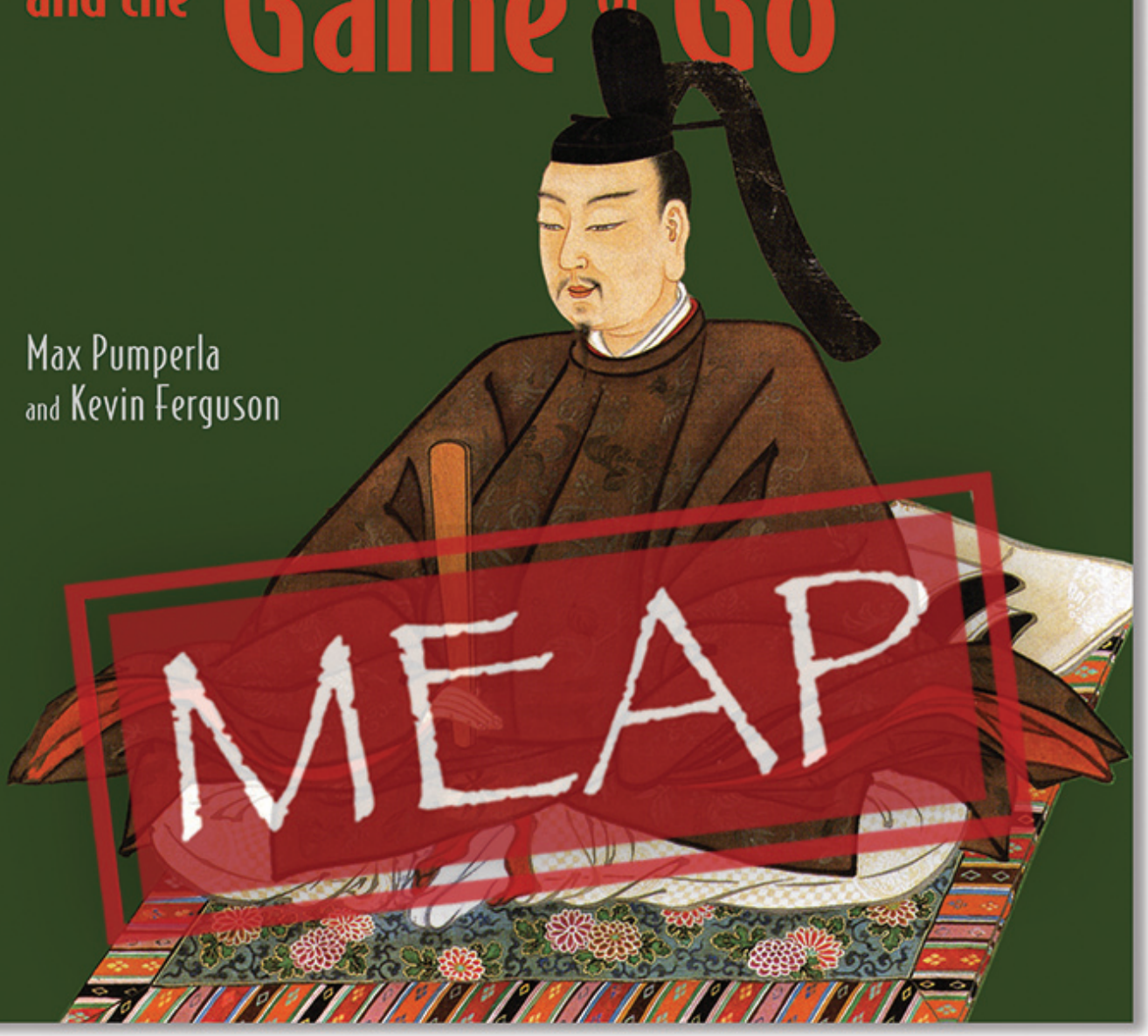
MANNING

Deep Learning and the Game of Go

Max Pumperla
and Kevin Ferguson



MEAP Edition
Manning Early Access Program
Learning and the Game of Go
Version 1

For more information on this and other Manning titles go to
www.manning.com

welcome

When AlphaGo hit the news in early 2016, we were extremely excited about this groundbreaking advancement in computer Go. At the time, it was largely conjectured that human-level artificial intelligence for the game of Go was at least 10 years in the future. We followed the games meticulously and didn't shy away from waking up early or staying up late to watch the broadcasted games live. Indeed, we had good company—millions of people around the globe were captivated by the games against Fan Hui, Lee Sedol, and later, Ke Jie and others.

Shortly after the emergence of AlphaGo, we picked up work on a little open source library we coined BetaGo (https://github.com/maxpumperla/betago) to see if we could implement some of the core mechanisms running AlphaGo ourselves. The idea of BetaGo was to bring AlphaGo to regular developers. While we were realistic enough to accept that we didn't have the resources (time, computing power, or intelligence) to compete with Deepmind's incredible achievement, it has been a lot of fun to create our own Go bot. Since then, we've had the privilege to speak about computer Go at quite a few occasions.

