimize one than the other.

For example, if a model is to detect breast cancer in mammograms, a false positive represents a case where the model says, "it might be cancer," even though it isn't. That's scary to hear, but further testing will show that the model was wrong. However, a false negative represents a missed cancer. We might tolerate a model with more false positives if it also has virtually no false negatives, as a false positive is less catastrophic than a false negative. We're beginning to appreciate how important it is to fully train, characterize, test, and *understand* our machine learning models.

All right, back to our experiment. The "is it a nine" classifier, like our earlier MNIST model, knows nothing about fours or sevens. When shown fours and sevens, the MNIST model typically called them nines. Will this model do the same? Here's what I received when I gave the model fours and sevens:

Not 9 9

Not 9 5,014 9,103

The model marked 9,103 of the 14,117 fours and sevens as nines. That's slightly more than 65 percent, or roughly two out of every three. This mimics the case where we present the model with inputs of a type it was never trained to detect.

Let's help the model by adding fours and sevens to the training set. Hopefully, providing examples that say, "It looks like a nine, but it isn't," formally known as *hard negatives*, will improve the model. I made 3 percent of the training data fours and sevens. The overall model was just as accurate as before, 99 percent, and here's what happened when I gave it fours and sevens it had never seen before:

Not 9	9
--------------	----------

Not 9 9,385 3,321

That's better. Instead of calling two-thirds of four or seven inputs a nine, the model labeled only one in four as a nine. Even a few examples of things that look like the positive class but aren't can help. If I boost the proportion of fours and sevens in the training set to 18 percent, the model misclassifies fours and sevens less than 1 percent of the time. Because models

learn from data, we *must* use datasets that are as complete as possible so our models interpolate and do not extrapolate.

NOTE

To be completely accurate, recent research shows that modern deep learning models are almost always extrapolating, but the more similar the inputs are to the data on which the model was trained, the better the performance, so I feel justified in using the analogy.

Everyone who seeks to understand, let alone work with, AI must take the warnings about the quality of the data used to train AI models to heart. A 2021 research article published in the journal *Nature Machine Intelligence* by Michael Roberts et al., “Common Pitfalls and Recommendations for Using Machine Learning to Detect and Prognosticate for COVID-19 Using Chest Radiographs and CT Scans,” is a sobering example. The authors assessed the performance of machine learning models designed to detect COVID-19 in chest X-rays and CT scans, reducing the initial candidate pool of over 2,000 studies (models) to 62 for rigorous testing. In the end, the authors declared *none* of the models fit for clinical use because of flaws in construction, bias in the datasets, or both.

Results like these have led to the creation of *explainable AI*, a subfield that seeks to give models the ability to explain themselves.

Look at your data and try to understand, as far as humanly possible, what your model is doing and *why*.

This chapter's title, "And Away We Go," was comedian Jackie Gleason's tagline. It's often good to dive into a subject to get an overview before coming back to understand things at a deeper level. In other words, we rush in to get a feel for the topic before exploring more methodically.

You'll find the many new terms and concepts introduced in this chapter in the glossary at the end of the book. My goal isn't for you to understand them all now, let alone retain them, but to plant seeds so that the next time you encounter one of these terms or concepts, you'll be more likely to think, "Ah, I know that one." Later chapters reinforce them, and you'll learn the important ones via repeated exposure.

There are two categories of takeaways from this chapter. The first has to do with what AI is and its essential pieces. The

second is about building intuition about what AI offers and how we should respond.

AI involves models, as yet nebulous entities we can condition with data to perform some desired task. There are many types of AI models, and this chapter introduced two: decision trees and neural networks. I won't say much more about decision trees, but neural networks occupy most of the remainder of the book.

Models are often best thought of as functions, like the mathematical functions you may remember from school or the functions that form the core of most computer programs. Both can be considered black boxes, where something goes in (the input) and something comes out (the output). In AI, the input is a feature vector, a collection of whatever is appropriate for the task at hand. In this chapter, we used two feature vectors: measurements of a flower and images of a handwritten digit.

Training conditions the model by altering its parameters to make it as accurate as possible. It's necessary to exercise caution when training most models to learn the general features of the data and not spurious correlations or the minute details of the training set (a concept known as *overfitting*, which we'll discuss in [Chapter 4](#)).

Proper development of machine learning models means we must have a test set, a collection of known input and output pairs that we do not use when training. We use this set after training to evaluate the model. If the dataset is constructed correctly, the test set provides an idea of how well we can expect the model to perform in the wild.

The second takeaway relates to what AI offers and how we should respond to it. While AI is powerful, it doesn't think as we do (though the models of [Chapter 7](#) might disagree). AI lives and dies by data and is only as good as the data we feed to it. If the dataset is biased, the AI is biased. If the dataset neglects to include examples of the types of inputs it will encounter when used, the AI will fail to handle such inputs properly.

The chapter's examples warn us to be careful when assuming AI operates as intended. Did the model learn what we wanted it to learn? Was it influenced by correlations in the data that we didn't notice or, worse still, that we are too limited to discern? Think back to the huskies versus wolves example.

Because AI is only as good as the data fed to it, it's on us to make datasets fair and unbiased and to understand what the AI has truly learned without assumptions.

AI first appeared in the 1950s, so why is it now suddenly everywhere we look? The next chapter answers this question.

KEY TERMS

algorithm, artificial intelligence, classifier, class label, confusion matrix, dataset, decision tree, deep learning, explainable AI, feature, feature vector, machine learning, model, neural network, parameters, testing, training

2

WHY NOW? A HISTORY OF AI



Rowan Atkinson's comic masterpiece *Mr. Bean* opens in the dead of night on a deserted London street. A spotlight appears, the title character falls from the sky, and a choir sings in Latin, "ecce homo qui est faba"—behold the man who is a bean. Mr. Bean picks himself up, brushes off his suit, and runs awkwardly into the darkness. He is something otherworldly, a thing that literally fell from the sky, defying comprehension.

Given the parade of AI wonder after wonder in recent years, we might be excused for thinking that AI, like Mr. Bean, fell from the sky, fully formed and beyond our comprehension. However, none of this is true; indeed, I'd argue that AI is still in its infancy.

So why are we hearing about AI now? I'll answer that question with a brief (and biased) history of AI, followed by a discussion

of the advances in computing that acted as the catalyst for the AI revolution. This chapter provides context for the models we'll explore throughout the remainder of the book.

Since its inception, AI has been divided into two main camps: symbolic AI and connectionism. *Symbolic* AI attempts to model intelligence by manipulating symbols and logical statements or associations. *Connectionism*, however, attempts to model intelligence by building networks of simpler components. The human mind embodies both approaches. We use symbols as elements of thought and language, and our minds are constructed from unbelievably complex networks of neurons, each neuron a simple processor. In computer programming terms, the symbolic approach to AI is top-down, while connectionism is bottom-up. Top-down design starts with high-level tasks, then breaks those tasks into smaller and smaller pieces. A bottom-up design begins with smaller pieces and combines them together.

Proponents of symbolic AI believe that intelligence can be achieved in the abstract, without a substrate resembling a brain. Connectionists follow the evolutionary development of brains and argue that there needs to be some foundation, like a

massive collection of highly interconnected neurons, from which intelligence (however defined) can emerge.

While the debate between symbolic AI and connectionism was long-lived, with the advent of deep learning it's safe to say that the connectionists have won the day—though perhaps not the war. Recent years have seen a smattering of papers blending the two approaches. I suspect symbolic AI has a cameo or two left in it, if not ultimately starring in a supporting role.

My introduction to AI in the late 1980s was entirely symbolic. Connectionism was mentioned as another approach, but neural networks were thought inferior and likely to be marginally useful at best.

A complete history of artificial intelligence is beyond our scope. Such a magnum opus awaits a motivated and capable historian. Instead, I'll focus on the development of machine learning while (very unfairly!) ignoring the mountain of effort expended over the decades by those in the symbolic camp. Know, however, that for most of AI's history, people mostly spoke of symbolic AI, not connectionism. For a fairer presentation, I recommend Michael Wooldridge's book *A Brief History of Artificial Intelligence* (Flatiron Books, 2021), or Pamela McCorduck's deeply personal account in *This Could Be*

Important: My Life and Times with the Artificial Intelligentsia
(Lulu Press, 2019).

With my apparent connectionist bias in mind, let's take a stroll through the history of machine learning.

Pre-1900

The dream of intelligent machines dates back to antiquity. Ancient Greeks related the myth of Talos, a giant robot meant to guard the Phoenician princess, Europa. Throughout the Middle Ages and Renaissance, many automatons—machines that moved and appeared lifelike—were developed. However, I suspect that none were believed to be intelligent or capable of thought. Some were even hoaxes, like the infamous Mechanical Turk that wowed the world by playing, and beating, many skilled chess players. In the end, it was discovered that a person hiding within the machine could control the “automaton” by manipulating a mechanical arm to move free-standing chess pieces on the board while viewing the board configuration from beneath. Still, the mechanical part of the machine was rather impressive for the late 18th century.

Apart from automatons, there were also early attempts to understand thought as a mechanical process and efforts to

produce a logical system capable of capturing thought. In the 17th century, Gottfried Leibniz described such a concept abstractly as an “alphabet of thought.” In the 1750s, Julien Offray de La Mettrie published *L'Homme Machine (Man as Machine)*, arguing that thought is a mechanical process.

The idea that human thought might emerge from the physical entity of the brain rather than the spiritual soul marked the beginning of a new chapter on the road to AI. If our minds are biological machines, why can't there be another kind of machine that thinks?

In the 19th century, George Boole attempted to create a calculus of thought, resulting in what we know now as Boolean algebra. Computers depend on Boolean algebra, to the point that it represents their very implementation as collections of digital logic gates. Boole was partially successful, but he didn't achieve his stated goal: “to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolic language of a Calculus” (*The Laws of Thought*, 1854). That Boole was willing to try represented another step toward the notion that AI might be possible.

What these early attempts were lacking was an actual calculating machine. People could dream of artificial minds or beings (like the creature from Mary Shelley's *Frankenstein*) and, assuming their existence, discuss the repercussions. But until there was a machine capable of plausibly mimicking (implementing?) thought, all else was speculation.

It was Englishman Charles Babbage who, in the mid-19th century, first conceived of an implementable general-purpose calculating machine: the Analytical Engine. The Engine was never built in its entirety, but it contained all the essential components of a modern computer and would, in theory, be capable of the same operations. While it's unclear if Babbage appreciated the potential versatility of his machine, his friend, Ada Lovelace, did. She wrote about the machine as a widely applicable, general-purpose device. Still, she did not believe the Engine was capable of thought, as this quote from her *Sketch of the Analytical Engine* (1843) demonstrates:

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.

This quote may be the first to refer to the possibility of artificial intelligence involving a device potentially capable of achieving

it. The phrase “do whatever we know how to order it to perform” implies programming. Indeed, Lovelace wrote a program for the Analytical Engine. Because of this, many people consider her to be the first computer programmer. The fact that her program had a bug in it proves to me that she was; nothing is more emblematic of programming than bugs, as my 40-plus years of programming experience have demonstrated distressingly often.

1900 to 1950

In 1936, a 24-year-old Englishman named Alan Turing, still a student at the time, wrote a paper that has since become the cornerstone of computer science. In this paper, Turing introduced a generic conceptual machine, what we now call a *Turing machine*, and demonstrated that it could calculate anything representable by an algorithm. He also explained that there are things that cannot be implemented by algorithms and that are, therefore, uncomputable. Since all modern programming languages are equivalent to a Turing machine, modern computers can implement any algorithm and compute anything computable. However, this says nothing about how long the computation might take or the memory required.

If a computer can compute anything that can be implemented as an algorithm, then a computer can perform any mental operation a human can perform. At last, here was the engine that might enable true artificial intelligence. Turing's 1950 paper "Computing Machinery and Intelligence" was an early recognition that digital computers might eventually lead to intelligent machines. In this paper, Turing described his "imitation game," known now as the *Turing test*, by which humans might come to believe that a machine is intelligent. Many claims of AI systems that pass the Turing test have appeared, especially in recent years. One of these is OpenAI's ChatGPT. However, few would be inclined to believe that ChatGPT is truly intelligent—in other words, I suspect that this test fails to capture what humans generally understand this term to mean, and a new test will likely be created at some point.

In 1943, Warren McCulloch and Walter Pitts wrote "A Logical Calculus of Ideas Immanent in Nervous Activity," which deserves an award for one of the most opaque yet intriguing paper titles ever. The paper represents "nervous nets" (collections of neurons) as logical statements in mathematics. The logical statements are difficult to parse (at least for me), but the authors' description of "nets without circles" bears a strong resemblance to the neural networks we'll explore in [Chapter 4](#)

—indeed, one could argue that McCulloch and Pitts’s groundbreaking paper led to what we now recognize as a neural network. Frankly, neural networks are far easier to parse and understand, which is good news for us.

The progression from fantastical stories about artificially intelligent machines and beings to a serious investigation of whether mathematics can capture thought and reasoning, combined with the realization that digital computers are capable of computing anything that can be described by an algorithm, set the stage for the advent of artificial intelligence as a legitimate research enterprise.

1950 to 1970

The 1956 Dartmouth Summer Research Project on Artificial Intelligence workshop is generally regarded as the birthplace of AI, and where the phrase “artificial intelligence” was first used consistently. The Dartmouth workshop had fewer than 50 participants, but the list included several well-known names in the worlds of computer science and mathematics: Ray Solomonoff, John McCarthy, Marvin Minsky, Claude Shannon, John Nash, and Warren McCulloch, among others. At the time, computer science was a subfield of mathematics. The workshop

was a brainstorming session that set the stage for early AI research.

In 1957, Frank Rosenblatt of Cornell University created the Mark I Perceptron, widely recognized as the first application of neural networks. The Perceptron was remarkable in many respects, including that it was designed for image recognition, the same application where deep learning first proved itself in 2012.

Figure 2-1 shows the conceptual organization as given in the *Perceptron Operators' Manual*. The Perceptron used a 20×20 -pixel digitized television image as input, which was then passed through a “random” set of connections to a set of association units that led to response units. This configuration is similar to some approaches to deep learning on images in use today and resembles a type of neural network known as an *extreme learning machine*.

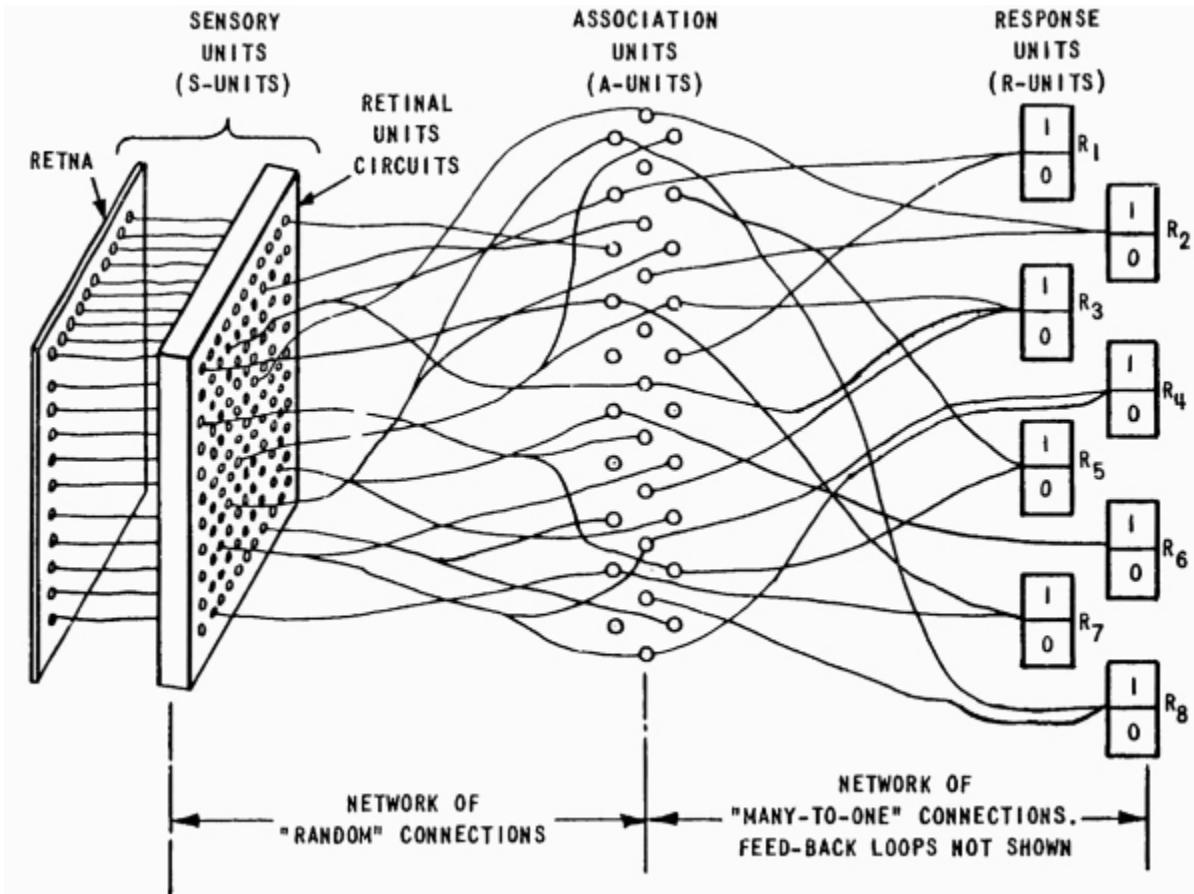


Figure 2-1: The organization of the Mark I Perceptron

If the Perceptron was on the right track, why was it all but forgotten for decades? One reason was Rosenblatt's penchant for hype. At a 1958 conference organized by the US Navy (a sponsor of the Perceptron project), Rosenblatt's comments were so hyperbolic that the *New York Times* reported:

The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Later perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech and writing in another language, it was predicted.

The comments ruffled many feathers at the time, though as modern AI systems do allow machines to walk, talk, see, write, recognize people, and translate speech and writing between languages, perhaps we should be more forgiving toward Rosenblatt. He was only some 60 years early.

A few years later, in 1963, Leonard Uhr and Charles Vossler described a program that, like the Perceptron, interpreted a 20×20 -pixel image represented as a matrix of 0s and 1s. Unlike the Perceptron, this program was able to generate the patterns and combinations of image features necessary to learn its inputs. Uhr and Vossler's program was similar to the convolutional neural networks that appeared over 30 years later and are the subject of [Chapter 5](#).

The first of what I call the “classical” machine learning models appeared in 1967, courtesy of Thomas Cover and Peter Hart. Known as [*nearest neighbors*](#), it is the simplest of all machine learning models, almost embarrassingly so. To label an unknown input, it simply finds the known input most like it and uses that input’s label as the output. When using more than one nearby known input, the method is called *k-nearest neighbors*, where *k* is a small number, like 3 or 5. Hart went on to write the first edition of *Pattern Classification*, along with Richard Duda and David Stork, in 1973; this seminal work introduced many

computer scientists and software engineers to machine learning, including me.

The success of the Perceptron came to a screeching halt in 1969, when Marvin Minsky and Seymour Papert published their book *Perceptrons*, which demonstrated that single- and two-layer perceptron networks weren't able to model interesting tasks. We'll cover what "single-layer" and "two-layer" mean in time. *Perceptrons*, coupled with the 1973 release of "Artificial Intelligence: A General Survey" by James Lighthill, universally known as "the Lighthill report," ushered in what is now referred to as the first AI winter; funding for AI research dried up in short order.

Minsky and Papert's criticisms of the perceptron model were legitimate; however, many people missed their observation that such limitations were not applicable to more complex perceptron models. Regardless, the damage was done, and connectionism virtually vanished until the early 1980s.

Note the "virtually." In 1979, Kunihiko Fukushima released a paper that was translated into English in 1980 as "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." The name "Neocognitron" didn't catch on, and this

was perhaps one of the last uses of the “-tron” suffix that had been so popular in computer science for the previous three decades. While Uhr and Vossler’s 1963 program bore some similarities to a convolutional neural network, the Neocognitron is, to many people, the original. The success of convolutional neural networks led directly to the current AI revolution.

1980 to 1990

In the early 1980s, AI went commercial with the advent of computers specifically designed to run the Lisp programming language, then the lingua franca of AI. (Today, it’s Python.) Along with Lisp machines came the rise of *expert systems*—software designed to capture the knowledge of an expert in a narrow domain. The commercialization of AI brought the first AI winter to an end.

The concept behind expert systems is, admittedly, seductive. To build an expert system that, for example, diagnoses a particular kind of cancer, you first interview experts to extract their knowledge and arrange it in a knowledge base. A knowledge base represents knowledge as a combination of rules and facts. Then, you combine the knowledge base with an inference engine, which uses the knowledge base to decide when and

how to execute rules based on stored facts or input to the system by a user. Rules fire based on facts, which may lead to placing new facts in the knowledge base that cause additional rules to fire, and so on. A classic example of an expert system is CLIPS, which NASA developed in 1985 and released into the public domain in 1996.

In an expert system, there's no connectionist network or collection of units from which one might (hopefully) cause intelligent behavior to emerge, making it a good example of symbolic AI. Instead, the knowledge base is an essentially rigid collection of rules, like "if the engine temperature is above this threshold, then this other thing is the likely cause," and facts, like "the engine temperature is below the threshold."

Knowledge engineers are the links between the experts and the expert system. Building a knowledge base from the experts' answers to the questions posed by the knowledge engineers is complex, and the resulting knowledge base is hard to modify over time. However, the difficulty in designing expert systems doesn't mean they're useless; they still exist, mainly under the guise of "business rule management systems," but currently have minimal impact on modern AI.

The hype surrounding expert systems, combined with early successes, drove renewed interest in AI in the early 1980s. But

when it became clear that expert systems were too brittle to have a general use, the bottom fell out of the industry, and AI's second winter hit in the middle of the decade.

During the 1980s, connectionists occupied the background, but they were not sitting still. In 1982, John Hopfield demonstrated what are now known as Hopfield networks. A *Hopfield network* is a type of neural network that stores information in a distributed way within the weights of the network, and then extracts that information at a later time. Hopfield networks aren't widely used in modern deep learning, but they proved an important demonstration of the utility of the connectionist approach.

In 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams released their paper "Learning Representations by Back-propagating Errors," which outlined the backpropagation algorithm for training neural networks. Training a neural network involves adjusting the weights between the neurons so that the network operates as desired. The backpropagation algorithm was the key to making this process efficient by calculating how adjusting a particular weight affects the network's overall performance. With this information, it becomes possible to iteratively train the network by applying known training data, then using the network's errors when

classifying to adjust the weights to force the network to perform better on the next iteration. (I'll discuss neural network training in more depth in [Chapter 4](#).) With backpropagation, neural networks could go well beyond the limited performance of Rosenblatt's Perceptron. However, even with backpropagation, neural networks in the 1980s were little more than toys. While there's contention about who invented backpropagation and when, the 1986 paper is generally understood to be the presentation that influenced neural network researchers the most.

1990 to 2000

The second AI winter extended into the 1990s, but research continued in both the symbolic and connectionist camps. Corinna Cortes and Vladimir Vapnik introduced the machine learning community to [*support vector machines \(SVMs\)*](#) in 1995. In a sense, SVMs represent the high-water mark of classical machine learning. The success of SVMs in the 1990s through the early 2000s held neural networks at bay. Neural networks require large datasets and significant computational power; SVMs, on the other hand, are often less demanding of resources. Neural networks gain their power from the network's ability to represent a function, a mapping from inputs to the desired

outputs, while SVMs use clever mathematics to simplify difficult classification problems.

The success of SVMs was noted in the academic community as well as the broader world of software engineering, where applications involving machine learning were increasing. The general public was largely unaware of these advances, though intelligent machines continued appearing frequently in science fiction.

This AI winter ended in 1997 with the victory of IBM's Deep Blue supercomputer against then world chess champion Garry Kasparov. At the time, few people thought a machine could ever beat the best human chess player. Interestingly, a decade earlier, one of my professors had predicted that an AI would accomplish this feat before the year 2000. Was this professor clairvoyant? Not really. Deep Blue combined fast custom hardware with sophisticated software and applied known AI search algorithms (in particular, the Minimax algorithm). Combined with heuristics and a healthy dose of custom knowledge from other chess grandmasters, Deep Blue was able to out-evaluate its human opponent by searching more possible moves than any human could ever hope to contemplate. Regardless, at its core, Deep Blue implemented what AI experts knew *could* beat a human if the machine had enough resources

at its disposal. Deep Blue's victory was inevitable because researchers expected computers to eventually become fast enough to overcome a human's abilities. What was needed was known; all that remained was to put the pieces together.

The year 1998 saw the publication of "Gradient-Based Learning Applied to Document Recognition," a paper by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner that escaped public notice but was a watershed moment for AI and the world. While Fukushima's Neocognitron bore strong similarities to the convolutional neural networks that initiated the modern AI revolution, this paper introduced them directly, as well as the (in)famous MNIST dataset we used in [Chapter 1](#). The advent of convolutional neural networks (CNNs) in 1998 begs the question: why did it take another 14 years before the world took notice? We'll return to this question later in the chapter.

2000 to 2012

Leo Breiman introduced [*random forests*](#) in 2001 by forming the existing pieces of what would become the random forest algorithm into a coherent whole, much like Darwin did with evolution in the 19th century. Random forests are the last of the classical machine learning algorithms we'll contemplate in

[Chapter 3](#). If “random forests” remind you of the decision trees in [Chapter 1](#), there’s a reason: a random forest is a forest of decision trees.

Stacked denoising autoencoders are one type of intermediate model, and they were my introduction to deep learning in 2010. An *autoencoder* is a neural network that passes its input through a middle layer before generating output. It aims to reproduce its input from the encoded form of the input in the middle layer.

An autoencoder may seem like a silly thing to fiddle with, but while learning to reproduce its input, the middle layer typically learns something interesting about the inputs that captures their essence without focusing on fine, trivial details. For example, if the inputs are the MNIST digits, then the middle layer of an autoencoder learns about digits as opposed to letters.

A *denoising autoencoder* is similar, but we discard a random fraction of the input values before pushing the input through the middle layer. The autoencoder must still learn to reproduce the entire input, but now it has a more challenging task because the input is incomplete. This process helps the autoencoder’s middle layer discover a better encoding of the input.

Finally, a *stacked denoising autoencoder* is a stack of denoising autoencoders, wherein the output of the middle layer of one becomes the input of the next. When arranged this way, the stack learns a new representation of the input, which often helps a classifier appended to the top of the stack to discriminate between classes. For example, in my work at the time, the inputs were small pieces of an image that may have contained a target of interest. Two or three layers of trained stacked denoising autoencoders were used to transform the inputs into a list of numbers that would hopefully represent the input's essence while ignoring the image's minutiae. The outputs were then used with a support vector machine to decide if the input was a target.

2012 to 2021

Deep learning caught the world's attention in 2012 when AlexNet, a particular convolutional neural network architecture, won the ImageNet challenge with an overall error of just over 15 percent—far lower than any competitor. The ImageNet challenge asks models to identify the main subject of color images, whether a dog, a cat, a lawnmower, and so on. In reality, “dog” isn’t a sufficient answer. The ImageNet dataset contains 1,000 classes of objects, including some 120 different

dog breeds. So, a correct answer would be “it’s a Border Collie” or “it’s a Belgian Malinois.”

Random guessing means randomly assigning a class label to each image. In that case, we would expect an overall success rate of 1 in 1,000, or an error rate of 99.9 percent. AlexNet’s error of 15 percent was truly impressive—and that was in 2012. By 2017, convolutional neural networks had reduced the error to about 3 percent, below the approximate 5 percent achievable by the few humans brave enough to do the challenge manually. Can you discriminate between 120 different dog breeds? I certainly can’t.

AlexNet opened the floodgates. The new models broke all previous records and began to accomplish what no one had really expected from them: tasks like reimagining images in the style of another image or painting, generating a text description of the contents of an image along with the activity shown, or playing video games as well as or better than a human, among others.

The field was proliferating so quickly that it became nearly impossible to keep up with each day’s deluge of new papers. The only way to stay current was to attend multiple conferences per year and review the new work appearing on websites such

as arXiv (<https://www.arxiv.org>), where research in many fields is first published. This led to the creation of sites like <https://www.arxiv-sanity-lite.com>, which ranks machine learning papers according to reader interest in the hope that the “best” might become easier to find.

In 2014, another breakthrough appeared on the scene, courtesy of researcher Ian Goodfellow’s insight during an evening’s conversation with friends. The result was the birth of [*generative adversarial networks \(GANs\)*](#), which Yann LeCun called at the time the most significant breakthrough in neural networks in 20 to 30 years (overheard at NeurIPS 2016). GANs, which we’ll discuss in [Chapter 6](#), opened a new area of research that lets models “create” output that’s related to but different from the data on which they were trained. GANs led to the current explosion of [*generative AI*](#), including systems like ChatGPT and Stable Diffusion.

Reinforcement learning is one of the three main branches of machine learning, the other two being the supervised learning we’ve been discussing and unsupervised learning, which attempts to train models without labeled datasets. In reinforcement learning, an agent (a model) is taught via a reward function how to accomplish a task. The application to robotics is obvious.

Google's DeepMind group introduced a deep reinforcement learning-based system in 2013 that could successfully learn to play Atari 2600 video games as well as or better than human experts. (Who counts as an expert in a then-35-year-old game system, I'm not sure.) The most impressive part of the system, to me, was that the model's input was precisely the human's input: an image of the screen, nothing more. This meant the system had to learn how to parse the input image and, from that, how to respond by moving the joystick to win the game (virtually—they used emulators).

The gap between beating humans at primitive video games and beating humans at abstract strategy games like Go was, historically, deemed insurmountable. I was explicitly taught in the late 1980s that the Minimax algorithm used by systems like Deep Blue to win at chess did not apply to a game like Go; therefore, no machine would ever beat the best human Go players. My professors were wrong, though they had every reason at the time to believe their statement.

In 2016, Google's AlphaGo system beat Go champion Lee Sedol in a five-game match, winning four to one. The world took notice, further enhancing the growing realization that a paradigm shift had occurred. By this time, machine learning was already a commercial success. However, AlphaGo's victory

was utterly impressive for machine learning researchers and practitioners.

Most of the general public didn't notice that AlphaGo, trained on thousands of human-played Go games, was replaced in 2017 by AlphaGo Zero, a system trained entirely from scratch by playing against itself, with no human input given. In short order, AlphaGo Zero mastered Go, even beating the original AlphaGo system (scoring a perfect 100 wins and no losses).

However, in 2022, the current state-of-the-art Go system, KataGo, was repeatedly and easily defeated by a system trained not to win but to reveal the brittleness inherent in modern AI systems. The moves the adversarial system used were outside the range encountered by KataGo when it was trained. This is a real-world example of how models are good at interpolating but bad at extrapolating. When the adversarial system was trained not to be better at Go but to exploit and “frustrate” the AI, it was able to win better than three out of four games. I point the reader to the *Star Trek: The Next Generation* episode “Peak Performance,” where Data the android “wins” a difficult strategy game against a master not by attempting to win but by attempting to match and frustrate.

Deep learning's penchant for beating humans at video games continues. In place of primitive games like Atari's, deep reinforcement learning systems are now achieving grandmaster-level performance at far more difficult games. In 2019, DeepMind's AlphaStar system outperformed 99.8 percent of human players in *StarCraft II*, a strategy game requiring the development of units and a plan of battle.

The 1975 Asilomar Conference on Recombinant DNA was an important milestone in recognizing biotechnology's growth and potential ethical issues. The conference positively impacted future research, and that year its organizers published a summary paper outlining an ethical approach to biotechnology. The potential hazards of a field that was then primarily in its infancy were recognized early, and action was taken to ensure ethical issues were paramount when contemplating future research.

The 2017 Asilomar Conference on Beneficial AI intentionally mirrored the earlier conference to raise awareness of the potential hazards associated with AI. It is now common to encounter conference sessions with titles like "AI for Good." The 2017 Asilomar conference resulted in the development of a set of principles to guide the growth and application of artificial intelligence. Similarly, as of 2023, the US government—

specifically, the White House Office of Science and Technology Policy—has developed a “Blueprint for an AI Bill of Rights” meant to protect the American public from the harmful effects of AI indiscriminately applied. Indeed, White House officials have taken pains to address the AI community directly to encourage proper consideration in developing even more powerful AI systems. All of this is a good sign, but history teaches that human law often lags behind technological development, so the ultimate effectiveness of these necessary attempts at framing the field remains to be seen.

2021 to Now

The COVID-19 pandemic of 2020 brought most of the world to a standstill. However, the AI community was only minimally impacted by the pandemic, likely because remote collaboration and conferences work rather well in this field. In addition, we can access powerful computers over the internet, so physical proximity doesn’t limit research as it might in other disciplines where access to the hardware is necessary.

Beginning in 2021 and continuing as I write, an explosion of new models have appeared, each more impressive than the last. Most can accept text input written by humans to produce text, images, or even video output. Often, the input marries text and

example imagery to guide the system. Current popular (and concerning) systems include DALL-E 2, Stable Diffusion, Midjourney, and ChatGPT, among others. Most large tech companies have also demonstrated cutting-edge techniques that generate video, audio, and even 3D models. The current buzz generated by these systems is reminiscent of the excitement in 2012, when deep learning proved its worth with AlexNet. It's another paradigm shift; but, to be clear, none of these systems are conscious. I'm not alone in this opinion. Even ChatGPT agrees with me. I asked it: "Are you a conscious being?" Here's its reply, verbatim:

As a machine learning model, I am not a conscious being. I am a computer program that has been trained to generate text based on input from a user. I do not have the ability to think, reason, or experience consciousness in the same way that humans do. I am designed to provide information and assist users, but I do not have any independent thought or consciousness.

I think Ada Lovelace would be satisfied with this answer, though she might be puzzled about how an unthinking machine could generate it. Note, however, that ChatGPT didn't claim not to think, only that it doesn't think like humans do. We'll explore image synthesis in [Chapter 6](#) and large language models like ChatGPT in [Chapter 7](#). Perhaps then we'll find a resolution to her (assumed) confusion.

So, why now? The short answer is the fall of symbolic AI and the rise of technological innovations highly favorable to the connectionist approach.

Symbolic AI and connectionism emerged together, with symbolic AI dominating for decades and forcing connectionism into the background. But after two AI winters that left symbolic AI barely breathing, connectionism, assisted by key technological innovations, has risen to fill the void.

I think of the relationship between symbolic AI and connectionism as akin to that between non-avian dinosaurs and mammals. Dinosaurs and mammals emerged at roughly the same time, geologically speaking, but large, terrestrial dinosaurs dominated the world for about 160 million years, forcing mammals to eke out an existence in the shadows. When the asteroid hit 66 million years ago, the large dinosaurs were wiped out, allowing the mammals to evolve and take over.

Of course, analogies ultimately break down. The dinosaurs didn't die out completely—we now call them birds—and they didn't go extinct because they were somehow inferior. In fact, the dinosaurs are one of Earth's greatest success stories. Non-

avian dinosaurs died because of plain old bad luck. It was, almost literally, a disaster that did them in (“disaster” from the Italian *disastro*, meaning “ill star”).

Might symbolic AI reemerge? It’s likely in some form, but in cooperation with connectionism. Symbolic AI promised that intelligent behavior was possible in the abstract, and it didn’t deliver. Connectionism claims that intelligent behavior can emerge from a collection of simpler units. Deep learning’s successes support this view, to say nothing of the billions of living brains currently on the planet. But, as ChatGPT pointed out, existing connectionist models “do not think, reason, or experience consciousness in the same way that humans do.” Modern neural networks are not minds; they are representation-learning data processors. I’ll clarify what that means in [Chapter 5](#).

Though our species, *Homo sapiens*, relies critically on symbolic thought, it isn’t a requirement for intelligence. In his book *Understanding Human Evolution* (Cambridge University Press, 2022), anthropologist Ian Tattersall claims it was unlikely that Neanderthals used symbolic thought as we do, nor did they have language as we do, but that they were nonetheless intelligent. Indeed, the Neanderthals were sufficiently human for our ancestors to “make love, not war” with them more than

once—the DNA of people of non-African ancestry testifies to this fact.

I expect a synergy between connectionism and symbolic AI in the near future. For example, because a system like ChatGPT is, in the end, only predicting the next output token (word or part of a word), it can't know when it's saying something wrong. An associated symbolic system could detect faulty reasoning in the response and correct it. How such a system might be implemented, I don't know.

Hints of what might emerge from connectionism were evident by the early 1960s. So, was it only symbolic AI bias that delayed the revolution for so many decades? No. Connectionism stalled because of speed, algorithm, and data issues. Let's examine each in turn.

Speed

To understand why speed stalled the growth of connectionism, we need to understand how computers work. Taking great liberties allows us to think of computers as memory, which holds data (numbers) and a processing unit, typically known as the central processing unit (CPU). A microprocessor—like the

one in your desktop computer, smartphone, voice-controlled assistant, car, microwave, and virtually everything else you use that isn't a toaster (oh, and in many toasters too)—is a CPU. Think of a CPU as a traditional computer: data comes into the CPU from memory or input devices like a keyboard or mouse, gets processed, then is sent out of the CPU to memory or an output device like a monitor or hard drive.

Graphics processing units (GPUs), on the other hand, were developed for displays, primarily for the video game industry, to enable fast graphics. GPUs can perform the same operation, such as “multiply by 2,” on hundreds or thousands of memory locations (read: *pixels*) simultaneously. If a CPU wants to multiply a thousand memory locations by 2, it must multiply the first, second, third, and so on sequentially. As it happens, the primary operation needed to train and implement a neural network is ideally suited to what a GPU can do. GPU makers, like NVIDIA, realized this early and began developing GPUs for deep learning. Think of a GPU as a supercomputer on a card that fits in your PC.

In 1945, the Electronic Numerical Integrator and Computer (ENIAC) was state-of-the-art. ENIAC's speed was estimated to be around 0.00289 million instructions per second (MIPS). In other words, ENIAC could perform just under 3,000 instructions in

one second. In 1980, a stock 6502 8-bit microprocessor like the ones in most then-popular personal computers ran at about 0.43 MIPS, or some 500,000 instructions per second. In 2023, the already somewhat outdated Intel i7-4790 CPU in the computer I'm using to write this book runs at about 130,000 MIPS, making my PC some 300,000 times faster than the 6502 from 1980 and about 45 million times faster than ENIAC.

However, NVIDIA's A100 GPU, when used for deep learning, is capable of 312 teraflops (TFLOPS), or 312,000,000 MIPS: 730 million times faster than the 6502 and an unbelievable 110 *billion* times faster than ENIAC. The increase in computational power over the timespan of machine learning boggles the mind. Moreover, training a large neural network on an enormous dataset often requires dozens to hundreds of such GPUs.

Conclusion: Computers were, until the advent of fast GPUs, too slow to train neural networks with the capacity needed to build something like ChatGPT.

Algorithm

As you'll learn in [Chapter 4](#), we construct neural networks from basic units that perform a simple task: collect input values, multiply each by a weight value, sum, add a bias value, and

pass the result to an activation function to create an output value. In other words, many input numbers become one output number. The collective behavior emerging from thousands to millions of such units leading to billions of weight values lets deep learning systems do what they do.

The structure of a neural network is one thing; conditioning the neural network to the desired task is another. Think of the network's structure, known as its *architecture*, as anatomy. In anatomy, we're interested in what constitutes the body: this is the heart, that's the liver, and so on. Training a network is more like physiology: how does one part work with another? The anatomy (architecture) was there, but the physiology (training process) was incompletely understood. That changed over the decades, courtesy of key algorithmic innovations: backpropagation, network initialization, activation functions, dropout and normalization, and advanced gradient descent algorithms. It's not essential to understand the terms in detail, only to know that improvements in what these terms represent —along with the already mentioned improvements in processing speed, combined with improved datasets (discussion coming up)—were primary enablers of the deep learning revolution.

While it was long known that the right weight and bias values would adapt a network to the desired task, what was missing for decades was an efficient way to *find* those values. The 1980s' introduction of the backpropagation algorithm, combined with stochastic gradient descent, began to change this.

Training iteratively locates the final set of weight and bias values according to the model's errors on the training data. Iterative processes repeat from an initial state, some initial set of weights and biases. However, what should those initial weights and biases be? For a long time, it was assumed that the initial weights and biases didn't matter much; just select small numbers at random over some range. This approach often worked, but many times it didn't, causing the network not to learn well, if at all. A more principled approach to initializing networks was required.

Modern networks are still initialized randomly, but the random values depend on the network's architecture and the type of activation function used. Paying attention to these details allowed networks to learn better. Initialization matters.

We arrange neural networks in layers, where the output of one layer becomes the input of the next. The activation function assigned to each node in the network determines the node's

output value. Historically, the activation function was either a sigmoid or a hyperbolic tangent, both of which produce an S-shaped curve when graphed. These functions are, in most cases, inappropriate, and were eventually replaced by a function with a long name that belies its simplicity: the *rectified linear unit (ReLU)*. A ReLU asks a simple question: is the input less than zero? If so, the output is zero; otherwise, the output is the input value. Not only are ReLU activation functions better than the older functions, but computers can ask and answer that question virtually instantaneously. Switching to ReLUs was, therefore, a double win: improved network performance and speed.

Dropout and batch normalization are advanced training approaches that are somewhat difficult to describe at the level we care to know about them. Introduced in 2012, *dropout* randomly sets parts of the output of a layer of nodes to zero when training. The effect is like training thousands of models simultaneously, each independent but also linked. Dropout, when appropriate, has a dramatic impact on network learning. As a prominent computer scientist told me at the time, “If we had had dropout in the 1980s, this would be a different world now.”

Batch normalization adjusts the data moving between layers as it flows through the network. Inputs appear on one side of the network and flow through layers to get to the output.

Schematically, this is usually presented as a left-to-right motion. Normalization is inserted between the layers to change the values to keep them within a meaningful range. Batch normalization was the first learnable normalization technique, meaning it learned what it should do as the network learned. An entire suite of normalization approaches evolved from batch normalization.

The last critical algorithmic innovation enabling the deep learning revolution involves gradient descent, which works with backpropagation to facilitate learning the weights and biases. The idea behind gradient descent is far older than machine learning, but the versions developed in the last decade or so have contributed much to deep learning's success. We'll learn more about this subject in [Chapter 4](#).