We strongly believe that the principles underpinning AlphaGo can be taught to a general software engineering audience in a practical manner. Enjoyment and understanding of Go comes from playing it and experimenting with it. It can be argued that the same holds true for machine learning, or any other discipline for that matter.

In this book, we hope to use the game of Go as a gateway to the exciting world of deep learning. We start with the classical AI principles for board games. Right from the start, you'll have a working Go AI that you can play against—although it will be very weak at first. Then we'll introduce some techniques from the world of deep learning and reinforcement learning. As you learn each technique, you can incorporate it into your Go AI and watch it improve.

If you share some of our enthusiasm for either Go or machine learning, hopefully both, at the end of this book, we've done our job. If, on top of that, you know how to build and ship a Go bot and run your own experiments, many other interesting artificial intelligence applications will be accessible to you as well. Enjoy the ride!

—Max Pumperla and Kevin Ferguson

brief contents

1 Toward deep learning: a machine learning introduction

This chapter covers:

- What is machine learning and how does it differ from traditional programming?
- What kinds of problems can and cannot be solved with machine learning?
- How does machine learning fit into artificial intelligence in general?
- What is the structure of a machine learning system?
- What disciplines of machine learning are there and what are examples of them?
- What are the guiding principles of deep learning?

As long as there have been computers, programmers have been interested in *artificial intelligence* (or "AI"): implementing human-like behavior on a computer. Games have long been been a popular subject for AI researchers. During the personal computer era, AIs have overtaken humans at checkers, backgammon, chess, and almost all classic board games. But the ancient strategy game Go remained stubbornly out of reach for computers for decades. Then in 2016, Google DeepMind's AlphaGo AI challenged 14-time world champion Lee Sedol and won four out of five games. The next revision of AlphaGo was completely out of reach for human players: it won 60 straight games, taking down just about every notable Go player in the process.

AlphaGo's breakthrough was enhancing classical AI algorithms with machine learning. More specifically, it used modern techniques known as *deep learning* — algorithms that can organize raw data into useful layers of abstraction. These techniques are not limited to games at all. You will also find deep learning in applications for identifying images, understanding speech, translating natural languages, and guiding robots. Mastering the foundations of deep learning will equip you to understand how all these applications work.

Why write a whole book about computer Go? You might suspect the authors are die-hard Go nuts — OK, guilty as charged. But the real reason to study Go, as opposed to chess or backgammon, is that a really strong Go AI requires deep learning. A top-tier chess engine such as Stockfish is full of chess-specific logic; you need a certain amount of knowledge about the game to write something like that. With deep learning, we can teach a computer to imitate strong Go players, even if we don't understand what they're doing. And that's a powerful technique that opens up all kinds of applications, both in games and in the real world.

Chess and checkers AIs are designed around reading out the game farther and more accurately than human players can. There are two problems with applying this technique to Go. First, we can't read very far ahead, because there are too many moves to consider. Second, if we could read ahead, we don't know how to evaluate if the result is good. It turns out that deep learning is the key to unlocking both these problems.

This book provides a practical introduction to deep learning by covering the techniques that powered AlphaGo. We don't need to study the game of Go in much detail to do this; instead, we'll look at the general principles of how a machine can learn. This chapter introduces the idea of machine learning and what kinds of problems it can (and cannot) solve. We work through some examples that illustrate the major branches of machine learning, and show how deep learning has brought machine learning into new domains.

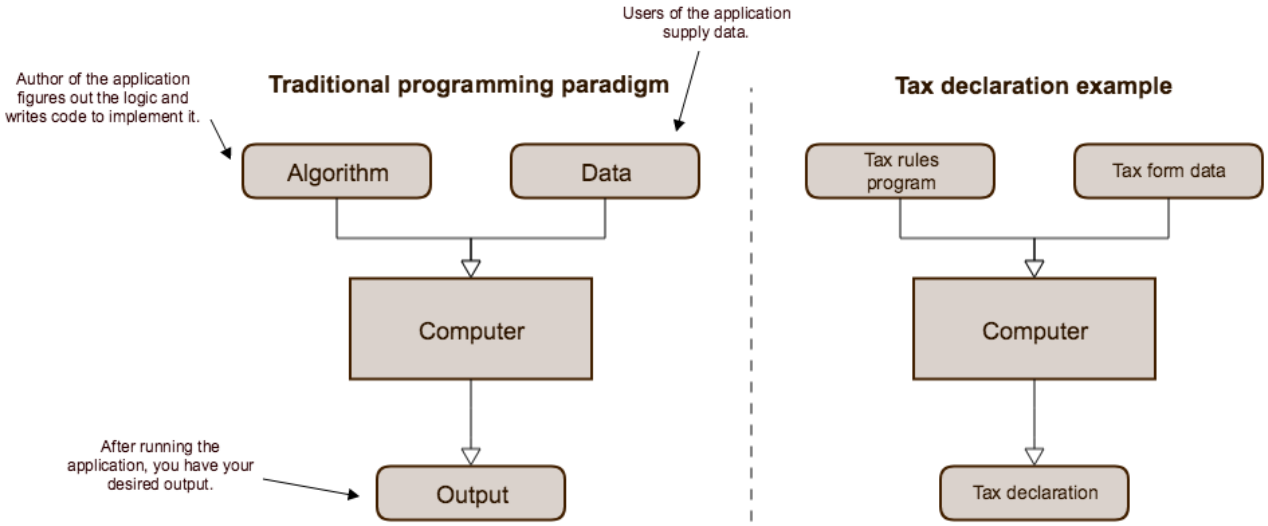
1.1 What is machine learning?

Consider the task of identifying a photo of a friend. This is effortless for most people, even if the photo is badly lit, your friend got a haircut, or is wearing a new shirt. But suppose you wanted to program a computer to do the same thing. Where would you even begin? This is the kind of problem that machine learning can solve.

Traditionally, computer programming is about applying clear rules to structured data. A human developer programs a computer to execute a set of instructions on data and out comes the desired result. Think of a tax form: every box has a well-defined meaning, and there are detailed rules about how to make various calculations from them. Depending on where you live, these rules may be extremely complicated. It's easy for people to make a mistake here, but this is exactly the kind of task that computer programs excel at.

In contrast to the traditional programming paradigm, *machine learning* is a family of techniques for inferring a program or algorithm from example data, rather than implementing it directly. So, with machine learning we still feed our computer data, but instead of imposing instructions and expecting output, *we provide the expected output and let the machine find an algorithm by itself*.

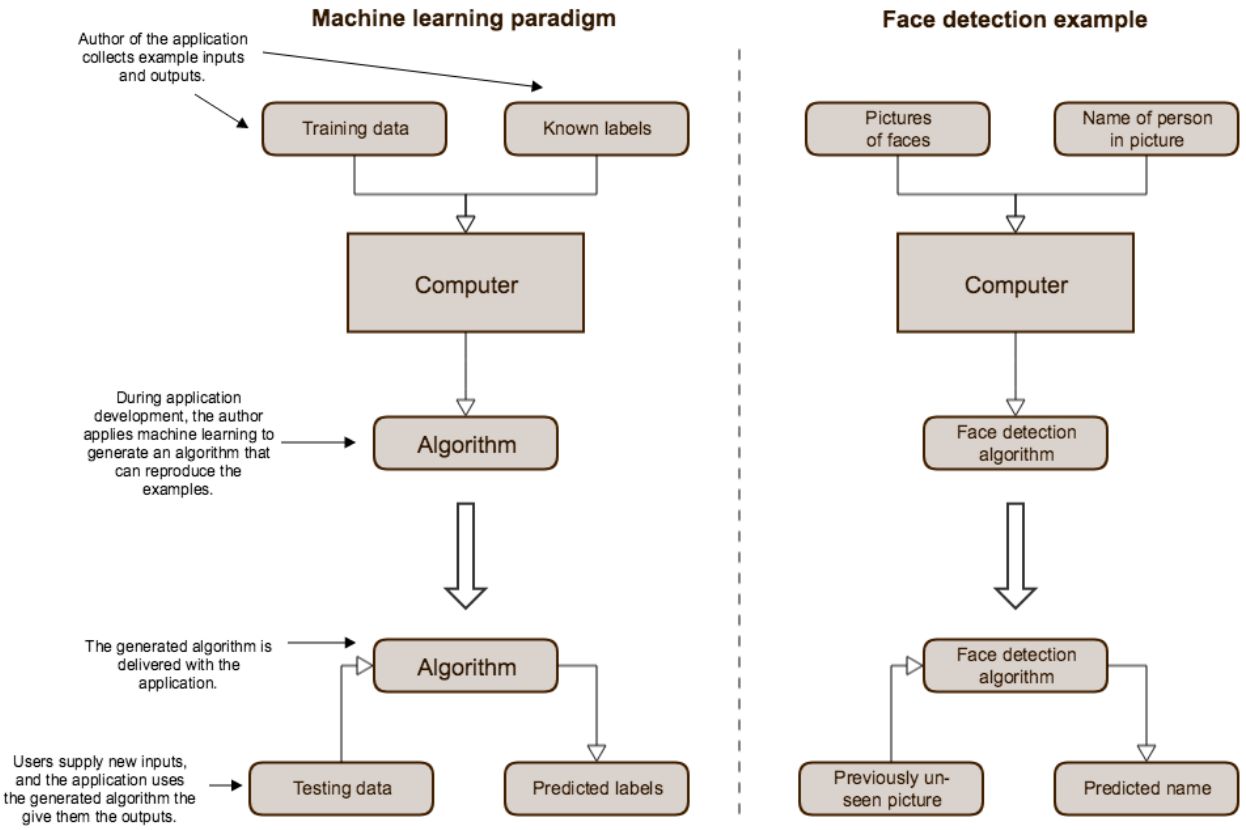
Figure 1.1. An illustration of the standard programming paradigm that most software developrs are familiar with. The developer identifies the algorithm and implements the code; the users supply the data.