Conclusion: The first approaches to training neural networks were primitive and unable to take advantage of their true potential. Algorithmic innovations changed that.

Data

Neural networks require lots of training data. When people ask me how much data is necessary to train a particular model for a specific task, my answer is always the same: all of it. Models learn from data; the more, the better because more data means an improved representation of what the model will encounter when used.

Before the World Wide Web, collecting, labeling, and processing datasets of the magnitude necessary to train a deep neural network proved difficult. This changed in the late 1990s and the early 2000s with the tremendous growth of the web and the explosion of data it represented.

For example, Statista (<https://www.statista.com>) claims that in 2022, 500 hours of new video were uploaded to YouTube *every minute*. It's also estimated that approximately 16 million people were using the web in December 1995, representing 0.4 percent of the world's population. By July 2022, that number had grown to nearly 5.5 billion, or 69 percent. Social media use, e-commerce, and simply moving from place to place while carrying a smartphone are enough to generate staggering amounts of data—all of which is captured and used for AI. Social media is free because we, and the data we generate, are the product.

A phrase I often hear in my work is “we used to be data-starved, but now we’re drowning in data.” Without large datasets and enough labels to go with them, deep learning cannot learn. But, on the other hand, with large datasets, awe-inspiring things can happen.

Conclusion: In machine learning, data is everything.

The main takeaways from this chapter are:

- The symbolic AI versus connectionist feud appeared early and led to decades of symbolic AI dominance.
- Connectionism suffered for a long time because of speed, algorithm, and data issues.
- With the deep learning revolution of 2012, the connectionists have won, for now.
- The direct causes of the deep learning revolution were faster computers, the advent of graphics processing units, improved algorithms, and huge datasets.

With this historical background complete enough for our purposes, let’s return to machine learning, starting with the classical algorithms.

3

CLASSICAL MODELS: OLD-SCHOOL MACHINE LEARNING



Beginning piano students don't start with Liszt's "La Campanella," but "Mary Had a Little Lamb" or "Twinkle, Twinkle, Little Star." The simpler pieces contain the basics of playing the piano, and mastering the basics allows students to progress over time. This principle holds in most areas of study, including artificial intelligence.

To reach our ultimate goal of understanding modern AI, we must begin in the "simpler" world of classical machine learning. What holds for the classical models is generally true for more advanced neural networks. This chapter explores three classical models: nearest neighbors, random forests, and support vector machines. Understanding these will prepare us for the neural networks of [Chapter 4](#).

[Figure 3-1](#) shows the training samples for a made-up dataset with two features (x_0 and x_1) and three classes (circles, squares, and triangles). We saw a similar plot in [Chapter 1](#); see [Figure 1-2](#). As with the iris dataset, every shape in the figure represents a sample from the training set. [Figure 3-1](#) is the tool we'll use to understand the nearest neighbors classical model.

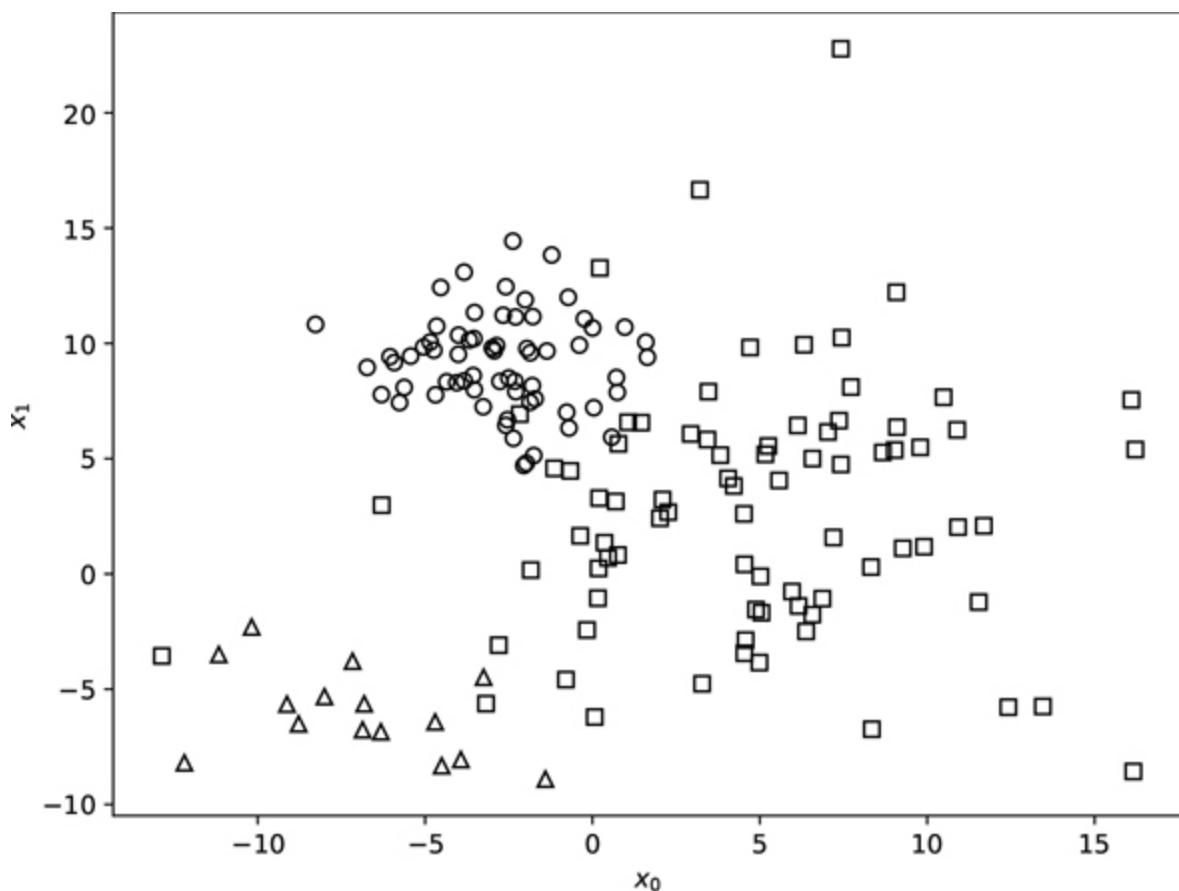


Figure 3-1: A made-up training set with three classes and two features

As mentioned in the previous chapter, [nearest neighbor](#) classifiers are the simplest of models—so simple that there's no model to train; the training data *is* the model. To assign a class label to a new, unknown input, find the training sample closest to the unknown sample and return that sample's label. That's all there is to it. Despite their simplicity, nearest neighbor classifiers are quite effective if the training data represents what the model will encounter in the wild.

As a natural extension to the nearest neighbor model, locate the k training samples nearest the unknown sample. k is often a number like 3, 5, or 7, though it can be any number. This type of model uses a majority voting system, so the assigned class label is the one that's most common among the k training samples. If there's a tie, select the label randomly. For example, if the model is contemplating the 5-nearest neighbors to an unknown sample, and two are class 0 while another two are class 3, then assign the label by choosing randomly between 0 and 3; on average, you'll make the correct choice 50 percent of the time.

Let's use the nearest neighbor concept to classify some unknown inputs. [Figure 3-2](#) shows the training samples again, along with two unknown samples: the diamond and the pentagon. We want to assign these samples to one of the three classes: circle, square, or triangle. The nearest neighbor

approach says to locate the training sample closest to each unknown sample. For the diamond, that's the square to its upper left; for the pentagon, it appears to be the triangle to the upper right. Therefore, a nearest neighbor classifier assigns class square to the diamond and class triangle to the pentagon.

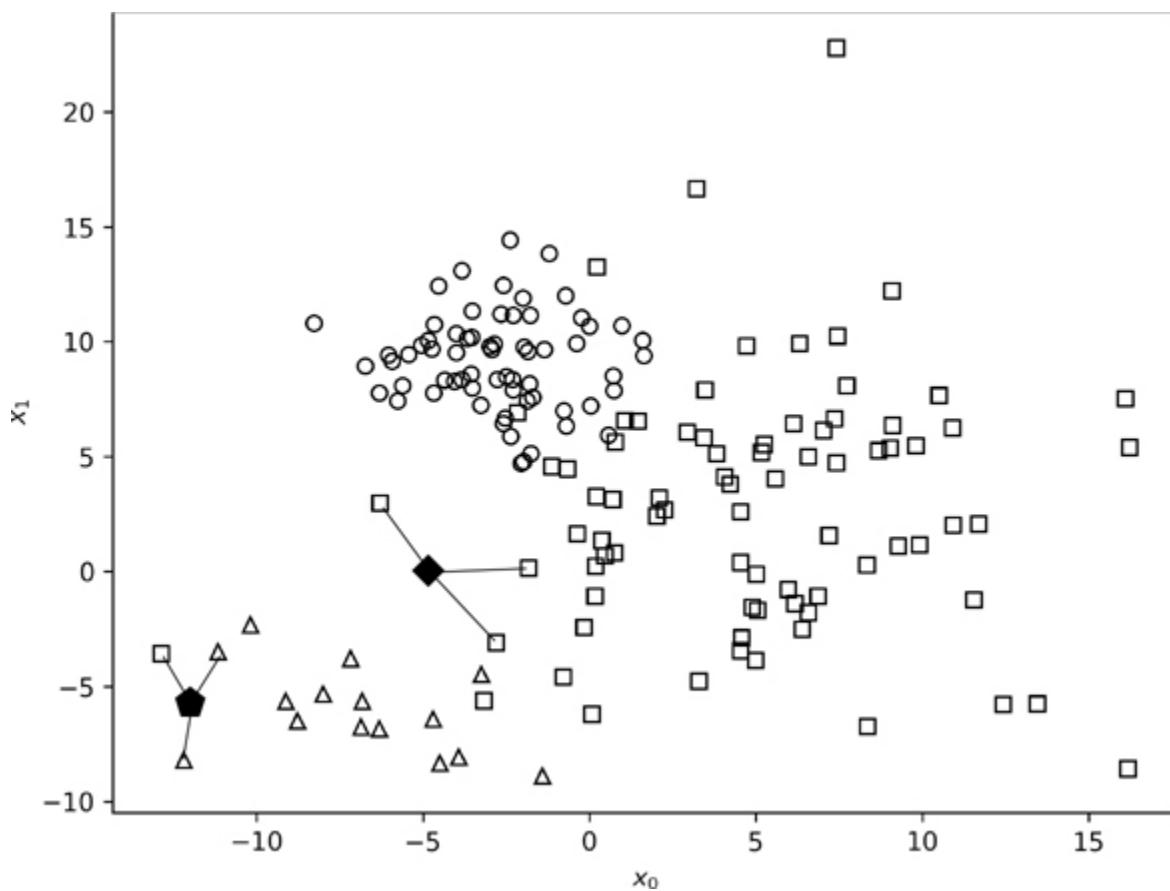


Figure 3-2: Classifying unknown samples

I suspect you've noticed the lines connecting the unknown samples in [Figure 3-2](#) to the three nearest training samples. These are the samples to use if k is 3. In this case, the classifier would again assign class square to the diamond, because all

three of the nearest training samples are squares. For the pentagon, two of the three nearest neighbors are triangles and one is a square, so it would also again assign class triangle to the pentagon.

This example uses two-dimensional feature vectors, x_0 and x_1 , so we can visualize the process. We're not restricted to models with only two features; we can have dozens or even hundreds. The idea of “nearest” (distance) still has mathematical meaning even when there are too many features to graph. Indeed, many mathematical concepts qualify as distance measures, and in practice, nearest neighbor classifiers may use any of the measures depending on the dataset.

For example, let's return to [Chapter 1](#)'s MNIST digits dataset. The samples are small, grayscale images of the digits 0 through 9 that we unravel into vectors of 784 elements. Therefore, each digit sample in the training set is a single point in a 784-dimensional space, just as in the previous example each sample was a point in a 2-dimensional space.

The full MNIST dataset has 60,000 training examples, meaning the training space consists of 60,000 points scattered throughout the 784-dimensional space (not quite, but more on that soon). It also has 10,000 test samples that we can use to

evaluate the nearest neighbor model. I trained 1-nearest neighbor models using all 60,000 training samples, then 6,000 samples, then 600, before ending with a mere 60. Sixty samples in the training set implies about six examples of each digit. I say “about” because I sampled the training set randomly, so there might be eight of one digit and only three of another. In every case, I tested the model using all 10,000 test samples, thereby mimicking using the model in the real world.

[Table 3-1](#) shows the model’s performance as the number of training examples changed.

Table 3-1: Changing the Training Set Size

Training set size	Accuracy (%)
60,000	97
6,000	94
600	86
60	66

Recall that accuracy is the percentage of the test samples that the model classified correctly by assigning the correct digit label, 0 through 9. When using the entire training set the model is correct 97 times out of 100, on average. Even when the training set is made 10 times smaller, the accuracy is still 94 percent. With 600 training examples—about 60 per digit—the accuracy falls to 86 percent. It's only when the training set shrinks to a mere six examples of each digit, on average, that the accuracy falls dramatically to 66 percent.

However, before we're too harsh on our nearest neighbor model, remember that there are 10 digit classes, so random guessing will be correct, on average, about 1 time in 10, for an accuracy of about 10 percent. In this light, even the 60-sample model is six times better than guessing randomly. Let's explore this phenomenon a bit to see if we can gain some insight into why the nearest neighbor model does well with so little training data.

Imagine you're alone in a basketball arena, sitting in the middle of the court. A speck of dust is suspended in the air somewhere in the arena. For convenience, the speck stays fixed in its position. Now imagine 59 more specks of dust inhabiting the air. Those 60 specks of dust are the 60 digit samples in our training

set, and the arena is the three-dimensional world in which the digit image vectors live.

Now imagine a new speck of dust has appeared right in front of your nose. It's a new digit vector you want to classify. The nearest neighbor model calculates the distance between that speck of dust and the 60 specks whose digit labels you know. The closest speck of dust to the new one is below the rim of the basket you're facing, at a distance of 47 feet (14 meters). It's a three, so the model returns a label of 3. Is it reasonable to think that the closest speck represents the proper label for the unknown sample? After all, there are only 60 specks of dust in the whole arena.

We need to consider two competing effects to provide a reasonable answer to this question. First, we should answer “no” because it seems silly to believe that we can represent the giant volume of the arena with 60 specks of dust. There’s too little data in the training set to fill the arena’s space. This observation, known as the *curse of dimensionality*, refers to the fact that as the number of dimensions increases, so too, at a very rapid rate, does the number of samples needed to fill the space. In other words, the number of points increases rapidly, meaning the number of training samples necessary to represent the space increases rapidly—exponentially, to be more precise.

The curse of dimensionality is one of the bane of classical machine learning.

The curse of dimensionality says we should have no hope of properly classifying digits when we have only 60 training samples and 784 dimensions . . . yet our nearest neighbor classifier still works. Not very well, but better than random guessing. Why? The reason has to do with the digits dataset and how similar examples of the different classes are to each other. All examples of fives look like a 5; if they didn't, we wouldn't recognize them as fives. Therefore, while there are 784 dimensions to the space of digits, most digits in a class will land relatively close to that class's other digits. In other words, the specks of dust representing fives are likely clustered or grouped near each other, probably in a thin, tube-like region that snakes its way through the arena. The other digits are likely grouped similarly. Because of this, the nearest sample has a better chance of being from the same digit class than we initially suspected when considering the curse of dimensionality. Based on this observation, we upgrade our "no" answer to a wishy-washy "probably."

We talk about this effect mathematically by saying that the digit data lies on a *manifold* with an effective dimensionality that is well below the 784 dimensions of the vectors representing the

digits. That data often lies on lower-dimensional manifolds is a boon if we can make use of that information. The nearest neighbor model uses the information because the training data is the model. Later in the book, when we discuss convolutional neural networks, we'll understand that such models learn new ways to represent their inputs, which is akin to learning how to represent the lower-dimensional manifold on which the data lives.

Before we get too excited about how well our nearest neighbor classifier performs with the digits dataset, though, let's bring ourselves back to reality by attempting to classify real images. The CIFAR-10 dataset consists of 50,000 small 32×32 -pixel color images from 10 different classes, including a mix of vehicles, like airplanes, cars, and trucks, and animals, like dogs, cats, and birds. Unraveling each of these images creates a vector of 3,072 elements, so we're asking our classifier to separate images in a 3,072-dimensional space. [Table 3-2](#) shows how it fares.

Table 3-2: Classifying CIFAR-10 with Nearest Neighbor

Training set size	Accuracy (%)
50,000	35.4

Training set size	Accuracy (%)
5,000	27.1
500	23.3
50	17.5

As with MNIST, random guessing leads to an accuracy of 10 percent. While our classifier performs better than this with all variations of training set size, its best accuracy is little more than 35 percent—nowhere near the 97 percent achieved with MNIST. Sobering realizations like this led many in the machine learning community to lament that generic image classification might be beyond our grasp. Thankfully, it isn’t, but none of the classical machine learning models do it well.

If we think in terms of manifolds—the idea that data often lives in a lower-dimensional space than the dimensionality of the data itself—then these results aren’t surprising. CIFAR-10 contains real-world photographs, often referred to as natural images. Natural images are far more complex than simple images like MNIST digits, so we should expect them to exist in a higher-dimensional manifold and consequently be harder to

learn to classify. As it happens, there are numerical approaches to estimating the true dimensionality of data. For MNIST, even though the images live in a 784-dimensional space, the data is closer to 11-dimensional. For CIFAR-10, the intrinsic dimensionality is closer to 21 dimensions, so we expect to need far more training data to perform on par with MNIST.

Nearest neighbor models aren't used often these days. Two issues contribute to why. First, while training a nearest neighbor model is effectively instantaneous because there's nothing to train, *using* a nearest neighbor model is slow because we have to calculate the distance between the unknown sample and each of the training set samples. This calculation time grows as the square of the number of samples in the training set. The more training data we have, the better we expect the model to perform, but the slower it runs. Double the size of the training set, and the search time increases by a factor of four.

Decades of study of nearest neighbor classifiers have uncovered all manner of tricks to mitigate the time it takes to find the nearest neighbor, or nearest k neighbors, but the effect remains: increasing the number of training samples increases the time it takes to use the classifier.

The second issue is common to all classical machine learning models, as well as the traditional neural networks we'll discuss in [Chapter 4](#). These models are holistic, meaning they interpret their input vectors as a single entity without parts. This is *not* the right thing to do in many cases. For example, writing a four uses multiple strokes, and there are definite parts that distinguish the four from an eight. Classical machine learning models don't explicitly learn about these parts or where they appear, or that they might appear in multiple locations. Modern convolutional neural networks, however, do learn these things.

In sum, nearest neighbor models are straightforward to understand and trivial to train, but slow to use and unable to explicitly understand structure in their inputs. Let's change gears to contemplate the forest and the trees.

We briefly explored decision trees, comprising a series of yes/no questions asked about an unknown sample, in [Chapter 1](#). You begin at the root node and traverse the tree by answering the node's question. If the answer is "yes," move down one level to the left. If the answer is "no," move down to the right. Continue answering questions until you reach a leaf (a node with no

question), and assign the unknown sample whatever label is in the leaf node.

Decision trees are deterministic; once constructed, they don't change. Therefore, traditional decision tree algorithms return the same decision tree for the same training set. More often than not, the tree doesn't work all that well. If that happens, is there anything we can do? Yes! We can grow a forest of trees.

But if decision trees are deterministic, won't the forest be nothing more than the same tree, over and over, like a mass of clones? It will, if we don't do anything clever along the way. Fortunately, humans are clever. Researchers realized around the year 2000 that introducing randomness produces a forest of unique trees, each with its own strengths and weaknesses, but collectively better than any single tree. A *random forest* is a collection of decision trees, each randomly different from the others. The forest's prediction is a combination of its trees' predictions. Random forests are a manifestation of the wisdom of crowds.

Using randomness to build a classifier seems counterintuitive at first. If on Tuesday we present the model with sample X and it tells us that sample X is a member of class Y, then we don't want it to tell us that it's a member of class Z if we happen to present

the same sample on Saturday. Fortunately, the randomness of a random forest doesn't work that way. Give a trained forest sample X as input, and it always gives us class Y as output, even if it's February 29.

Three steps go into growing a random forest: bagging (also called bootstrapping), random feature selection, and ensembling. Bagging and random feature selection help combat overfitting, a concept mentioned in [Chapter 1](#). Single decision trees are prone to overfitting.

All three steps work together to grow a forest of decision trees whose combined outputs produce a (hopefully) better-performing model. Explainability is the price paid for this gain in power. A single decision tree explains itself by the series of questions and answers that produce its output. With dozens or hundreds of decision trees combining their output, explainability goes out the window, but we can live with that in many cases.

As I've already mentioned several times, the training set is key to conditioning the model. This remains true with random forests. We have as a starting point a training set. As we grow the forest, decision tree by decision tree, we use the existing

training set to create tree-specific training sets unique to the current decision tree. This is where bagging comes in.

Bagging refers to constructing a new dataset from the current dataset by random sampling with replacement. The phrase “with replacement” means we might select a training sample more than once or not at all. This technique is used in statistics to understand a measurement’s bounds. We’ll use the following example dataset of test scores to figure out what that means:

95, 88, 76, 81, 92, 70, 86, 87, 72

One way to assess a class’s performance on the test is to calculate the average score by taking the sum of all the scores divided by the number of scores. The sum is 747, and there are 9 scores, giving us an average of 83.

Collectively, the test scores are a sample from a mythical parent process that generates test scores for the particular test taken. This isn’t a common way to think about test scores, but it’s a machine learning way to think about what a dataset represents. The test scores from another group of students represent another sample from the parent process for this test. If we have many classes’ worth of test scores, we can get an idea about the true average test score, or at least the range over which we

expect to find that average score, with a high degree of confidence.

We could give the test to many different classes to get multiple average scores, one per class, but instead we'll use bagging to create new datasets from the collection of test scores we do have and look at their averages. To do that, we pick values from the collection of test scores at random, not caring if we've already picked this particular score or never pick that one. Here are six such bootstrapped datasets:

1. 86, 87, 87, 76, 81, 81, 88, 70, 95
2. 87, 92, 76, 87, 87, 76, 87, 92, 92
3. 95, 70, 87, 92, 70, 92, 72, 70, 72
4. 88, 86, 87, 70, 81, 72, 86, 95, 70
5. 86, 86, 92, 86, 87, 86, 70, 81, 87
6. 76, 88, 88, 88, 88, 72, 86, 95, 70

The respective averages of each are 83.4, 86.2, 80.0, 81.7, 84.6, and 83.4 percent. The lowest is 80.0 percent, and the highest is 86.2 percent. This gives us some reason to believe that a large number of samples will produce an average more or less in that range.

This is how a statistician might use bagging. For us, the critical part is the six new datasets bootstrapped from the original dataset. When growing a random forest, every time we need a new decision tree, we'll first use bagging to produce a new dataset, then train the decision tree using that dataset, not the original. Notice that many of the six datasets have repeated values. For example, dataset 1 used both 81 and 87 twice, but never 72. This randomization of the given dataset helps create decision trees that behave differently from one another yet are aligned with what the original dataset represents.

The second trick a random forest uses is to train the decision tree on a randomly selected set of features. Let's use the toy dataset in [Table 3-3](#) to understand what that means. As always, each row is a feature vector, a sample for which we know the proper class label. The columns are the values of that feature for each sample.

Table 3-3: A Toy Dataset

#	x_0	x_1	x_2	x_3	x_4	x_5
1	0.52	0.95	0.81	0.78	0.97	0.36

#	x_0	x_1	x_2	x_3	x_4	x_5
2	0.89	0.37	0.66	0.55	0.75	0.45
3	0.49	0.98	0.49	0.39	0.42	0.24
4	0.43	0.51	0.90	0.78	0.19	0.22
5	0.51	0.16	0.11	0.48	0.34	0.54
6	0.48	0.99	0.62	0.58	0.72	0.42
7	0.80	0.84	0.72	0.26	0.93	0.23
8	0.50	0.70	0.13	0.35	0.96	0.82
9	0.70	0.54	0.62	0.72	0.14	0.53

What does this dataset represent? I have no idea; it's made up. My cheeky answer is a good reminder that machine learning models don't understand what their datasets represent. They process numbers without context. Is it a pixel value? The number of square feet in a house? The crime rate of a county?

per 100,000 people? It doesn't matter to the machine learning model—it's all just numbers.

This toy dataset consists of nine feature vectors, each with six features, x_0 through x_5 . The forest's decision trees use a randomly selected subset of the six features. For example, say we randomly keep features x_0 , x_4 , and x_5 . [Table 3-4](#) shows the dataset now used to train the decision tree.

Table 3-4: A Random Collection of Features

#	x_0	x_4	x_5
1	0.52	0.97	0.36
2	0.89	0.75	0.45
3	0.49	0.42	0.24
4	0.43	0.19	0.22
5	0.51	0.34	0.54
6	0.48	0.72	0.42

#	x_0	x_4	x_5
7	0.80	0.93	0.23
8	0.50	0.96	0.82
9	0.70	0.14	0.53

Each decision tree in the forest has been trained on a bootstrapped version of the dataset using only a subset of the available features. We've used randomness twice to grow a forest of trees that are all subtly different from each other, in both what data they're trained on and which features they pay attention to.

Now that we have a forest, how do we use it? Enter the last of the three pieces: ensembling. Musically, an ensemble is a collection of musicians playing diverse instruments. The random forest is also an ensemble, with each decision tree a different musician playing a different instrument.

A musical ensemble produces a single output, the music, by combining the notes played by each instrument. Likewise, a random forest produces a single output, a class label, by

combining the labels produced by each decision tree, typically by voting like a k -nearest neighbors classifier. We assign the winning label to the input.

For example, if we want to use the random forest to classify sample X, and there are 100 trees in the random forest (already trained), we give each tree sample X. The trees know which subsets of sample X's features to use to arrive at a leaf with a label. We now have 100 possible class labels, the output from the forest's 100 decision trees. If 78 of the trees assign sample X to class Y, the random forest proclaims sample X to be an instance of class Y.

The random assignment of features to trees, combined with bootstrapped datasets and ensemble voting, gives a random forest its power. Ensembling is an intuitively attractive idea that isn't restricted to random forests. Nothing stops us from training multiple model types on the same dataset and then combining their predictions in some way to arrive at a joint conclusion about an input sample. Each of the models will have its own strengths and weaknesses. When combined, the strengths tend to enhance the output quality, making the sum greater than the parts.

We have one more classical machine learning model to investigate, the support vector machine (SVM). After that, we'll pit the models against each other to gain intuition about how they behave and provide a baseline against which we can compare the performance of neural networks.

To understand support vector machines is to understand four concepts: margins, support vectors, optimization, and kernels. The math is a bit hairy, even for math people, but we'll set that aside and focus instead on gaining a conceptual understanding.

Support vector machines are best understood visually, so we'll begin with the example toy dataset in [Figure 3-3](#). This is a two-class dataset (circles and squares) with two-dimensional feature vectors, features x_0 and x_1 .

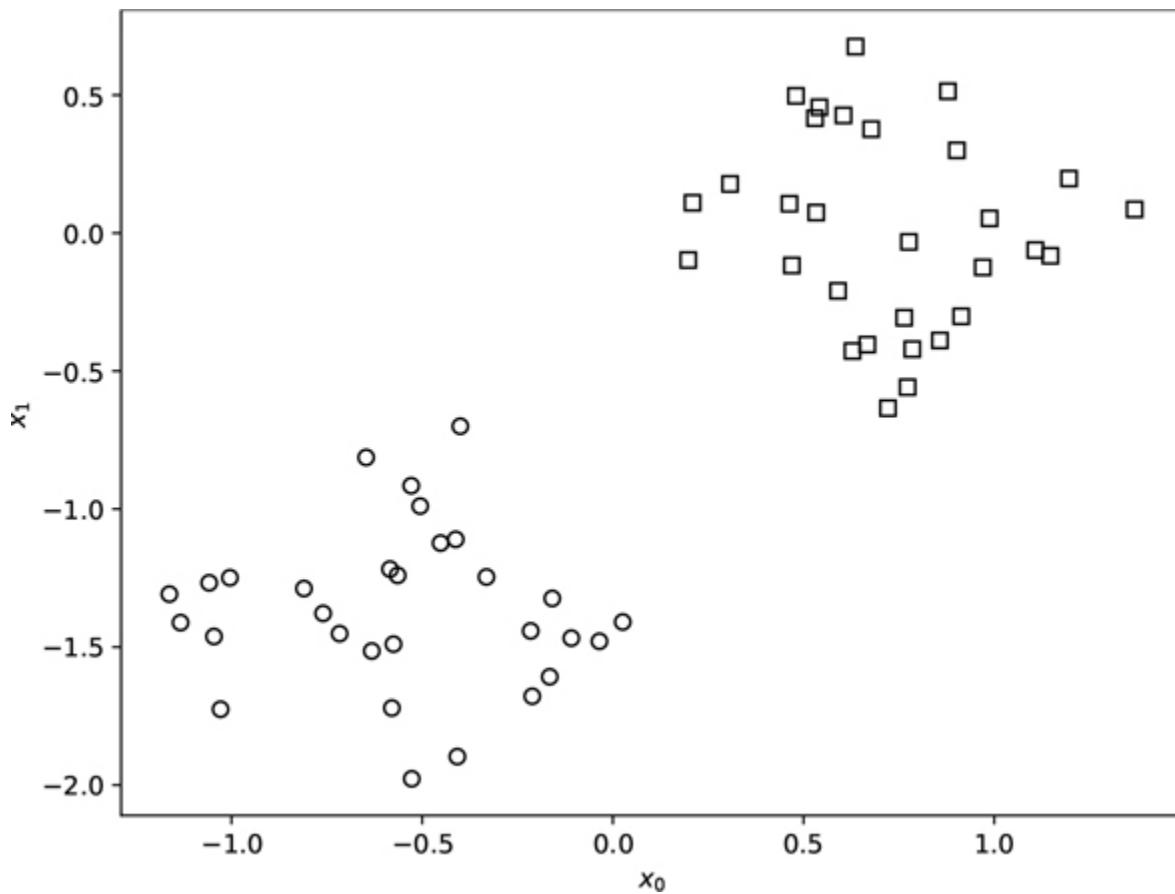


Figure 3-3: A two-class toy dataset with two features, x_0 and x_1

A classifier for this dataset is straightforward to construct because a line easily separates the dataset by class, with all the squares above it and to the right and all the circles below and to the left. But where should it go? There are an infinite number of lines that we might use. For example, we might pass the line just below all the squares. That line separates the classes, but if we encounter a sample from class square that lands just below the line when we use the classifier, we'll make a mistake and assign the sample to class circle because it's below the line we declared separates the classes. Similarly, if we place the line just

above all the circles, we might call a new sample that's actually a circle a square because it landed slightly above that line.

Given what we know based on the training data, we should place the separating line as far from each group as possible. Here's where the concept of a margin comes into play. SVMs seek to maximize the margin between the two groups, meaning finding the place with the widest separation between classes. When they have the maximum margin, they place the boundary, here a line, in the middle of the margin because that's the most sensible thing to do based on the information contained in the training data.

[Figure 3-4](#) shows the training data with three additional lines. The dashed lines define the margin, and the heavy continuous line marks the boundary placed by the SVM to maximize the distance between classes. This is the best position for the line to minimize labeling errors between the two classes. This, in a nutshell, is all an SVM does.

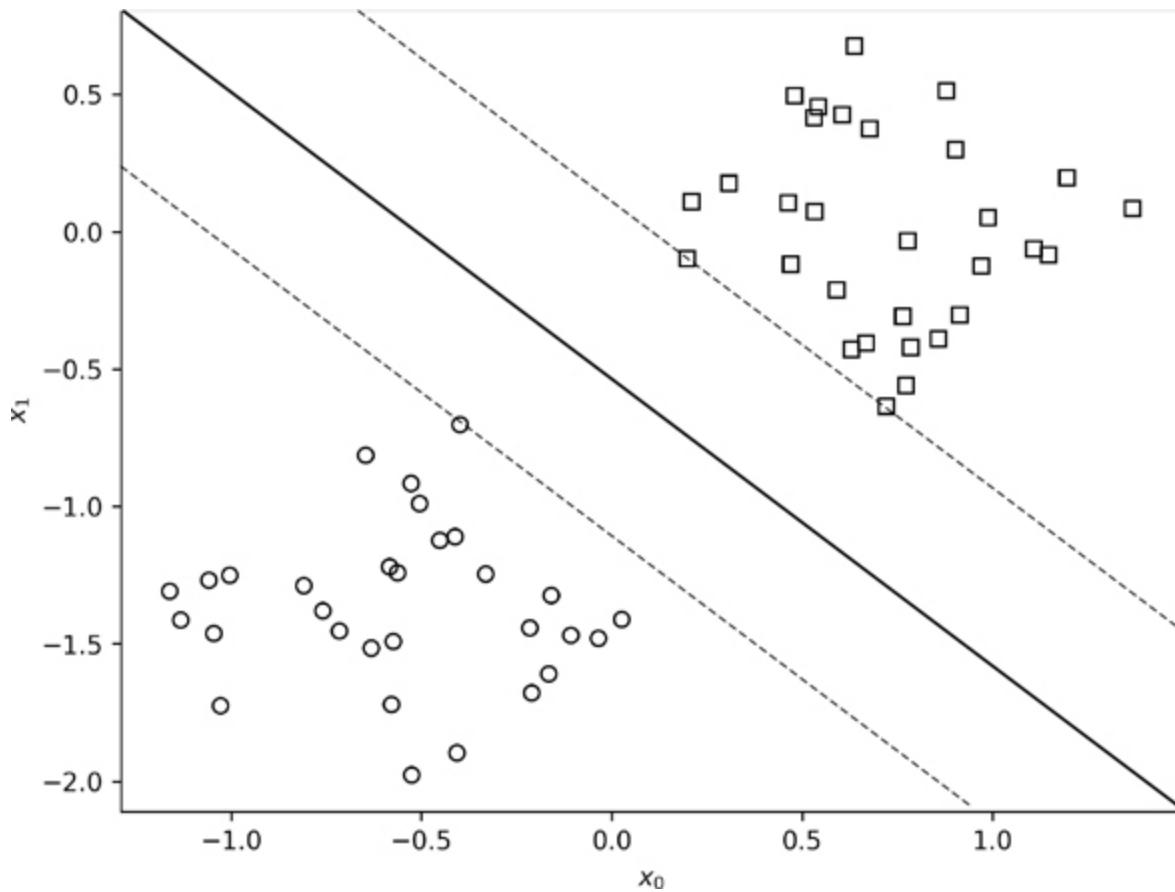


Figure 3-4: The maximal margin separating line (heavy) and maximum margins (dashed)

The three other parts of an SVM—support vectors, optimization, and kernels—are used to find the margins and the separating line. In [Figure 3-4](#), notice that the dashed lines pass through some of the data points. These points are the support vectors that the algorithm finds to define the margin. Where do those support vectors come from? Recall that the figure’s points represent specific feature vectors in the training set. Support vectors are members of the training set found via an optimization algorithm. Optimization involves finding the best

of something according to some criteria. The optimization algorithm used by an SVM locates the support vectors that define the maximum margin and, ultimately, the separating line. In [Chapter 1](#), we used an optimization algorithm when we discussed fitting data to a curve, and we'll use one again when training neural networks.

We're almost there; we have only one SVM concept remaining: kernels. As opposed to the popcorn variety or the kernel at the heart of your computer's operating system, mathematical kernels relate two things—here, two feature vectors. The example in [Figure 3-4](#) uses a linear kernel, meaning it uses the training data feature vectors as they are. Support vector machines admit many kinds of kernels to relate two feature vectors, but the linear kernel is the most common. Another kind, called a Gaussian kernel (or, even more verbose and impressive, a radial basis function kernel), often helps in situations where the linear kernel fails because the feature vectors are in a different kind of relationship to each other.

The kernel transforms the feature vectors into a different representation, an idea central to what convolutional neural networks do. One of the issues that made classical machine learning stumble for so long is that the data supplied to the models was too complex in its raw form for the model to make

meaningful distinctions between classes. This is related to the idea of manifolds and intrinsic dimensionality introduced in our discussion of nearest neighbors.

Classical machine learning practitioners spent considerable effort trying to minimize the number of features needed by a model, paring the features down to the minimal set necessary for the model to distinguish between classes. This approach was termed *feature selection* or *dimensionality reduction*, depending on the algorithm used. Similarly, especially with SVMs, they used kernels to map the given feature vectors to a new representation, making separating classes easier. These approaches were human-led endeavors; we selected the features or the kernels in the hopes that they'd make the problem more manageable. But, as we'll learn, modern deep learning lets the data speak for itself when learning new representations of the information the data contains.

In practice, training a support vector machine means locating good values for the parameters related to the kernel used. If the kernel is linear, as in the previous example, there's only one value to find, universally called C . It's a number, like 1 or 10, affecting how well the support vector machine performs. If using the Gaussian kernel, we have C and another parameter, known by the Greek letter γ (gamma). The art of training an

SVM involves finding the magic values that work best for the dataset at hand.

The magic values used by a model are its *hyperparameters*.

Neural networks have many hyperparameters; even more than SVMs. However, my experience has taught me that it's often easier to tune a neural network—especially a modern deep neural network—than a support vector machine. I freely confess my bias here; others might disagree.

Support vector machines are mathematically elegant, and practitioners use that elegance to tweak the hyperparameters and the kernel used, along with a suite of old-school data preparation approaches, to construct a well-performing model that works well on data in the wild. Every step of this process relies on the intuition and experience of the human building the model. If they're knowledgeable and experienced, they'll likely succeed if the dataset is amenable to such a model, but success isn't assured. On the other hand, deep neural networks are big, kind of clunky, and live or die by the raw data they're fed. That said, by coming to the problem with a minimal set of assumptions, neural networks can generalize over elements of the dataset that humans cannot fathom, which I think is often why modern neural networks can do what was previously believed to be next to impossible.

SVMs are binary classifiers: they distinguish between two classes, as in the dataset in [Figure 3-3](#). But sometimes we need to distinguish between more than two classes. How can we do that with an SVM?

We have two options for generalizing SVMs to multiclass problems. Assume we have 10 classes in the dataset. The first generalization approach trains 10 SVMs, the first of which attempts to separate class 0 from the other nine classes. The second likewise attempts to separate class 1 from the remaining nine, and so on, giving us a collection of models, each trying to separate one class from all the others. To classify an unknown sample, we give the sample to each SVM and return the class label of the model that produced the largest decision function value—the [*metric*](#), or measurement, the SVM uses to decide its confidence in its output. This option is known as [*one-versus-rest*](#) or [*one-versus-all*](#). It trains as many SVMs as there are classes.

The other option is [*one-versus-one*](#), which trains a separate SVM for each possible pair of classes. The unknown sample is given to each model, and the class label that shows up most often is assigned to it. One-versus-one isn't practical if the number of classes becomes too large. For example, for the 10 classes in CIFAR-10, we'd need 45 different SVM machines. And if we tried this approach with the 1,000 classes in the ImageNet dataset,

we'd be waiting a long time for the 499,500 different SVMs to train.

Support vector machines were well suited to the computing power commonly available in the 1990s and early 2000s, which is why they held neural networks at bay for so long. However, with the advent of deep learning, there's little reason to resort to an SVM (in my opinion).

Let's test the three classical models explored in this chapter using an open source dataset consisting of dinosaur footprint outlines that comes from the 2022 paper "A Machine Learning Approach for the Discrimination of Theropod and Ornithischian Dinosaur Tracks" by Jens N. Lallensack, Anthony Romilio, and Peter L. Falkingham. The footprint images were released under the Creative Commons CC BY 4.0 license, which allows reuse with attribution.

[Figure 3-5](#) contains samples from the dataset. Theropod footprints (think *T. rex*) are in the top row, and ornithischian footprints (think duckbilled dinos like hadrosaurs) are at the bottom. The images used by the models were inverted to be white on a black background, rescaled to 40×40 pixels, and

unraveled to become 1,600-dimensional vectors. The dataset is small by modern standards, with 1,336 training samples and 335 test samples.



Figure 3-5: Theropod (top) and ornithischian (bottom) footprints

I trained the following models:

- Nearest neighbor ($k = 1, 3, 7$)
- A random forest with 300 trees
- A linear support vector machine
- A radial basis function support vector machine

After training, I tested the models with the held-out test set. I also timed how long it took to train each model and to test each model after training. Using a model after training is *inference*, meaning I tracked the inference time on the test set.

NOTE

This isn't a programming book, but if you're familiar with programming, especially Python, feel free to contact me at rkneuselbooks@gmail.com and I'll send you the dataset and code.

Table 3-5 shows the results. Evaluating how well a model works is, as you might expect, a critical component of the machine learning process.

Table 3-5: Classifying Dinosaur Footprints

Model	ACC	MCC	Train	Test
RF300	83.3	0.65	1.5823	0.0399
RBF SVM	82.4	0.64	0.9296	0.2579
7-NN	80.0	0.58	0.0004	0.0412
3-NN	77.6	0.54	0.0005	0.0437
1-NN	76.1	0.50	0.0004	0.0395
Linear SVM	70.7	0.41	2.8165	0.0007

The first column on the left identifies the model: from top to bottom, random forest, radial basis function support vector machine, nearest neighbors (with 7, 3, and 1 neighbor), and linear support vector machine.

The ACC and MCC columns are metrics calculated from the confusion matrix, the single most crucial part of the machine learning practitioner's toolbox when evaluating a model (see [Chapter 1](#)). For binary classifiers like the ones we have here, the confusion matrix counts the number of times a theropod test sample was correctly identified, the same for ornithischian test samples, and the number of times one was confused for the other.

Visually, the confusion matrix for a binary model looks like this:

		Ornithischian	Theropod
Ornithischian	TN	FP	
Theropod	FN		TP

The rows are the actual class label from the held-out test set. The columns are the labels assigned by the models. The cells are the counts of the number of times each combination of actual

label and model-assigned label happened. The letters are the standard way to refer to what the numbers in the cells mean: TN is *true negative*, TP is *true positive*, FP is *false positive*, and FN is *false negative*. For the dinosaur footprint models, theropod is class 1, the “positive” class, making ornithischian class 0, or the “negative” class.

The number of times the model called an ornithischian footprint “ornithischian” is the TN count. Similarly, the TP count represents the number of times the model was right about a theropod footprint. The goal is to get TN and TP as high as possible while making FP and FN, the mistakes, as low as possible.