To build a computer program that can identify who's in a photo, we can apply an algorithm that analyzes a large collection of images of your friend and generates a function that matches them. If we do this correctly, the generated function will also match new photos that we've never seen before. Of course, it will have no knowledge of its purpose; all it can do is identify things that are similar to the original images we fed it.

In this situation, we call the images we provide the machine *training data* and the names of the person on the picture *labels*. Once we have *trained* an algorithm for our purpose, we can use it to *predict* labels on new data to test it. Figure 1.2 displays this example alongside a schema of the machine learning paradigm.

Figure 1.2. An illustration of the machine learning paradigm. During development, we generate an algorithm from a data set, and then incorporate that into our final application.



Machine learning comes in when rules aren't clear; it can solve problems of the "I'll know it when I see it" variety. Instead of programming the function directly, we provide data that indicates what the function should do, and then methodically generate a function that matches our data.

In practice, you usually combine machine learning with traditional programming to build a useful application. For our face detection app, we have to instruct the computer on how to find, load, and transform the example images before we can apply a machine learning algorithm. Beyond that, we might use hand-rolled heuristics to separate headshots from photos of sunsets and latte art; then we can apply machine learning to put names to faces. Often a mixture of traditional programming techniques and advanced machine learning algorithms will be superior to either one alone.

1.1.1 How does machine learning relate to AI?

Artificial intelligence, in the broadest sense, refers to any technique for making

computers imitate human behavior. AI includes a huge range of techniques, including:

- Logic production systems, which apply formal logic to evaluate statements
- Expert systems, in which programmers try to directly encode human knowledge into software
- Fuzzy logic, which defines algorithms to help computers process imprecise statements

These sorts of rules-based techniques are sometimes called *classical AI* or *GOFAI* ("good old-fashioned AI").

Machine learning is just one of many fields in AI, but today it is arguably the most successful one. In particular, the subfield of deep learning is behind some of the most exciting breakthroughs in AI, including tasks that eluded researchers for decades. In classical AI, researchers would study human behavior and try to encode rules that match it. Machine learning and deep learning flip the problem on its head: now we collect examples of human behavior, and apply generalized algorithms to extract the rules.

Deep learning is so ubiquitous that some people in the community use "AI" and "deep learning" interchangeably. For clarity, we will use "AI" to refer to the general problem of imitating human behavior with computers, and "machine learning" or "deep learning" to refer to those specific algorithms.

1.1.2 What you can and cannot do with machine learning

Machine learning is a specialized technique. You wouldn't use machine learning to update database records or render a user interface. Traditional programming should be preferred in the following situations.

- *Traditional algorithms solve the problem directly*. If you can directly write code to solve a problem, it will be easier to understand, maintain, test, and debug.
- *You expect perfect accuracy*. All complex software contains bugs, of course. But in traditional software engineering, we expect to methodically identify and fix bugs. That's not always possible with machine learning. You can improve machine learning systems, of course; but focusing too much on a specific error often makes the overall system worse.
- *Simple heuristics work well*. If you can implement a rule that's good enough with just a few lines of code, do so and be happy. A simple heuristic, implemented clearly, will be easy to understand and maintain. Functions that are implemented with machine learning are opaque and require a separate training process to update. (On the other hand, if you are maintaining a complicated sequence of heuristics, that's a good candidate to replace with machine learning.)

Often there is a fine line between problems that are feasible to solve with traditional programming and problems that are virtually impossible to solve, even

with machine learning. Detecting faces in images versus tagging faces with names is just one example we have seen. Determining what language a text is written in versus translating that text into a given language is another such example.

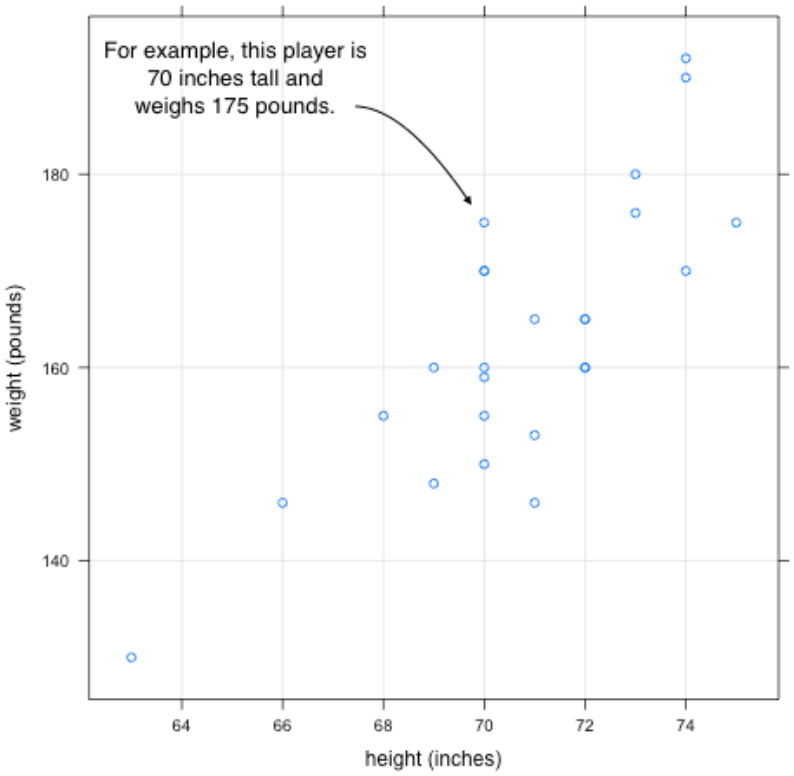
There are also situations in which we often resort to traditional programming, where machine learning might actually help, for instance when the complexity of the problem is extremely high. When confronted with highly complex, information-dense scenarios, humans tend to settle for rules of thumbs and narratives: think macro economics, stock market predictions, or politics. Process managers and so-called experts can often vastly benefit from enhancing their intuition with insights gained from machine learning. Often, there is more structure in real-world data than anticipated and we are just beginning to harvest the benefits of automation and augmentation in many of these areas.

1.2 Machine learning by example

The goal of machine learning is to construct a function that would be hard to implement directly. We do this by selecting a *model*, a large family of generic functions. Then we need a procedure for selecting a function from that family that matches our goal; this process is called *training* or *fitting* the model. We'll work through a very simple example.

Let's say we collected the height and weight of some people and plotted them on a graph. Figure 1.3 shows some data points that were pulled from the roster of a professional soccer team:

Figure 1.3. A simple example data set. Each point on the graph represents a soccer player's height and weight. Our goal is to fit a model to these points.



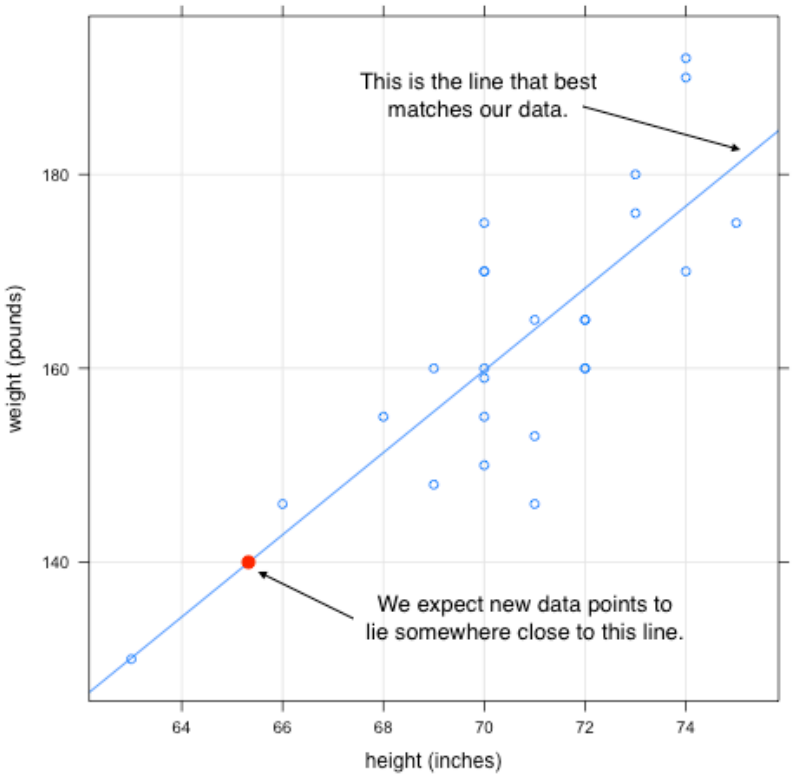
Suppose we want to describe these points with a mathematical function. First, notice the points more or less make a straight line going up and to the right. If you think back to high school algebra, you may recall that functions of the form f(x) = ax + b describe straight lines. So you might suspect that we could find values of a and b so that ax + b matches our data points fairly closely. The values of a and b are the *parameters* or *weights* that we need to figure out. This is our model. We can write some Python code that can generate any function in this family:

```
class GenericLinearFunction(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def evaluate(self, x):
        return self.a * x + self.b
```

How would we find out the right values of a and b? There are rigorous algorithms to do this, but for a quick and dirty solution, you could just draw a line through our

graph with a ruler and try to work out its formula.

Figure 1.4. First we noted that our data set roughly follows a linear trend. Then we found the formula for a specific line that fits the data.