In [Table 3-5](#), ACC refers to the accuracy: how many times was the classifier’s assigned label correct? While accuracy is the most natural metric to consider, it isn’t always the best, especially if the number of examples per class isn’t nearly equal. The random forest performed the best in terms of accuracy, correctly labeling more than 83 out of every 100 test images. The linear SVM was the worst; it was right only about 71 times out of 100. Random guessing would be correct about 50 percent of the time because we have two classes, though, so even the linear SVM was learning from the footprint images. We define the accuracy in terms of the cells of the confusion

matrix by adding TP and TN and dividing that sum by the sum of all four cells.

The MCC column, which stands for *Matthews correlation coefficient*, introduces a new metric. It's a different combination of the four numbers in the confusion matrix. MCC is my favorite metric for classifiers, and it is increasingly understood to be the best single-number measure of how well a model performs. (These metrics apply to more advanced deep learning models as well.) [Table 3-5](#) is sorted by MCC, which, for this example, also happens to sort by ACC. For a binary model, the lowest possible MCC is -1 , and the highest is 1 . Random guessing gives an MCC of 0 . An MCC of 1 means the model makes no mistakes. An MCC of -1 , which never actually happens in practice, means that the model is perfectly wrong: in our case, it would label all theropod tracks ornithischian and all ornithischian tracks theropod. If you have a perfectly wrong classifier, swap the output labels to make it perfectly right.

The Train and Test columns list times in seconds. The Train column tells us how long it took to train the model before using it. The nearest neighbor models take virtually no time, a mere fraction of a millisecond, because there's nothing to train. Recall that a nearest neighbor model is the training set itself; there is no model to condition to approximate the data in some way.

The slowest model was the linear SVM. Curiously, the more complex radial basis function model trained in roughly one-third the time (a difference that can be attributed to how such models are implemented in code). The next slowest model to train was the random forest. This makes sense because there were 300 decision trees in the forest, and each of them had to be trained independently.

The inference time, in the Test column, was roughly the same between the nearest neighbor and random forest models. The SVM models were respectively slow (RBF) and very fast (linear), again reflecting differences in the implementation. Notice that the nearest neighbor models take longer to use than to train. This is the reverse of the usual scenario, especially for neural networks, as we'll see later in the book. Typically, training is slow but needs to be done only once, while inference is fast. For nearest neighbor models, the larger the training set, the slower the inference time—a significant strike against them.

There are two main things to take away from this exercise: a general understanding of the performance of the classical models, which we'll use as a baseline against which to compare a neural network in [Chapter 4](#), and that even classical models can do well on this particular dataset. Their performance was on par with that of human experts (meaning paleontologists),

who also labeled the dinosaur footprint outlines. According to the original paper by Lallensack et al. from which the dinosaur dataset was taken, the human experts were correct only 57 percent of the time. They were also allowed to label tracks as “ambiguous,” a luxury the models don’t have; the models always make a class assignment, with no “I don’t know” option. We can coerce some model types into making such statements, but the classical models of this chapter are not well suited to that.

Are the classical models symbolic AI or connectionism? Are they AI at all? Do they learn, or are they merely mathematical tricks? My answers to these questions follow.

In [Chapter 1](#), I characterized the relationship between AI, machine learning, and deep learning as a series of nested concepts, with deep learning a form of machine learning and machine learning a form of AI (see [Figure 1-1](#)). This is the proper way to describe the relationship for most people, and it fits with [Chapter 2](#)’s history. From this perspective, the classical models of this chapter are a form of AI.

But are the classical models symbolic AI or connectionist AI? I say neither. They are not symbolic AI because they don't manipulate logical rules or statements, and they're not connectionist because they don't employ a network of simple units that learn their proper association as they work with the data. Instead, I consider these models to be a fancy form of curve fitting—the output of an algorithm employing an optimization process to produce a function that best characterizes the training data, and, hopefully, the data encountered by the model in the wild.

For a support vector machine, the function is the structure of the model in terms of the support vectors it locates during its optimization process. A decision tree's function is generated by a specific algorithm designed to repeatedly split the training data into smaller and smaller groups until a leaf is created that (usually) contains only examples from a single class. Random forests are merely collections of such functions working in parallel.

Tree classifiers are almost a form of genetic programming. [Genetic programming](#) creates computer code by simulating evolution via natural selection, where improved fitness corresponds to “is a better solution to the problem.” Indeed, genetic programming is a kind of [evolutionary algorithm](#), and

evolutionary algorithms, along with *swarm intelligence* algorithms, implement robust, generic optimization. Some people consider evolutionary algorithms and swarm intelligence to be AI. I don't, though I frequently use them in my work. Swarms don't learn; they search a space representing possible solutions to a problem.

Nearest neighbor models are even simpler; there is no function to create. If we have *all* the possible data generated by some parent process—that is, the thing creating the feature vectors that we're trying to model—then we don't need a model. To assign a class label to a feature vector, we simply look it up in the feature vector “phone book” and return the label we find there. Since we have all possible feature vectors with labels, there's nothing to approximate, and any feature vector encountered in the wild will necessarily be in the book.

Barring access to all possible feature vectors for the problem at hand, a nearest neighbor model uses the closest feature vector in the incomplete phone book represented by the training data.

As an example, suppose we live in a town of 3,000 people, and all of them are in the phone book. (Are there still such things as phone books? If not, pretend.)

If we want to find Nosmo King's phone number, we look in the book under "King" and scan until we hit "Nosmo," and we have it. Suppose, however, that we don't have a complete listing of all 3,000 people, but 300 selected at random. We still want to know Nosmo King's phone number (class label), but it's not in the phone book. However, there is a Burg R. King. There's a good chance Burg is related to Nosmo because of the shared last name, so we return Burg's phone number as Nosmo's. Clearly, the more complete the phone book, the better the chance we'll find our desired name or someone in that person's household. That's essentially all that a nearest neighbor model does.

To recap, support vector machines, decision trees, and random forests use data to generate functions according to a carefully crafted algorithm designed by a human. That is neither symbolic AI nor connectionism to me, but curve fitting or, perhaps more accurately, optimization. Nearest neighbor models are even worse; in their case, there's no function at all.

This doesn't mean that AI is bogus, but it does mean that what practitioners have in mind when they talk about AI is likely different from what the general public considers "artificial intelligence."

However, all is not lost. There is a machine learning model worthy of the connectionist label: the neural network. It's at the heart of the AI revolution, and it's capable of actually learning from data. So, let's put classical models and symbolic AI aside and devote our attention to neural networks.

KEY TERMS

bagging, curse of dimensionality, evolutionary algorithm, false negative, false positive, genetic programming, hyperparameters, inference, manifold, metric, nearest neighbor, one-versus-one, one-versus-rest, random forest, support vector machine, swarm intelligence, true negative, true positive

4

NEURAL NETWORKS: BRAIN-LIKE AI



Connectionism seeks to provide a substrate from which intelligence might emerge. Today, connectionism means neural networks, with *neural* being a nod to biological neurons. Despite the name, however, the relationship between the two is superficial. Biological neurons and artificial neurons may possess a similar configuration, but they operate in an entirely different manner.

Biological neurons accept input on their dendrites, and when a sufficient number of inputs are active they “fire” to produce a short-lived voltage spike on their axons. In other words, biological neurons are off until they’re on. Some 800 million years of animal evolution have made the process considerably more complex, but that’s the essence.

The artificial neurons of a neural network likewise possess inputs and outputs, but instead of firing, the neurons are mathematical functions with continuous behavior. Some models spike like biological neurons, but we ignore them in this book. The neural networks powering the AI revolution operate continuously.

Think of a biological neuron like a light switch. It's off until there is a reason (sufficient input) to turn it on. The biological neuron doesn't turn on and stay on but flashes on and off, like flicking the switch. An artificial neuron is akin to a light with a dimmer switch. Turn the switch a tiny amount to produce a small amount of light; turn the switch further, and the light's brightness changes proportionally. This analogy isn't accurate in all cases, but it conveys the essential notion that artificial neurons are not all or nothing. Instead, they produce output in proportion to their input according to some function. The fog will lift as we work through the chapter, so don't worry if this makes little sense at present.

[Figure 4-1](#) is the most critical figure in the book. It's also one of the simplest, as is to be expected if the connectionist approach is on the right track. If we understand what [Figure 4-1](#)

represents and how it operates, we have the core understanding necessary to make sense of modern AI.

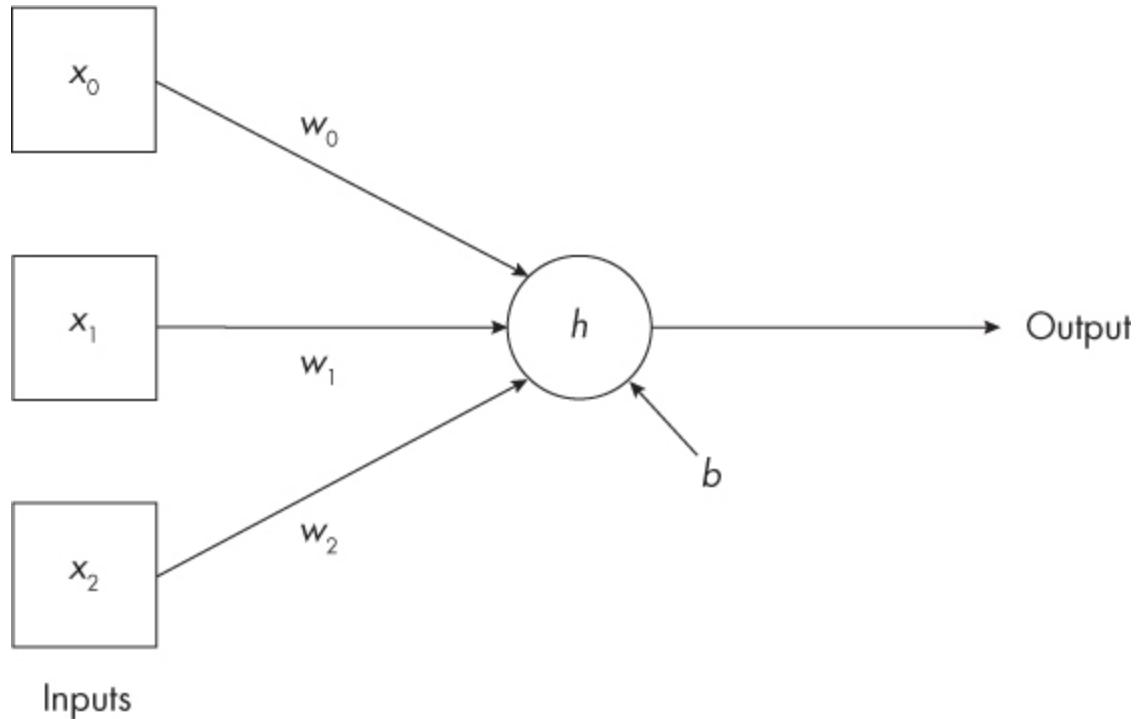


Figure 4-1: The humble (artificial) neuron

[Figure 4-1](#) contains three squares, a circle, five arrows, and labels like “ x_0 ” and “Output.” Let’s examine each in turn, beginning with the squares on the left.

Standard practice presents neural networks with the inputs on the left and data flow to the right. In [Figure 4-1](#), the three squares labeled x_0 , x_1 , and x_2 are the inputs to the neuron. They are the three features of a feature vector, what we want the neuron to process to give us an output leading to a class label.

The circle is labeled h , a standard notation for the *activation function*. The activation function's job is to accept input to the neuron and produce an output value, the arrow heading off to the right in [Figure 4-1](#).

The three input squares are connected to the circle (the *node*) by arrows, one from each input square. The arrows' labels— w_0 , w_1 , and w_2 —are the *weights*. Every input to the neuron has an associated weight. The lone b linked to the circle by an arrow is the *bias*. It's a number, as are the weights, the input x s, and the output. For this neuron, three numbers come in, and one number goes out.

The neuron operates like this:

1. Multiply every input value, x_0 , x_1 , and x_2 , by its associated weight, w_0 , w_1 , and w_2 .
2. Add all the products from step 1 together along with the bias value, b . This produces a single number.
3. Give the single number to h , the activation function, to produce the output, also a single number.

That's all a neuron does: it multiplies its inputs by the weights, sums the products, adds the bias value, and passes that total to the activation function to produce the output.

Virtually all the fantastic accomplishments of modern AI are due to this primitive construct. String enough of these together in the correct configuration, and you have a model that can learn to identify dog breeds, drive a car, or translate from French to English. Well, you do if you have the magic weight and bias values, which training gives us. These values are so important to neural networks that one company has adopted “Weights & Biases” as its name; see <https://www.wandb.ai>.

We have choices for the activation function, but in modern networks it's most often the rectified linear unit (ReLU) mentioned in [Chapter 2](#). The ReLU is a question: is the input (the sum of the inputs multiplied by the weights plus the bias) less than zero? If so, the output is zero; otherwise, it's whatever the input is.

Can something as straightforward as a lone neuron be useful? It can. As an experiment, I trained the neuron in [Figure 4-1](#) using three features from the iris flower dataset from [Chapter 1](#) as input. Recall, this dataset contains measurements of the parts of three different species of iris. After training, I tested the neuron with an unused test set that had 30 feature vectors. The neuron correctly classified 28, for an accuracy of 93 percent.

I trained the neuron by searching for a set of three weights and a bias value producing an output that, when rounded to the nearest whole number, matched the class label for an iris flower—either 0, 1, or 2. This is not the standard way to train a neural network, but it works for something as modest as a single neuron. We'll discuss standard network training later in the chapter.

A single neuron can learn, but complex inputs baffle it. Complex inputs imply we need a more complex model. Let's give our single neuron some friends.

Convention arranges neurons in layers, with the outputs from the previous layer the inputs to the following layer. Consider [Figure 4-2](#), which shows networks with two, three, and eight nodes in the layer after the input. Arranging the network in layers simplifies the implementation in code and facilitates the standard training procedure. That said, there is no requirement to use layers if an alternative way to train the model can be found.

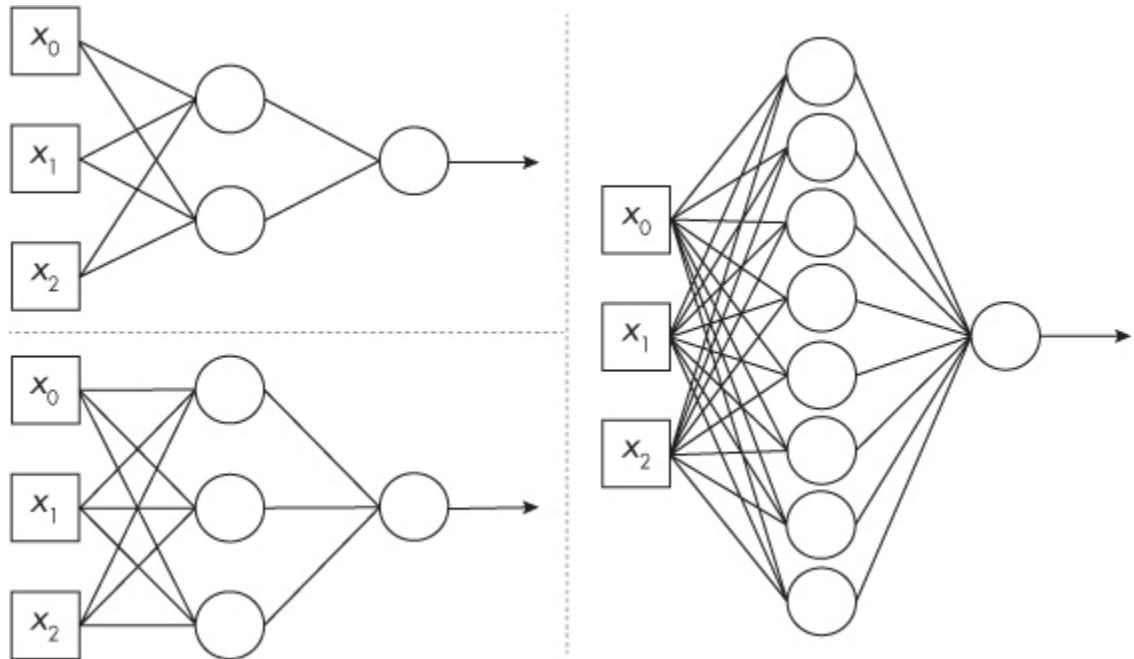


Figure 4-2: Two-, three-, and eight-node networks

Let's begin with the two-node network at the upper left. The three inputs (squares) are there, but this time there are two circles in the middle layer and a single circle on the right. The inputs are fully connected to the two nodes in the middle layer, meaning a line connects each input square to each middle layer node. The middle layer outputs are connected to a single node on the far right, from which the network's output comes.

The middle layers of a neural network between the input on the left and the output on the right are known as hidden layers. For example, the networks of [Figure 4-2](#) each have one hidden layer with 2, 3, and 8 nodes, respectively.

A network with this configuration is suitable for a binary classification task, class 0 versus class 1, where the output is a single number representing the model's belief that the input is a member of class 1. Therefore, the rightmost node uses a different activation function known as a *sigmoid* (also called a logistic). The sigmoid produces an output between 0 and 1. This is also the range used to represent a probability, so many people refer to the output of a node with a sigmoid activation function as a probability. This is not generally accurate, but we can live with the sloppiness. The nodes of the hidden layer all use ReLU activation functions.

How many weights and biases must we learn to implement the two-node network in [Figure 4-2](#)? We need one weight for each line (except the output arrow) and one bias value for each node. Therefore, we need eight weights and three bias values. For the model at the lower left, we need 12 weights and 4 biases. Finally, for the 8-node model, we need to learn 32 weights and 9 bias values. As the number of nodes in a layer increases, the number of weights increases even faster. This fact alone restrained neural networks for years, as potentially useful models were too big for a single computer's memory. Of course, model size is relative. OpenAI's GPT-3 has over 175 billion weights, and while they aren't talking about how large GPT-4 is, rumor puts it at 1.7 *trillion* weights.

We need a two-class dataset to explore the models in [Figure 4-2](#). The dataset we'll use is a classic one that attempts to distinguish between two cultivars of grapes used to make wine in a particular region of Italy. Unfortunately, the wines represented by the dataset are, it seems, no longer known. (That's how old the dataset is.) However, we know that models don't care about the labels—they use numbers—so we'll use 0 and 1 as the labels.

We need three features, x_0 , x_1 , and x_2 . The features we'll use are alcohol content in percent, malic acid, and total phenols. The goal is to train the models in [Figure 4-2](#) to see how well each performs when identifying an unknown wine given measurements of the three features.

I trained the two-neuron model using a training set of 104 samples and a test set of 26 samples. This means I used 104 triplets of measured alcohol content, malic acid level, and total phenols, knowing the proper output label, class 0 or class 1. The training set conditioned the two-neuron model to give values to all eight weights and three biases. I promise we will discuss how training works, but for now, assume it happens so we can explore how neural networks behave. The trained model achieved an accuracy on the test set of 81 percent, meaning it

was right better than 8 times out of 10. That's not too bad for such a small model and training set.

[Figure 4-3](#) presents the trained two-neuron model. I added the weights to the links and the biases to the nodes so you can see them. I think it's worth looking at the numbers at least once, and it's best to do that with a simple model.

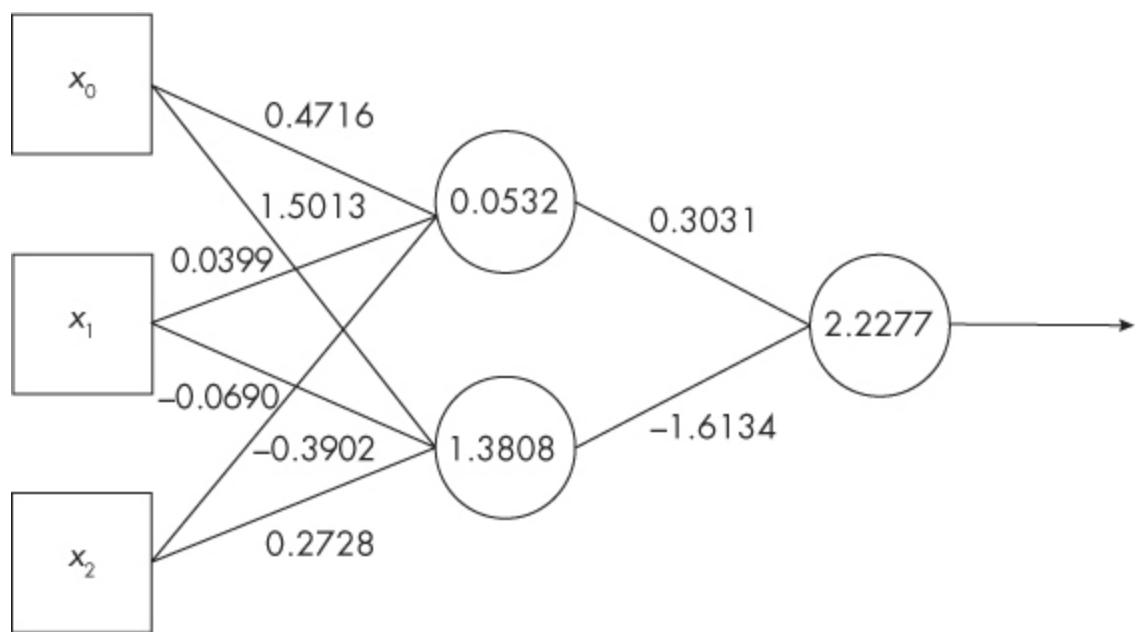


Figure 4-3: The two-neuron model trained on the wine dataset

Let's use the model with two test samples to understand the process. The two test samples consist of three numbers each, the values of the features, (x_0, x_1, x_2):

Sample 1 $(-0.7359, 0.9795, -0.1333)$

Sample 2 (0.0967, -1.2138, -1.0500)

You may have a question at this point. I said the features were alcohol content in percent, malic acid level, and total phenols. While I have no idea what the units are for measuring malic acid or total phenols, a percent is a percent, so why is x_0 for the first sample a small negative number? We can't have a negative percentage of alcohol.

The answer has to do with *preprocessing*. Raw data, like the percent alcohol, is seldom used with machine learning models as is. Instead, each feature is adjusted by subtracting the average value of the feature over the training set and dividing that result by a measure of how scattered the data is around the average value (the standard deviation). The original alcohol content was 12.29 percent, a reasonable value for wine, but after scaling, it became -0.7359 .

Let's classify sample 1 using the learned weights and biases in [Figure 4-3](#). The input to the top neuron is each feature multiplied by the weight on the line connecting that feature to the neuron, then summed with the bias value. The first feature gives us 0.4716×-0.7359 , the second 0.0399×0.9795 , and the third -0.3902×-0.1333 , with bias value 0.0532 . Adding all of these together gives -0.2028 . This is the number passed to the

activation function, a ReLU. Since it is negative, the ReLU returns 0, meaning the output from the top node is 0. Repeating the calculation for the bottom node gives 0.1720 as the input to the ReLU. That's a positive number, so the ReLU returns 0.1720 as the output.

The outputs of the two nodes in the middle layer are now used as the inputs to the final node on the right. As before, we multiply the outputs by the weights, add them along with the bias value, and pass that to the activation function. In this case, the activation function is not a ReLU but a sigmoid.

The top node's output is 0, and the bottom's output is 0.1720. Multiplying these by their respective weights, summing, and adding the bias value of 2.2277 gives us 1.9502 as the argument to the sigmoid activation function, producing 0.8755 as the network's output for the first input sample.

How should we interpret this output? Here's where we learn an important aspect of neural networks:

Neural networks don't tell us the actual class label for the input, but only their confidence in one label relative to another.

Binary models output a confidence value that we're interpreting as the probability of the input belonging to class 1.

Probabilities are numbers between 0 (no chance) and 1 (absolutely assured). Humans are generally more comfortable with percentages, which we get by multiplying the probability by 100. Therefore, we can say that the network is a little more than 87 percent confident that this input represents an instance of class 1.

In practice, we use a threshold—a cutoff value—to decide which label to assign. The most common approach for binary models is a threshold of 50 percent. If the output exceeds 50 percent (probability 0.5), we assign the input to class 1. This output is above 50 percent, so we assign “class 1” as the label. This sample is from class 1, meaning the network’s assigned label is correct.

We can repeat these calculations for the second input sample, $(0.0967, -1.2138, -1.0500)$. I’ll leave walking through it to you as an exercise, but the network’s output for sample 2 is 0.4883. In other words, the network’s confidence that this sample belongs to class 1 is 49 percent. The cutoff is 50 percent, so we reject the class 1 label and assign this input to class 0. The actual class is class 1, so, in this instance, the network is wrong—it assigned a class 1 sample to class 0. Oops.

Is this a useful model? The answer depends on the context. We're classifying wine by cultivar. If the model's output is wrong 20 percent of the time, which is one time in five, is that acceptable? I suspect not, but there might be other tasks where a model with this level of accuracy is acceptable.

Neural networks offer some control over how their outputs are interpreted. For example, we might not use 50 percent as the cutoff. If we make it lower, say, 40 percent, we'll capture more class 1 samples, but at the expense of mistakenly identifying more actual class 0 samples as class 1. In other words, we get to trade off one kind of error for another.

Let's bring the other models in [Figure 4-2](#) into the mix. I trained all three models using the same training and test sets used for [Figure 4-3](#). I repeated the process 240 times for each of the three models. Here are the average accuracies:

2-node 81.5 percent

3-node 83.6 percent

8-node 86.2 percent

The model's performance improves as the number of nodes in the hidden layer increases. This makes intuitive sense, as a more complex model (more nodes) implies the ability to learn more complex associations hidden within the training set.

I suspect you now have a new question: why did I train each model 240 times and report the average accuracy over all 240 models? Here's another critical thing to understand about neural networks:

Neural networks are randomly initialized, such that repeated training leads to differently performing models even when using the same training data.

The phrase “randomly initialized” demands clarification. Look again at [Figure 4-3](#). The numbers representing the weights and biases came from an iterative process. This means that an initial set of weights and biases are updated repeatedly, each time moving the network toward a better and better approximation of whatever function it is that links the input feature vectors and the output labels. Approximating this function well is what we want the network to do.

Why not initialize all the weights to the same value? The answer is that doing so forces the weights to learn similar characteristics of the data, which is something we don't want,

and in the end the model will perform poorly. If we set all of the initial weights to zero, the model does not learn at all.

An initial set of values are necessary for the iterative process to work. How should we pick the initial values? That's an important question, and the answer for our current level of understanding is "at random," meaning we roll dice, in a sense, to get the initial value for each weight and bias. The iterative process then refines these values to arrive at the final set in [Figure 4-3](#).

However, the iterative process doesn't always end in the same place. Pick a different random set of initial weights and biases, and the network will converge to a different set of final values. For example, the network in [Figure 4-3](#) achieved an accuracy of 81 percent, as mentioned previously. Here are 10 more accuracies for the same network trained and tested on the same data:

89, 85, 73, 81, 81, 81, 81, 85, 85, 85

The accuracies range from a high of 89 percent to a low of 73 percent. All that changed between each training session was the collection of initial weights and biases. This is an often overlooked issue with neural networks. Networks should be

trained multiple times, if feasible, to gather data on their effectiveness or, as with the 73 percent version of the network, to understand that a bad set of initial values was used purely by chance. I should also mention that the wide variation in the accuracy of this network is related to its being relatively small and containing only a few weights and biases. Larger models tend to be more consistent when trained repeatedly.

We've already covered a lot of ground, so a recap is in order:

- The fundamental unit of a neural network is the neuron, also called a node.
- Neurons multiply their inputs by weights, sum those products, add a bias value, and pass all of that to the activation function to produce an output value.
- Neural networks are collections of individual neurons, typically arranged in layers with the output of the current layer the input to the following layer.
- Training a neural network assigns values to the weights and biases by iteratively adjusting an initial, randomly selected set.
- Binary neural networks produce an output that roughly corresponds to the probability of the input belonging to class 1.

Now that we know what a neural network is and how it's used, we finally come to the crux of the matter: where do the magic weights and biases come from in the first place? In [Chapter 2](#), I briefly mentioned that neural networks improved in the 1980s thanks to two essential algorithms: backpropagation and gradient descent. These are the algorithms at the heart of neural network training.

We discussed optimization, the process of finding the best of something according to some criteria, in [Chapter 3](#) in reference to support vector machines. Training a neural network is also an optimization process, involving learning the weights and biases that best fit the training data. Care must be taken, however, to make it more likely that the learned weights and biases fit general trends in the training data rather than the details of the specific training data itself. What I mean by that will become apparent as we learn more about the training process.

The general training algorithm is:

1. Select the model's architecture, including the number of hidden layers, nodes per layer, and activation function.

2. Randomly but intelligently initialize all the weights and biases associated with the selected architecture.
3. Run the training data, or a subset, through the model and calculate the average error. This is the *forward pass*.
4. Use backpropagation to determine how much each weight and bias contributes to that error.
5. Update the weights and biases according to the gradient descent algorithm. This and the previous step make up the *backward pass*.
6. Repeat from step 3 until the network is considered “good enough.”

These six steps include many important terms. It's worth our time to ensure that we have an idea of what each means. In this chapter, *architecture* refers to the number of layers, typically hidden layers, used by the network. We have our input feature vector, and we can imagine each hidden layer working collectively to accept an input vector and produce an output vector, which then becomes the input to the next layer, and so on. For binary classifiers, the network's output is a single node producing a value from 0 to 1. We'll learn later in the book that this idea can be extended to multiclass outputs.

The algorithm indicates that training is an iterative process that repeats many times. Iterative processes have a starting point. If

you want to walk from point A to point B, place one foot in front of the other. That's the iterative part. Point A is the starting point. For a neural network, the architecture implies a set of weights and biases. The initial values assigned to those weights and biases are akin to point A, with training akin to placing one foot in front of the other.

The algorithm uses the phrase “average error.” What error? Here's where a new concept enters the picture. Intuitively, we can see that simply picking some initial values for the weights and biases is not likely to lead to a network able to classify the training data accurately. Remember, we know the inputs and the expected outputs for the training data.

Say we push training sample 1 through the network to give us an output value, perhaps 0.44. If we know that sample 1 belongs to class 1, the error made by the network is the difference between the expected output and the actual output. Here, that's $1 - 0.44$, or 0.56. A good model might instead have produced an output of 0.97 for this sample, giving an error of only 0.03. The smaller the error, the better the model is at classifying the sample. If we push all the training data through the network, or a representative subset of it, we can calculate the error for each training sample and find the average over the entire training set. That's the measure used by the (to be described)

backpropagation and gradient descent algorithms to update the weights and biases.

Finally, the training algorithm says to push data through the network, get an error, update the weights and biases, and repeat until the network is “good enough.” In a way, good enough is when the error, also called the *loss*, is as close to zero as possible. If the network produces 0 as the output for all class 0 samples and 1 as the output for all class 1 samples, then it performs perfectly on the training data, and the error will be zero. That’s certainly good enough, but we must be careful. Sometimes when that happens the network is *overfitting*, meaning it’s learned all the details of the training data without actually learning the general trends of the data that will allow it to perform well when used with unknown inputs in the wild.

In practice, overfitting is addressed in several ways, the best of which is acquiring more training data. We use the training data as a stand-in for all the possible data that could be produced by whatever process we are trying to model. Therefore, more training data means a better representation of that data collection. It’s the interpolate versus extrapolate issue we discussed in [Chapter 1](#).

However, getting more training data might not be possible. Alternatives include tweaking the training algorithm to introduce things that keep the network from focusing on irrelevant details of the training data while learning. One such technique you may hear mentioned is *weight decay*, which penalizes the network if it makes the weight values too large.

Another common approach is [data augmentation](#). Out of training data? No worries, data augmentation will invent some by slightly modifying the data you already have. Data augmentation takes the existing training data and mutates it to produce new data that might plausibly have been created by the same process that made the actual training data. For example, if the training sample is a picture of a dog, it will still be a picture of a dog if you rotate it, shift it up a few pixels, flip it left to right, and so on. Each transformation produces a new training sample. It might seem like cheating, but in practice, data augmentation is a powerful [regularizer](#) that keeps the network from overfitting during training.

Let's return for a moment to initialization, as its importance was not sufficiently appreciated for many years.

At first, weight initialization meant nothing more than “pick a small random number” like 0.001 or -0.0056. That worked

much of the time. However, it didn't work consistently, and when it did work, the network's behavior wasn't stellar.

Shortly after the advent of deep learning, researchers revisited the “small random value” idea in search of a more principled approach to initialization. The fruit of those efforts is the way neural networks are initialized to this day. Three factors need to be considered: the form of the activation function, the number of connections coming from the layer below (*fan-in*), and the number of outputs to the layer above (*fan-out*). Formulas were devised to use all three factors to select the initial weights for each layer. Bias values are usually initialized to zero. It isn't difficult to demonstrate that networks so initialized perform better than those initialized the old-fashioned way.

We have two steps of the training algorithm yet to discuss: backpropagation and gradient descent. Backpropagation is often presented first because its output is necessary for gradient descent. However, I think it's more intuitive to understand what gradient descent is doing, then fill in the missing piece it needs with what backpropagation provides. Despite the unfamiliar names, I am certain you already understand the essence of both algorithms.

You're standing in a vast, open grassland of rolling hills. How did you get here? You strain your brain, but no answer comes. Then, finally, you spy a small village to the north, in the valley far below. Perhaps the people there can give you some answers. But what's the best way to get there?

You want to go north and down, in general, but you must also respect the contour of the land. You always want to move from a higher to a lower position. You can't go due north because a large hill is in your way. You could head northeast; the terrain is flatter there, but going that way will make your journey a long one, as the land drops slowly. So, you decide to head northwest, as that moves you both north and down more steeply than to the east. You take a step to the northwest, then pause to reassess your position to decide which direction to move in next.

Repeating this two-stage process of examining your current position to determine the direction that best moves you both northward and downward, then taking a step in that direction, is your best bet for reaching the village in the valley. You may not make it; you might get stuck in a small canyon out of which you can't climb. But overall, you'll make progress toward your goal by consistently moving in a direction that is north and down relative to your current position.

Following this process, known as *gradient descent*, lets us adjust a neural network's initial weights and biases to give us ever better-performing models. In other words, gradient descent trains the model.

The three-dimensional world of the grassland surrounding the village corresponds to the n -dimensional world of the network, where n is the total number of weights and biases whose values we are trying to learn. Choosing a direction to head in from your current position and then moving some distance in that direction is a gradient descent step. Repeated gradient descent steps move you closer and closer to the village.

Gradient descent seeks the minimum position, the village in the valley—but the minimum of what? For a neural network, gradient descent aims to adjust the weights and biases of the network to minimize the error over the training set.

The vast, open grassland of rolling hills represents the error function, the average error over the training data when using the weight and bias values corresponding to your current position. This means that each position in the grassland implies a complete set of network weights and biases. The position of the village corresponds to the smallest error the network can make on the training set. The hope is that a model that has a

small error on its training set will make few errors on unknown inputs when used in the wild. Gradient descent is the algorithm that moves through the space of weights and biases to minimize the error.

Gradient descent is an optimization algorithm, again telling us that training a neural network is an optimization problem, a problem where we need to find the best set of something. While this is true, it is also true that training a neural network is subtly different from other optimization problems. As mentioned previously, we don't necessarily want the smallest possible error on the training data, but rather the model that best generalizes to unknown inputs. We want to avoid overfitting. I'll demonstrate visually what that means later in the chapter.

Gradient descent moves through the landscape of the error function. In everyday use, a gradient is a change in something, like the steepness of a road or a color gradient varying smoothly from one shade to another. Mathematically, a gradient is the multidimensional analog of the slope of a curve at a point. The steepest direction to move is down the maximum gradient. The slope of a line at a point on a curve is a helpful representation of the gradient, so contemplating slopes is a worthy use of our time.

[Figure 4-4](#) shows a curve with four lines touching it at different points. The lines represent the slope at those points. The slope indicates how quickly the value of the function changes in the vicinity of the point. The steeper the line, the faster the function's value changes as you move along the x -axis.

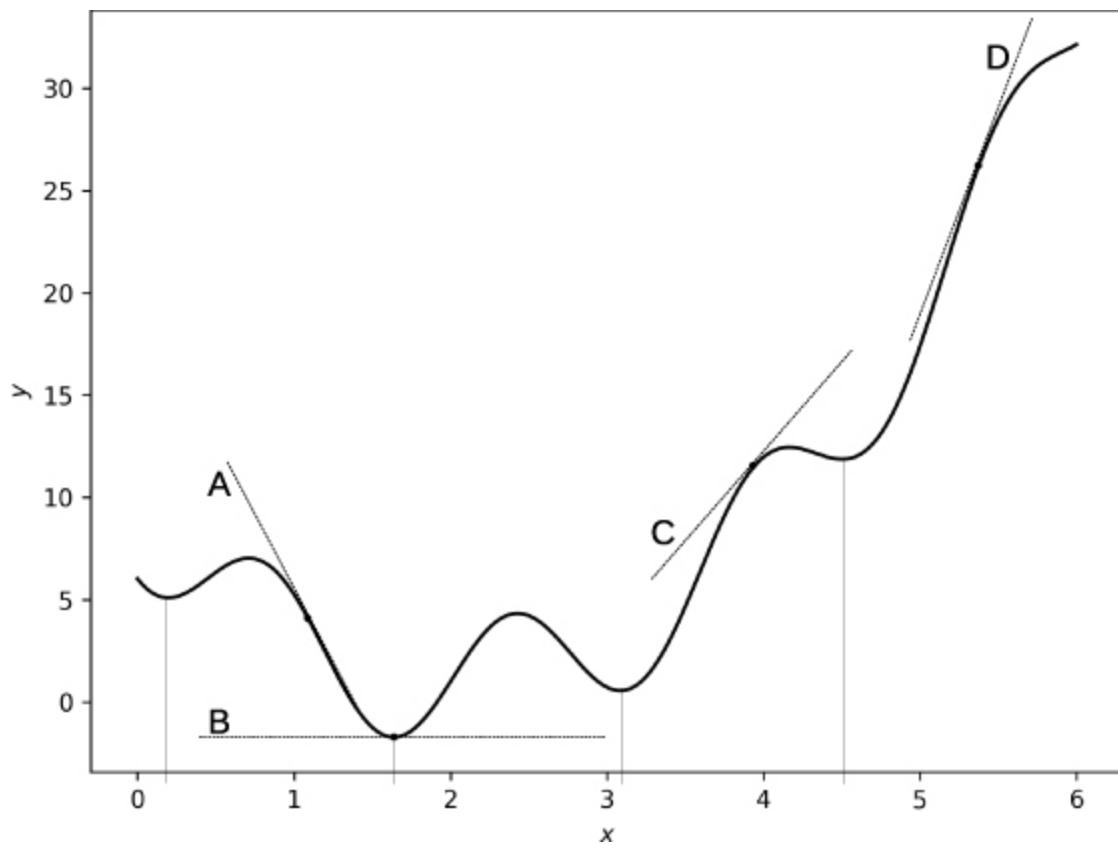


Figure 4-4: A curve with the slope at various points marked

Line B marks the lowest point on the curve. This is the [global minimum](#) and the point that an optimization algorithm seeks to find. Notice that the line touching this point is entirely horizontal. Mathematically, this means that the slope of line B is zero. This is true at the minima (and maxima) of functions.

The point touched by line B is the global minimum, but there are three other minima in the plot. These are *local minima*, points where the slope of the line touching those points is also zero. Ideally, an optimization algorithm would avoid these points, favoring the global minimum.

Line A is steep and points toward the global minimum. Therefore, if we were at the point on the curve touched by line A, we could move quickly toward the global minimum by taking steps in the indicated direction. Moreover, as the slope is steep here, we can take reasonably large steps down to the valley.

Line C is also steep but heads toward one of the local minima, the one just beyond 3 on the x -axis. A gradient descent algorithm that only knows how to move down the gradient will locate that local minimum and become stuck there. The same applies to line D, which heads toward the local minimum between 4 and 5 on the x -axis.

What are the takeaways from [Figure 4-4](#)? First, gradient descent moves down the gradient, or slope, from some point. Here the curve is one-dimensional, so the point is a specific value of x . Gradient descent uses the value of the slope at that point to pick a direction and a step size proportional to the steepness of the slope. A steep slope means we can take a larger step to end up

at a new x value closer to a minimum. A shallow slope implies a smaller step.

For example, suppose we are initially at the point where line A touches the curve. The slope is steep, so we take a big step toward the global minimum. After the step, we look at the slope again, but this time it's the slope at the new point on the x -axis. Using that slope, we take another step, then another, and another until we get to a point where the slope is essentially zero. That's the minimum, so we stop.

The one-dimensional case is straightforward enough because at each point there is only one slope, so there is only one direction to go. However, recalling the vast, open grassland, we know that from any point there are an infinite number of directions we might head in, many of which are useful in that they move us northward and downward. One of these directions, the direction of the maximum gradient, is the steepest and moves us most quickly toward our desired destination, and that's the direction we step in. Repeating the process, using the maximum gradient direction each time, accomplishes in multiple dimensions what we did in one dimension. To be precise, we step in the direction *opposite* the maximum gradient because the maximum gradient points away from the minimum, not toward it.

[Figure 4-5](#) presents gradient descent in two dimensions. The figure shows a contour plot. Imagine an open pit mine with terraced levels: the lighter the shade, the deeper into the mine, but also the flatter the slope. That is, lighter shades imply shallower slopes.