If you eyeball a couple points that line passes through, you can calculate a formula for the line; you'll get something like f(x) = 4.2x - 137. Now we have a specific function that matches our data. If we measure the height of a new person, we could then use our formula to estimate their weight. It won't be exactly right, but it may be close enough to be useful. We can turn our `GenericLinearFunction` into a specific function:

```
height_to_weight = GenericLinearFunction(a=4.2, b=-137)
height_of_new_person = 73
estimated_weight = height_to_weight.evaluate(height_of_new_person)
```

This should be a pretty good estimate, so long as our new person is also a professional soccer player. All the people in our data set were adult men, in a fairly narrow age range, who train for the same sport every day. If we try to apply our function to female soccer players, or Olympic weightlifters, or babies, we'll get wildly inaccurate results. Our function is only as good as our training data.

This is the basic process of machine learning. Here, our model is the family of all functions that look like f(x) = ax + b. And in fact, even something that simple is a useful model that statisticians use all the time. As we tackle more complex problems, we will use more sophisticated models and more advanced training techniques. But the core idea in the same: first describe a large family of possible functions, then identify the best function from that family.

1.2.1 Using machine learning in software applications

In the previous section, we looked at a purely mathematical model. How can we apply machine learning to a real software application?

Suppose you're working on a photo-sharing app, where users have uploaded millions of pictures with tags. You'd like to add a feature that suggests tags for a new photo. This feature is a perfect candidate for machine learning.

First, we have to be specific about what function we're trying to learn. If we had a function like

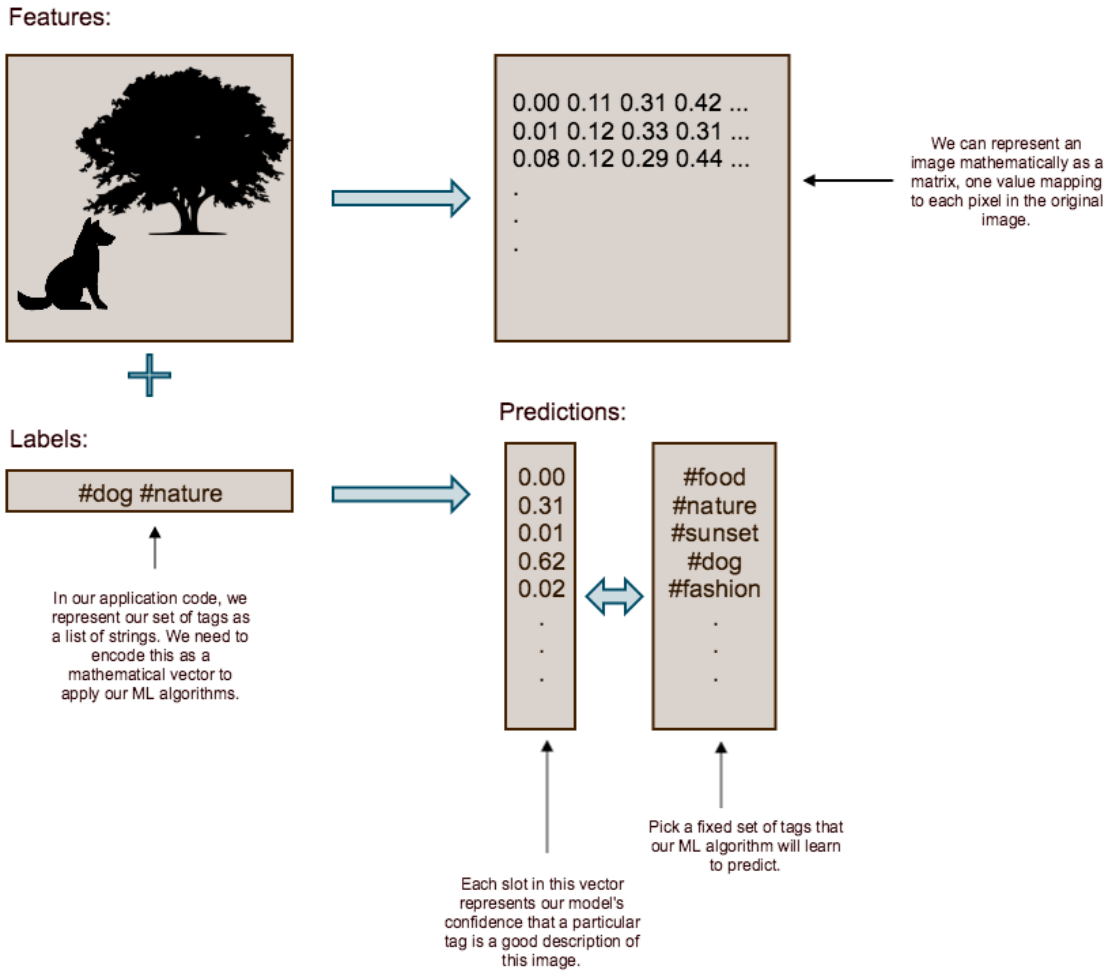
```
def suggest_tags(image_data):
    """Recommend tags for an image.

    Input: image_data is a photo in bitmap format

    Returns: a ranked list of suggested tags
    """
```

then the rest of the work is relatively straightforward. But it's not at all obvious how to start implementing a function like `suggest_tags`. That's where machine learning comes in.