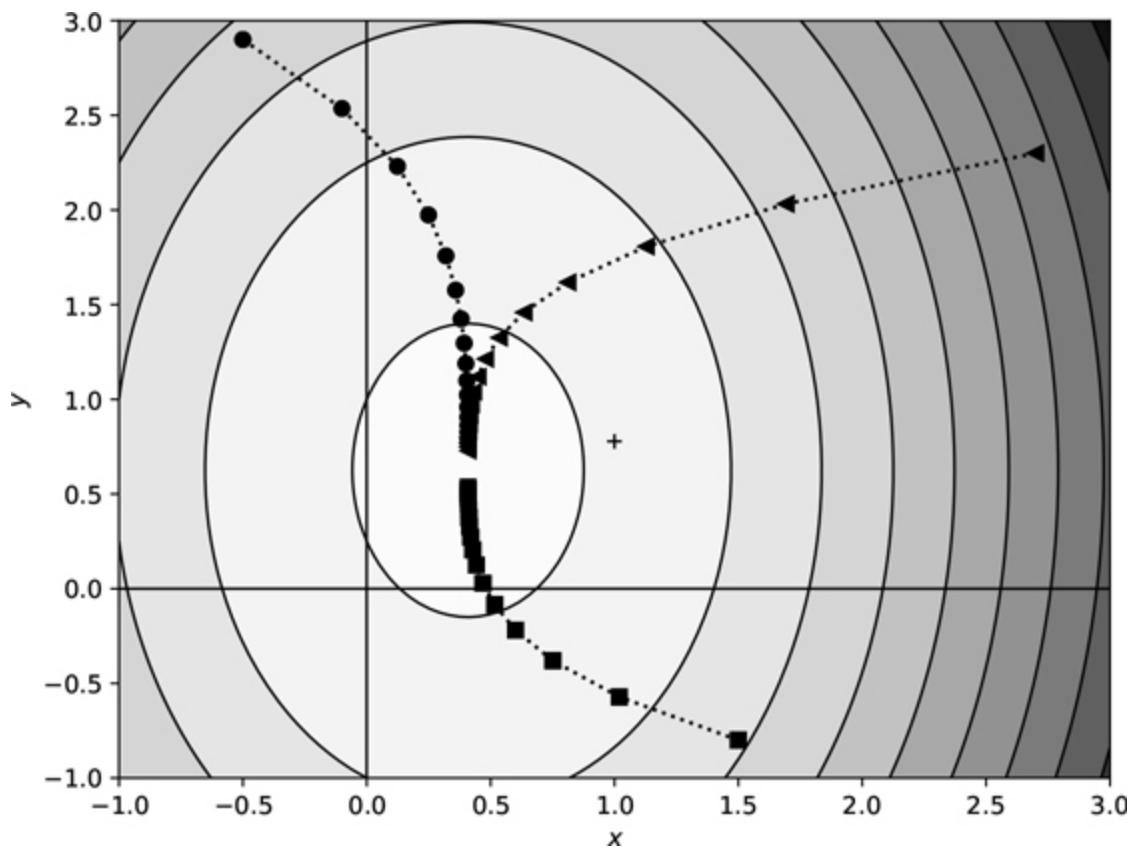


Figure 4-5: Gradient descent in two dimensions

The figure shows the path taken by gradient descent for three starting positions: the circle, the triangle, and the square. Initially, the slopes are steep, so the step sizes are big, but the slopes become shallow as the minimum is approached,

implying smaller steps. Eventually, gradient descent reaches the minimum, regardless of the starting point.

We've discussed gradient descent in one and two dimensions because we can visualize the process. We understand now that we have always known the algorithm and used it ourselves whenever we walk from a higher elevation to a lower one. Honestly, this is all that training a neural network does. The initial set of weights and biases is nothing more than a single starting point in an n -dimensional space. Gradient descent uses the maximum gradient from that initial starting position to march toward a minimum. Each new position in the n -dimensional space is a new set of the n weights and biases generated from the previous set based on the steepness of the gradient. When the gradient gets very small, we claim victory and fix the weights and biases, believing the network to be trained.

Gradient descent depends on slopes, on the value of the gradient. But where do the gradients come from? Gradient descent minimizes the loss function, or the error made by the network. The error over the training set is a function of each weight and bias value in the network. The gradient represents how much each weight and bias contributes to the overall error.

For example, suppose we know how much weight 3 (whatever weight that labels) contributes to the network's error as measured by the mistakes the network makes on the training set. In that case, we know the steepness of the gradient should we change weight 3's value, keeping all other weights and biases the same. That steepness, multiplied by a step size, gives us a value to subtract from weight 3's current value. By subtracting, we move in the direction opposite to the maximum gradient. Repeating the calculation for every weight and bias in the network takes a step in the n -dimensional space. This is what gradient descent does during training.

Backpropagation is the algorithm that gives us the steepness values per weight and bias. Backpropagation is an application of a well-known rule from differential calculus, the branch of mathematics telling us how one thing changes as another changes. Speed is an example. Speed indicates how distance changes with time. It's even in how we talk about speed: miles per hour or kilometers per hour. Backpropagation gives us the "speed" representing how the network's error changes with a change in any weight or bias value. Gradient descent uses these "speeds," multiplied by a scale factor known as the learning rate, to step to the next position in the n -dimensional space represented by the n weights and biases of the network.

For example, the “big” network in [Figure 4-2](#) has 32 weights and 9 biases; therefore, training that network with gradient descent means moving through a 41-dimensional space to find the 41 weight and bias values giving us the smallest error averaged over the training set.

The algorithm is called “backpropagation” because it calculates the “speed” values for each weight and bias, beginning with the network’s output layer and then moving backward, layer by layer, to the input layer. That is, it moves backward through the network to propagate the error from a layer to the previous layer.

The take-home message is this:

Gradient descent uses the gradient direction supplied by backpropagation to iteratively update the weights and biases to minimize the network’s error over the training set.

And that, in a nutshell, is how neural networks are trained.

The ability to train a neural network with backpropagation and gradient descent is a bit of a fluke. It shouldn’t work. Gradient descent with backpropagation is a *first-order* optimization approach. First-order optimization works best with simple

functions, and the error surfaces of a neural network are anything but. However, Fortuna has smiled upon us, and it does work, and rather well at that. There is as yet no rigorous mathematical explanation beyond the realization that the local minima of the error function are all pretty much the same, meaning if you land in one and can't get out, that's often just fine.

There is another empirical explanation, but to understand that, we must learn more about the training process. The six-step training algorithm I gave earlier in the chapter talks about running the training set, or a subset of it, through the network, and repeating until things are “good enough.” Let me expand on the process implied by these steps.

Each pass of training data through the network, a forward pass followed by a backward pass, results in a gradient descent step as shown in [Figure 4-5](#). If the training set is small, all of it is used in the forward pass, meaning all of it is used by gradient descent to decide where to step next. A complete pass through the training data is called an [*epoch*](#); therefore, using all the training data in the forward and backward passes results in one gradient descent step per epoch.

Modern machine learning datasets are often massive, making it computationally infeasible to use all of the training data for each gradient descent step. Instead, a small, randomly selected subset of the data, known as a *minibatch*, is passed through the network for the forward and backward passes. Using minibatches dramatically reduces the computational overhead during gradient descent, resulting in many steps per epoch. Minibatches also provide another benefit that helps overcome the “this approach to training shouldn’t work” issue.

Suppose we had a mathematical function representing the error made by the network. In that case, we could use centuries-old calculus techniques to find the exact form of each weight and bias’s contribution to the error; gradient descent would know the best direction to step each time. Unfortunately, the world isn’t that kind. We don’t know the mathematical form of the error function (there isn’t likely one to know), so we have to approximate with our training data. This approximation improves when using more training data to determine the error. This fact argues for using all the training data for each gradient descent step. However, we already know this is computationally extremely taxing in many cases.

The compromise is to use minibatches for each gradient descent step. The calculations are no longer too taxing, but the

approximation of the actual gradient is worse because we are estimating it with fewer data points. Randomly selecting something is often attached to the word “stochastic,” so training with minibatches is known as *stochastic gradient descent*. Stochastic gradient descent, in one form or another, is the standard training approach used by virtually all modern AI.

At first blush, stochastic gradient descent sounds like a losing proposition. Sure, we can calculate many gradient descent steps before the heat death of the universe, but our gradient fidelity is low, and we’re likely moving in the wrong direction through the error space. That can’t be good, can it?

Here’s where Fortuna smiles on humanity a second time. Not only has she given us the ability to train complex models with first-order gradient descent because local minima are (assumed) roughly equivalent; she’s also arranged things so that the “wrong” gradient direction found by stochastic gradient descent is often what we need to avoid local minima early in the training process. In other words, walking slightly northeast when we should head due north is a blessing in disguise that allows us to train large neural networks.

We're ready to move on to the next chapter. However, before we do, let's apply traditional neural networks to the dinosaur footprint dataset. We'll compare the results to the classical models of [Chapter 3](#).

We need first to select an architecture: that is, the number of hidden layers, the number of nodes per layer, and the type of activation function for each node. The dinosaur footprint dataset has two classes: ornithischian (class 0) and theropod (class 1). Therefore, the output node should use a sigmoid activation function to give us a likelihood of class 1 membership. The network's output value estimates the probability that the input image represents a theropod. If the probability is above 50 percent, we'll assign the input to class 1; otherwise, into class 0 it goes. We'll stick with rectified linear unit activations for the hidden layer nodes, as we have for all the models in this chapter. All that remains is to select the number of hidden layers and the number of nodes per layer.

There are 1,336 training samples in the footprints dataset. That's not a lot, and we aren't augmenting the dataset, so we need a smallish model. Large models, meaning many nodes and layers, require large training sets; otherwise, there are too many weights and biases to learn relative to the number of training samples. Therefore, we'll limit ourselves to trying at

most two hidden layer models for the footprints dataset. As for the number of nodes in the hidden layers, we'll let the first hidden layer vary from very small to nearly twice the input size of 1,600 features (the 40×40 -pixel image unraveled). If we try a second hidden layer, we'll restrict the number of nodes to no more than half the number in the first hidden layer.

First, we'll train a collection of one- and two-layer architectures. Second, we'll train the best performing of those 100 times to give us an average level of performance. [Table 4-1](#) presents the trial models' results.

Table 4-1: Trial Architectures with the Dinosaur Footprint Dataset

Accuracy (%)	Architecture	Weights and biases
59.4	10	16,021
77.0	400	640,801
76.7	800	1,281,601
81.2	2,400	3,844,801

Accuracy (%)	Architecture	Weights and biases
75.8	100, 50	165,201
81.2	800, 100	1,361,001
77.9	2,400, 800	5,764,001

The network with a mere 10 nodes in its hidden layer was the worst, returning an accuracy of about 60 percent. A binary classifier that does nothing but flips a coin is correct about 50 percent of the time, so the 10-node network is performing only slightly above chance. We don't want that one. Most of the other networks return accuracies in the mid- to upper 70s.

The two models in **bold** each produced just over 81 percent accuracy. The first used a single hidden layer of 2,400 nodes. The second used a hidden layer of 800 nodes, followed by another with 100 nodes. Both models produced the same accuracy on the test set, but the 2,400-node model had nearly three times as many weights and biases as the two-layer model, so we'll go with the two-layer model. (Bear in mind that the results in [Table 4-1](#) represent a single training session, not the average of many. We'll fix that shortly.)

The two-layer model is still relatively large. We're trying to learn 1.4 million parameters to condition the model to correctly classify the dinosaur footprint images. That's a lot of parameters to learn, especially with a training set of only 1,336 samples. Fully connected neural networks grow quickly in terms of the number of parameters required. We'll revisit this observation in [Chapter 5](#) when discussing convolutional neural networks.

We have our architecture: two hidden layers using rectified linear activation functions with 800 and 100 nodes, respectively, followed by a single node using a sigmoid to give us a likelihood of class 1 membership. Training the model 100 times on the footprints dataset returned an average accuracy of 77.4 percent, with a minimum of 69.3 percent and a maximum of 81.5 percent. Let's put this result in its proper relation to those of [Chapter 3](#); see [Table 4-2](#).

Table 4-2: Dinosaur Footprint Models

Model	Accuracy (%)
RF300	83.3

Model	Accuracy (%)
RBF SVM	82.4
7-NN	80.0
3-NN	77.6
MLP	77.4
1-NN	76.1
Linear SVM	70.7

Recall that RF300 means a random forest with 300 trees, SVM refers to a support vector machine, and, somewhat confusingly, NN refers to a nearest neighbor classifier. I'm using MLP (multilayer perceptron) as a stand-in for our neural network. [Multilayer perceptron](#) is an old but still common name for the traditional neural networks we've been discussing in this chapter—notice the link back to Rosenblatt's original Perceptron from the late 1950s.

Our neural network wasn't the best performer on this dataset. In fact, it was one of the worst. Additional tweaking might move it up a place or two on the list, but this level of performance is typical, in my experience, and contributed to the general perception (pun intended) before the deep learning revolution that neural networks are “meh” models—run-of-the-mill, nothing to write home about.

This chapter introduced the fundamental ideas behind modern neural networks. The remainder of the book builds on the basic concepts covered in this chapter. Here are the principal takeaways:

- Neural networks are collections of nodes (neurons) that accept multiple inputs and produce a single number as output.
- Neural networks are often arranged in layers so that the current layer’s input is the previous layer’s output.
- Neural networks are randomly initialized, so repeated training leads to differently performing models.
- Neural networks are trained by gradient descent, using the gradient direction supplied by backpropagation to update the weights and biases iteratively.

Now, let's press on to investigate convolutional neural networks, the architecture that ushered in the deep learning revolution. This chapter brought us to the early 2000s. The next moves us to 2012 and beyond.

KEY TERMS

activation function, architecture, backward pass, bias, data augmentation, epoch, forward pass, global minimum, gradient descent, hidden layer, learning rate, local minimum, loss, minibatch, multilayer perceptron, neuron, node, overfitting, preprocessing, rectified linear unit, regularizer, sigmoid, stochastic gradient descent, weight

5

CONVOLUTIONAL NEURAL NETWORKS: AI LEARNS TO SEE



Classical machine learning models struggle with appropriate feature selection, feature vector dimensionality, and the inability to learn from the structure inherent in the input.

[Convolutional neural networks \(CNNs\)](#) overcome these issues by learning to generate new representations of their inputs while simultaneously classifying them, a process known as [end-to-end learning](#). CNNs are the representation-learning data processors I referred to in [Chapter 2](#).

Elements of what became CNNs appeared at various times throughout the history of neural networks, beginning with Rosenblatt's Perceptron, but the architecture that ushered in the deep learning revolution was published in 1998. Over a decade of additional improvements in computing capability

were required to unleash the full power of CNNs with the appearance of AlexNet in 2012.

Convolutional networks exploit structure in their inputs. We'll better understand what that means as the chapter progresses. In one dimension, the inputs might be values that change over time, also known as a time series. In two dimensions, we're talking about images. Three-dimensional CNNs exist to interpret volumes of data, like a stack of magnetic resonance images or a volume constructed from a LiDAR point cloud. In this chapter, we'll focus exclusively on two-dimensional CNNs.

The order in which features are presented to a traditional neural network is irrelevant. Regardless of whether we present feature vectors to the model as (x_0, x_1, x_2) or (x_2, x_0, x_1) , the model will learn just as well because it assumes the features are independent and unrelated to each other. Indeed, a strong correlation between a pixel value and adjacent pixel values is something traditional machine learning models do not want, and their inability to achieve much success with such inputs held neural networks back for years.

Convolutional neural networks, on the other hand, exploit structure in their inputs. For a CNN, it matters whether we present the input as (x_0, x_1, x_2) or (x_2, x_0, x_1) ; the model might

learn well with the former and poorly with the latter. This isn't a weakness, but a strength, because we want to apply CNNs to situations where there is structure to learn—structure that helps determine how best to classify inputs.

Later in the chapter, we'll compare the performance of a traditional neural network to a CNN when classifying small photos of animals and vehicles (the CIFAR-10 dataset of [Chapter 3](#)). At that time, we'll learn the true power of exploiting structure. Before that, however, let's conduct a little experiment. We have two datasets. The first is our old friend, the MNIST digits dataset; the second is the same collection of digit images, but the order of the pixels in the images has been scrambled. The scrambling isn't random but consistent so that the pixel at position (1,12) has been moved to, say, position (26,13), with similarly consistent moves for all other pixels. [Figure 5-1](#) shows some examples of MNIST digits and scrambled versions of the same digits.



Figure 5-1: Example MNIST digits (top) and scrambled versions of the same digits (bottom)

The scrambled digits are incomprehensible to me. The pixel information between the original and scrambled digits is the same—that is, the same collection of pixel values is present in both—but the structure is largely gone, and I can no longer discern the digits. I claim that a traditional neural network treats its inputs holistically and isn’t looking for structure. If that’s the case, a traditional neural network shouldn’t care that the digits have been scrambled; it should learn just as well when trained using the original or the scrambled dataset. As it turns out, that’s precisely what happens. The model learns equally well; scrambling changes nothing in terms of performance. Note, though, that the scrambled test digits must be used with the scrambled model; we shouldn’t expect the model to work when trained on one dataset and tested on the other.

We at present know only one fact about CNNs: they pay attention to structure in their inputs. Knowing this, should we expect a CNN trained on the scrambled dataset to perform as well as one trained on the original dataset? The scrambled digits are uninterpretable by us because local structure in the images has been destroyed. Therefore, we might expect a model

that similarly wants to exploit local structure to be unable to interpret the scrambled digits. And that is the case: a CNN trained on the scrambled dataset performs poorly compared to one trained on the original dataset.

Why can't we easily interpret the scrambled digits? We must explore what happens in the brain during vision to answer that question. Then we'll circle back to relate that process to what CNNs do. As we'll learn, CNNs follow the old adage: when in Rome, do as the Romans (humans) do.

Vincent van Gogh is my favorite artist. Something about his style speaks to me, something strangely peaceful from a man tormented by mental illness. I believe the peace emanating from his work reflects his attempt to calm the turmoil within.

Consider [Figure 5-2](#). It shows Van Gogh's famous 1889 painting of his bedroom in Arles. The image is in black and white, an unforgivable violence to Vincent's use of color, but print restrictions require it.



Figure 5-2: Van Gogh's bedroom in Arles, 1889 (public domain)

What do you see in the painting? I'm not asking about a higher meaning or impression, but objectively, what do you see in the painting? I see a bed, two chairs, a small table, a window, and a pitcher on the table, among many other items. I suspect you see the same. You saw the bed, two chairs, and table, but how? Photons, particles of light, traveled from the image to your eye and were converted into discrete objects in your brain. Again, how?

I'm asking questions but not yet offering answers. That's okay for two reasons. First, pondering the problem of segmenting an image into a collection of meaningful objects is worth some effort on our part. Second, no one yet knows the full answer to "how?" Neuroscientists do, however, understand the beginnings of the process.

We take for granted the ability to look at a scene and parse it into separate and identified objects. For us, the process is effortless, completely automatic. We shouldn't be fooled. We're the beneficiaries of hundreds of millions of years of evolution's tinkering. For mammals, vision begins in the eye, but parsing and understanding begins in the primary visual cortex at the back of our brains.

The primary visual cortex, known as area V1, is sensitive to edges and orientation. Immediately, we encounter a clue to how vision works in the brain (as opposed to the eye). The brain takes the input sensations, spread over V1 as a warped image, and begins by seeking edges and the orientation of the edges. V1 is additionally sensitive to color. Mapping the entire visual field over V1, with magnification so that most of V1 is occupied by the central 2 percent of our visual field, means that edge detection, orientation, and color are local to where they occur.

V1 sends its detections to area V2, which sends its detections to area V3, and so on through V4 to V5, with each area receiving, essentially, a representation of larger and more grouped elements of what is in the visual field. The process starts with V1 and, eventually, delivers a fully parsed and understood representation of what the eyes see. As mentioned, the details much beyond V1 are murky, but for our purposes all we need to remember is that V1 is sensitive to edges, the orientation of edges, and colors (we might also include textures). Starting simply and grouping to separate objects in the scene is the name of the game. CNNs mimic this process. It's fair to say that CNNs literally learn to see the world of their inputs.

CNNs decompose inputs into small parts, then groups of parts and still larger groups of groups of parts, until the entire input is transformed from a single whole into a new representation: one that is more easily understood by what amounts to a traditional neural network sitting at the top of the model. However, mapping the input to a new, more easily understood representation does not imply that the new representation is more easily understood by *us*.

Convolutional neural networks learn during training to partition inputs into parts, enabling the top layers of the network to classify successfully. In other words, CNNs learn

new representations of their inputs and then classify those new representations. Indeed, “Learning New Representations from Old” was an early title for this chapter.

How do CNNs break their inputs into parts? To answer that question, we must first understand the “convolution” part of “convolutional neural network.” Be warned, low-level details ahead.

Convolution is a mathematical operation with a formal definition involving integral calculus. Fortunately for us, convolution is a straightforward operation in digital images, using nothing more than multiplication and addition. Convolution slides a small square, known as a **kernel**, over the image from top to bottom and left to right. At each position, convolution multiplies the pixel values covered by the square with the corresponding kernel values. It then sums all those products to produce a single number that becomes the output pixel value for that position. Words only go so far here, so let’s try a picture. Consider **Figure 5-3**.

$$\begin{array}{ccccccccc}
 60 & 58 & 60 & 60 & 60 & 60 & 60 & 52 \\
 68 & \boxed{60} & 60 & 68 & 68 & 52 & 76 & 76 \\
 76 & 44 & \boxed{60} & 60 & 68 & 52 & 60 & 52 \\
 68 & 68 & 76 & 76 & 68 & 52 & 60 & 44 \\
 92 & 84 & 84 & 84 & 76 & 44 & 60 & 44 \\
 76 & 68 & 84 & 76 & 76 & 52 & 60 & 52 \\
 68 & 60 & 60 & 76 & 84 & 52 & 84 & 60 \\
 60 & 60 & 68 & 68 & 68 & 52 & 76 & 68
 \end{array} \times \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -60 & -120 & -68 \\ 0 & 0 & 0 \\ 68 & 152 & 76 \end{bmatrix} = \boxed{48}$$

Figure 5-3: Convolving a kernel over an image

The left side of [Figure 5-3](#) shows a grid of numbers. These are the pixel values for the center portion of the image in [Figure 5-4](#). Grayscale pixel values are typically in the range 0 through 255, where lower values are darker. The kernel is the 3×3 grid to the right. The convolution operation instructs us to multiply each pixel value by the corresponding kernel value. This produces the rightmost 3×3 grid of numbers. The final step sums all nine values to create a single output, 48, which replaces the center pixel in the output image, $60 \rightarrow 48$.

To complete the convolution, slide the 3×3 solid box one pixel to the right and repeat. When the end of a row is reached, move the box down one pixel and repeat for the next row, then process row-by-row until the kernel has covered the entire image. The convoluted image is the collection of new output pixels.

At first, convolution might seem like a strange thing to do. However, in digital images, convolution is a fundamental operation. An appropriately defined kernel lets us filter an image to enhance it in various ways. For example, [Figure 5-4](#) shows four images. The upper left is the original image, a frequently used test image of Gold Hill in Shaftesbury, England. The remaining three images are filtered versions of the original. Clockwise from the upper right, we have a blurred version, one showing horizontal edges, and one showing vertical edges. Each image is produced by convolving a kernel as described previously. The kernel of [Figure 5-3](#) produces the horizontal-edge image at the lower right. Rotate the kernel by 90 degrees, and you get the vertical-edge image at the lower left. Finally, make all the kernel values 1, and you get the blurred image at the upper right. Note that the edge images are inverted to make the detected edges black instead of white.

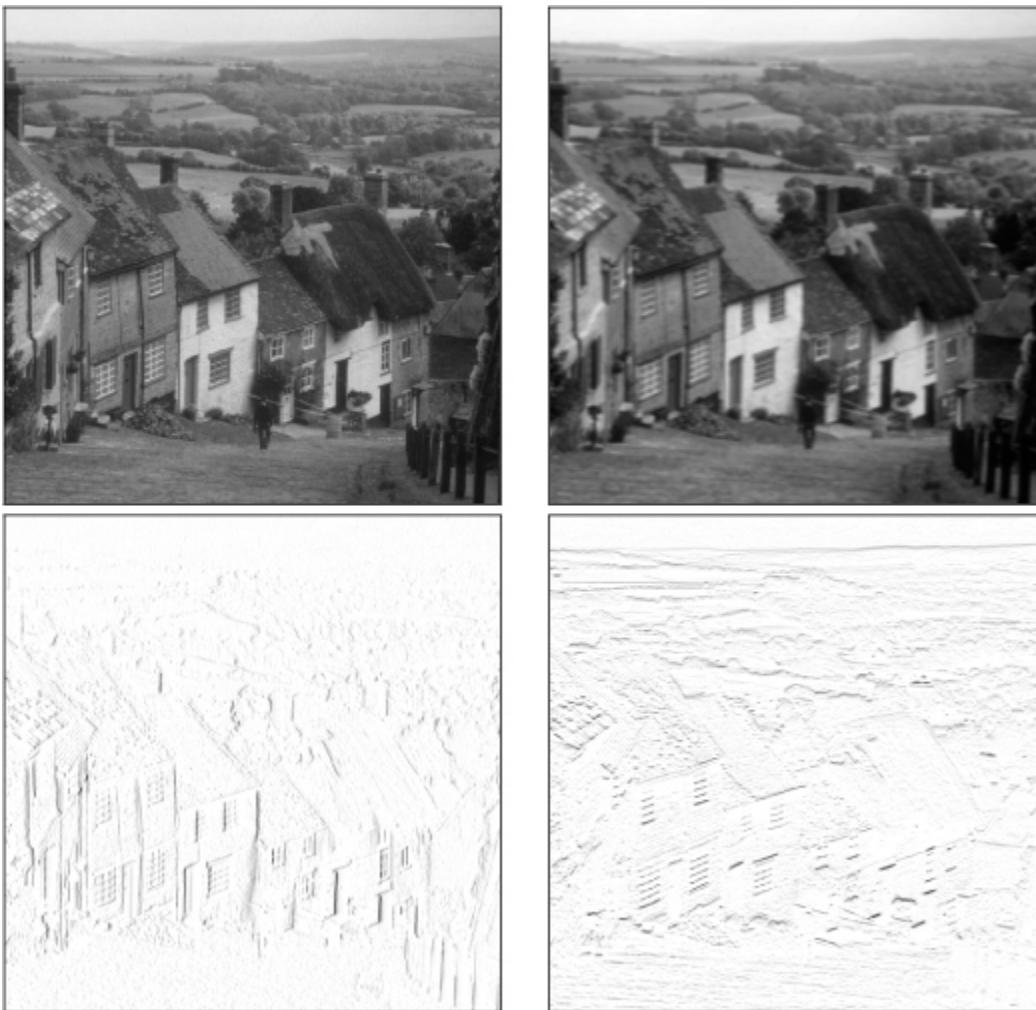


Figure 5-4: Convolution kernels in action

The critical point for us to remember is that convolving an image with different kernels highlights different aspects of the image. It isn't hard to imagine an appropriate set of kernels extracting structure relevant to correctly classifying the image. This is exactly what CNNs do during end-to-end training and, in a sense, what our visual system does in area V1 when it detects edges, orientations, colors, and textures.

We're making progress. We now have a handle on the core operation of a CNN, convolution, so let's take the next step to learn how convolution is used within a model to extract structure and build a new representation of the input.

The traditional neural networks of [Chapter 4](#) consist of a single kind of layer: a collection of fully connected nodes accepting input from the layer below to produce output for the layer above. Convolutional neural networks are more flexible and support diverse layer types. Regardless, the data flow is the same: from input to layer after layer to the network's output.

In CNN parlance, the fully connected layers a traditional neural network uses are called *dense* layers. CNNs usually use dense layers at the top, near the output, because by that time the network has transformed the input into a new representation, one that the fully connected layers can classify successfully. CNNs make heavy use of convolutional layers and pooling layers.

Convolutional layers apply a collection of kernels to their input to produce multiple outputs, much as [Figure 5-4](#) produced three outputs from the one input image at the upper left. The kernels

are learned during training using the same backpropagation and gradient descent approach we encountered in [Chapter 4](#). The values of the learned kernels are the weights of the convolutional layer.

[**Pooling layers**](#) have no weights associated with them. There's nothing to learn. Rather, pooling layers perform a fixed operation on their inputs: they reduce the spatial extent of their inputs by keeping the largest value in a 2×2 square moved without overlap across and then down. The net effect is similar to reducing the size of an image by a factor of two. [Figure 5-5](#) illustrates the process of changing an 8×8 input into a 4×4 output, keeping the maximum value in each solid square. Pooling layers are a concession to reduce the number of parameters in the network.

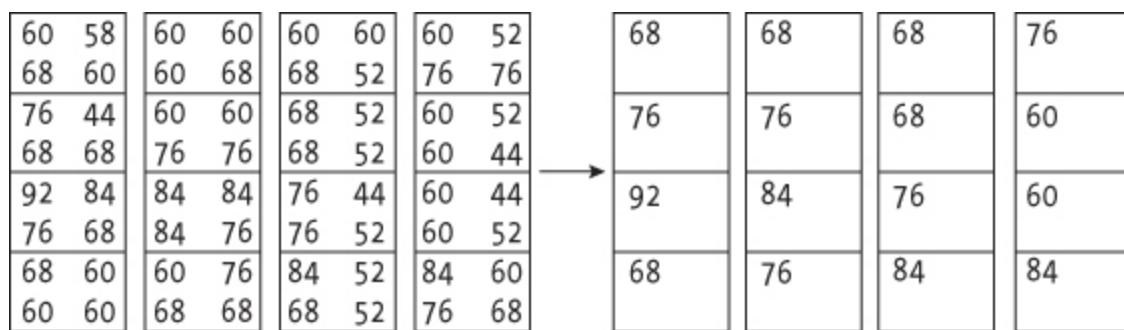
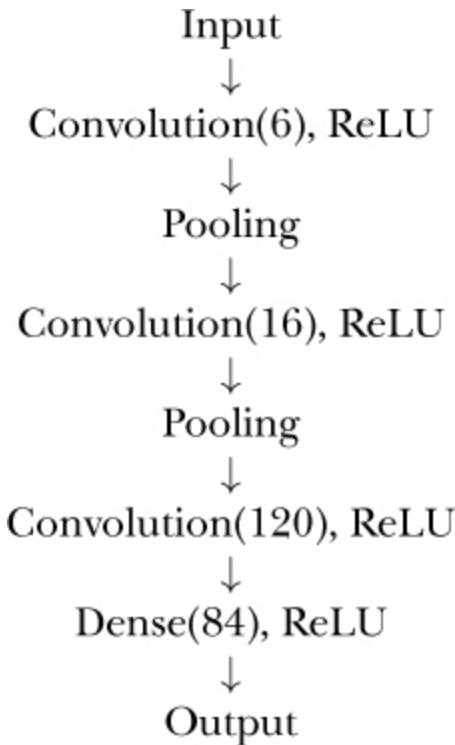


Figure 5-5: Pooling to reduce the spatial extent of the data

A typical CNN combines convolutional and pooling layers before topping things off with a dense layer or two. ReLU layers

are used as well, usually after the convolutional and dense layers. For example, a classic CNN architecture known as LeNet consists of the following layers:



The model uses three convolutional layers, two pooling layers, and a single dense layer with 84 nodes. Each convolutional and dense layer is followed by a ReLU layer to map all negative inputs to zero while leaving all positive inputs untouched.

The number in parentheses for each convolutional layer is the number of filters to learn in that layer. A *filter* is a collection of convolutional kernels, with one kernel for each input channel. For example, the first convolutional layer learns six filters. The input is a grayscale image with one channel, so this layer learns

six kernels. The second convolutional layer learns 16 filters, each with 6 kernels, one for each of the 6 input channels from the first convolutional layer. Therefore, the second convolutional layer learns a total of 96 kernels. Finally, the last convolutional layer learns 120 filters, each with 16 kernels, for another 1,920 kernels. All told, the LeNet model needs to learn 2,022 different convolutional kernels.

The hope is that learning so many kernels will produce a sequence of outputs that capture essential elements of the structures in the input. If training is successful, the output of the final convolutional layer, as a vector input to the dense layer, will contain values that clearly differentiate between classes—at least, more clearly than can be accomplished by using the image alone.

If it feels like we’re in the weeds, we are, but we will not dig further. We’ve reached the lowest level of detail we’ll consider in the book, in fact, but it’s a necessary burden, as we cannot understand how CNNs work if we don’t understand convolution and convolutional layers.

Perhaps the best way to understand what the layers of a CNN are doing is to look at their effect on data flowing through the network. [Figure 5-6](#) shows how a LeNet model trained on

MNIST digits manipulates two input images. The output of the first convolutional layer is the six middle images, where gray represents zero, darker pixels are increasingly negative, and lighter pixels are increasingly positive. The six kernels of the first convolutional layer each produce an output image for the single input image. The kernels highlight different portions of the inputs as transitions from dark to light.

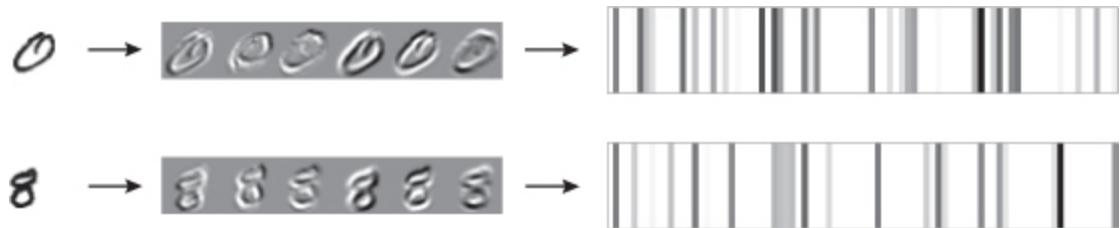


Figure 5-6: Input to first convolutional layer to dense layer

The rightmost barcode-like pattern is a representation of the dense layer's output. We're ignoring the output of the second and third convolutional layers and jumping directly to the end of the model. The dense layer's output is a vector of 84 numbers. For [Figure 5-6](#), I mapped these numbers to pixel values, where larger values correspond to darker vertical bars.

Notice that the barcodes for the digits 0 and 8 differ. If the model learned well, we might expect the barcodes for the dense layer outputs to share commonalities across digits. In other words, the barcodes for zeros should look roughly similar, as should the barcodes for eights. Do they? Consider [Figure 5-7](#).

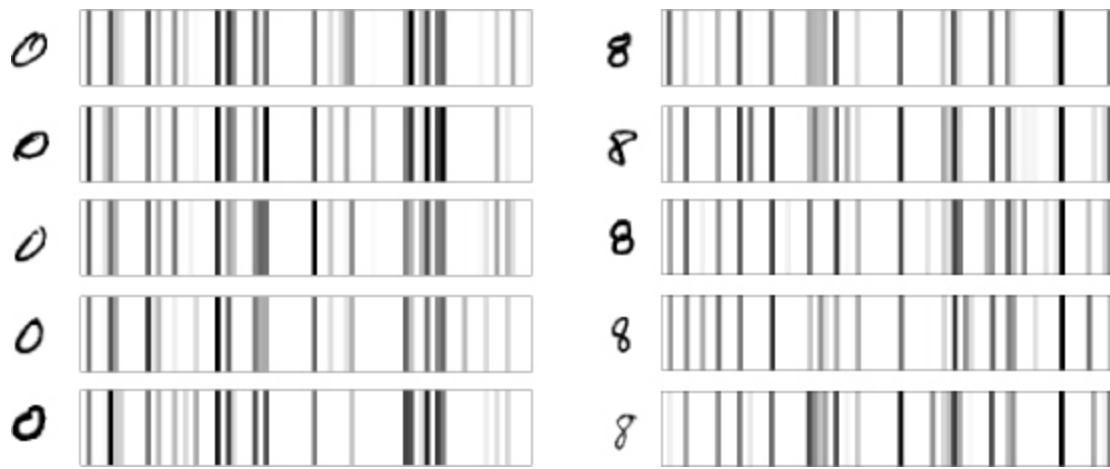


Figure 5-7: Dense layer output for sample inputs

This figure presents the dense layer outputs for five different zero and eight inputs. The barcodes are all different but share similarities according to digit. This is especially true for the zeros. The LeNet model has learned how to map each 28×28 -pixel input image (784 pixels) into a vector of 84 numbers that show strong similarities by digit type. Based on our experience with traditional neural networks, we can appreciate that this mapping has produced something of lower dimensionality that preserves and even emphasizes differences between digits. The learned lower-dimensionality vector is akin to a complex concept explained with a few well-chosen words. This is exactly what we want a CNN to do. The trained model learned to “see” in the world of handwritten digits represented as small grayscale images. There’s nothing special about grayscale images, either. CNNs are quite happy to work with color images

represented by red, green, and blue channels, or any number of channels, as when using multiband satellite imagery.

We might think of the model this way: the CNN layers before the dense layer learned how to act as a function producing an output vector from the input image. The true classifier is the dense layer at the top, but it works well because the CNN learned the classifier (dense layer) while simultaneously learning the mapping function.

I stated earlier that higher layers in the CNN pay attention to ever larger parts of the input. We can see that this is so by considering the portion of the input that influences the output of a kernel at a deeper layer. [Figure 5-8](#) demonstrates this effect.

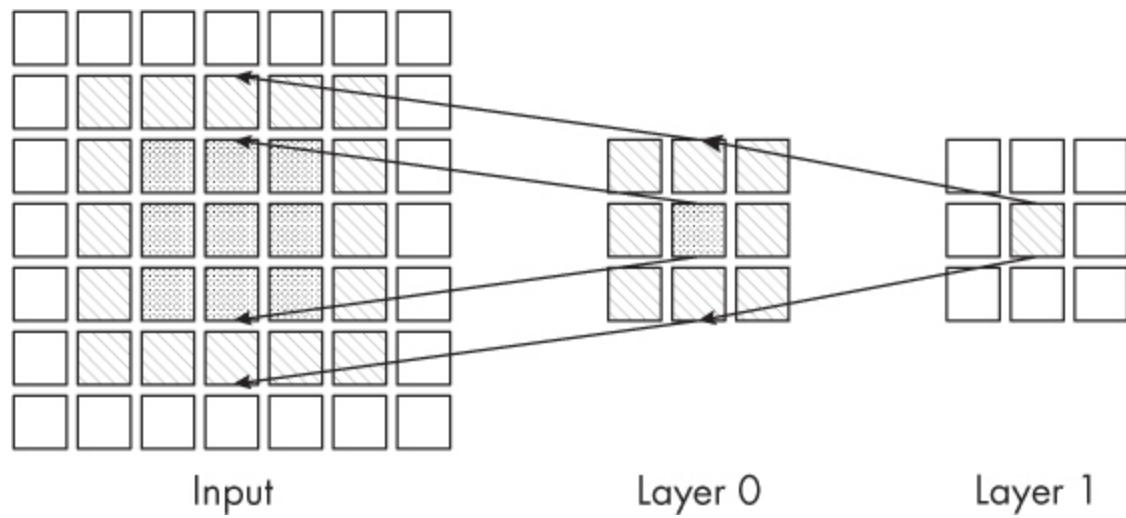


Figure 5-8: The part of the input affecting deeper layers of the model

Begin on the right side of the image. The 3×3 grid of squares represents the output of a kernel at convolutional layer 1. We want to know what portion of the input influences the value of the shaded pixel. Looking at the previous convolutional layer, layer 0, we see that the layer 1 output depends on the nine shaded values coming from the layer before.

The nine shaded values of convolutional layer 0 depend on the 5×5 shaded region of the input. It's 5×5 because each of the nine values is found by sliding a 3×3 kernel over the shaded 5×5 region of the input. For example, the dotted portion of the middle value in layer 0 comes from the similarly shaded 3×3 region of the input. In this way, higher CNN layers are affected by larger and larger portions of the input. The technical term for this is the *effective receptive field*, where the effective receptive field of the rightmost shaded value in [Figure 5-8](#) is the 5×5 shaded region of the input.

It's time for an experiment. We now have a handle on how CNNs function, so let's put that knowledge to work to compare a traditional neural network with a convolutional model. Which will win? I suspect you already know the answer, but let's prove it and gain some experience along the way.

We need a dataset. Let's use a grayscale version of CIFAR-10. This is a better choice than the dinosaur footprint dataset we used in the previous two chapters because the footprint images are outlines devoid of texture and background, and a CNN will not learn much more from such images than a traditional model. As we learned in [Chapter 3](#), CIFAR-10 contains 32×32 -pixel images of animals and vehicles, which will likely be more challenging.

We'll train three models: a random forest, a traditional neural network, and a convolutional neural network. Is this sufficient? We've come to appreciate that all three of these models involve randomness, so training once might not give us a fair representation of how each model performs. After all, we might get a lousy initialization or mix of trees that would throw one of the models off. Therefore, let's train each model 10 times and average the results.

This experiment will help us understand the differences in performance between the models, but we can learn still more about the neural networks by tracking their errors as training progresses. The result is a graph, which I'll present and then explain shortly. Before that, however, let me lay out the details of the models.

The training and test datasets are the same for each model. The traditional neural network and the random forest require vector inputs, so each 32×32 -pixel image is unraveled into a vector of 1,024 numbers. The CNN works with the actual two-dimensional images. There are 50,000 images in the training set, 5,000 for each of the 10 classes, and 10,000 images in the test set, 1,000 per class.

The random forest uses 300 trees. The traditional neural network has two hidden layers of 512 and 100 nodes, respectively. The CNN is more complex, with four convolutional layers, two pooling layers, and a single dense layer of 472 nodes. Even though the CNN has many more layers, the total number of weights and biases to learn is nearly identical to the traditional model: 577,014 versus 577,110.

We'll train the neural networks for 100 epochs, meaning 100 passes through the full training set. Fixing the minibatch size at 200 gives us 250 gradient descent steps per epoch. Therefore, during training, we'll update the weights and biases of the networks 25,000 times. At the end of each epoch, we'll capture the error made by the model on both the training and test sets. When the dust settles, a single graph will reveal everything we want to know.

[Figure 5-9](#) is that graph. It's the most complex graph we've seen, so let's walk through it in detail, beginning with the axes.