If this were an ordinary Python function, we'd expect it to take some kind of `Image` object as input and perhaps return a list of strings as output. Machine learning algorithms are not so flexible about their inputs and outputs; they generally work on vectors and matrices. So as a first step, we need to represent our input and output mathematically.

If we resize the input photo to some fixed size, say 128x128 pixels, then we can encode it as a matrix with 128 rows and 128 columns: one float value per pixel. What about the output? One option is to restrict the set of tags we will identify; we could select perhaps the 1000 most popular tags on the app. The output could then be a vector of size 1000, where each element of the vector corresponds to a particular tag. If we allow the output values to vary anywhere between 0 and 1, we can generate ranked lists of suggested tags.

Figure 1.5. Machine learning algorithms operate on mathematical structures, like vectors and matrices. Our photo tags are stored in a standard computer data structures: a list of strings. This is one possible scheme for encoding that list as a mathematical vector.



This data preprocessing step we just carried out is an integral part of every machine learning system. Usually, we load the data in raw format and carry out some preprocessing steps to create *features*, i.e. input data that can be fed into a machine learning algorithm.

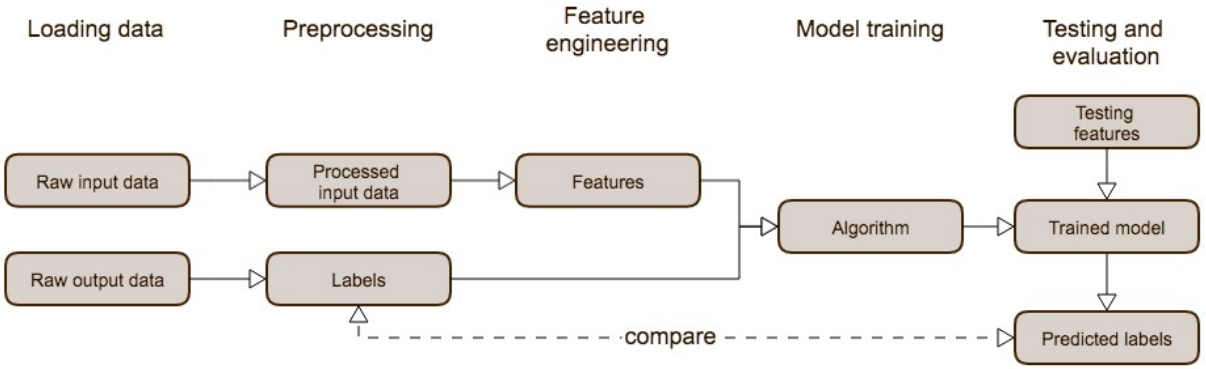
1.2.2 Supervised learning

Next, we need an algorithm for training our model. In this case, we have millions of correct examples already — all the photos that users have already uploaded and manually tagged in our app. We can learn a function that attempts to match these

examples as closely as possible, and we hope that it will generalize to new photos in a sensible way. This technique is known as *supervised learning*, so-called because the *labels* of human-curated examples provide guidance for the training process.

When training is complete, we can deliver the final learned function with our application. Every time a user uploads a new photo, we pass it into the trained model function and get a vector back. We can match each value in the vector back to the tag it represents; then we can select the tags with the largest values and show them to the user. Schematically, the procedure we just outlined, can be represented as follows in figure 1.6.

Figure 1.6. A machine learning pipeline for supervised learning



How do we test our trained model? The standard practice is to set aside some of our original labeled data for that purpose. Before starting training, we can set aside a chunk of our data, say 10%, as a *validation set*. The validation set is *not* included as part of the training data in any way. Then we can apply our trained model to the images in the validation set and compare the suggested tags to the known good tags. This lets us compute the accuracy of our model. If we want to experiment with different models, we have a consistent metric for measuring which is better.

In game AI, we can extract labeled training data from records of human games. And online gaming is a huge boon for machine learning: when people play a game online, the game server may save a computer-readable record. Some examples of how to apply supervised learning to games are:

- Given a collection of complete records of chess games, represent the game state in vector or matrix form and learn to predict the next move from data.
- Given a board position, learn to predict the likelihood of winning for that state.