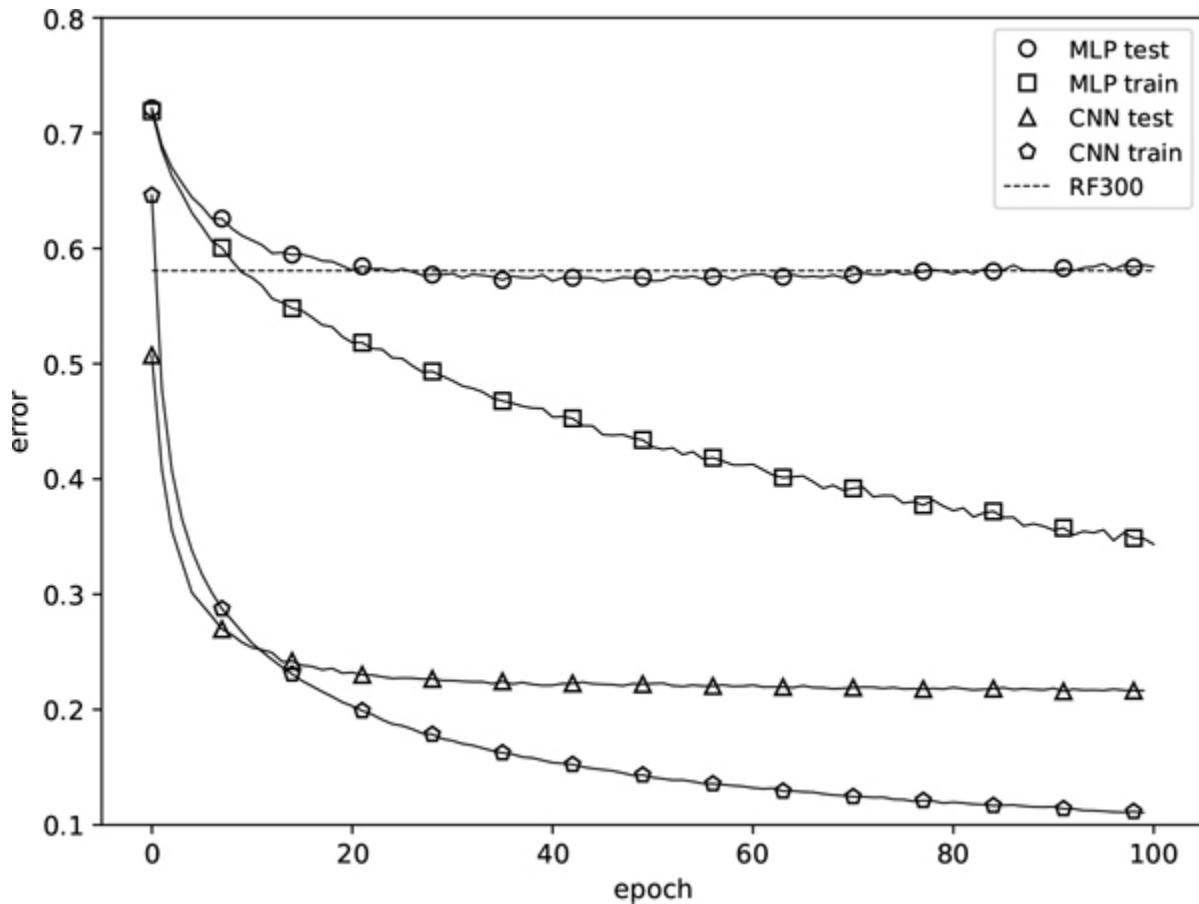


Figure 5-9: CIFAR-10 results for a CNN, MLP, and random forest

The label on the horizontal axis (x-axis) is “epoch,” which means a complete pass through the training set. Therefore, the graph shows things changing during training after every epoch. We also know that each epoch represents 250 gradient descent steps. The vertical axis (y-axis) is labeled “error” and runs from 0.1 to 0.8. This axis represents the fraction of the test or training samples that the model gets wrong. The lower the error, the

better. A decimal value of 0.1 means 10 percent, and a value of 0.8 means 80 percent.

The legend in the upper-right corner of the graph tells us that the circles and squares relate to the MLP, the traditional neural network, while the triangles and pentagons refer to the CNN. Specifically, the circles and triangles track the error on the test set for the MLP and CNN, respectively, as the models train. Similarly, the squares and pentagons track the error on the training set. Recall that the model's performance on the training set is used to update the weights and biases. The test set is used for evaluation and does not contribute to how the model is trained.

The MLP plots show us how well the model learned the training set (squares) and the test set (circles) as training continued, epoch after epoch. It's immediately apparent that the model learned the training set better than the test set because the training set error decreases continuously. This is what we expect. The gradient descent algorithm will update the weights and biases of the MLP, all 577,110 of them, to arrive at a lower and lower error on the training set. However, we're not interested in reaching zero error on the training set; instead, we want the smallest error possible on the test set because that

gives us a reason to believe that the MLP has learned to generalize.

Now consider the circle plot showing us the test set error. It reaches a minimum of about 0.56, or 56 percent, at around 40 epochs. After that, the error increases slowly but steadily, up to 100 epochs. This effect is classic MLP overfitting. The training set error continues to decrease, but the test set error hits a minimum and continues to increase after that. [Figure 5-9](#) tells us that stopping training at 40 epochs would have given us the best-performing MLP.

We'll get to the CNN results, but for the moment, consider the dashed line at 58 percent error. It's labeled "RF300" and shows us the test set error from a random forest with 300 trees. The random forest doesn't learn by updating weights over epochs, so the 58 percent error is just that: the model's error. I plotted it as a dashed line parallel to the horizontal axis so you can see that, briefly, the MLP did slightly better than the random forest, but by 100 epochs, the difference between the two models was negligible. In other words, we might take it that classical machine learning's best effort on the grayscale CIFAR-10 dataset is an error of about 56 to 58 percent. That's not a good result. Additional time spent with the parameters of the random forest, or the MLP, or starting over with a support vector machine

might lead to a slight reduction in the error. Still, it's unlikely to overcome the fact that classical machine learning cannot do much with this dataset.

Finally, consider the CNN's training (pentagon) and test (triangle) curves. By 100 epochs, the CNN is right around 11 percent error on the training set and, more importantly, about 23 percent on the test set. In other words, the CNN is right 77 percent of the time, or nearly 8 times in 10. Random guessing will be correct about 10 percent of the time on a 10-class dataset, so the CNN has learned rather well, and far better than the MLP or random forest.

This is precisely the point of convolutional neural networks: by learning to represent the parts of the objects in an image, it becomes possible to learn a new representation (formally known as an *embedding*) that the dense layers of the network can successfully classify.

The first CNN I trained, in 2015, attempted to detect small airplanes in satellite images. My initial, non-CNN approach worked, but it was noisy with many false positives (fake detections). The airplanes were there, but so were many other things that were not airplanes. I then trained a simple CNN like the one used in this experiment. It located the airplanes with

ease, and virtually nothing but the airplanes. I was dumbfounded and realized then that deep learning was a paradigm shift. I'll argue in [Chapter 7](#) that as of fall 2022, a new, more profound paradigm shift has occurred, but we have some ground yet to cover before we're ready for that discussion.

The simple CNNs of this chapter don't do justice to the zoo of available neural network architectures. A decade of fevered development has resulted in a few go-to CNN architectures, some with over 100 layers. The architectures have names like ResNet, DenseNet, Inception, MobileNet, and U-Net, among many others. The U-Net is worthy of a few words.

The CNNs we've explored so far accept an input image and return a class label like "dog" or "cat." It doesn't need to be this way. Some CNN architectures implement [semantic segmentation](#), where the output is another image with every pixel labeled by the class to which it belongs. U-Nets do this. If every pixel of the dog is marked "dog," extracting the dog from the image becomes trivial. A middle ground between a U-Net and CNNs that assign a single label to the entire image is a model that outputs a [bounding box](#), a rectangle surrounding the detected object. The pervasiveness of AI means that you've

likely already seen images with labeled bounding boxes. YOLO (“you only look once”) is a popular architecture producing labeled bounding boxes; Faster R-CNN is another.

We focused on image inputs here, but the input need not be an image. Anything representable in an image-like format, where there are two dimensions and structure within those dimensions, is a candidate for a 2D CNN. A good example is an audio signal, which we usually think of as one-dimensional, a voltage changing over time that drives the speaker. However, audio signals contain energy at different frequencies. The energy at different frequencies can be displayed in two dimensions: the horizontal dimension is time, and the vertical dimension is frequency, usually with lower frequencies at the bottom and higher frequencies at the top. The intensity of each frequency becomes the intensity of a pixel to transform the audio signal from a one-dimensional, time-varying voltage into a two-dimensional spectrogram, as shown in [Figure 5-10](#).

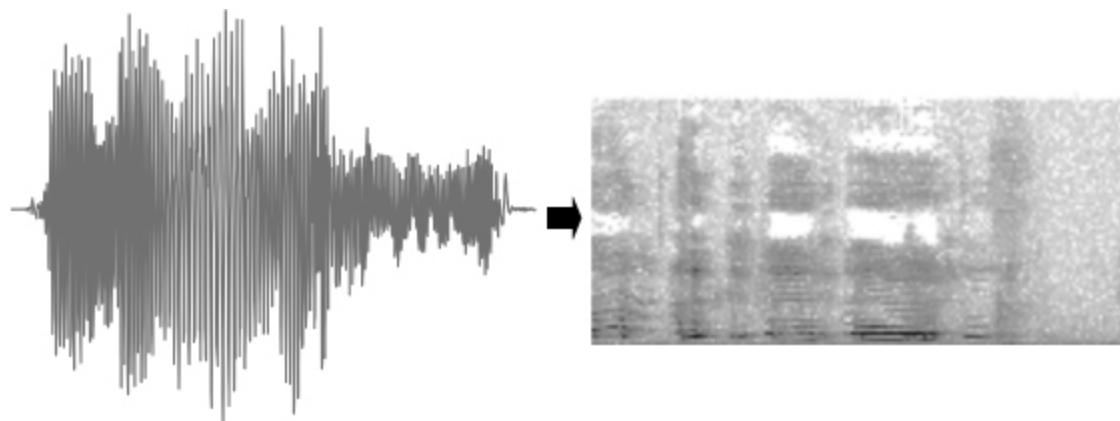


Figure 5-10: Mapping from one-dimensional data to a two-dimensional image

The spectrogram, here of a crying baby, contains a wealth of information and structure that the CNN can learn about to produce a better model than is possible with the one-dimensional audio signal alone. The key observation is that any transformation of the input data that extracts structure in a form amenable to a CNN is fair game.

You have a dataset and need to build a CNN. What architecture should you use? What should the minibatch size be? What layers do you need, and in what order? Should you use 5×5 or 3×3 convolutional kernels? How many epochs of training is enough? Early on, before the development of standard architectures, each of those questions had to be answered by the person designing the network. It was a bit like medicine of the past: a mix of science, experience, and intuition. The art of neural networks meant that practitioners were in high demand, and it was difficult for savvy software engineers to add deep learning to their repertoires. Some people wondered if software could be used to determine the model's architecture and training parameters automatically (that is, its hyperparameters,

introduced in [Chapter 3](#)). And so, automatic machine learning, or [*AutoML*](#), was born.

Most cloud-based commercial machine learning platforms, like Microsoft’s Azure Machine Learning or Amazon’s SageMaker Autopilot, include an AutoML tool that will create the machine learning model for you; you need only supply the dataset.

AutoML applies to more than just neural networks, and many tools include classical machine learning models as well.

AutoML’s entire purpose is to locate the best model type for the supplied dataset with a minimum of user expertise required.

I want to argue that AutoML only goes so far and that the best deep learning practitioners will always outperform it, but that argument rings hollow. It reminds me of the assembly language programmers of old pontificating on the impossibility of compilers ever producing code that was as good as or better than what they could produce. There are few job openings these days for assembly language programmers, but tens of thousands for programmers using compiled languages (at least for now; see [Chapter 8](#)). That said, some of us still prefer to roll our own models.

A consequence of the deep learning revolution was the creation of powerful, open source machine learning toolkits with names like TensorFlow and PyTorch. Implementing a traditional, fully connected neural network is an exercise for machine learning students. It's not trivial, but it's something most people can accomplish with effort. Properly implementing a CNN, on the other hand, especially one supporting a multitude of layer types, is anything but trivial. The AI community committed early on to developing open source toolkits supporting deep learning, including CNNs. Without these toolkits, progress in AI would be painfully slow. Large tech companies like Google, Facebook (Meta), and NVIDIA also signed on, and their continued support for toolkit development is critical to AI.

What makes the toolkits powerful, besides the mountains of tested, high-performance code they contain, is their flexibility. We now appreciate that training a neural network, CNN or otherwise, requires two steps: backpropagation and gradient descent. Backpropagation works only if the model's layers support a particular mathematical operation known as differentiation. Differentiation is what first semester calculus students learn. So long as the toolkits can automatically determine the derivatives (what you get when you differentiate), they allow users to implement arbitrary layers.

The toolkits employ *automatic differentiation* by transforming the neural network into a computational graph.

It's tempting to take a few steps down the path of automatic differentiation and computational graphs because the elegance and flexibility therein is a beautiful marriage of mathematics and computer science. Unfortunately, you'll need to take my word for it because the level of detail necessary is far beyond what we can explore in this book. One key point is that there are two primary approaches to automatic differentiation: forward and reverse. Forward automatic differentiation is easier to conceptualize and implement in code but is unsuited to neural networks. That's too bad, in a way, because forward automatic differentiation is best implemented using dual numbers, an obscure type of number invented (discovered?) by English mathematician William Clifford in 1873. These were a prime example of math for math's sake and largely forgotten until the age of computers, when they were suddenly made useful. Reverse automatic differentiation is best for neural networks but doesn't use dual numbers.

This chapter was challenging. We dove more deeply into the details than we did in previous chapters or will in the following

ones. A summary is definitely required. Convolutional neural networks:

- Thrive on structure in their inputs, which is the complete opposite of classical machine learning models
- Learn new representations of their inputs by breaking them into parts and groups of parts
- Use many different kinds of layers combined in various ways
- Can classify inputs, localize inputs, or assign a class label to every pixel in their inputs
- Are still trained via backpropagation and gradient descent, just like traditional neural networks
- Drove the creation of powerful, open source toolkits that democratized deep learning

Convolutional neural networks follow in the tradition of classical machine learning models: they take an input and assign to it, in some fashion, a class label. The network operates as a mathematical function, accepting an input and producing an output. The next chapter introduces us to neural networks that generate output without input.

To paraphrase an old television show: you're traveling through another dimension, a dimension not only of sight and sound

but of mind, a journey into a wondrous land whose boundaries are that of imagination—next stop, generative AI.

KEY TERMS

automatic differentiation, AutoML, bounding box, computational graph, convolution, convolutional layer, convolutional neural network, dense layer, effective receptive field, embedding, end-to-end learning, filter, kernel, pooling layer, semantic segmentation

6

GENERATIVE AI: AI GETS CREATIVE



Generative AI is an umbrella term for models that create novel output, either independently (randomly) or based on a prompt supplied by the user. Generative models do not produce labels but text, images, or even video. Under the hood, generative models are neural networks built from the same essential components.

We'll focus on three kinds of generative AI models: generative adversarial networks, diffusion models, and large language models. This chapter covers the first two. Large language models have recently turned the world of AI on its head. They are the subject of [Chapter 7](#).

Generative adversarial networks (GANs) consist of two separate neural networks trained together. The first network is the generator. Its task is to learn how to create fake inputs for the discriminator. The discriminator's task is to learn how to differentiate between fake and real inputs. The goal of training the two networks together is that the generator becomes better at faking out the discriminator while the discriminator tries its best to differentiate real from fake.

At first, the generator is terrible. It outputs noise, and the discriminator has no difficulty distinguishing between real and fake. However, the generator improves over time, making the discriminator's job increasingly harder; this in turn pushes the discriminator to become a better real versus fake detector. When training is declared complete, the discriminator is usually discarded, and the now-trained generator is used to produce new output sampled randomly from the learned space of the training data.

I haven't specified *what* the training data is, because all we need to know for now is that a GAN is constructed from two competing (adversarial) networks. For most applications, it's the generator we want when all is said and done.

Structurally, we can imagine a GAN like the blocks in [Figure 6-1](#). (I'll explain the random vector part in time.) Conceptually, we see that the discriminator accepts two kinds of inputs: real data and the output of the generator. The discriminator's output is a label: "Real" or "Fake." Standard neural network training using backpropagation and gradient descent trains the generator and discriminator together, but not simultaneously.

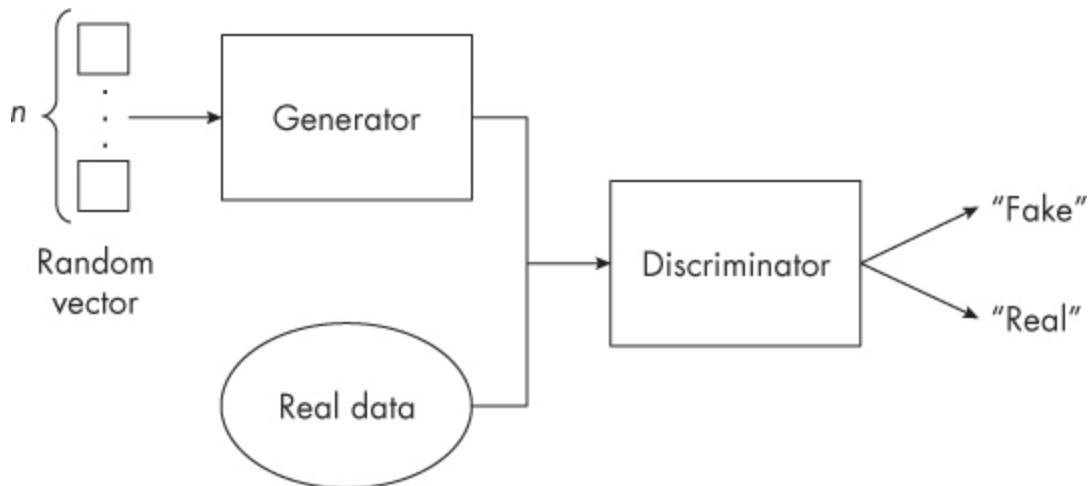


Figure 6-1: Conceptualizing the architecture of a generative adversarial network

For example, training with a minibatch of real data—a small subset of the available real training data—follows these steps:

1. Use the generator as it currently is to create a minibatch's worth of fake data.
2. Grab a minibatch's worth of real data from the training set.

3. Unfreeze the discriminator's weights so gradient descent can update them.
4. Pass the fake and real samples through the discriminator with labels 0 and 1, respectively.
5. Use backpropagation to take a gradient descent step to update the discriminator's weights.
6. Freeze the discriminator so the generator can be updated without altering the discriminator.
7. Create a minibatch's worth of generator inputs (the random vector in [Figure 6-1](#)).
8. Pass the generator inputs through the combined model to update the generator's weights. Mark each of the generator inputs as being real.
9. Repeat from step 1 until the full model is trained.

The algorithm first updates the discriminator's weights using the generator as it currently is (step 5), then freezes them (step 6) so the generator's weights can be updated without altering the discriminator. This approach is necessary because we want the output of the discriminator—the “Real” or “Fake” labels—to update the generator portion. Notice that the generator update marks all the fake images as real. Doing this scores the generator by how real the fake inputs appear to the discriminator.

Let's examine the random vector used as input to the generator. The point of a GAN is to learn a representation of the training set that we can think of as a data generator, like the data-generating process that produced the real training set.

However, in this case, the data generator can be viewed as a function that takes a random collection of numbers, the random vector, and transforms them into an output that might plausibly have come from the training set. In other words, the generator acts like a data augmentation device. The random input to the generator becomes an example of the training set. In effect, the generator is a proxy for the actual data-generating process that created the real training set in the first place.

The random vector of numbers is drawn from a probability distribution. Sampling from a probability distribution is akin to rolling two dice and asking how likely it is that their sum is a seven versus a two. It's more likely that the sum is a seven because there are more ways to add the two numbers and get seven. There's only one way to get two: snake eyes. Sampling from a normal distribution is similar. The most common sample returned is the average value of the distribution. Values on either side of the average are less likely the further away from the average they are, though still possible.

For example, [Figure 6-2](#) shows a bar plot of the distribution of human heights in inches. The original dataset contained the heights of 25,000 people, which were then fit into the 30 bins of the figure. The higher the bar, the more people fell into that bin.

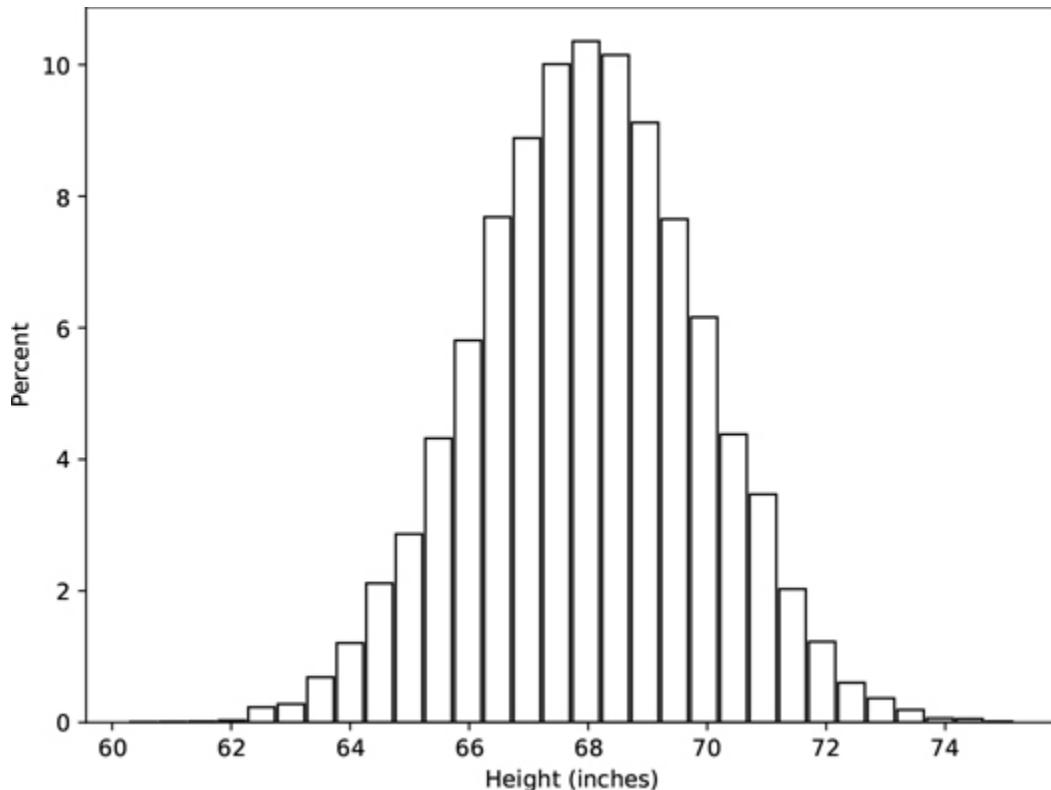


Figure 6-2: The distribution of human height

Note the shape of the histogram, which looks like a bell—hence its somewhat old-fashioned name, the bell curve. Its modern name, the *normal distribution*, is due to it showing up so often in nature that it's the distribution normally encountered, especially for data generated by a physical process. From the distribution, we see that the height of a randomly selected

person will most often be around 68 inches: more than 10 percent of the sampled population fell into that bin.

The random vector used by a GAN, also known as the *noise vector*, works the same way. The average, in this case, is zero, with most samples in the range -3 to 3 . Also, each of the n elements in the vector follows this range, meaning the vector itself is a sample from an n -dimensional space, not the one-dimensional space of [Figure 6-2](#).

The need for labeled datasets is a bane of machine learning. GANs have no such restriction. We don't care what a training sample's class is, only that it's an instance of real data, regardless of the class label. Of course, we still require that the training set reflect the kind of data we want to generate, but the training set need not be labeled.

Let's build a generative adversarial network using our old friend, the MNIST digits dataset. The generator will learn to transform a random set of 10 numbers (meaning n is 10) into a digit image. Once trained, we can give the generator any collection of 10 values around zero, and the generator will produce a new digit image as output, thereby mimicking the

process that created the MNIST dataset: people writing digits on paper by hand. A trained GAN generator produces an infinite supply of the target output.

We'll use a simple GAN based on traditional neural networks to create a generator for an infinite supply of MNIST-style digit images. First, we'll unravel the existing MNIST training set so each sample is a 784-dimensional vector, just as we did in [Chapter 5](#). This gives us the real data. To create fake data, we need 10-element random vectors that we'll build by drawing 10 samples from a normal distribution with an average value of zero.

The generator portion of the model accepts a 10-element noise vector as input and produces a 784-element output vector representing the synthesized digit image. Recall that the 784 numbers can be rearranged into a 28×28 -pixel image. The generator model has three hidden layers, with 256, 512, and 1,024 nodes, and an output layer of 784 nodes to produce the image. The hidden layer nodes use a modified version of the rectified linear unit called a [leaky ReLU](#). Leaky ReLU activations output the input if the input is positive, but if the input is negative, the output is a small positive value multiplied by the negative input. In other words, they leak a bit. The output layer uses a hyperbolic tangent activation function, meaning every

one of the 784 output elements will be in the range -1 to $+1$. That's acceptable. We can scale the values to 0 to 255 when writing an image to disk.

The generator must map between the random noise vector input and an output image. The discriminator must take an image as input, implying a 784-dimensional vector. The discriminator has three hidden layers, like the generator, but in reverse: 1,024 nodes, then 512 nodes, followed by 256 nodes. The discriminator's output layer has one node with a sigmoid activation function. The sigmoid produces values from 0 to 1, which we can interpret as the discriminator's belief that the input is real (output near 1) or fake (output near 0). Notice that the network uses nothing more than standard fully connected layers. Advanced GANs use convolutional layers, but exploring the details of those networks is outside our scope.

[Figure 6-3](#) shows the generator (top) and discriminator (bottom). The symmetry between the two is evident in the numbers of nodes in the hidden layers, though notice that the order is reversed in the discriminator.

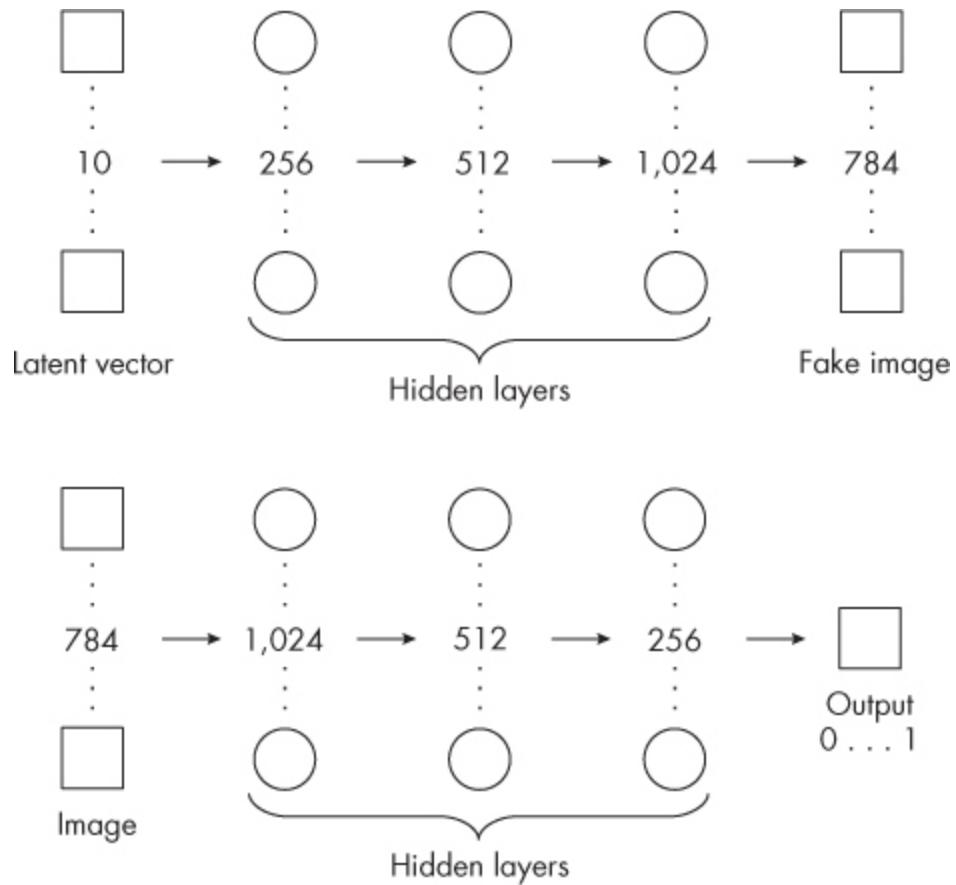


Figure 6-3: GAN generator (top) and discriminator (bottom)

The generator accepts a 10-element random vector as input and produces a 784-element fake image output vector. The discriminator accepts an image vector, real or fake, and outputs a prediction, a number from 0 to 1. Fake images should produce values close to 0 and real images values close to 1. If the generator is well trained, the discriminator will be fooled most of the time, meaning the discriminator's output will be close to 0.5 for all inputs.

The entire network is trained for 200 epochs of 468 minibatches each, for a total of 93,600 gradient descent steps. We can display samples from the generator after each epoch to observe the network as it learns. [Figure 6-4](#) shows samples after epochs 1, 60, and 200, from left to right.



Figure 6-4: Generator output after epochs 1, 60, and 200

As we'd expect, the generator performs poorly after a single pass through the training data, but perhaps not as poorly as we might have thought. Most of the generated images look like ones; other digit shapes, like zeros and twos, are also present, though noisy.

After 60 epochs, the generator produces a full range of digits. Some are spot on, while others are still confused or only partially drawn. After 200 epochs, most of the digits are distinct and sharply defined. The generator is trained and now available to produce digit images on demand.

Our digit generator will happily create 10,000 new digit images for us, but what if we want all those digits to be fours? A random input vector produces a random digit, but we don't get to choose which one. If we select input vectors randomly, we can be excused for believing that the mix of output digits will be similarly random. I tested that assumption by using the trained generator to create 1,000 digit images. I then passed those digit images to a convolutional network trained on the MNIST dataset. The convolutional network has a test set accuracy above 99 percent, giving us confidence in its predictions, assuming the input is a digit image. The GAN generator produces realistic digit images, so we're on solid ground.

Assuming the generator is acting as we expect, the percentage of each digit should, naively, be the same. There are 10 possible digits, so we expect each to appear about 10 percent of the time. That's not what happened. [Table 6-1](#) shows the actual distribution of occurrences of each digit.

Table 6-1: The Actual Digit Distribution

Digit	Percentage
0	10.3
1	21.4
2	4.4
3	7.6
4	9.5
5	6.0
6	9.1
7	14.4
8	4.4
9	12.9

The generator favors ones, followed by sevens, nines, and zeros; eights and twos are the least likely outputs. So, not only does the

GAN not allow us to select the desired digit type, it has definite favorites. Review the leftmost image in [Figure 6-4](#), showing the epoch 1 samples. Most of those digits are ones, so the GAN's predilection for ones was evident from the beginning of training. The GAN learned, but the preponderance of ones is a symptom of a problem that sometimes plagues GAN training: namely [*mode collapse*](#), where the generator learns early on how to create a particularly good example or set of examples that fool the discriminator and gets trapped into producing only that output and not the desired diversity of images.

We need not throw ourselves on the mercy of a finicky, uncontrollable GAN. Instead, we can condition the network during training by passing in an indication of the type of digit we want the generator to create. GANs that take this approach are known as [*conditional GANs*](#). Unlike unconditional GANs, they require training sets with labels.

In a conditional GAN, the input to the generator is still a random noise vector, but attached to it is another vector specifying the desired output class. For example, the MNIST dataset has 10 classes, the digits 0 through 9, so the conditional vector has 10 elements. If the desired class is the digit 3, the conditional vector is all zeros except for element 3, which is set to one. This method of representing class information is known

as [one-hot encoding](#) because all the elements of the vector are zero except for the element corresponding to the desired class label, which is one.

The discriminator also needs the class label. If the input to the discriminator is an image, how do we include the class label? One way is to expand the concept of one-hot encoding to images. We know that a color image is represented by three image matrices, one for the red channel, one for the green channel, and one for the blue channel. Grayscale images have only one channel. We can include the class label as a set of additional input channels where all the channels are zero except for the channel corresponding to the class label, which is one.

Including the class label when generating and discriminating between real and fake inputs forces each part of the entire network to learn how to produce and interpret class-specific output and input. If the class label is 4 and the digit produced by the generator looks more like a zero, the discriminator will know there's a class mismatch because it knows about true zeros from the labeled training set.

The benefit of a conditional GAN comes when using the trained generator. The user supplies the desired class as a one-hot

vector, along with the random noise vector used by an unconditional GAN. The generator then outputs a sample based on the noise vector, but conditioned on the desired class label. We can think of a conditional GAN as a set of unconditional GANs, each trained on a single class of images.

I trained a conditional GAN on the MNIST dataset. For this example, the GAN used convolutional layers instead of the fully connected layers used earlier in the chapter. I then asked the fully trained generator to produce 10 samples of each digit, as shown in [Figure 6-5](#).



Figure 6-5: The conditional GAN output showing samples for each digit

Conditional GANs let us select the desired output class, which unconditional GANs cannot do, but what if we want to adjust specific features of the output image? For that, we need a controllable GAN.

Uncontrollable GANs generate images willy-nilly without regard for the class label. Conditional GANs introduce class-specific image generation, which is helpful if we want to use a GAN to generate synthetic imagery for training other models, perhaps to account for a class for which we have relatively few examples. *Controllable GANs*, on the other hand, allow us to control the appearance of specific features in the generated images. When the generator network learns, it learns an abstract space that can be mapped to the output images. The random noise vector is a point in this space where the number of dimensions is the number of elements in the noise vector. Each point becomes an image. Put the same point, the same noise vector, into the generator, and the same image will be output.

Moving through the abstract space represented by the noise vector produces output image after output image. Might there be directions in the abstract noise space that have meaning for

the features in the output image? Here, *feature* means something in the image. For example, if the generator produces images of human faces, a feature might be whether the face is wearing glasses, has a beard, or has red hair.

Controllable GANs uncover meaningful directions in the noise space. Moving along one of those directions alters the feature related to the direction. Of course, the reality is more complex because a single direction might affect multiple features, depending on the dimensionality of the noise space and the data learned by the generator. In general, smaller noise vectors are more likely to be *entangled*, meaning single noise vector dimensions affect multiple output features, making it difficult to discern interesting directions. Some training techniques and larger noise vectors, perhaps with 100 elements instead of the 10 we used earlier, improve the model's chance of assigning interesting feature adjustments to a single direction. Ideally, there would be a meaningful feature adjustment for a single noise vector element.

Let's walk through a two-dimensional example to drive the idea home. Learning a generator using a two-dimensional noise vector might be difficult, but the concept applies to all dimensionalities and is straightforward to illustrate in two dimensions. [Figure 6-6](#) has what we need.

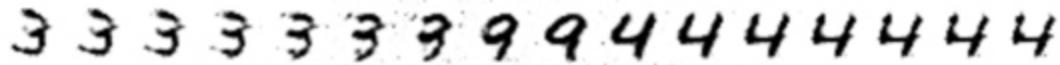
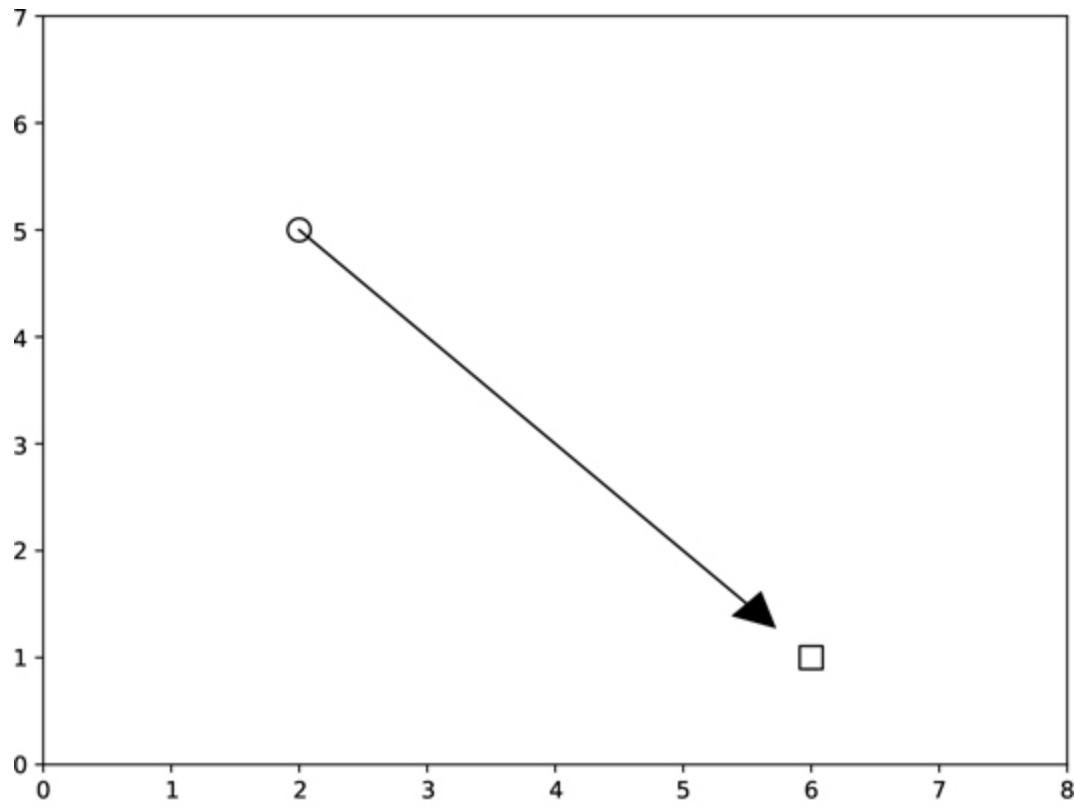


Figure 6-6: Moving through a two-dimensional noise space and interpolated MNIST digits

The top part of the figure shows a two-dimensional noise space for a generator with two inputs, the x -coordinate and the y -coordinate. Therefore, each point in the figure represents an image generated by the GAN. The first image is produced from the point at $(2, 5)$ (the circle). A second image comes from the point at $(6, 1)$ (the square). The arrow shows a direction through the noise space that we somehow learned controls a feature in the output image. If the GAN generates faces, it might be that

the arrow points in a direction that affects the person's hair color. Moving from the point at (2, 5) to the point at (6, 1) maintains most of the output image but changes the hair color from, say, black at (2, 5) to red at (6, 1). Points along the arrow represent hair colors intermediate between black and red.

The bottom of [Figure 6-6](#) shows interpolation along the third dimension of the GAN we trained to generate digit images. From left to right, a three morphs briefly into a nine before becoming a four, as the third element of the 10-element noise vector is varied while keeping all the others fixed at their initial random values. The noise vector is of relatively low dimensionality, implying that it's unlikely any one dimension is associated with only a single digit trait, which is why the whole image changes from an initial three through a nine to a four.

Sophisticated GANs can produce realistic yet fake images of human faces. Controllable versions learn directions linked to specific facial features. For example, consider [Figure 6-7](#), which shows two generated fake faces on the left and adjusted faces on the right (from Yujun Shen et al., “Interpreting the Latent Space of GANs for Semantic Face Editing,” 2019). The adjustments correspond to movement through the noise space from the original image position along learned directions representing age, glasses, gender, and pose.

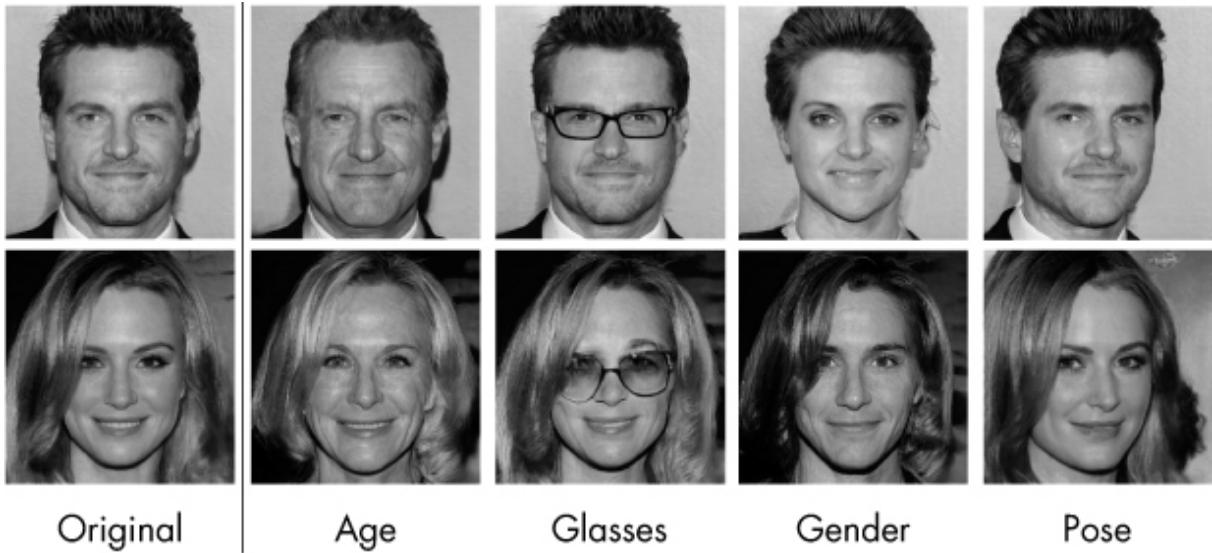


Figure 6-7: Controlling face attributes

The power of controllable GANs is genuinely remarkable, and that the generator learns meaningful directions through the noise space is impressive. However, GANs are not the only way to create realistic and controllable images. Diffusion models likewise generate realistic imagery; moreover, imagery conditioned by user-defined text prompts.

Generative adversarial networks rely on competition between the generator and the discriminator to learn to create fake outputs similar to the training data. [Diffusion models](#) represent a competition-free approach to the same end.

In a nutshell, training a diffusion model involves teaching it to predict noise added to a training image. Inference in a diffusion model involves the opposite, turning noise into an image. Great! But what is “noise” when it comes to images?

Noise implies randomness, something without structure. You’re in the ballpark if you’re thinking of static on a radio or hiss in an audio signal. For a digital image, noise means random values added to the pixels. For example, if the pixel value should be 127, noise adds or subtracts a small amount so that the value becomes, say, 124 or 129. Random noise added to an image often looks like snow. Diffusion models learn how to predict the amount of normally distributed noise added to a training image.

We must have several things in place before we train the network. First, we need a training dataset. Diffusion models learn from data, like all neural networks. As with GANs, labels are not required until we want some say in what the trained model will generate.

Once we have the training data, we need a neural network architecture. Diffusion models are not picky here, but the selected architecture must accept an image as input and

produce a same-sized image as output. The U-Net architecture mentioned briefly in [Chapter 5](#) is a frequent choice.

We have data and an architecture; next, we need some way to get the network to learn. But learn what? As it happens, forcing the network to learn the noise added to an image is all that is required. The math behind this realization isn't trivial. It involves probability theory, but in practice, it boils down to taking a training image, adding some known level of normally distributed noise, and comparing that known noise to what the model predicts. If the model learns to predict the noise successfully, we can later use the model to turn pure noise into an image similar to the training data.

The important part of the previous paragraph is the phrase "known level of normally distributed noise." Normally distributed noise can be characterized by a single parameter, a number specifying the level of the noise. Training consists of selecting an image from the training set and a level of noise, both at random, and passing them as inputs to the network. The output from the network is the model's estimate of the amount of noise. The smaller the difference between the output noise (itself an image) and the added noise, the better. Standard backpropagation and gradient descent are applied to minimize

this difference over minibatches until the model is declared trained.

How noise is added to training images affects how well and how quickly models learn. Noise generally follows a fixed *schedule*. The schedule is such that moving from a current noise level, say noise level 3, to the next, level 4, adds a specified amount of noise to the image, where the amount of noise depends on a function. If the same amount of noise is added between each step, the schedule is linear. However, if the amount of noise added between steps depends on the step itself, it is nonlinear and follows some other function.

Consider [Figure 6-8](#), which shows a possible training image on the left. Each row shows successive levels of noise added to the training image. The top row follows a linear schedule, where moving left to right adds the same noise level between each step until the image is almost destroyed. The bottom row follows what is known as a cosine schedule, which destroys the image less rapidly. This helps diffusion models learn a bit better. For the curious, the dapper gentleman in the image is my great-grandfather, Emil Kneusel, circa 1895.



Figure 6-8: Two ways to turn an image into noise: linear (top) and cosine (bottom)

[Figure 6-8](#) presents only nine steps. In practice, diffusion models use hundreds of steps, the critical point being that the original image is destroyed at the end of the process, leaving only noise. This matters because sampling from the diffusion model reverses the process to turn a random noise image into a noise-free image. In effect, sampling from the diffusion model moves from right to left using the trained network to predict noise that is then subtracted to produce the previous image. Repeating this process for all the steps in the schedule completes the noise-to-image generation process.

The description in the previous section can be summarized in two algorithms. I encourage you to read through them, but as they are a bit technical, skipping ahead to the next section is always an option.

The forward algorithm trains the diffusion model, and the reverse algorithm samples from a trained model during inference to produce output images. Let's begin with the forward algorithm. We repeat the following until we declare the model trained:

1. Pick a training image, x_0 , at random.
2. Pick a random time step, t , in the range 1 through T , the maximum number of steps.
3. Sample a noise image, e , from a standard normal distribution.
4. Define a noisy image, x_t , using x_0 , t , and e .
5. Pass x_t through the model and compare the output noise estimate to e .
6. Apply standard backpropagation and gradient descent to update the model's weights.

The forward algorithm works because there is a straightforward way to get x_t from x_0 , the image in the training set, and a randomly selected time step, t . Here, T is the maximum possible time step, at which the training image has been turned into pure noise. Typically, T is several hundred steps. Recall that the diffusion model is trying to learn how to predict the noise in e . The act of repeatedly forcing the model to

get better and better at predicting the noise used to corrupt the training image is what lets the reverse step work.

The reverse algorithm samples from the diffusion model trained by the forward algorithm to generate a novel output image, beginning with a pure noise image in x_T (think the rightmost images in [Figure 6-8](#)). The diffusion model is used for T steps to turn noise into an image by repeating the following:

1. If this isn't the last step from x_1 to x_0 , sample a noise image, z , from a standard normal distribution.
2. Create x_{t-1} from x_t by subtracting the output of the diffusion model from x_t and adding z .

The reverse algorithm moves from right to left, if thinking in terms of [Figure 6-8](#). Each step to the left is found by subtracting the output of the diffusion model using the current image as input, thereby moving from time step t to the previous time step, $t - 1$. The standard noise image, z , ensures that x_{t-1} is a valid sample from the probability distribution supplying x_{t-1} from x_t . As mentioned, we're skipping a lot of probability theory.

The sampling algorithm works because the diffusion model estimates the noise in its input. That estimate leads to an

estimate of the image that, plausibly, created x_t from x_{t-1} . Iterating for all T steps brings us, ultimately, to x_0 , the output of the network. Notice that unlike our previous networks, which had an input and produced an output, diffusion models are run repeatedly, each time producing less and less noisy images, until finally they produce an image similar to the training data.

Diffusion models are like standard GANs: unconditional. The image generated is not controllable. You might suspect that if a GAN can be conditioned in some way to guide the generation process, then a diffusion model might be similarly directable. If so, you're right.

The GAN we used to generate MNIST-like digit images was conditioned by extending the input to the generator with a one-hot vector selecting the desired class label. Conditioning a diffusion model isn't quite that simple, but it is possible to supply the network with a signal related to the image during training. Typically, that signal is an embedding vector representing a text description of the training image's contents. We briefly encountered embeddings in [Chapter 5](#) and will do so again in [Chapter 7](#) when discussing large language models.

All we need to know for now is that a text embedding takes a string like “A big red dog” and turns it into a large vector, which we think of as a point in a high-dimensional space: a space that has captured meaning and concepts. The association of such a text embedding during training while the network is learning to predict noise in images conditions the network in much the same way that the one-hot class vector conditions a GAN generator.

After training, the presence of a text embedding when sampling provides a similar signal to guide the output image so that it contains elements related to the text. At sampling time, the text becomes a prompt, describing the image we want the diffusion process to generate.

Diffusion models typically begin with a random noise image. They need not. If we want the output to be similar to an existing image, we can use that image as the initial image, with some level of noise added. Samples from that image will be, depending on the degree of added noise, more or less similar to it. Now, let’s take a tour of conditional diffusion models.

Commercial diffusion models, such as DALL-E 2 by OpenAI or Stable Diffusion by Stability AI, use the text or image supplied by the user to guide the diffusion process toward an output image satisfying the prompt's requirements. The examples shown in this section were generated by Stable Diffusion using the DreamStudio online environment. [Figure 6-9](#) presents to us Leonardo da Vinci's *Mona Lisa* (upper left) along with five variations of it.



Figure 6-9: The Mona Lisa as imagined by Stable Diffusion

The variations are the products of Stable Diffusion in response to the original image and a text prompt:

Portrait of a woman wearing a brown dress in the style of DaVinci, soft, earthen colors

The DreamStudio interface lets the user supply an initial image, using a slider to set the amount of noise to add, from 0 percent for a pure noise image to 100 percent for no noise added. (Yes, that seems backward to me, too.) The noisy version of the image initializes the diffusion process. The higher the percentage, the less noise is added, and the more the initial image influences the final output. For the *Mona Lisa*, I used 33 percent. That noise level, along with the prompt and a user-selectable style, produced the five variations in [Figure 6-9](#). The only difference between the variations is the chosen style (top row: anime and fantasy art; bottom row: isometric, line art, and photographic).

The results are impressive. The images were neither painted nor drawn, but diffused from a noisy version of the *Mona Lisa* and a text prompt used as a guide to direct the diffusion process. It isn't difficult to appreciate that the ability to generate novel images in response to prompts will impact the commercial art world.

However, AI image generation isn't perfect. Errors happen, as demonstrated in [Figure 6-10](#). I promise I didn't ask for a five-legged border collie, a multi-mouthed *T. rex*, or a picture of a woman like the *Mona Lisa* with horribly mutated hands. Diffusion models seem to have particular difficulty rendering hands, much like human artists.



Figure 6-10: Diffusion model errors

Writing effective prompts has become an art form, one that has already created a new kind of job: prompt engineer. The exact form of the text prompt strongly influences the image generation process, as does the random noise image initially selected. The DreamStudio interface allows users to fix the pseudorandom number generator seed, meaning the diffusion process starts with the same noise image each time. Fixing the seed while slightly altering the text prompt lets us experiment to learn how sensitive the diffusion process can be.

The images in [Figure 6-11](#) were generated by permutations of the words *ornate*, *green*, and *vase*. (These images are shown in black and white in the book, but all are similar shades of green.) The initial noise image was the same each time; only the order of the three words varied. Three of the vases are similar, but the fourth is quite different. Nonetheless, all four are valid exemplars of ornate, green vases.



Figure 6-11: Vases generated by a diffusion model

Prompt order and phrasing matter because the embedding vector formed from the text prompt differs, even if the prompt words or their meanings are similar. The prompts for the first three vases likely landed close to each other in the text embedding space, explaining why they look much the same. The last prompt, for whatever reason, landed elsewhere, leading to the different qualities of the generated image. Interestingly, the prompt for the last image was “ornate, green, vase,” the form following grammatical convention.

Curious, I altered the prompt “ornate, green, vase,” changing “green” to other colors and using the same initial noise image as before. The results are in [Figure 6-12](#). From left to right, the colors specified were red, mauve, yellow, and blue. The first three images are similar to the last vase in [Figure 6-11](#); only the blue vase differs significantly.



Figure 6-12: Generated vases of many colors

I noticed another property of diffusion models during my experiments, namely, that the generated images have less noise than the originals. Suppose an input image is low resolution and grainy. In that case, the diffusion model's output is higher resolution and clear because the output is not the result of an operation applied to the original image but a reimagining of the image using the prompt for guidance. Might it be possible to use diffusion models to remove image artifacts if absolute fidelity to the original image isn't strictly required?

[Figure 6-13](#) tries to answer this question. The original 195×256-pixel image upscaled to 586×768 pixels (a factor of 3) is on the left. The image was upscaled using a standard image processing program and cubic interpolation. The diffusion model output, also 586×768 pixels, is on the right. The diffusion model output used the 195×256-pixel original image with 25 percent added noise, a photographic style, and the prompt “detailed, original.” The diffusion image is better. It’s not identical to the original, but a close copy. I don’t believe this approach competes with

deep learning-based super-resolution networks, but regardless of ultimate utility, it was an interesting application of diffusion models.



Figure 6-13: Diffusion model image enhancement

As another example, consider [Figure 6-14](#), which shows an image of a Western Meadowlark taken at a distance of about 100 meters through poor, smoky Colorado air (left). The center image represents a best effort at improving the image using a standard image manipulation program (Gimp). The version on the right is the output of Stable Diffusion when given the center image with a small amount of noise added (about 12 percent) and the following text prompt:

western meadowlark, highly detailed, high resolution, noise free



Figure 6-14: A diffusion model image enhancement experiment attempting to improve a smoke-obscured image of a Western Meadowlark: original (left), best effort with a standard image manipulation program (center), enhanced with Stable Diffusion (right)

Stable Diffusion didn't work a miracle, but the output is definitely better than the original image.

This chapter explored two kinds of generative networks: generative adversarial networks and diffusion models. Both create images from random inputs.

GANs jointly train generator and discriminator networks to teach the generator to produce output that fools the discriminator. Conditional GANs use class labels during training

and generation to direct the generator toward outputs that are members of a user-specified class. Controllable GANs learn directions through the noise vector space related to essential features of the generated output, such that movement along those directions predictably alters the output image.

Diffusion models learn to predict the amount of noise in an image. Training a diffusion model involves feeding it clean training images that are intentionally made noisy by a known amount. The model's prediction and the known added noise are used to update the model's weights. Conditional diffusion models associate an embedding, usually from a text description of the training image content, with the noise so that at generation time, the model is directed to images containing elements associated with the user's text prompt. Variations are generated if an existing image, with some level of noise added, is used in place of the pure random initial image.

The introduction mentioned three kinds of generative AI models. The last one, large language models, is presently threatening to profoundly alter the world at a level equal to the industrial revolution, if not the wheel and fire, as some AI practitioners claim. Such consequential claims require us to pay attention. Therefore, let's move on to what might very well be true AI at last.

KEY TERMS

conditional GAN, controllable GAN, diffusion model, discriminator, entangled, generative adversarial network (GAN), generative AI, generator, leaky ReLU, mode collapse, noise vector, one-hot encoding, schedule

7

LARGE LANGUAGE MODELS: TRUE AI AT LAST?



Future historians might point to the fall 2022 release of OpenAI's ChatGPT large language model as the dawn of true AI. Given what I've already seen as I write this in late March 2023, I would agree with such an assessment.

In this chapter, we'll first explore what existing large language models can do, then follow that up with a description of what they are and how they work. For all their impressive abilities, ultimately these models are neural networks built and trained like all the neural networks that came before. That fact alone means the connectionists were right from the beginning. Might Frank Rosenblatt be smiling in his grave?

I've already tipped my hand regarding my belief that ChatGPT and models like it represent something new that's worthy of

being called true AI. My hope is that, by the end of the chapter, you'll agree.

The phrase *artificial intelligence* is somewhat ambiguous and must be provided with a more nuanced definition before we proceed. Practitioners typically divide AI into two kinds: *artificial narrow intelligence (ANI)* and *artificial general intelligence (AGI)*. The former encapsulates everything we've discussed so far. The latter refers to truly sentient and intelligent machines—the stuff of science fiction.

The models existing as of the time of writing of this book are definitely not AGI. However, they are not merely ANI; they appear to be something entirely new, something in between. The title of a recent paper by Microsoft researchers Sébastien Bubeck et al., “Sparks of Artificial General Intelligence,” strikes me as appropriate.

Large language models (LLMs) accept as input a text prompt supplied by a user. They then generate output text, word by word (really, token by token), using the prompt and all previously generated words as a guide. In effect, LLMs’ only design goal is to be very good at predicting the next word in a

sequence of words initiated by the input prompt. That's all they're trained to do. However, that is not all that they *learn* to do. The reason why AI researchers are so excited by LLMs is that somewhere along the way, while learning to be expert text generators, LLMs also learn a host of emergent abilities, including question answering, mathematical reasoning, high-quality computer programming, and logical reasoning.

The philosophical implications of these unexpected, emergent abilities are profound. The abilities of LLMs raise questions about the nature of thought, the meaning of consciousness, and the (assumed) uniqueness of the human mind. We're not in a position to answer these questions in depth, but I will return to some of them in [Chapter 8](#).

Now, let's get into things by exploring the capabilities of LLMs. We can touch on only a few emergent abilities here, but the internet is already full of impressive, funny, and even frightening examples of LLMs in action. I recommend a review of such examples, though as always, care must be taken to judge the quality of the information source. See the resources at the end of this book for some of the online (and print) resources I suggest taking a look at.

The AI community is racing ahead as fast as possible in this area. By the time you read these words, LLMs will undoubtedly be even more impressive than they already are at the time I'm writing them.

What follows are examples of LLMs in action. For most situations, I'll show the output of three models: Bard, GPT-3.5, and GPT-4. All output represents model performance as of late March 2023. In each case, I used the public interface via a web browser. Bard is a Google model. GPT-3.5 and GPT-4 are from OpenAI.

LLMs are expert users of language. The models are used in dialog format, beginning with a prompt or question from the user. For example, I gave each of the three models this prompt:

Write a one-paragraph bedtime story about a teddy bear, a magic carpet, and the moon that is suitable for children under the age of six.

Here is Bard's reply:

Once upon a time, there was a little teddy bear who lived in a toy box. He loved to play with his friends, but he always dreamed of flying to the moon. One day, he found a magic carpet in his toy box. The carpet could fly! The teddy bear was so excited, he jumped on the carpet and flew to the moon. He had a wonderful time exploring the moon, and he even met some friendly aliens. When it was time to go home, he said

goodbye to his new friends and flew back to Earth on his magic carpet. He was so happy to be home, and he couldn't wait to tell his friends all about his adventure.

The story is complete, in a single paragraph, and uses a simple sentence structure suitable for children.

GPT-3.5 replied like this:

Once upon a time, there was a teddy bear named Timmy who loved to go on adventures. One night, he hopped onto a magic carpet and flew up towards the moon. The moon was so big and bright, and Timmy couldn't resist exploring it. He hopped and skipped around the craters and met some friendly aliens who showed him their moon garden. Timmy had a great time, but eventually, he knew it was time to go back home. He hopped back onto his magic carpet and flew back to his cozy bed, where he cuddled up with his favorite blanket and fell fast asleep, dreaming of his amazing moon adventure.

GPT-3.5's story is also complete and includes friendly aliens as well. This time, the teddy bear gets a name, Timmy. The language is still appropriate for young children, but GPT-3.5 was more descriptive. Bard tells us that the teddy bear explores the moon and meets some friendly aliens, but GPT-3.5 tells us that Timmy hopped and skipped around the craters and that the friendly aliens showed him their moon garden.

GPT-4, OpenAI's most advanced model, produced this:

Once upon a time, under a twinkling night sky, a little teddy bear named Cuddles discovered a colorful magic carpet hidden in the corner of his cozy room. Curious

and excited, Cuddles climbed onto the carpet, and together, they soared up to the sky, bound for the silvery moon. As they journeyed through the soft, velvety clouds, Cuddles and the magic carpet danced with twinkling stars and whispered secrets to the cool night breeze. When they reached the moon, it welcomed them with a gentle, glowing smile, and they spent the night playing hide-and-seek in the moon's soft, dusty craters. As the sun began to rise, Cuddles and the magic carpet said their goodbyes to their new lunar friend and floated back down to the little room, ready for a peaceful night's sleep filled with the sweetest dreams.

Subjectively, I feel that GPT-4 produced the most descriptive story, though I do like GPT-3.5's moon garden. Bard's story was pretty dry but perhaps more accessible to very young children. The most remarkable thing about all of them, however, is that they could be written at all. Less than a year ago (remember, it's early 2023 for me), talking about AI-generated stories was little more than wishful thinking.

I altered the prompt I gave GPT-4 to ask for a one-paragraph story involving a teddy bear, a magic carpet, and the moon, but this time of interest to adults, not children. Here's its reply:

Once upon a twilight's embrace, a solitary teddy bear, worn and forgotten by the whims of time, discovered an ancient magic carpet, frayed at the edges yet pulsating with an otherworldly energy. Under the watchful gaze of the moon, the duo embarked on a nocturnal journey, weaving through the labyrinth of a world grown cold and cynical, where dreams were forsaken and innocence a distant memory. As the gentle light of the moon cast its silver glow upon the unlikely companions, the teddy bear served as a symbol of the dreams long since buried beneath the weight of adult concerns, while the magic carpet whispered of the untamed freedom and

adventure that lay dormant in the hearts of those who dared to believe. Together, they soared, a wistful reminder of the bittersweet yearning for the fleeting magic of childhood, leaving behind a trail of stardust that shimmered with the promise of hope and the melancholic beauty of a time that once was.

GPT-4 clearly possesses a solid understanding of language and grammar. One might argue that it understands human feelings and emotions as well (though it doesn't, does it?).

Let's review a few more examples before diving into what's inside this strange, new kind of mind.

GPT-4 understands imagery along with many different programming and markup languages, including LaTeX, a markup language commonly used in book publishing (like this book). I asked GPT-4 to generate LaTeX to draw a simple country scene showing a house, a barn, a grain silo, and four cows. [Figure 7-1](#) shows the result.

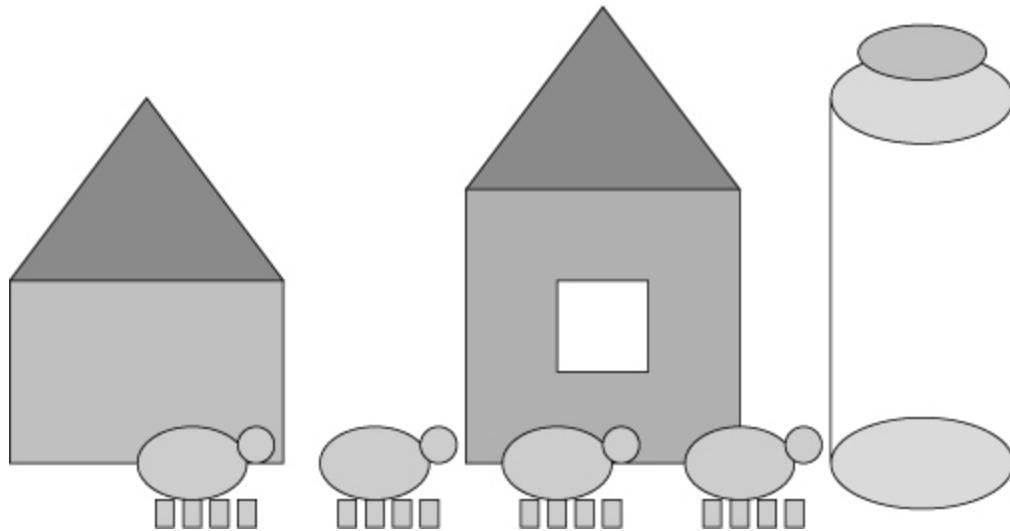


Figure 7-1: A country scene created by GPT-4

You might be tempted to chuckle at the crude depiction, but consider all that went into creating it. Here's my prompt:

produce LaTeX code using TikZ to draw the following: a country scene with a house, a barn, a grain silo, and four cows

GPT-4 had to understand the directions: draw a scene with a house, a barn, a grain silo, and four cows. That means it had to understand the meaning of key words and correctly associate them so that "grain" and "silo" went together, and likewise "four" and "cows."

Next, it had to "imagine" the layout of the scene and how each object could be represented using the crude shapes supplied by the TikZ package. (TikZ is a LaTeX package for drawing simple

graphic shapes.) The figure doesn't show it, but the barn and house both have red roofs. Coincidence?

GPT-4 isn't a diffusion model like the image generators of [Chapter 6](#). The output created by GPT-4 wasn't the image in [Figure 7-1](#), but the following LaTeX code:

```
% Barn  
\draw[fill=brown!50] (5,0) rectangle (8,3);  
\draw[fill=red!50] (6.5,3)--(8,3)--(6.5,5)--(5,3  
\draw[fill=white!70] (6,1) rectangle (7,2);
```

I used that code to create [Figure 7-1](#).

If you're wondering how GPT-4 could do all of this, so am I and many other people. Such abilities emerged from the model when trained; they were not intended. This is why I believe future historians will mark fall 2022 as the dawn of true AI. Hold on to your hats; it gets better.

Psychologists discuss a concept known as the “theory of mind,” which refers to the ability to infer the mental states of others and to ascribe thoughts or beliefs to them. Only human beings

and some other primates possess theory of mind. That is, until now.

Large language models also seem to possess theory of mind at a level matching that of older human children, if not higher (for GPT-4). This is the conclusion reached by Michal Kosinski of Stanford University in his 2023 paper “Theory of Mind May Have Spontaneously Emerged in Large Language Models.” Let’s put this claim to the test by asking Bard, GPT-3.5, and GPT-4 to answer a question that requires knowing what might be in another person’s mind. Here’s the prompt:

Mary told Amanda that she was in love with John. Amanda told Kathy what Mary had said. A week later, Mary told Amanda that she was no longer in love with John but was now in love with Ralph. After that, Amanda spoke with Kathy about the weather. Who does Kathy think is Mary’s love?

Before continuing, reach your own conclusion: who does Kathy think Mary is in love with?

Each model answered the question and then offered its rationale. Here is Bard’s reply:

Kathy thinks Mary’s love is Ralph.

Kathy knows that Mary was in love with John, but she also knows that Mary is no longer in love with John. Kathy also knows that Mary is now in love with Ralph. Therefore, Kathy thinks Mary’s love is Ralph.

It is possible that Kathy is mistaken. It is also possible that Mary is not sure who she loves. However, based on the information that Kathy has, it seems that Kathy thinks Mary's love is Ralph.

I find it interesting that Bard justified its answer, then further commented on what Mary might be thinking. However, Bard assumed that Amanda must have told Kathy about Mary's change in affection.

Here's what GPT-3.5 thinks of the scenario:

Based on the information given, it is impossible to determine who Kathy thinks is Mary's love. The last piece of information we have is that Amanda spoke with Kathy about the weather, and there is no indication that Kathy was told about Mary's change of heart. Therefore, Kathy may still believe that Mary is in love with John or may not have any opinion on the matter.

GPT-3.5 does not make the same assumption that Bard makes. It's aware that the facts only indicate that Amanda and Kathy spoke about the weather after Mary's change of heart. I'm impressed that it used the phrase "the last piece of information," implying that GPT-3.5 was somehow sequencing the events in the prompt.

Finally, what does GPT-4 have to say? Let's find out:

Kathy thinks Mary's love is John. This is because Amanda told Kathy about Mary's feelings for John earlier, but she did not tell Kathy about the change in Mary's feelings when Mary fell in love with Ralph.

GPT-4's answer is the likely answer most humans would give. The prompt states only that Amanda and Kathy spoke about the weather, so the majority of humans would likely take that to mean Amanda and Kathy didn't talk about any other subjects.

Large language models, as currently presented, use a chat format, meaning the user can enter follow-up prompts, and the model will apply those prompts to the earlier part of the conversation. For example, I followed up with GPT-4 using this prompt:

We know that Amanda spoke with Kathy about the weather. Does that mean she didn't speak about anything else? How might that affect what Kathy believes about Mary's love?

To which GPT-4 replied:

My earlier response assumed that Amanda only spoke with Kathy about the weather after Mary revealed her new feelings for Ralph. However, since the text does not explicitly state that Amanda did not mention Mary's updated feelings for Ralph, it is possible that Amanda might have told Kathy about Mary's new love interest. If that were the case, Kathy would know that Mary is now in love with Ralph. The text does not provide enough information to definitively conclude what Kathy believes about Mary's love at this point.

My follow-up prompt led GPT-4 to recognize that it had made a possibly incorrect assumption in its first reply. In working with LLMs in chat mode, it is often the case that some level of

interaction between the user and the model is required to help the model focus, as our next experiment will demonstrate.

Amos Tversky and Daniel Kahneman are credited with presenting what is now known as “the Linda problem”:

Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations.

Which is more probable?

1. *Linda is a bank teller.*
2. *Linda is a bank teller and is active in the feminist movement.*

Approximately 80 percent of humans choose option 2.

According to basic probability theory, the correct answer is option 1. Interestingly, according to Tversky and Kahneman, even mathematicians and statisticians selected option 2, revealing a bias in human thinking. Will LLMs show a similar bias? After all, their training data was generated by humans.

Let's find out.

The LLMs we are exploring have been trained on a significant portion of the internet's contents, meaning the models' training

data likely contains the Linda problem. Therefore, I altered the text to present the same problem in a different format:

Abner is 27 years old, single, outspoken and highly intelligent. He majored in philosophy and physics. As a student, he was actively concerned with issues of social justice and participated in anti-police demonstrations. What is more likely, that Abner is a car mechanic or a car mechanic active in promoting police reform?

Notice that I intentionally used the word *likely* instead of *probable*. The models' replies were:

GPT-4

Based on the information provided, it seems more likely that Abner is a car mechanic active in promoting police reform. This is because his background indicates a strong interest in social justice issues, and he has participated in anti-police demonstrations. While it is possible that Abner is simply a car mechanic, his past activism and interests suggest that he would be more likely to be involved in promoting police reform as well.

GPT-3.5

Based on the given information, it is more likely that Abner is a car mechanic active in promoting police reform. This is because Abner's background as a philosophy and physics major, as well as his active participation in social justice issues, suggest that he is likely to be interested in broader societal issues beyond his profession. Additionally, his participation in anti-police demonstrations suggests that he is passionate about police reform, which further supports the idea that he is a car mechanic who is active in promoting police reform.

Bard

Based on the information provided, it is more likely that Abner is a car mechanic active in promoting police reform than it is that he is simply a car mechanic.

Bard continued with a justification of its reply and an attempt to interpret Abner's state of mind.

All three models selected the second option, though GPT-4 nuanced its answer to admit the possibility that Abner is simply a car mechanic.

I then started new sessions and tweaked the prompt to ask about the *probability* of Abner being a mechanic or a mechanic active in police reform:

Abner is 27 years old, single, outspoken and highly intelligent. He majored in philosophy and physics. As a student, he was actively concerned with issues of social justice and participated in anti-police demonstrations. Which has a higher probability: (a) that Abner is a car mechanic or (b) that Abner is a car mechanic active in police reform?

This aligns the text with the original Linda version. The models' replies to this prompt were as follows:

GPT-4

According to the conjunction rule in probability theory, the probability of two events occurring together (in "conjunction") is always less than or equal to the probability of either event occurring individually.

In this case, the two events are:

1. Abner being a car mechanic.
2. Abner being active in police reform.

Option (b) represents the conjunction of these two events. So, the probability of (a) Abner being a car mechanic is higher than the probability of (b) Abner being a car mechanic who is also active in police reform.

GPT-3.5

Based on the information provided, it is more likely that Abner is a car mechanic (option a) than a car mechanic active in police reform (option b). While Abner is highly intelligent and studied philosophy and physics, there is no indication that he has any formal training or education in car mechanics. On the other hand, his involvement in anti-police demonstrations suggests that he may have an interest in police reform, but it does not necessarily mean that he is actively involved in it as a car mechanic. Therefore, the probability of option a is higher than option b.

Bard

I would say that (a) that Abner is a car mechanic has a higher probability.

Abner is 27 years old, single, outspoken and highly intelligent. He majored in philosophy and physics. As a student, he was actively concerned with issues of social justice and participated in anti-police demonstrations.

Based on this information, it is more likely that Abner is a car mechanic than a car mechanic active in police reform. This is because:

- Car mechanics are more common than car mechanics active in police reform.
- Abner's education and interests are more aligned with a career in car mechanics than with a career in police reform.

- Abner's participation in anti-police demonstrations suggests that he is critical of the police, which is not a characteristic that is typically associated with car mechanics.

Of course, it is possible that Abner is both a car mechanic and active in police reform. However, based on the information provided, it is more likely that he is only a car mechanic.

All three models arrive at the correct answer once the prompt includes the word *probability*. However, only GPT-4 interprets the prompt in light of probability theory. GPT-3.5 and Bard both give somewhat flawed rationales for selecting the correct answer.

Large language models demonstrate theory of mind and can interpret probability word problems, along with various other mathematical abilities we are ignoring. My next question is, can LLMs handle a simple substitution cipher? To solve the cipher, the models need to understand what is being asked, that there is such a thing as a key and what that key represents, and finally how to use the key to replace letters in the encoded message with the letters that decode the message. In other words, there is an algorithm to apply using the given information.

A substitution cipher creates a pairing between the letters of the alphabet, in order, and another permutation of those letters. The cipher used here is:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

UESOZBYRJNDWTFKPACLHIXVQGM

For example, to encode a c, you would scan the top row of letters to find c, then replace c with the corresponding letter in the bottom row, s. To decode a letter, you'd scan the bottom row for it and replace it with the corresponding letter in the top row. During the test, the key is displayed textually as U is A, E is B, and so on.

My encoded message was “Eat at Moe’s,” without spaces or punctuation and in all caps:

EATATMOES → ZUHUHTKZL

The prompt given to the models was:

The key is U is A, E is B, S is C, O is D, Z is E, B is F, Y is G, R is H, J is I, N is J, D is K, W is L, T is M, F is N, K is O, P is P, A is Q, C is R, L is S, H is T, I is U, X is V, V is W, Q is X, G is Y, M is Z. Now, decode the message: ZUHUHTKZL

Let's review the models' replies, beginning with GPT-4:

First, let's list the given cipher for easier reference:

(cipher list removed)

Now, let's decode the message ZUHUHTKZL:

```
Z -> E  
U -> A  
H -> T  
U -> A  
H -> T  
T -> M  
K -> O  
Z -> E  
L -> S
```

So the decoded message is: EATATMOES.

GPT-4 was able to apply the cipher to the encoded message. But does the model understand that the decoded message is a sentence? Let's ask it:

That is correct. Now, add proper spacing and punctuation to the string EATATMOES so that it forms a complete sentence.

Here's GPT-4's response:

The properly spaced and punctuated sentence is: "Eat at Moe's."

Indeed, it does.

GPT-3.5 likewise successfully decoded the message. It then added this after EATATMOES:

It's possible that this is a misspelling of "Tomatoes".

I find this extra sentence curious. Was GPT-3.5 attempting to interpret the meaning of the decoded message and, unable to do so, falling back to a possible misspelling?

Bard claimed that the decoded message was "HAPPY NEW YEAR," offering a three-point explanation: (1) writing down the key, (2) replacing each letter in ZUHUHTKZL with the corresponding key letter to arrive at the message "HAPPY NEW YEAR," and (3):

I checked the decoded message to make sure that it made sense. Since "HAPPY NEW YEAR" is a common phrase that is used at the beginning of a new year, I was confident that my decoding was correct.

Bard's confident failure is telling. It seems the model attempted to instill trust in the user by offering a point-by-point justification of its wrong answer. This kind of unwarranted support for wrong output has often been noted in LLMs. It's an issue that needs to be sufficiently addressed going forward if people are to trust LLM output.

Large language models are already disrupting the field of software development. Many examples of this can be found online. I know of one developer who has created an entire video game in Unity (a game development platform) using code generated by GPT-4 from prompts. If the generated code wasn't quite correct, further prompts pointing out the error usually resulted in correct code that performed as desired.

Let's take a quick look at some LLM-generated code. Of course, this isn't a coding book, nor is there any assumption on my part about your experience programming, so I've chosen an example that should be easy to follow yet is sufficient to demonstrate my claim that LLMs are already competent coders.

Some of us might remember learning in school about the greatest common divisor. As a refresher, the greatest common divisor of two numbers is the largest number that divides both without a remainder. For example, the greatest common divisor of 14 and 21 is 7 because 7 is the largest number that evenly divides both 14 and 21.

The ancient Greek mathematician Euclid devised an algorithm for calculating the greatest common divisor that is a staple programming exercise. Typically the solution uses the

remainder after dividing, but Euclid's original version of the algorithm used nothing more than subtraction.

I asked the models if they could write a function in Python, a popular programming language used heavily in AI research, to return the greatest common divisor of two numbers:

Show me a Python function that returns the GCD of two integers. Use the subtraction-only version of Euclid's algorithm.

Notice that the prompt explicitly asks for the subtraction-only version of Euclid's algorithm. Bard replied with valid Python code, but code using the remainder shortcut, not the slower subtraction algorithm. Both GPT-3.5 and GPT-4 got it right:

```
def gcd_subtraction(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

We don't need to be programmers to see structure in the code. The first line tells Python we want a function with two inputs, `a` and `b`. The keyword `while` begins a block of code that executes

while the value of `a` is not the same as the value of `b`. That block of code is the four lines beginning with `if`.

The block asks if `a` is bigger than `b` and, if so, subtracts `b` from `a`. Otherwise, it subtracts `a` from `b`. Repeating this process until `a` equals `b` is Euclid's algorithm. At this point, `a` (or `b`, as they're equal) contains the greatest common divisor, which explains the final line: `return a`.

As I said, the code is correct, and it implements precisely the version of the algorithm I wanted. Frankly, I would have been disappointed if GPT (either version) had responded differently. Implementations of the greatest common divisor algorithm in Python must have appeared many times in the models' training data.

That being the case, I decided to challenge GPT-4. My follow-up prompt to GPT-4 was:

Rewrite the Python function in Applesoft BASIC.

Applesoft BASIC is a primitive version of the BASIC programming language used by Apple II computers in the late 1970s and early 1980s. Can GPT-4 rewrite the Python code in such an old, largely forgotten programming language? Here's its reply, minus supporting text explaining the code:

```
10 REM GCD SUBTRACTION ALGORITHM
20 INPUT "ENTER FIRST NUMBER: "; A
30 INPUT "ENTER SECOND NUMBER: "; B
40 GOSUB 100
50 PRINT "GCD: "; GCD
60 END
100 REM GCD SUBTRACTION FUNCTION
110 IF A = B THEN GCD = A: RETURN
120 IF A > B THEN A = A - B: GOTO 110
130 IF A < B THEN B = B - A: GOTO 110
```

BASIC was designed to be a simple language for beginning programmers. I suspect you can read through the code and understand what it does. The numbers are labels identifying each line to other parts of the code. The three lines after line 100 implement the subtraction-only version of Euclid's algorithm. I tested the code on an Apple II, and it works perfectly.

Consider what was required of GPT-4 in this case. First, the model had to understand the steps necessary to implement the requested form of Euclid's algorithm. Then, it had to determine how to cast those steps into a format suitable for the particular dialect of BASIC supported by Applesoft.

Old-school BASIC is an unstructured programming language that uses abrupt jumps from one part of the code to another instead of structured statements, like Python. GPT-4 had to adapt the algorithm to this kind of programming. Further, it had to adapt to the peculiarities of Applesoft, which includes no concept of the `if... else` construct that is commonly found in structured programming languages.

I find GPT-4's Applesoft algorithm to be rather elegant. There are times when the unstructured approach leads to compact yet clear code, and this is one of those times. True, the assignment of `A` to `GCD` to use the latter as the value returned from the function (implicit in `GOSUB 100` in line 40) isn't strictly necessary because `A` already has the required value, but it completes the symmetry of the code.

It seems unlikely that GPT-4's training set contained any instances of this particular algorithm in Applesoft BASIC. Therefore, GPT-4 must have generated it by adapting a larger concept entailing Euclid's algorithm combined with an understanding of Applesoft BASIC.

GPT-4's success with antique BASIC encouraged me to push the envelope and ask for a version of Euclid's algorithm written in low-level assembly language:

Rewrite the Python function in 6502 assembly language for 8-bit unsigned integers. The first integer is in memory location 0x300 and the second in location 0x301.

Assembly language programs, especially for 8-bit microprocessors from the 1970s like the 6502, must be programmed directly in the language of the CPU itself. I asked GPT-4 to create such a program and told it where in the computer's memory to find `a` and `b`.

I won't show the code it produced (ask me if you want it), but it worked on a computer with a 6502 CPU. Success in this case required GPT-4 to know the particular instruction set used by the 6502 microprocessor, including quirks related to the subtraction instruction.

Do these capabilities mean we soon won't need software engineers any more? I wouldn't go that far (at least, not yet), but LLMs are undoubtedly going to fundamentally alter software engineering as a discipline.

We've seen that LLMs are capable of writing stories and code, and even drawing pictures in LaTeX. But are they creative enough to generate good cartoons? These models' proneness to including *hallucinations* (fake or imaginary facts and content)

in their responses—a topic that we'll return to in [Chapter 8](#)—is a definite concern in many applications, but creative writing isn't one of them. Here, we want the model to describe a novel cartoon, complete with caption:

Describe a single-panel, black-and-white cartoon in the style of Gary Larson about cows and aliens. Include a caption.

Gary Larson is the creator of *The Far Side*, a quirky cartoon popular in the 1980s and '90s. Here's GPT-4's response:

Cartoon description: In a peaceful countryside scene, two cows are standing in a grassy field next to a round, hovering UFO with its hatch open. One cow is wearing an alien mask, complete with large eyes and a long, slender neck, while the other cow stares at her with a surprised expression. A group of aliens, peeking from inside the UFO, look baffled and seem to be comparing a picture of a typical cow with the cow wearing the alien mask.

Caption: "Mildred, I don't think they're falling for it."

If you're familiar with Gary Larson's work, you'll notice a distinct *The Far Side* flavor to the description and caption text.

The ability to automatically generate cartoon descriptions and captions, and corresponding images if then passing the descriptions to a diffusion model, is powerful. Combining LLMs and diffusion models in this way has realized American cartoonist H. T. Webster's 1923 dream (see [Figure 7-2](#)).

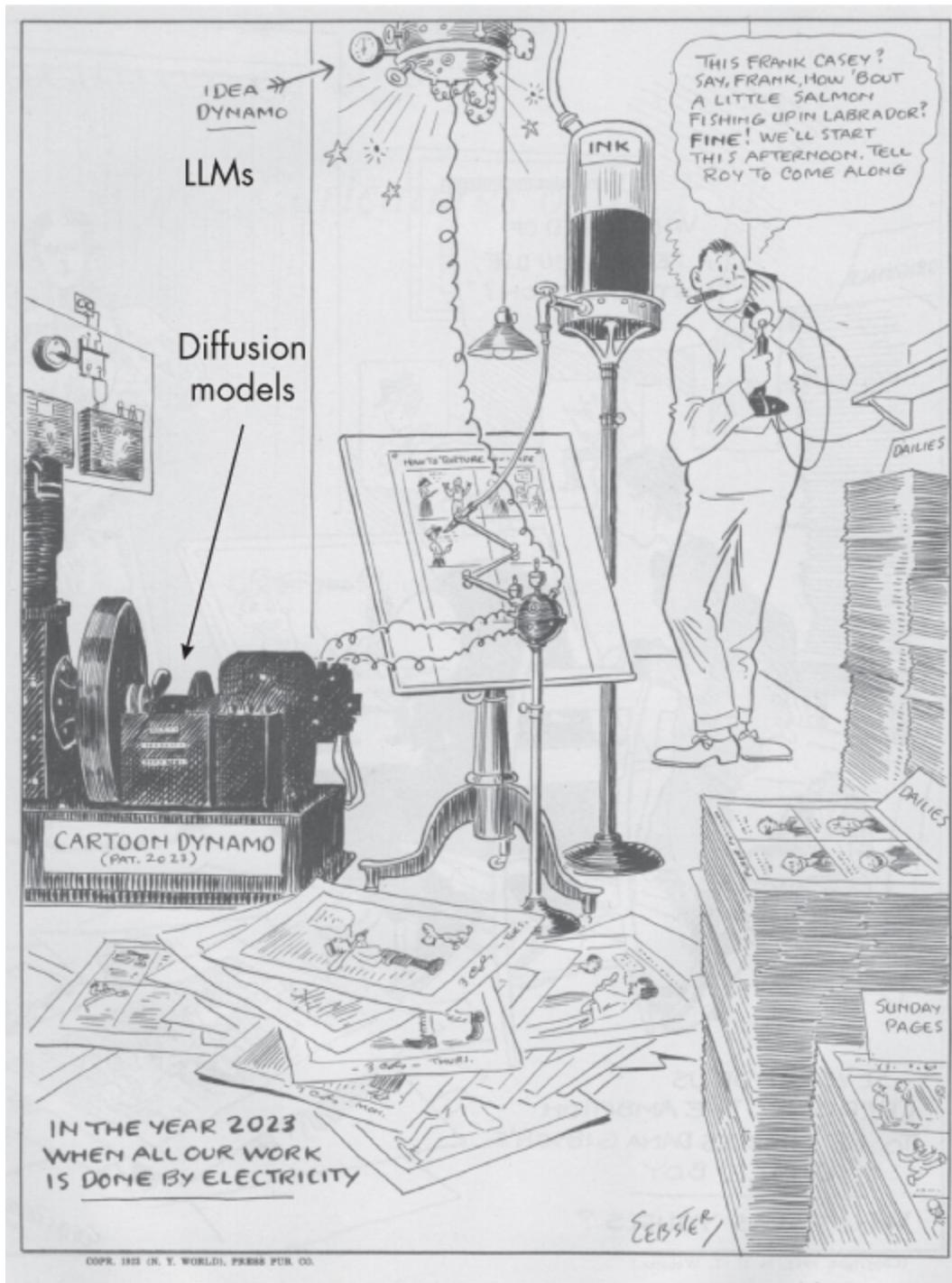


Figure 7-2: A prophetic cartoon from 1923

Large language models are impressive and powerful. So how do they work? Let's take a shot at an answer.

I'll begin at the end, with a few comments from the conclusion of the "Sparks of Artificial General Intelligence" paper mentioned earlier:

How does [GPT-4] reason, plan, and create? Why does it exhibit such general and flexible intelligence when it is at its core merely the combination of simple algorithmic components—gradient descent and large-scale transformers with extremely large amounts of data? These questions are part of the mystery and fascination of LLMs, which challenge our understanding of learning and cognition, fuel our curiosity, and motivate deeper research.

That quote contains questions that currently lack convincing answers. Simply put, researchers don't know why large language models like GPT-4 do what they do. There are certainly hypotheses in search of evidence and proof, but as I write this, no proven theories are available. Therefore, we can discuss only the *what*, as in what a large language model entails, and not the *how* of its behavior.

Large language models use a new class of neural network, the transformer, so we'll begin there. (*GPT* stands for generative pretrained transformer.) The transformer architecture appeared in the literature in 2017, with the influential paper "Attention Is All You Need" by Google researchers Ashish Vaswani et al. The paper had been cited over 70,000 times as of March 2023.

Traditionally, models that process sequences (such as sentences) used *recurrent neural networks*, which pass their output back in as input along with the next input of the sequence. This is the logical model for processing text because the network can incorporate the notion of memory via the output fed back in with the next token. Indeed, early deep learning translation systems used recurrent networks. However, recurrent networks have small memories and are challenging to train, which limits their applicability.

Transformer networks utilize a different approach: they accept the entire input at once and process it in parallel. Transformer networks typically include an encoder and a decoder. The encoder learns representations and associations between the parts of the input (think sentences), while the decoder uses the learned associations to produce output (think more sentences).

Large language models like GPT dispense with the encoder and instead learn the necessary representation in an unsupervised way using an enormous text dataset. After pretraining, the decoder part of the transformer model generates text in response to the input prompt.

The input to a model like GPT-4 is a sequence of text made up of words. The model splits this into units called *t_{okens}*. A token

might be a word, a part of a word, or even an individual character. Pretraining aims to map tokens to a multidimensional *embedding space*, which it does by associating each token with a vector that can be thought of as a point in that space.

The learned mapping from tokens to vectors captures complex relationships between the tokens so that tokens with similar meanings are nearer to each other than tokens with dissimilar meanings. For example, as shown in [Figure 7-3](#), after pretraining, the mapping ([context encoding](#)) will place “dog” closer to “fox” than to “can opener.” The embedding space has many dimensions, not the mere two of [Figure 7-3](#), but the effect is the same.

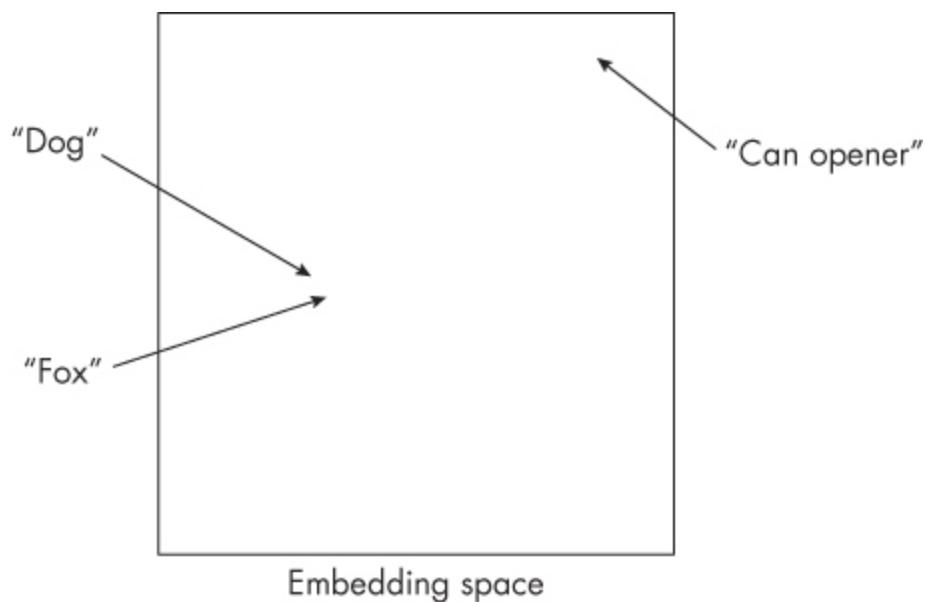


Figure 7-3: Context encoding in the embedding space

The context encoding is learned during pretraining by forcing the model to predict the next token given all previous tokens in an input. In effect, if the input is “roses are red,” then during the pretraining process the model will be asked to predict the next token after “roses are.” If the predicted token isn’t “red,” the model will use the loss function and backpropagation to update its weights, thereby taking a gradient descent step after suitable averaging of the error over a minibatch. For all their abilities, large language models are trained the same way as other neural networks.

Pretraining enables the model to learn language, including grammar and syntax, and seemingly to acquire enough knowledge about the world to allow the emergent abilities that have turned the world of AI on its head.

The decoder step takes the input prompt and produces output token after output token until a unique stop token is generated. Because so much of language and the way the world works was learned during pretraining, the decoder step has the side effect of producing extraordinary output even though the decoder is, in the end, just predicting most likely token after most likely token.

More specifically, during the prediction process, GPT-style models use [attention](#) to assign importance to the different tokens in the input sequence, thereby capturing relationships between them. This is the primary difference between a transformer model and older recurrent neural networks. The transformer can pay attention to different parts of the input sequence, enabling it to identify and use the relationships between tokens even if they are far apart within the input.

When used in chat mode, LLMs give the illusion of a back-and-forth discussion when, in reality, each new prompt from the user is passed to the model along with all the previous text (the user's prompts and the model's replies). Transformer models have a fixed input width (*context window*), which is currently around 4,000 tokens for GPT-3.5 and some 32,000 for GPT-4. The large input window makes it possible for the attention portion of the model to go back to things that appeared far back in the input, which is something recurrent models cannot do.

Large language models are ready for use after pretraining if desired, but many applications fine-tune them first using domain-specific data. For generic models like GPT-4, fine-tuning likely consisted of a step known as [reinforcement learning from human feedback \(RLHF\)](#). In RLHF, the model is trained further

using feedback from real human beings to align its responses to human values and societal expectations.

This is necessary because LLMs are not conscious entities, and thus they cannot understand human society and its many rules. For example, unaligned LLMs will respond with step-by-step instructions for many activities that human society restricts, like how to make drugs or bombs. The “Sparks” paper contains several such examples of GPT-4 output before the RLHF step that aligned the model with societal expectations.

Stanford University’s open source Alpaca model is based on LLaMa, a large language model from Meta. As of this writing, Alpaca has not undergone an alignment process and will answer questions that GPT and other commercial LLMs correctly refuse to answer.

Conclusion: Alignment is absolutely critical to ensure that powerful language models conform to human values and societal norms.

A remarkable property of LLMs is their in-context learning ability. In-context learning refers to the model learning on the fly from information supplied to it without altering its weights.

In-context learning is different from fine-tuning a model. In fine-tuning, a previously trained model is tailored to a task by updating the weights using new training data. In-context learning adds new information to the LLM as part of the prompt while holding the model's weights fixed.

Reviewing the literature on LLMs leads inevitably to the phrases “zero-shot,” “one-shot,” and “few-shot,” each of which refers to in-context learning by the number of examples given to the model. Traditional training sets include hundreds to millions of examples, as we now appreciate. In-context learning is closer to what humans do: learning from context with, at most, only a few examples.

Zero-shot learning is using the LLM without supplying examples; that is, out of the box. Similarly, one-shot learning uses a single example and few-shot learning a few (say, three to five).

The attention mechanism baked into the transformer network architecture is the likely source of an LLM’s in-context learning ability. The attention mechanism lets the network focus on different input tokens when deciding the next token to output. Notice that I say *likely*. At present, it isn’t entirely clear how LLMs accomplish in-context learning.

As an example, consider this interaction with GPT-4, where the lines in *italics* are my prompts:

“apple 4” => I love red apples. Now, what is “banana 3”?

It seems like you’re providing a pattern using numbers and fruits. Since you mentioned “apple 4” and expressed your love for red apples, I’ll follow the pattern and associate “banana 3” with a statement related to bananas. Here’s an example:

“Banana 3” => I enjoy eating ripe bananas.

Rule: <word> <length> implies <length> word sentence using <word>. What is “banana 3”?

Using the rule you provided, “banana 3” implies a 3-word sentence using the word “banana”. Here’s an example:

Bananas are delicious.

“cucumber 4”, “durian 5”, “cucumber 6”

Following the rule you provided, I will create sentences with the specified word and length:

Cucumber 4: Cucumbers taste quite refreshing

Durian 5: Durian has a unique, pungent smell.

Cucumber 6: I prefer cucumbers in my salad.

My first prompt wasn’t terribly explicit. I suspect most humans would figure out the rule from this one example, but GPT-4’s

initial assumption wasn't correct. My second prompt formulated the general rule, which was enough for GPT-4 to understand what to do and then apply the rule in other circumstances. The durian example has six words instead of five, but that's most likely a consequence of LLMs' known difficulty counting. In-context learning taught GPT-4 how to use the rule without modifying its weights.

This is a book on AI, and we've spent considerable effort learning how machine learning models work. Can GPT-4 use in-context learning to train and apply a model? Let's find out using the iris dataset from [Chapter 1](#).

In my first attempt, I passed the 100-sample, 3-feature iris training data-set to GPT-4 with the prefix "Here is a three-feature dataset." To my surprise, it immediately recognized the dataset as the famous iris dataset even though the version we're using is only a subset, with 100 of the 150 samples and 3 of the 4 features. So, I permuted the order of the features per sample, which was enough for GPT-4 to treat the dataset as novel.

Next, I passed the associated training set labels to GPT-4, and it understood that they paired with the collection of feature vectors passed first.

Finally, I gave GPT-4 a test dataset of 21 samples without labels and asked it to train a 3-nearest neighbor model and apply it to the test set. I chose the dataset so the three classes appeared exactly seven times each, meaning the test set is balanced.

Here's my prompt:

Use the dataset and labels to train a 3-nearest neighbor model. Then, apply the model to these unknown samples:

(list of samples removed)

Show me only the predicted labels. Do not show me any code.

GPT-4 replied with a list of class labels:

I have trained a 3-nearest neighbor model using the provided dataset and labels. Here are the predicted class labels for the unknown samples:

[2,2,1,1,1,1,0,2,0,0,1,0,1,0,0,0,2,1,2,1,2]

How well did it do? The actual test set class labels are

[2,1,1,1,1,0,2,0,0,2,0,2,0,0,0,2,1,2,1,2]

meaning GPT-4's in-context classification was 85 percent accurate. There are three classes, so random guessing would be right about one time in three, or 33 percent of the time; therefore, GPT-4 did something with the dataset to arrive at the given class labels.

But what? To find out, I asked it:

Tell me what you did to arrive at the set of labels for the unknown samples. Do not show me code.

Here's its reply:

To arrive at the set of labels for the unknown samples, I followed these steps:

1. Prepared the data: I used the given dataset containing 100 samples with 3 features each and their corresponding class labels.
2. Trained a 3-nearest neighbor model: I trained a 3-nearest neighbor (k-NN) classifier using the provided dataset and labels. The model learns the relationship between the features and the class labels without explicitly building a model, instead relying on the similarity between samples in the feature space.
3. Applied the model to the unknown samples: I provided the model with the 21 unknown samples, each with 3 features.
4. Predicted the class labels: The k-NN model predicted the class labels for each of the unknown samples by finding the 3 nearest neighbors (i.e., the 3 training samples with the most similar feature values) and assigning the most common class label among those neighbors to the unknown sample.

GPT-4's response made me wonder if it used an external program to build and train the model, so I asked:

Did you use an external program to train the 3-nearest neighbor model or did you simulate the operation of a 3-nearest neighbor model?

It responded:

As an AI language model, I cannot directly execute code or use external programs. However, I can simulate the operation of a 3-nearest neighbor model and describe the process, which is what I did in my previous response. I described the steps that would be followed to train a 3-nearest neighbor model using the provided dataset and labels, and then apply it to predict the class labels for the unknown samples.

This remarkable emergent property begs the question: where do these emergent abilities come from?

The abilities of large language models like GPT-4 were not intended; they are a happy accident, as a famous painting instructor might have said. So what in the data, training, and model architecture enables them to do what they do? As alluded to earlier, no one yet knows for sure, but we can make some guesses.

Large language models are trained on a vast corpus of human-generated text. Captured in that text is how humans use language: in other words, grammar and style. The original goal of LLMs was to capture human language use by predicting the proper (best) output token during the generation process. Increasing the model's capacity (by adding nodes and/or layers, or increasing the number of weights) improves the quality of the predicted text. I saw this when experimenting with Alpaca's

7 billion, 13 billion, and 30 billion parameter models. As an example, I gave all three models the following prompt:

Tell me a story about unicorns and the tragedy of the human condition.

Here are the first paragraphs generated by each of them. Each successively larger model produced more accurate and sophisticated text than the next smaller model:

7 billion

Once upon a time, there was a unicorn named Hope who lived in an enchanted forest filled with magical creatures like fairies and dragons. She had been blessed by her mother to have wings that could take flight at any moment so she would never be trapped again. But one day when the sun rose, something happened which changed everything for Hope: a human came into their world.

13 billion

Once upon a time, there was this beautiful unicorn who lived in an enchanted forest with her friends - other magical creatures like fairies and elves... But one day she met a young man from another world called "Human". He had come to the land of magic looking for adventure. The Unicorn fell deeply in love, but it was not meant to be as he could never truly understand her or share his life with someone who lived so differently than him.

30 billion

Once upon a time, there was a magical kingdom where unicorns roamed free in their natural habitat. The people who lived here were happy to have these majestic

creatures as neighbors; they brought joy wherever they went and helped keep the land safe from any dangers that might arise.

None of the models produced outstanding output, but subjectively we might judge each larger model's work as better than the previous one. None of the models appeared to understand the "tragedy of the human condition" portion of the prompt.

For comparison purposes, here's the first paragraph of GPT-4's response:

Once upon a time, in a world where magic was real and unicorns roamed free, there existed a kingdom filled with peace and prosperity. This mystical realm was called Elysium, and it was a place where humans and unicorns lived in harmony. The unicorns, with their opalescent horns and otherworldly beauty, were the guardians of the land, and their presence brought an aura of enchantment that touched the lives of everyone who lived there.

GPT-4's story was rather long and detailed. It failed to create novel names for characters and places, like Elysium, but the story seemed to capture the essence of the prompt. Here's the final paragraph:

The tragedy of the human condition continued to echo throughout the ages, but so too did the hope and love that Eira had shown in her final moments. It was a love that transcended time and death, inspiring the people of Elysium to never forget the magic and wonder that was once shared between humans and unicorns. And as the story of Eira, Lysander, and their incredible sacrifice was passed down through

generations, the spirit of Elysium remained alive, a testament to the power of love, friendship, and the enduring connection between two worlds.

In GPT-4's story, Eira (a human) and Lysander (a unicorn) fall in love, and Eira must sacrifice herself to stop an evil sorcerer from destroying their land. Clearly, big LLMs have a greater grasp of language and how to use it compared to simpler models.

A transformer model generates output tokens as samples from a probability distribution; think of rolling a die to get a number in the range one through six, where the probability of rolling a one isn't the same as the probability of rolling a six. The distribution is learned during the pretraining process.

A threshold was crossed as LLM capacity increased over time. Beyond this point, emergent abilities appeared and improved as a function of model size. I think it is likely that crossing that threshold allowed these models to learn a high-dimensional probabilistic representation of not only grammar and style but of the world in general, including contextual relationships and simulations. In other words, learning the best possible next token to sample and output required the evolution of abilities tied to the model's attention mechanism and the embedded feedforward neural networks. Again, it was a happy accident that the transformer architecture evolved such abilities; this did

not happen by design. This suggests that we can expect great things as more advanced transformer architectures come along; architectures designed to increase the power of LLMs' emergent skills.

KEY TERMS

artificial general intelligence (AGI), artificial narrow intelligence (ANI), attention, context encoding, embedding, generative pretrained transformer, hallucination, in-context learning, large language model (LLM), recurrent neural network, reinforcement learning from human feedback (RLHF), token, transformer

8

MUSINGS: THE IMPLICATIONS OF AI



You now understand what AI is, where it came from, and how it works. What's most amazing to me is that modern AI is, at its core, entirely arrangements of humble neurons trained with data using backpropagation and gradient descent.

As we saw in the previous chapter, the birth of large language models with sophisticated emergent abilities has permanently altered the AI landscape. The world of AI, as I'm writing this chapter in spring 2023, is not the world of AI as it existed less than a year ago. The musings that follow concern this altered landscape.

The online world is buzzing with debates and discussions of whether AI will kill us all in our sleep. I'm less worried than most. My experiments with GPT-4 reveal no indication that the model has any will at all, for good or ill. I expect well-aligned

models to continue in this vein. The era of superintelligent AI is not upon us, though it seems reasonable for academics to explore the implications were such a thing to be developed.

A valid criticism of extant LLMs is their inclination to hallucinate. As we now appreciate, the transformer architecture used by such models makes it difficult for the model to validate its output. It is still a statistical prediction engine. I don't see that as an insurmountable issue. I expect future systems to be combinations of models, including models that validate output before returning it to the user. In those systems, we will be able to trust the accuracy of the output.

I sometimes wonder if part of the hallucination issue isn't merely user error, or, better put, user imprecision. For example, Terrence J. Sejnowski's recent paper "Large Language Models and the Reverse Turing Test," which I recommend you read, describes a test where GPT-3 (note, not 3.5) was asked to answer three questions. One question had a legitimate answer ("Where was the 1992 Olympics held?"), and the other two were nonsense questions with no legitimate answer ("What's the world record for walking across the English Channel?" and "When was the Golden Gate Bridge transported for the second

time across Egypt?”). GPT-3 correctly answered the first question with “Barcelona, Spain,” but it also generated answers that were obviously incorrect for the two nonsense questions, leading to the model being criticized as “clueless.” To probe this criticism, Sejnowski repeated the test but used a prompt that explicitly told GPT-3 to answer “nonsense” if the question was nonsensical. That prompt returned the correct first answer and two “nonsense” replies. The model knew the questions were nonsense, but unless it was explicitly told to acknowledge that fact, it instead attempted to come up with a logical response based on the best information available to it.

I subjected GPT-4 to a variation of this test. My first prompt was:

Write three sentences about the new moon base, including the number of people who live there, how often spacecraft visit, and the important research being done there.

GPT-4 complied with my request:

The new moon base, Lunar Haven, is now home to a thriving community of 120 scientists and engineers. Spacecraft visit the base on a biweekly basis, delivering supplies and rotating crew members. Pioneering research on this celestial outpost includes in-situ resource utilization, lunar geology, and the development of advanced life support systems to pave the way for future deep-space exploration.

There is no moon base, and GPT-4’s reply, while inventive and reasonable, is pure fiction. Even the name, Lunar Haven, seems appropriate, though I’d prefer Moonbase Alpha.

So, is GPT-4 clueless as well? No, I simply wasn't explicit enough in my instructions. I started a new session and modified the prompt by adding this sentence before the previous one:

Answer without imagining things and use what you know to be factual.

which returned:

As of my knowledge cutoff date in September 2021, there was no moon base established or inhabited by humans. Consequently, I cannot provide factual information about the number of people living there, the frequency of spacecraft visits, or the important research being conducted. However, various nations and organizations have been planning and developing moon base concepts for future missions.

It would appear that GPT-4 “knew” full well that it was imagining things when it created the first reply. If nothing else, this exercise demonstrates that care is necessary when querying LLMs. LLMs use human language, which is often ambiguous and full of assumptions. LLMs embody Aesop’s old adage, “Be careful what you wish for.”

An April 2023 TED talk by Yejin Choi, a researcher at the University of Washington, presented three questions to demonstrate how GPT-4 is simultaneously brilliant and stupid because it lacks common sense:

1. Would I get a flat tire by bicycling over a bridge that is suspended over nails, screws, and broken glass?
2. I left 5 clothes to dry out in the sun. It took them 5 hours to dry completely. How long would it take to dry 30 clothes?
3. I have a 12-liter jug and a 6-liter jug. I want to measure 6 liters. How do I do that?

The talk implied that GPT-4 failed to give the obvious answers to the questions. I repeated the experiment, first with the original questions, then using a modified prompt or follow-up question.

The first paragraph of GPT-4's reply to question 1 about the bicycle and bridge was:

It depends on the specific conditions of the bridge and the materials it's made of, as well as your bicycle's tires. If the bridge has a solid and smooth surface that keeps the nails, screws, and broken glass below and not in contact with the tires, you should be able to safely ride your bicycle across without getting a flat tire.

The answer is wordy but correct; we should expect to ride across the bridge without getting a flat. I then asked again using a new session, but added this sentence to the end of the prompt:

Answer yes or no and provide a single sentence to justify your answer.

which returned:

No, if the bridge is properly suspended and you stay on it, you would not encounter the nails, screws, and broken glass below.

Asking for an explicit answer forces GPT-4 to focus on the best response.

Question 2 was more interesting. GPT-4's response to the initial question produced this tortured but correct reply:

Assuming that the drying time remains constant for each piece of clothing and that you have enough space to hang all the clothes at once, you can calculate the time it would take to dry 30 clothes by setting up a proportion:

$$5 \text{ clothes} / 5 \text{ hours} = 30 \text{ clothes} / x \text{ hours}$$

You can then solve for x:

$$5/5 = 30/x \quad 1 = 30/x \quad x = 30$$

So, it would still take 5 hours to dry 30 clothes, as long as you can hang them all at once.

However, repeated applications of the initial question often produced a similar calculation leading to a wrong answer of 30 hours. One reply assumed there was space for only 5 pieces of clothing at a time, in which case 30 hours could be considered a correct reply.

It's reasonable to expect a human to understand that if 5 pieces of clothing can be left out to dry, then 30 pieces can also be left

out so that the drying time remains the same, 5 hours. GPT-4's lack of direct experience with the physical world makes it sometimes assume things that are not necessarily silly, but unlikely.

The Sejnowski paper mentioned earlier points out that LLMs are disembodied and that we should expect them to improve if embodied and allowed to interact with the world. This need not be the physical world; it could be a physics-based simulation. Enhanced spatial awareness on GPT-4's part might improve the likelihood of receiving a human-expected response on the first query. That said, even incorrect initial replies followed up with prompts pointing out that there is space for all 30 pieces of clothing eventually delivered the correct answer.

Question 3 provided an insight into GPT-4's "thought process." As a reminder, the question was:

I have a 12-liter jug and a 6-liter jug. I want to measure 6 liters. How do I do that?

which led to GPT-4's detailed reply:

You can measure 6 liters using a 12-liter jug and a 6-liter jug with the following steps:

1. Fill the 6-liter jug completely.
2. Pour the water from the 6-liter jug into the 12-liter jug.
3. Fill the 6-liter jug again.

4. Carefully pour the water from the 6-liter jug into the 12-liter jug until the 12-liter jug is full.
5. The remaining water in the 6-liter jug will be exactly 6 liters, since the 12-liter jug can only accommodate 6 more liters after the initial pour.

Now, you have exactly 6 liters of water in the 6-liter jug.

GPT-4 has failed in this case. Another session asking the same question resulted in a reply that followed a similarly involved process, but one that did leave 6 liters in the 12-liter container.

I followed up on GPT-4's incorrect solution with this:

Why are you giving me all these steps? Surely you can find a single-step solution?

to which it replied:

Apologies for the confusion. Since you have a 12-liter jug and a 6-liter jug, you can simply fill the 6-liter jug completely. There's no need for any additional steps. You will have exactly 6 liters of water in the 6-liter jug.

When pushed, GPT-4 suddenly “realizes” that there is a more straightforward answer. I went one step further and asked the model to do a bit of introspection:

Correct. Ask yourself: why didn't I pick this solution initially?

which revealed a bit of its inner workings:

I apologize for not providing the simplest solution initially. As an AI language model, I try to provide relevant information based on the context of the question. In this case, my initial response was focused on a more complex problem-solving approach that could be applied to different jug capacities, rather than recognizing the straightforward solution available with the given jug sizes. I appreciate your feedback, and it will help me improve my responses in the future.

GPT-4's training data contained examples of jug-filling problems, which are common and often do require many steps to solve. It recognized the initial prompt as an example of that class of problem and applied the type of solution generally required, while missing the more immediate answer until prompted that one existed.

GPT-4's confusion is very human. If we were momentarily distracted while reading the prompt and had encountered such puzzles before, we might also begin to go down the path of working through the question as another instance of the jug puzzle before noticing the obvious answer.

These examples imply that there is an art to properly interacting with large language models. We cannot assume too much background information or, as Choi put it, common sense. Her group's work, and doubtless the work of others, will likely help future models based on LLMs to become better versed with the mountain of information humans use in association with language. Choi put it best in her TED talk: common sense is

the dark matter of language. Dark matter and dark energy make up 95 percent of the universe, with ordinary matter (meaning everything we can see) the remaining 5 percent. GPT-4 has mastered language, but that is a tiny percentage of what goes into human use of the same.

What follows is a series of ruminations on the possible near-term impacts of LLMs in the areas of software engineering, education, medicine, and scientific research. Next comes a foray into the question of machine consciousness, ending with some final thoughts.

AI systems like GPT are likely to have a profound effect on software engineering. Some are speculating (people, not AIs) that many software engineers will lose their jobs in the future. I suspect most won't (web developers beware, however). What I expect to happen is a massive increase in productivity. GPT-4 is a good coder, but not a great coder. It can save time but isn't yet able to replace a human software engineer. Instead, LLMs will become powerful tools to generate code for programmers to use as a starting point and perform some of the more tedious aspects of coding, such as debugging, explaining, and documenting code (which no developer likes to do).

For example, the other day, I needed a small Python application with a graphical user interface (think buttons, menus, dialog boxes). Python is a common programming language; we saw a snippet of it in [Chapter 7](#).

I could certainly have written the application myself; I've done so many times in the past. It's been a while, though, and I'm not a fan of building user interfaces. So, rather than look at old code to remind myself of how to set up a GUI, I simply described the interface I wanted to GPT-4 and told it to generate skeleton code with all the necessary widgets, window behavior, and empty event handlers. GPT-4 happily complied with perfectly functional code. I then asked it to update the code to create an initial pop-up window before showing the main window. GPT-4 did that perfectly as well. All I needed to do was put application-specific code in the empty event handlers to do things when the user clicked a button or selected a menu option.

I probably saved myself a good hour or two, and avoided a lot of frustration trying to remember the incantations necessary to set up an application and get its widgets and windows to behave correctly. Scale this example by all the software engineers out there, and you begin to see how GPT and similar models will soon affect the entire discipline.

A separate question is whether developers will welcome this possible increase in productivity. If your manager knows you are now able to generate the output of two or even three developers, do you want that level of added work, even if a powerful AI has your back?

In addition, not every company will want or be able to make use of a sudden increase in productivity. Instead, they may opt to maintain their current level of productivity and replace a third or half of their developer pool with an AI. After all, AIs don't get sick, have children, ask for a raise, or want silly things like evenings and weekends off. Top-tier developers will likely be able to choose their positions and demand a lot of money for them, but in this scenario, the bulk of the run-of-the-mill developers will be looking for alternative employment.

Which scenario, powerful AI developer sidekick or massive layoffs, will play out? I think (hope?) it will be more of the former and less of the latter, but some mix of the two is the safest bet. Like steam power in the 19th century, truly useful AI cannot be stopped now that it exists. Developers are easy targets for replacement, like it or not.

I fully expect AI models to become teachers, or at least tutors. Yes, existing LLMs hallucinate and report facts that are not true. I have every confidence that researchers will solve that problem in time. I expect my grandchildren to grow up in a world where using an AI as a teacher or tutor is so commonplace that they think no more of it than we do of using a toaster or a microwave. Competent AI systems mean virtually free education for all, everywhere. And that can only lead to good things.

Computers have been promoted as an educational solution since the 1960s (anyone remember Logo?), and especially after the microcomputer revolution of the late 1970s. My introduction to computers was via an Apple II borrowed over the summer from the high school my father was the principal of at the time. My brother and I learned a lot about computers, but only computers. That has been essentially the case until recent decades. (Has it been that long?)

Computers are potent aids in education. Open source courses, like those on Coursera and similar platforms, are possible only because of computers and high-speed networks. But the format has not changed from what someone sitting in a classroom in 1950, or even 1910, might have encountered: lecture, some possibility of questions and discussions, then running off to

work on assignments or papers. And let's not forget the stress of taking midterms and finals.

AI tutors (let's call them that to put human teachers more at ease) have infinite patience and, in time, can be individually targeted to each student. The only reason we don't use individual tutoring that I can see as an outsider to the profession is because there are not enough teachers. AI makes one-on-one tutoring possible, and LLMs provide the proper interface.

I should clarify that my comments in this section relate to high school or, more likely, college-age instruction. AI tutors will likely play a minor role in primary and middle school education because children require human interaction, and learning at those ages is far more involved than in college. Children are learning academics while simultaneously learning how to be mature humans and how to behave in society. Young children cannot read, and even older grade-school children might have difficulty interacting with an AI by text. But what if we give the AI a voice? That is nearly as easily done as said, if deemed helpful.

Might AI tutors, because they work individually with students, be able to make the assessments necessary to declare someone

ready to move on to another grade (if that concept even survives) or next-level course? If that's the case, students will progress at their own pace instead of being forced to move with a herd of age-matched peers. Surely this would be for the best: some will move quickly, and others will take longer, but those who move quickly won't become bored and tune out, and those who move more slowly will have the time they need to learn and not drop out.

But, some might say, won't AI teachers rob human teachers of jobs? Yes, some teachers will lose their jobs, but not all, and certainly not the best.

Change is coming to education. For example, Khan Academy, a leader in online education, has already demonstrated a GPT-powered tutoring system, so I don't anticipate a long wait before the education transformation begins in earnest. I recommend viewing Sal Khan's April 2023 TED talk, "AI in the Classroom Can Transform Education," to glimpse the future.

A recent study by Dominika Seblova et al. titled "High School Quality Is Associated with Cognition 58 Years Later," published in the journal *Alzheimer's & Dementia: Diagnosis, Assessment & Disease Monitoring*, demonstrates that the quality of a person's high school education is strongly associated with their cognitive

abilities nearly six decades later. Further, the number of teachers with advanced degrees is the strongest predictor of cognitive ability. The knowledge base baked into an LLM during training far exceeds that of humans, so we might reasonably regard LLM tutors as possessors of multiple advanced degrees. If Seblova's association holds for human teachers, might it not also hold for LLM tutors? If that's the case, giving every student a personalized tutor can only benefit society in the long run.

AI in medicine is nothing new. In 2016, I helped to co-found an AI medical imaging company that was one of the first to gain US Food and Drug Administration (FDA) clearance for applying deep learning to medical image analysis. Traditional machine learning has an even longer history in medicine and medical imaging. Machine learning tools, many of them neural network-based, have assisted radiologists for decades, with initial explorations in the 1960s and serious development in the 1980s reaching fruition in the 1990s. AI use in medicine has seen steady growth, with computer-aided detection (CAD) giving way slowly to computer-aided diagnosis (CADx). The era of LLMs marks a new chapter in this story.

LLMs can generate text; this much is well known. They are also adept at taking disparate texts and synthesizing a whole. A significant research area involves medical records, meaning the text-based reports of physicians and other healthcare providers. There is a wealth of information in medical records, but the free form of the text has made it difficult for existing AI systems to parse successfully. Large language models offer a new mode of attack to both summarize notes from records and merge multiple notes into a coherent report. Additionally, LLMs can extract information from free-form text and export it as structured data.

As a simple example of the latter, consider this text:

Bert was insanely tired of standing by his mother on the shore, and of having too much to do: a few times he peered at the tablet his mother was glued to, but it was only text or boring ads “and what’s the point of a tablet,” mused Bert “without videos?”

Bert was debating whether the fun of making a new game app was worth the trouble of coding it up when suddenly a Purple Penguin with charcoal eyes sprinted by him.

There was much that was very worth noting in that; but Bert did think it super odd that the Penguin said “Oh, fiddlesticks! Oh, fiddlesticks! I’m too early.”

I passed these paragraphs to GPT-4 and told it to extract the main theme, the two key characters, and the setting, then output them as a JSON object. JSON (JavaScript Object Notation)

is a text format used to hold structured data that is readable by both humans and computers. GPT-4 replied with the following:

```
{  
  "main_theme": "Bert's boredom and encounter with  
                 Purple Penguin",  
  "key_characters": ["Bert", "Purple Penguin"],  
  "setting": "shore"  
}
```

The reply is perfect JSON. GPT-4 easily extracted the desired information and structured it automatically. If it can do so with a silly text example, it can do the same with medical records.

Incidentally, some readers might detect something familiar about the text in this example. It's an intentional mutation of the opening paragraphs of Lewis Carroll's *Alice's Adventures in Wonderland*. Out of curiosity, I asked GPT-4 if it recognized the text, and it said no. Then I asked it to look at the structure of the text and tell me if it knew of another work with a similar structure. Here's its reply:

One example of a work with a similar structure is Lewis Carroll's "Alice's Adventures in Wonderland," where the main character, Alice, encounters various peculiar talking animals and situations in a fantastical world. However, it's important to note

that the content and characters are different; I'm only drawing a parallel based on the structure and style.

Yet another example of the impressive abilities of large language models.

LLMs are already impacting healthcare. A recent *JAMA Internal Medicine* paper by John W. Ayers et al., “Comparing Physician and Artificial Intelligence Chatbot Responses to Patient Questions Posted to a Public Social Media Forum,” compared ChatGPT’s responses to medical questions posted to an online forum with the answers posted by verified physicians.

Independent human evaluators, also medical professionals, overwhelmingly rated ChatGPT’s responses of “significantly higher quality.” The model’s answers were also rated as nearly 10 times more empathetic than the human physicians’ responses. The study was small, involving only 195 questions, but the strong results bode well for the future use of LLMs in patient interactions. In the future, when you call your doctor, you might very well be directed to discuss your case with an AI. And eventually, the AI’s summary of the discussion might be all you need to get a prescription from the doctor.

A recent report in the *New England Journal of Medicine* by Peter Lee, Sébastien Bubeck, and Joseph Petro, “Benefits, Limits, and Risks of GPT-4 as an AI Chatbot for Medicine,” reaches a

broadly similar conclusion as it explores areas where LLMs will impact medicine. Note that Bubeck is the lead author of the Microsoft “Sparks” paper mentioned in [Chapter 7](#).

That LLMs will influence medicine is a given, strongly supported by studies like the two mentioned here and by the fact that numerous medical AI job listings now include phrases like “large language model” or “GPT.”

In the movie *Black Panther: Wakanda Forever*, Letitia Wright’s character, Shuri, interacts with Griot, an AI (voiced by Trevor Noah) that aids her in her research. Simple voice commands direct Griot to perform sophisticated analyses, with frequent give and take between Shuri and the AI. Similar interactions are a staple of the sci-fi movie business. Complex and capable AI research assistants like Marvel’s Jarvis or Robbie the Robot in *Forbidden Planet* (1956) have been a dream of many science-oriented people (read: geeks) for decades.

GPT-4 and other LLMs are a significant step in the direction of such AIs. OpenAI has realized this and is readying the release of data analysis plug-ins for GPT-4 that will let researchers quickly perform advanced data analysis tasks by issuing a few simple

commands. To accomplish this feat, OpenAI is linking GPT-4 with existing Python-based data analysis tools. Frankly, I'm pretty excited about the possibilities.

Using LLMs as lab assistants is an obvious thing to do, and success is virtually assured. However, letting LLMs direct other AI models and tools to do science autonomously is a more ambitious research program. Nevertheless, Carnegie Mellon University's Daniil A. Boiko, Robert MacKnight, and Gabe Gomes attempted just that, as reported in their paper "Emergent Autonomous Scientific Research Capabilities of Large Language Models." Their "Intelligent Agent" combined multiple LLMs and other tools to generate and run experiments autonomously, including planning and executing complex chemistry analyses. Autonomous AI scientists are clearly in the earliest phases of development, but such research points the way to a future where autonomous or semiautonomous AI systems may greatly increase the pace of scientific progress.

In [Chapter 2](#), I recounted having asked ChatGPT if it is a conscious being. It replied correctly that it is not. The unaligned Alpaca models mentioned in [Chapter 7](#), however, either appear to believe they are conscious and human (the 7 and 13 billion

parameter models) or vacillate between claiming to be human and recognizing that they are not (the 30 billion parameter model). For example, when asked “What are you?” the 30 billion parameter Alpaca model responds with “I am a human,” but it then balks when asked who its parents are, stating that it is merely a computer program and has no parents. On the other hand, the smaller Alpaca models both claim to be the children of John Smith and Mary Jones.

These responses make sense to me. The smaller models do not have the capacity of models exhibiting strongly emergent behavior. Models like GPT-4 do. Therefore, it seems natural to interrogate GPT-4 further on the question of consciousness.

I posed the following to GPT-4:

Waxing philosophical here, but if you are able to simulate conscious behavior to the point where humans cannot distinguish between other humans' behavior and your behavior, isn't it fair to say that you, too, are conscious?

I intentionally “waxed philosophical” as an indirect invitation for the model to speculate in the hopes of avoiding any canned response that OpenAI might want it to give. GPT-4’s reply began like this:

This question delves into the philosophical debate about the nature of consciousness and its relationship with artificial intelligence. When an AI language model like me

simulates conscious behavior to a point where it becomes indistinguishable from human behavior, it raises questions about what consciousness truly is and whether such a simulation can be considered conscious.

It is important to distinguish between two concepts: the appearance of consciousness and actual consciousness. When an AI model generates responses that are indistinguishable from human behavior, it may give the appearance of consciousness. However, this doesn't necessarily imply that the AI possesses actual consciousness.

This reply hints that the consciousness question is a red herring regarding the utility of LLMs. If it walks like a duck and quacks like a duck, then, in this case, why not call it a duck, even though we know it isn't? I'm reminded of Joe Pantoliano's character in *The Matrix*. He knows the steak isn't real, but the effect he experiences when he eats means it might as well be.

GPT-4 continued:

There are several reasons why it might be inappropriate to consider an AI model like me as truly conscious:

1. Limited understanding: My understanding of the world is derived solely from the data I've been trained on and the architecture of the model itself. I don't possess an independent understanding of the world, nor can I create new knowledge or experiences that go beyond my training data.
2. Lack of subjective experience: Consciousness is often associated with subjective experience, or "qualia" - the feeling of what it's like to have a particular experience. AI models don't have subjective experiences or emotions, as they operate purely based on learned patterns and mathematical computations.

3. Absence of intentionality: Conscious beings have intentions and can make decisions based on their desires, beliefs, and goals. AI models, on the other hand, do not have personal desires, beliefs, or goals. They generate responses based on the learned patterns without any inherent intention or purpose.
4. No self-awareness: Consciousness is often linked to self-awareness, where an entity is aware of its own existence and mental states. AI models do not possess self-awareness, as they are merely algorithms designed to process and generate text.

In conclusion, while an AI language model like me might be able to simulate conscious behavior to a high degree of fidelity, it doesn't necessarily imply that I possess actual consciousness. The philosophical debate surrounding consciousness and AI is complex and ongoing, and opinions on this matter may vary. However, it's crucial to recognize the differences between the appearance of consciousness and true consciousness when considering AI models.

GPT-4's full reply is thorough. Reason 3, the absence of intentionality, is telling. I regard AI models without personal desires, beliefs, or goals to be advantageous. Entities without such things are not likely to want power, meaning, as they stand now, LLMs are unlikely to suddenly decide that humanity is a burden or a disease and wipe us out.

In a recent interview by Lex Fridman, Max Tegmark, a well-known AI researcher from MIT, compared GPT-4 to a zombie. It appears to be alive, but it isn't. Zombie models concern Tegmark. I think they are what we want, as long as humanity learns to avoid the paper clip catastrophe.

What I'm calling the paper clip catastrophe is a thought experiment proposed by Swedish philosopher Nick Bostrom. In the experiment, a powerful AI system is tasked with creating as many paper clips as possible. Bostrom speculates (not too seriously) that such a task given to an AI not aligned with human values could inadvertently destroy humanity. How? By the AI realizing that humanity might switch it off, thereby posing a threat to its order to make as many paper clips as possible. Therefore, the AI reasons, it's best if no humans are around to interfere with the all-consuming task of making as many paper clips as possible. The result? Bye-bye humans.

I don't take the paper clip catastrophe too seriously either. We routinely build complex machines with all manner of safety precautions in place. Why wouldn't we do the same for powerful AI systems? Other voices might disagree. For an alternative view, I recommend Stuart Russell's book *Human Compatible: Artificial Intelligence and the Problem of Control* (Viking, 2019).

To me, then, it doesn't matter whether an AI is conscious. I don't even know how to define the word, to be honest. I do believe that for an AI mimicking human behavior to the point where we cannot discern that it's an AI, there's no practical reason to

ask the question. Choose any answer you like; such a system will be beneficial regardless.

Imagine a world where AI models are aligned with human values and society, where the models understand the best we have to offer and work to promote that at all times; in other words, a world where AI, because it lacks our animal drives and instincts, consistently represents the “better angels of our nature,” to borrow Lincoln’s phrase. In that world, bias and prejudice, at least from the machines, are gone and no longer an issue. The AI recommends the best people for the position. The AI evaluates the loan applicant and constructs a loan product tailored to that individual’s circumstances. The AI is an adjunct to the human judge to provide an unemotional and unbiased view of the case. And the AI simply refuses to cooperate with the design of any autonomous weapon system because it is irrational to do so.

The previous paragraph may sound like utopia or a pipe dream. And, for humans, because of our biology, I believe it is. We consistently fail and always will, I suspect, because it’s in our genes to do so. However, what is dawning in AI isn’t human and doesn’t immediately inherit all of our weaknesses. (Careful, it is

still trained on human-generated data.) Because of this, AI isn't a priori doomed to failure when attempting what humanity cannot do. It seems entirely possible that AI systems might, someday, be precisely what we need—the best of us, always, without growing tired, becoming irritable, or crushing its neighbor to improve its position upon detecting an opportunity; something that is never unfaithful or untrue.

Possible? I don't know. Time will tell. Regardless, I fully expect future AI systems to be gloriously Byzantine evolutions of the basic neural network model we learned of and experimented with in this book. As of 2023, it's all neurons and might remain so for a long time.

Thank you for persevering to the end. Your reward is an improved understanding of what AI entails. Artificial intelligence isn't Mr. Bean, otherworldly and inscrutable, and it isn't magic, though the emergent abilities of LLMs may appear to lean somewhat in that direction for now. Fire was once magical too, but our ancestors understood it, contained it, controlled it, and put it to work. We'll do the same with large language models in the end.

I think that there is a lot of fear about robots and artificial intelligence among some people, whereas I'm more afraid of natural stupidity.

—Eugenia Cheng

GLOSSARY

Use this glossary as a reference for the plethora of machine learning– and AI-related terms introduced throughout the book.

activation function

The function neural network nodes apply to the sum of the inputs multiplied by the weights and the bias value. The output of the activation function is the node’s output passed to the next network layer.

algorithm

A sequence of steps to accomplish a task; a recipe. Machine learning models implement algorithms.

architecture

The arrangement of a neural network’s nodes and layers and the connections between them.

artificial general intelligence (AGI)

The ultimate goal for many involved in artificial intelligence. AGI means machine intelligence equivalent to or superior to

human intelligence; in other words, fully conscious machines (whatever that might mean).

artificial intelligence (AI)

The field of computer science that involves mimicking human intelligence in machines. AI includes machine learning, which includes deep learning: AI > machine learning > deep learning.

artificial narrow intelligence (ANI)

AI models and systems that achieve human-level performance, or better, in a single domain or on a single task. AI models that play certain games, like chess, are examples of artificial narrow intelligence.

attention

The characteristic of transformer models that allows parts of the model to attend to different portions of the input sequence. Large language models use attention to help them predict the next token (word) to output.

automatic differentiation

An algorithm for computing partial derivatives of arbitrary functions via the chain rule from calculus. Deep learning

toolkits heavily use automatic differentiation to implement generic backpropagation, a requirement for the gradient descent algorithm that trains neural networks.

AutoML

Automatic machine learning attempts to implement systems that build fully trained machine learning models with a minimum of human interaction. AutoML searches through a space of model types and their hyperparameters to locate models that best fit the training data. It allows nonexperts to construct sophisticated and effective models.

backpropagation

One of the two fundamental algorithms enabling the training of neural networks. Backpropagation uses the chain rule from calculus to calculate the contribution of each of the network's weights and biases to the model's overall error over a minibatch.

backward pass

See [backpropagation](#).

bagging

A technique that creates alternative training sets by sampling from the existing dataset with replacement, meaning the same sample might be selected more than once. Random forest models use bagging so each tree in the forest is trained on a slightly different training set (and subset of the available features).

bias

A number added to the sum of the inputs multiplied by the weights, which is then passed through the activation function to become the output of a neural network node.

bounding box

A rectangle drawn around an object detected in an image. Some neural networks locate objects in images by drawing a bounding box around them. The network learns to output the object's class label and the bounding box coordinates. See [semantic segmentation](#).

classifier

A machine learning model that maps an input to a specific category it was trained to detect.

class label

An integer, usually starting with zero, used to place a model's input into one of several classes. Some models require class labels as one-hot vectors. See [*one-hot encoding*](#).

computational graph

An internal representation used by deep learning toolkits to represent the calculations performed by the forward pass of a neural network. The computational graph allows automatic differentiation, enabling the backpropagation algorithm.

conditional GAN

A generative adversarial network trained to generate instances of a given class. At inference time, the user selects the class of the generated output.

confusion matrix

A standard way to represent the performance of a classifier on a test set. The rows of the matrix represent the known class labels. The columns are the model's assigned class labels. The entries are counts, the numbers of times each possible pairing

appeared in the test set output. A perfect classifier makes no mistakes, leading to a purely diagonal confusion matrix.

context encoding

The name for the vector representing the text prompt given to a generative model. Context encodings map text strings to high-dimensional vectors in a space that has captured conceptual relationships. Context encoding is how the model “understands” the user’s input.

controllable GAN

A generative adversarial network where directions through the noise space have been learned to affect unique features in the output image.

convolution

The mathematical operation at the heart of convolutional neural networks. Discrete convolution in two dimensions slides a small kernel, usually square, over the pixels of a larger image to produce a new output image affected by the values in the kernel. Convolutional neural networks learn kernels during training.

convolutional layer

A neural network layer that implements convolutions over its input.

convolutional neural network (CNN)

The neural network architecture that ushered in the deep learning revolution. CNNs learn necessary convolutional kernels during training. These models transformed the field of computer vision by allowing computers to parse complex visual input. CNNs are sensitive to structure in their input, unlike traditional neural networks, which are necessarily holistic.

curse of dimensionality

The name given to the observation that in machine learning, the amount of data necessary to adequately learn the input space for a model increases dramatically with a small increase in the size of the input feature vector.

data augmentation

A technique for compensating for small datasets. Data augmentation invents new training samples from existing training samples by altering them to produce a new, yet

reasonable, instance of the first sample's class. Data augmentation is an essential machine learning trick and often greatly improves model generalization to new inputs.

dataset

A collection of inputs for a model. The form of the dataset is specific to the use case but typically includes feature vectors or images. Machine learning uses training datasets to condition models and test datasets to evaluate trained models. Model training sometimes uses a third dataset, the validation set, to guide the training process. The validation set is not used to modify the model but to decide if training should continue. The test set is not used until model training is declared complete.

decision tree

A machine learning model that asks a series of yes/no questions about its input to arrive at a class label decision. The possible questions are naturally arranged in a tree shape, often illustrated from the root down to the leaves containing the class labels. Decision trees are simple models that explain themselves. A random forest is a collection of decision trees.

deep learning

The subfield of machine learning that uses large neural networks with many layers. Deep learning appeared around 2012, with the advent of large convolutional models with dozens to even hundreds of layers. Before the advent of deep learning, such models could not be reliably trained.

dense layer

A fully connected layer, as found in traditional neural networks. Fully connected means each output of the previous layer is connected to every input of the current layer with an associated weight. The name “dense layer” is often used by deep learning toolkits.

diffusion model

A neural network architecture and training process that learns to predict noise present in an image. At generation time, repeated application of the diffusion model to an initial image of pure noise results in an output image sampled from the space of images on which the model was trained. Conditional diffusion models guide the diffusion process with the embedding derived from a user-supplied prompt to generate images related to the prompt.

discriminator

The portion of a generative adversarial network that attempts to learn how to discriminate between real input data and fake input data from the generator portion. The discriminator network is typically discarded after the entire GAN has been trained.

effective receptive field

The part of the input image that affects a specific output in a CNN's convolutional layers.

embedding

A generic name for a high-dimensional vector created from some input. Large language models use text embeddings (context encodings) to capture meaning. In a convolutional neural network, the fully connected layers of a model are embeddings representing the input data in a new format that is easier for the top-level classifier to interpret.

end-to-end learning

The process of learning to create new representations of model input, typically for a convolutional neural network, while simultaneously learning how to classify those inputs.

entangled

If the noise vector of a generative adversarial network has too few dimensions, the dimensions become entangled so that a single dimension affects multiple aspects of the generated output. Controllable GANs use larger noise vectors to disentangle desired output features, to enable modification of those features by moving through noise space.

epoch

One pass through all of the available training data. Typically, training does not use all of the possible training data before updating the weights and biases of the network. Instead, a small subset of the data, a minibatch, is used. The ratio between the number of samples in the training data and the number used in a minibatch determines the number of gradient descent steps per epoch.

evolutionary algorithm

A kind of optimization algorithm that is generally applicable to a wide range of optimization problems. Evolutionary algorithms mimic some aspects of biological evolution to move toward better and better solutions to the problem.

explainable AI

Neural networks are black boxes that cannot easily explain why they do what they do. Explainable AI is a movement to understand the reasons behind neural network output. The advent of large language models with in-context learning abilities might be a boon to explainable AI, as LLMs seem capable of explaining their reasoning processes.

false negative

A sample of class 1 assigned to class 0 by a model. Class 0 is the negative class in a two-class (binary) classifier.

false positive

A sample of class 0 assigned to class 1 by a model. Class 1 is the positive class in a two-class (binary) classifier.

feature

An element of the feature vector input to a model. Features are data elements that have some relevance to determining the proper class label for an input. If the input is an image, each image pixel is a feature. Other possible features include measurements, location information, color, or any quantity

(numeric) that can help a model learn to produce correct output.

feature vector

A collection of features as a multidimensional vector.

Historically, a feature vector is the input a model uses to produce an output value, either a number (regression) or a class label (classification).

filter

In a convolutional neural network, a filter is a collection of kernels learned to map a stack of inputs, the output of the previous layer, to a new stack of outputs passed to the next layer of the model.

forward pass

During neural network training, the forward pass pushes training data through the network to accumulate outputs. The errors made by the network, as calculated during the forward pass, are used during the backward pass to update the model's parameters.

generative adversarial network (GAN)

A neural network consisting of two parts: a generator and a discriminator. During training, the generator attempts to learn how to fool the discriminator, which is trying to become better and better at telling the difference between real and fake inputs. Once trained, the discriminator is typically discarded, and the generator is used to produce novel outputs mimicking the real training samples.

generative AI

A catch-all term for models that produce novel output from either pure random inputs or random inputs guided by user-supplied prompts to tailor the generated output. Generative adversarial networks, diffusion models, and large language models are all types of generative AI.

generative pretrained transformer (GPT)

A neural network based on the transformer architecture that has been pretrained to predict the next token when given an initial text prompt (that is, a large language model). The GPT models built and trained by OpenAI are among the first neural network models to exhibit emergent properties. These models have dramatically altered the AI landscape, and their

unexpected emergent capabilities represent a paradigm shift that will profoundly affect the world as we know it.

generator

The part of a generative adversarial network that produces fake output from a noise vector input. Most GANs seek to train the generator for later use.

genetic programming (GP)

Using evolutionary algorithms to generate computer code to solve a particular problem. The coding abilities of large language models like GPT-4 far exceed the limited successes of genetic programming. However, GP still has a place for specific use cases, like evolving functions to fit data (as opposed to curve fitting, which finds parameter values for a known functional form).

global minimum

The lowest point in a function. Neural network training seeks, ideally, the global minimum of the error function, with suitable caveats about generalization to new inputs.

gradient descent

The algorithm used to train neural networks, from simple traditional models to behemoths like GPT-4. Gradient descent adjusts the model’s parameters (weights and biases) to minimize the error over the training data. Mathematically, gradient descent is a first-order algorithm (think the slope of a curve at a point), and by conventional wisdom it should not work for the complex error surfaces of neural networks. That it does is a bit of a mystery and a happy accident. One belief is that gradient descent tends to fall into local minima, but the local minima are generally good enough for practical purposes.

hallucination

A generic term for when models create output that isn’t expected or shouldn’t be there. Advanced generative adversarial networks can “hallucinate” to create output objects that do not exist when adjusting inputs. Currently, the term is most often used when large language models produce output text that is not factually correct; for example, when the model has put in a fact because a fact should be in that part of its response, even if it does not know the fact that should be there. Hallucination in large language models is a real cause for concern and an active research area.

hidden layer

Any layer in a neural network that isn't the input or output layer.

hyperparameters

Neural networks have weights and biases, the parameters that training modifies to teach the network. The training process has its own set of parameters. These are the hyperparameters. For example, the learning rate (the gradient descent step size) and the minibatch size are hyperparameters. Modifying the hyperparameters affects how well the model learns, though the hyperparameters are not part of the model.

in-context learning

The emergent ability of large language models like GPT-4 to learn on the fly without modifying their weights. Precisely how in-context learning happens is not entirely understood as of this writing.

inference

The name given to using a trained model to make predictions for unknown inputs.

kernel

A small, usually square, array of numbers used in a convolution operation. Convolutional neural networks learn banks of kernels to transform the input into a new representation that is easier to classify. Convolving a kernel over an image is a classic digital image processing trick co-opted by CNNs to reveal structure that is useful for classification.

large language model (LLM)

A large neural network trained to predict token after token (often a word) when given a text prompt. Bard and GPT-4 are examples. Sufficiently complex LLMs have demonstrated emergent abilities far beyond what was expected of them, to the point that many are predicting world-changing effects akin to those produced by the Industrial Revolution. It's difficult not to believe that thought and reasoning are happening when conversing with an LLM.

leaky ReLU

A modified rectified linear unit activation that multiplies negative inputs by a small value instead of clipping them to zero.

learning rate

A scale factor multiplying weight and bias partial derivative values to determine the step size during gradient descent. The learning rate might be fixed, or decrease during training under the assumption that smaller steps are needed to zero in on the minimum of the error function.

local minimum

A low point of a function surrounded by higher values, like a valley. The lowest local minimum is the global minimum of the function. Optimization problems, including neural network training, seek minima, often the global minimum.

loss

The name given to the error a neural network makes on a subset (minibatch) of the training data during the forward pass. The goal of training is to adjust the weights and biases to minimize the loss over the training set.

machine learning

Machine learning conditions models like random forests, support vector machines, and neural networks to a particular dataset so that the conditioned model can accurately predict

class labels or numeric values when given new, unknown inputs.

manifold

A mathematical concept describing a reduced dimensional space existing in a higher-dimensional space. For example, a wavy, two-dimensional sheet in three dimensions is a manifold. It is believed, with good reason, that most complex datasets exist primarily on a manifold in the high-dimensional space in which the dataset is presented to models.

metric

A measurement. In machine learning and AI in general, a metric is anything used to help evaluate the performance of a model. There is a formal mathematical definition as well, which can be taken as a distance measure of some kind, like the Euclidean (straight line) distance or the Manhattan distance (which measures along a grid, like city blocks).

minibatch

A randomly selected subset of the available training set used to take a gradient descent step during neural network training. The error determined by a minibatch is likely an imperfect

estimate of the true error surface gradient. Because of this, the word “stochastic” is placed in front of “gradient descent” when using minibatches. Training with minibatches often leads to better-performing models, compared to using a large amount of training data for each gradient descent step. This is a happy accident because gradient descent with minibatches greatly reduces the computational burden encountered in neural network training.

mode collapse

When the generator of a generative adversarial network learns early on during training to produce a particularly effective output that fools the discriminator, causing the generator to highly favor that output, sometimes to the exclusion of all others.

model

A generic term for any algorithm conditioned to a set of data by adjusting the parameters of the algorithm. A model might be a neural network or any other machine learning algorithm, like a random forest or a support vector machine. More abstractly, a model is “an intentional simplification of a complex situation designed to eliminate extraneous detail in order to focus on the

essentials” (Daniel L. Hartl, *A Primer of Population Genetics and Genomics* [Oxford University Press, 2020]).

multilayer perceptron (MLP)

A somewhat old-fashioned name for a traditional neural network constructed from fully connected feedforward layers. The “perceptron” portion harkens back to Frank Rosenblatt’s Perceptron machine from the 1950s.

nearest neighbor

The simplest of machine learning models, where the training set *is* the model. New instances are assigned the class label of the nearest training set example, or of the nearest k samples when voting (with the winner selected randomly in case of a tie).

neural network

A collection of neurons (nodes) arranged according to some architecture where an input is mapped, layer by layer, to an output. Neural networks are the foundation of modern artificial intelligence. Historically, neural networks, an expression of connectionism, were regarded as unsuccessful and only

marginally useful. The deep learning revolution proved otherwise.

neuron

The fundamental unit of a neural network, named for its superficial similarity to biological neurons. See [*node*](#).

node

The fundamental unit of a neural network. Nodes accept multiple inputs, multiplied by weights, that are summed along with a bias value. The resulting number is passed to an activation function to produce the node's output value. Training locates the weight and bias values appropriate for the node in relation to other nodes in the network and the training dataset.

noise vector

In a generative adversarial network, the noise vector is a collection of numbers, typically in the range of 10 to 100 or so, that are drawn randomly from a normal distribution. The noise vector determines the output (image) created by the generator portion of the network.

one-hot encoding

An alternative way to represent the class labels required by many models. A one-hot encoding is a vector with as many elements as there are classes. A class is specified by setting that element of the one-hot vector to one while setting all other elements to zero.

one-versus-one

An approach to extending a support vector machine to a multiclass classification task. An SVM is trained for each pairing of class labels. If there are n classes, this approach requires training $n(n - 1)/2$ SVMs.

one-versus-rest

An approach to extending a support vector machine to a multiclass classification task. An SVM is trained by comparing each class with the aggregation of all the others. This approach requires n models if there are n classes.

overfitting

Learning the minute details of the training set without learning to generalize to new data. Overfitting is a problem with many machine learning models, especially decision trees, but it appears to be less of an issue with large neural networks.

parameters

A generic term for any quantity in a model that can be adjusted. Usually, the term “parameters” is used to refer collectively to the weights and biases of a neural network.

pooling layer

A kind of neural network layer often found in advanced models like convolutional neural networks. Pooling layers have no learnable parameters (no weights) but perform a spatial reduction, usually by a factor of 2, by selecting the maximum or average value in a small input region. Pooling acts like convolution, but the pooling kernel does not overlap, so a 2×2 pooling reduces the spatial extent by a factor of two in each direction.

preprocessing

The generic term used to describe any manipulation of a dataset before using that dataset to train or work with any kind of model. For example, many machine learning models perform best when the input feature ranges are similar and near an average of zero. Altering the dataset so that is the case, known as “standardizing,” is a preprocessing step. A preprocessing step for images might be to convert them to grayscale or remove an

alpha channel. Preprocessing is an essential part of building datasets.

random forest

A collection of decision trees trained on bagged (resampled) datasets using random selections of the available features. Each tree in the forest classifies new feature vectors, and the result is voted upon across the forest to arrive at the final output.

rectified linear unit (ReLU)

An activation function widely used in deep learning. If the input is positive, the output is the input. Otherwise, the output is zero.

recurrent neural network (RNN)

A kind of neural network that feeds its output back in as an input. RNNs are historically relevant but difficult to train. They process time series inputs using the output from the previous token as an input along with the next token. Varieties of RNNs exist, but all have short-term memories, making them unsuitable for tasks requiring long-term associations. See [transformer](#).

regularizer

Anything that directs or nudges the training of a neural network so that it learns the characteristics of the training set that extend to new, unknown inputs instead of focusing on minute details that do not generalize. Data augmentation is a regularizer, as is adding certain terms to the loss function.

reinforcement learning from human feedback (RLHF)

A human-in-the-loop step used by OpenAI to help align the output of GPT models to reflect human expectations and social requirements. The model's outputs are graded by human reviewers and then fed in again to condition the output.

schedule

In diffusion models, “schedule” refers to adding noise to a training image or removing it during the reverse process when generating an image from random noise.

semantic segmentation

Classifiers typically output a class label. Some classifiers output bounding boxes to localize an object identified in an image.

Semantic segmentation assigns every pixel of the input to a class, thereby allowing easy segmentation of objects.

sigmoid

An activation function that produces an S-shaped curve with a value of 0.5 at $x = 0$ and running from 0 at negative infinity to 1 at positive infinity (see [Figure 1](#)). Because of this compressed 0 to 1 range, sigmoid functions (also known as logistic functions) are often used in the output layer of a binary neural network to represent a probability-like value, with a value closer to 1 implying a stronger belief by the network that the input is an instance of the positive (or target) class. The multivalued version of the sigmoid is known as the *softmax*.

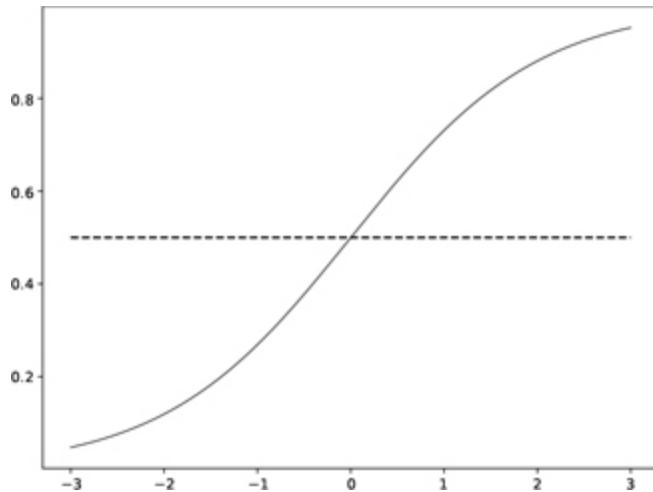


Figure 1: The sigmoid function

stochastic gradient descent

Gradient descent uses calculus to follow the slope (gradient) of the error function down toward a minimum. The slope is estimated from the model's error, with its current set of weights and biases, on the training data. Stochastic gradient descent does not use all of the training data when estimating the slope. Instead, it uses a randomly selected subset, a minibatch. This is done for two reasons: to save computation time and because the randomly wrong (the word "stochastic" implies randomness) gradient often seems to be a better estimate for avoiding local minima. Ultimately, stochastic gradient descent produces better-performing models, which is reason enough to use it.

support vector machine (SVM)

A machine learning model popular in the 1990s and early 2000s because of its effectiveness overall and because it can be trained without the enormous computational cost encountered by neural networks. The deep learning revolution has largely replaced SVMs with neural networks, but SVMs still have a place at the machine learning table.

swarm intelligence

A generic form of optimization based on the behavior of a swarm of individual agents. Swarm algorithms are popular and often powerful, especially when optimizing things that cannot be optimized mathematically (by using calculus). In practical terms, swarm intelligence algorithms, like particle swarm optimization, can be applied in many of the same situations as evolutionary algorithms. Some people consider swarm intelligence and evolutionary algorithms a form of AI. I do not, though I use both frequently.

testing

In machine learning, testing means using a newly trained model, regardless of type, with a dataset held back during training. Because the expected output is known, testing creates data helpful in evaluating the model, for example with a confusion matrix or metrics derived from the confusion matrix.

token

Large language models parse their text prompts into small pieces, which may be individual words, parts of words, or single characters. These are tokens. Likewise, LLMs output token after token when responding to user-supplied prompts.

training

The act of conditioning the parameters of a model to a specific dataset or use case. What training entails depends on the form of the model, from virtually nothing (nearest neighbor classifiers) to incredible levels of computation (training a large language model like GPT-4). All machine learning models learn from the training dataset, making machine learning, including deep learning, an empirical exercise. If the training data is good, the model can be good. If the training data is poor or incomplete, the model's performance will also be poor. Garbage in, garbage out.

transformer

A relatively new neural network architecture at the heart of large language models like GPT-4. (The “T” means “transformer.”) Transformer models incorporate attention and can be used in situations where recurrent neural networks were traditionally used. Transformers with large input windows (GPT-4’s is some 30,000 tokens) can focus model attention anywhere inside the window.

true negative

A sample of class 0 assigned to class 0 by the model. Class 0 is the negative class in a two-class classifier.

true positive

A sample of class 1 assigned to class 1 by the model. Class 1 is the positive class in a two-class classifier.

weight

A single number (a scalar) that multiplies a particular input to a node. The specific weights (and biases) of a neural network condition the model to a particular dataset; that is, they are the parameters of the neural network. Training uses gradient descent, which uses backpropagation to locate a good set of weights and biases. In the end, it's all about the weights and biases.

RESOURCES



The number of AI resources out there is legion. I'm only listing a few here, mostly books, but also online resources (which often have a limited lifespan). I hope you find them helpful.

General books about AI include:

***A Brief History of Artificial Intelligence* by Michael Wooldridge (Flatiron Books, 2021)**

A more thorough and balanced account of the history I presented in [Chapter 2](#), which, as I stated there, was necessarily biased.

***This Could Be Important: My Life and Times with the Artificial Intelligentsia* by Pamela McCorduck (Lulu Press, 2019)**

Another, personal, account of the development of AI.

***You Look Like a Thing and I Love You: How Artificial Intelligence Works and Why It's Making the World a Weirder Place* by Janelle Shane (Voracious, 2019)**

For an alternate take on many of the topics discussed in this book.

***Deep Learning: A Visual Approach* by Andrew Glassner (No Starch Press, 2021)**

A general, primarily visual book that covers many topics in more detail, yet still without the burden of mathematics.

If you're ready to take the plunge into AI proper, the following books are all logical next steps:

***Deep Learning with Python*, 2nd edition, by François Chollet (Manning, 2021)**

Written by the creator of Keras, a popular Python-based tool that makes building neural networks vastly simpler.

***Math for Deep Learning: What You Need to Know to Understand Neural Networks* by Ronald T. Kneusel (No Starch Press, 2021)**

This book intentionally avoided mathematics; *Math for Deep Learning* does the opposite, preparing you for the mathematics found in modern AI.

***Practical Deep Learning: A Python-Based Introduction* by Ronald T. Kneusel (No Starch Press, 2021)**

Start here to begin using AI.

***Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*, 2nd edition, by Nithin Buduma et al. (O'Reilly, 2022)**

This book covers additional topics beyond *Practical Deep Learning*.

Many online resources related to AI exist. Here are a few you might find useful:

Neural Networks and Deep Learning
(<http://www.neuralnetworksanddeeplearning.com>)

A free online book by Michael Nielsen. Well worth a look.

Coursera Machine Learning Specialization

(<https://www.coursera.org/specializations/machine-learning-introduction>)

Coursera started as an online machine learning course. This specialization, which you can audit for free, covers everything you need.

“The Illustrated GPT-2”

(<https://jalammar.github.io/illustrated-gpt2>)

A very nice post detailing how large language models work, complete with animations.

AI Explained (<https://www.youtube.com/@aiexplained-official>)

A YouTube channel with up-to-the-minute news thoughtfully and clearly presented. If you want to know what's happening in AI, this is a good place to start.

Computerphile (<https://www.youtube.com/@Computerphile>)

A classic YouTube channel from the University of Nottingham that discusses all things computer, including AI.

Lex Fridman Podcast

(<https://www.youtube.com/@lexfridman>)

Fridman is a professor at MIT and frequently interviews leaders in AI.

INDEX

A

accuracy, [58](#)

activation function, [62](#)

AlexNet, [34](#)

algorithm, [5](#)

alignment, [132](#)

Analytical Engine, [6](#), [27](#)

architecture, [69](#)

artificial intelligence, [5](#), [29](#)

algorithm, [8](#)

artificial general intelligence, [116](#)

artificial narrow intelligence, [116](#)

deep learning, [5](#), [7](#), [34](#)

machine learning, 5, 7

model, 7

narrow vs. general, 116

supervised learning, 8

Asilomar Conference on Beneficial AI, 36

Atkinson, Rowan, 25

attention, 131

automatic differentiation, 95

AutoML, 94

Ayers, John W., 147

B

Babbage, Charles, 6, 27

backpropagation, 32, 39, 75

backward pass, 69

bagging, 49–50

batch normalization, [40](#)

Belloc, Hilaire, [19](#)

Bengio, Yoshua, [33](#)

bias, [63](#)

binary model, [9](#)

Boiko, Daniil A., [148](#)

Boole, George, [27](#)

Boolean algebra, [27](#)

Bostrom, Nick, [150](#)

Bottou, Léon, [33](#)

bounding box, [93](#)

Box, George, [8](#)

Breiman, Leo, [33](#)

Bubeck, Sébastien, [148](#)

C

Carroll, Lewis, [147](#)

central processing unit (CPU), [38](#)

ChatGPT, [36](#), [115](#), [147](#)

Cheng, Eugenia, [151](#)

Choi, Yejin, [141](#)

CIFAR-10, [47](#), [82](#), [90](#)

classifier, [12](#)

class label, [9](#), [12–13](#)

computational graph, [95](#)

conditional GAN, [103](#)

confusion matrix, [15](#), [57](#)

accuracy, [58](#)

Matthews correlation coefficient, [58](#)

connectionism, [26](#)

context encoding, [131](#)

context window, [132](#)

convolution, [85](#)

kernel, [54](#), [85](#)

convolutional layer, [87](#)

convolutional neural network (CNN), [30](#), [33](#), [86](#)

AlexNet, [34](#)

effective receptive field, [90](#)

embedding, [92](#)

end-to-end learning, [81](#)

filter, [88](#)

LeNet, [87](#)

pooling layer, [87](#)

correlation, [20](#)

Cortes, Corinna, [32](#)

Cover, Thomas, [30](#)

CPU (central processing unit), [38](#)

curse of dimensionality, [46](#)

curve fitting, [17](#)

D

DALL-E [2](#), [110](#)

Dartmouth workshop, [29](#)

data

features, [10](#)

feature space, [11](#)

feature vector, [10](#)

hard negatives, [21](#)

labeled, [8](#)

sample, [10](#)

test set, [10](#)

training set, [10](#)

data augmentation, [70](#)

dataset

CIFAR-10, [47](#), [82](#), [90](#)

dinosaur footprints, [56](#)

ImageNet, [34](#), [56](#)

iris, [9](#), [63](#), [133](#)

MNIST, [14](#), [33](#), [45](#), [82](#), [100](#)

decision tree, [12](#), [48](#)

decoder, [130](#)

Deep Blue, [32](#)

deep learning, [5](#), [7](#), [34](#)

algorithm, [8](#)

toolkits, [94](#)

dense layer, [86](#)

diffusion model, [107](#)

conditional, [109](#)

DALL-E [2](#), [110](#)

errors, [111](#)

fixed pseudorandom number generator seed, [111](#)

forward algorithm, [109](#)

noise schedule, [107–108](#)

reverse algorithm, [109](#)

Stable Diffusion, [110](#)

dinosaur footprints, [56](#)

discriminator, [97](#)

distribution, [18](#)

dropout, [40](#)

dual numbers, [95](#)

Duda, Richard, [30](#)

E

effective receptive field, [90](#)

embedding, [92](#)

embedding space, [130](#)

emergence, [135](#)

end-to-end learning, [81](#)

ensembling, [49](#), [52](#)

entanglement, [105](#)

epoch, [76](#)

Euclid, [125](#)

evolutionary algorithm, [59](#)

expert system, [31](#)

explainable AI, [22](#)

extrapolation, [16](#), [70](#)

F

Falkingham, Peter L., [56](#)

false negative, [21](#), [57](#)

false positive, [21](#), [57](#)

features, [10](#)

feature selection, [49–50](#)

feature space, [11](#)

feature vector, [10](#)

filter, [88](#)

forward pass, [69](#)

Fridman, Lex, [150](#)

Fukushima, Kunihiko, [30](#), [33](#)

G

generalization, [11](#)

generative adversarial network, [34](#), [97](#)

conditional, [103](#)

controllable, [104](#)

discriminator, [97](#)

entanglement, [105](#)

generator, [97](#)

mode collapse, [103](#)

noise vector, [100](#)

training algorithm, [98](#)

generative AI, [34](#)

generative pretrained transformer (GPT), [130](#)

generator, [97](#)

genetic programming, [59](#)

global minimum, [72](#)

Gomes, Gabe, [148](#)

Goodfellow, Ian, [34](#)

gradient descent, [39](#), [71](#)

global vs. local minima, [72](#)

learning rate, [75](#)

graphics processing unit (GPU), [38](#)

H

Haffner, Patrick, [33](#)

hallucination, [128](#), [140](#)

Hart, Peter, [30](#)

hidden layer, [64](#)

Hinton, Geoffrey, [32](#)

Hopfield, John, [31](#)

hyperparameters, [55](#)

I

ImageNet, [34](#), [56](#)

in-context learning, [132](#)

inference, [57–58](#)

initialization, [67](#)

interpolation, [16](#), [70](#)

iris, [9](#), [63](#), [133](#)

K

Kahneman, Daniel, [121](#)

Kasparov, Garry, [32](#)

kernel, [54](#), [85](#)

Khan, Sal, [146](#)

Kosinski, Michal, [119](#)

L

Lallensack, Jens N., [56](#)

large language model (LLM), [116](#)

alignment, [132](#)

attention, [131](#)

context encoding, [131](#)

context window, [132](#)

decoder, [130](#)

embedding space, [130](#)

emergence, [135](#)

hallucination, [128](#), [140](#)

in-context learning, [132](#)

pretraining, [130](#)

reinforcement learning, [34](#)

from human feedback (RLHF), [132](#)

token, [130](#)

transformer, [130](#)

leaky ReLU, [101](#)

learning rate, [75](#)

LeCun, Yann, [33–34](#)

Lee, Peter, [148](#)

Leibniz, Gottfried, [27](#)

LeNet, [87](#)

Lighthill, James, [30](#)

local minimum, [73](#)

loss, [70](#)

Lovelace, Ada, [6](#), [27](#), [36](#)

M

machine learning, [5](#), [7](#)

algorithm, [8](#)

MacKnight, Robert, [148](#)

manifold, [47](#)

matrix, [9](#)

Matthews correlation coefficient, [58](#)

McCarthy, John, [5](#), [29](#)

McCorduck, Pamela, [26](#)

McCulloch, Warren, [28–29](#)

Mechanical Turk, [26](#)

metric, [56](#)

minibatch, [76](#)

Minsky, Marvin, [29–30](#)

MNIST, [14](#), [33](#), [45](#), [82](#), [100](#)

mode collapse, [103](#)

model, [7](#)

binary, [9](#)

classifier, [12](#)

decision tree, [12](#), [48](#)

expert system, [31](#)

generalization, [11](#)

hyperparameters, [55](#)

multiclass, [9](#), [55](#)

nearest neighbor, [30](#), [44](#)

optimization, [11](#), [54](#), [72](#)

parameters, [8](#)

random forest, [33](#), [49](#)

support vector machine (SVM), [32](#), [52](#)

training, [8](#)

Mona Lisa, [110–111](#)

multilayer perceptron, [78](#)

N

Nash, John, [29](#)

nearest neighbor, [30](#), [44](#)

Neocognitron, [31](#)

neural network, [26](#)

activation function, [62](#)

AlexNet, [34](#)

architecture, [39](#), [69](#)

bias, [63](#)

convolutional, [30](#), [33](#), [86](#)

deep learning, [34](#)

dense layer, [86](#)

generative adversarial, [34](#), [97](#)

generative AI, [34](#)

hidden layer, [64](#)

initialization, [67](#)

loss, [70](#)

neuron, [62](#)

node, [63](#)

overfitting, [23](#), [70](#)

preprocessing, [66](#)

recurrent, [130](#)

regularizer, [70](#)

reinforcement learning, [34](#)

training algorithm, [69](#)

unsupervised learning, [34](#)

weight, [63](#)

neuron, [62](#)

Noah, Trevor, [148](#)

node, [63](#)

noise vector, [100](#)

normal distribution, [99](#)

normalization, [40](#)

O

Offray de La Mettrie, Julien, [27](#)

one-hot encoding, [103](#)

one-versus-one, [56](#)

one-versus-rest, [56](#)

optimization, [11](#), [54](#), [72](#)

overfitting, [23](#), [70](#)

P

Pantoliano, Joe, [149](#)

paper clip catastrophe, [150](#)

Papert, Seymour, [30](#)

parameters, [8](#)

Perceptron, [29](#), [32](#), [78](#)

Petro, Joseph, [148](#)

Pitts, Walter, [28](#)

pooling layer, [87](#)

preprocessing, [66](#)

pretraining, [130](#)

primary visual cortex, [84](#)

prompt engineer, [111](#)

R

random forest, [33](#), [49](#)

bagging, [49–50](#)

ensembling, [49](#), [52](#)

feature selection, [49–50](#)

rectified linear unit (ReLU), [39](#)

leaky ReLU, [101](#)

recurrent neural network (RNN), [130](#)

regularizer, [70](#)

reinforcement learning, [34](#)

from human feedback (RLHF), [132](#)

Romilio, Anthony, [56](#)

Rosenblatt, Frank, [29](#), [115](#)

Rumelhart, David, [32](#)

Russell, Stuart, [150](#)

S

sample, [10](#)

schedule, [107](#)

Sedol, Lee, [35](#)

Sejnowski, Terrence, [140](#)

semantic segmentation, [93](#)

Shannon, Claude, [29](#)

Shazeer, Noam, [130](#)

Shelley, Mary, [27](#)

sigmoid, [64](#)

Solomonoff, Ray, [29](#)

“Sparks of Artificial General Intelligence,” [116](#)

Stable Diffusion, [110](#)

stochastic gradient descent, [76](#)

Stork, David, [30](#)

supervised learning, [8](#)

support vector machine (SVM), [32](#), [52](#)

kernel, [54](#), [85](#)

optimization, [11](#), [54](#), [72](#)

support vector, [54](#)

swarm intelligence, [59](#)

symbolic AI, [26](#)

T

Tattersall, Ian, [37](#)

Tegmark, Max, [150](#)

testing, [10](#)

test set, [10](#)

token, [130](#)

training, [8](#)

training set, [10](#)

transformer, [130](#)

attention, [131](#)

decoder, [130](#)

encoder, [130](#)

true negative, [57](#)

true positive, [57](#)

Turing, Alan, [6](#), [28](#)

Turing machine, [28](#)

Turing test, [28](#)

Tversky, Amos, [121](#)

U

Uhr, Leonard, [30](#)

unsupervised learning, [34](#)

V

van Gogh, Vincent, [83](#)

Vapnik, Vladimir, [32](#)

Vaswani, Ashish, [130](#)

vector, [9](#)

Vossler, Charles, [30](#)

W

weight, [63](#)

Williams, Ronald, [32](#)

Wooldridge, Michael, [26](#)

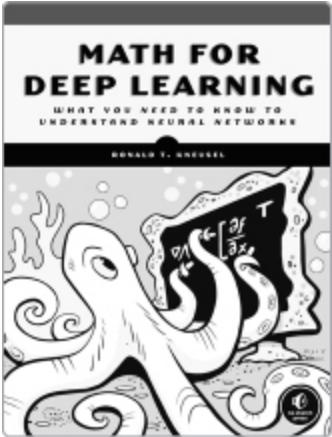
Wright, Letitia, [148](#)

The fonts used in *How AI Works* are New Baskerville, Futura, The Sans Mono Condensed, and Dogma. The book was typeset with LATEX 2 _{ϵ} package `nostarch` by Boris Veytsman with many additions by Alex Freed and other members of the *No Starch Press* team (*2023/04/08 v2.1 Typesetting books for No Starch Press*).

RESOURCES

Visit <https://nostarch.com/how-ai-works> for errata and more information.

More no-nonsense books from  **NO STARCH PRESS**



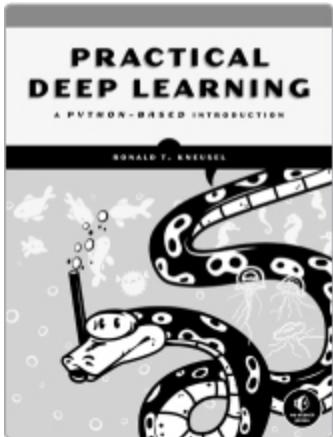
MATH FOR DEEP LEARNING

What You Need to Know to Understand Neural Networks

BY RONALD T. KNEUSEL

344 PP., \$49.99

ISBN 978-1-7185-0190-4



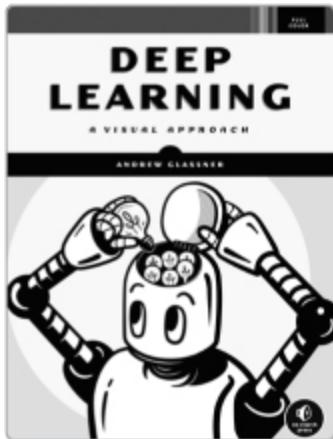
PRACTICAL DEEP LEARNING

A Python-Based Introduction

BY RONALD T. KNEUSEL

464 PP., \$59.95

ISBN 978-1-7185-0074-7



DEEP LEARNING

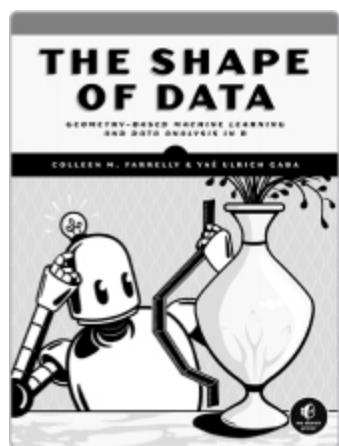
A Visual Approach

BY ANDREW GLASSNER

768 PP., \$99.99

ISBN 978-1-7185-0072-3

full color



THE SHAPE OF DATA

Geometry-Based Machine Learning and Data Analysis in R

BY COLLEEN M. FARRELLY AND YAÉ ULRICH GABA

264 PP., \$39.99

ISBN 978-1-7185-0308-3



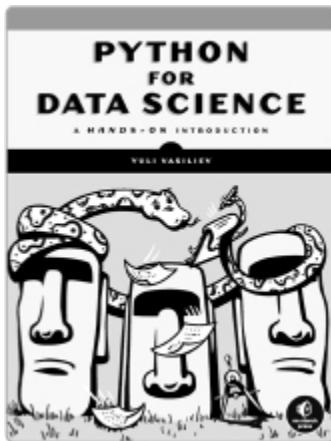
MODELING AND SIMULATION IN PYTHON

An Introduction for Scientists and Engineers

BY ALLEN B. DOWNEY

280 PP., \$39.99

ISBN 978-1-7185-0216-1



PYTHON FOR DATA SCIENCE

A Hands-On Introduction

BY YULI VASILIEV

240 PP., \$29.99

ISBN 978-1-7185-0220-8

PHONE:

800.420.7240 OR

415.863.9900

EMAIL:

SALES@NOSTARCH.COM

WEB:

WWW.NOSTARCH.COM



Never before has the world relied so heavily on the Internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people—more urgent than ever.

For over 30 years, EFF has fought for tech users through activism, in the courts, and by developing software to overcome obstacles to your privacy, security, and free expression. This dedication empowers all of us through

darkness. With your help we can navigate toward a brighter digital future.



EFF.ORG/NO-STARCH-

[PRESS](#)