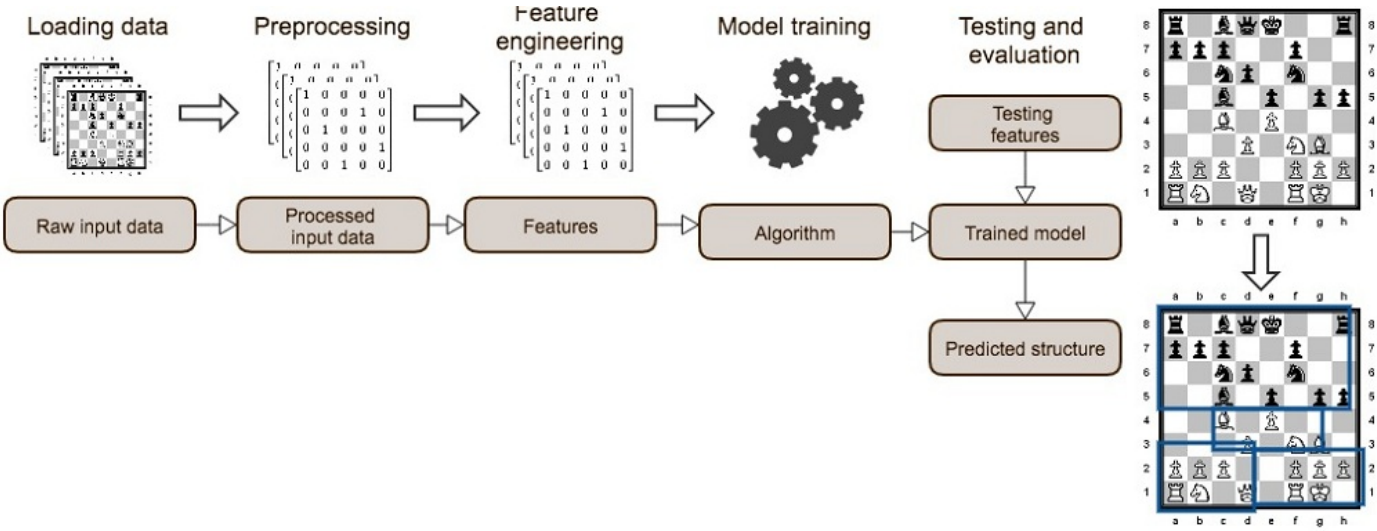
1.2.3 Unsupervised learning

In contrast to supervised learning, the subfield of machine learning called *unsupervised learning* does not come with any labels to guide the learning process. In unsupervised learning, the algorithm has to learn to find patterns in the input data on its own. The only difference to figure 1.6 is that we are missing the labels, so we can't evaluate our predictions the way we did before. All other components stay the same.

An example of this would be *outlier detection* — identifying data points that don't fit in with the general trend of the data set. In the soccer player data set, outliers would indicate players who don't match the typical physique of their teammates. For instance, we could come up with an algorithm that measures the distance of a height-width pair to the line we eyeballed. If a data point exceeds a certain distance to the average line, we declare it an outlier.

In board game AI, a natural question to ask is which pieces on the board belong together or form a group. In the next chapter we will see what this means for the game of Go in more detail. Finding groups of pieces that have a relationship is sometimes called *clustering* or *chunking*. In figure 1.7 we see an example of what this could look like for chess.

Figure 1.7. An unsupervised machine learning pipeline for finding clusters or chunks of chess pieces.



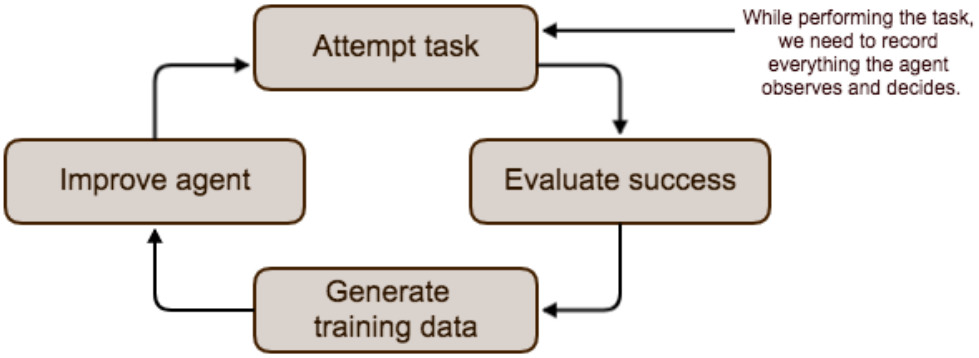
1.2.4 Reinforcement learning

Supervised learning is powerful, but finding quality training data can be a major obstacle.

Suppose we're building a house-cleaning robot. The robot has various sensors that can detect when it's near obstacles, and motors that let it scoot around the floor and steer left or right. We need a control system: a function that can analyze the sensor input and decide how it should move. But supervised learning is impossible here. There are no examples to use as training data — our robot doesn't even exist yet.

Instead, we can apply *reinforcement learning*, a sort of trial and error approach. We start with an inefficient or inaccurate control system, and then we let the robot attempt its task. During the task, we'll record all the inputs our control system sees and the decisions it makes. When it's done, we need a way to evaluate how well it did, perhaps by calculating the fraction of the floor it vacuumed and how far it drained its battery. That whole experience gives us a small chunk of training data, and we can use it to improve the control system. By repeating the whole process over and over, we can gradually home in on an efficient control function.

Figure 1.8. In reinforcement learning agents learn to interact with their environment by trial and error. We repeatedly have our agent attempt its task to get a supervised signal to learn from. Every cycle we can make an incremental improvement.



1.3 Deep learning

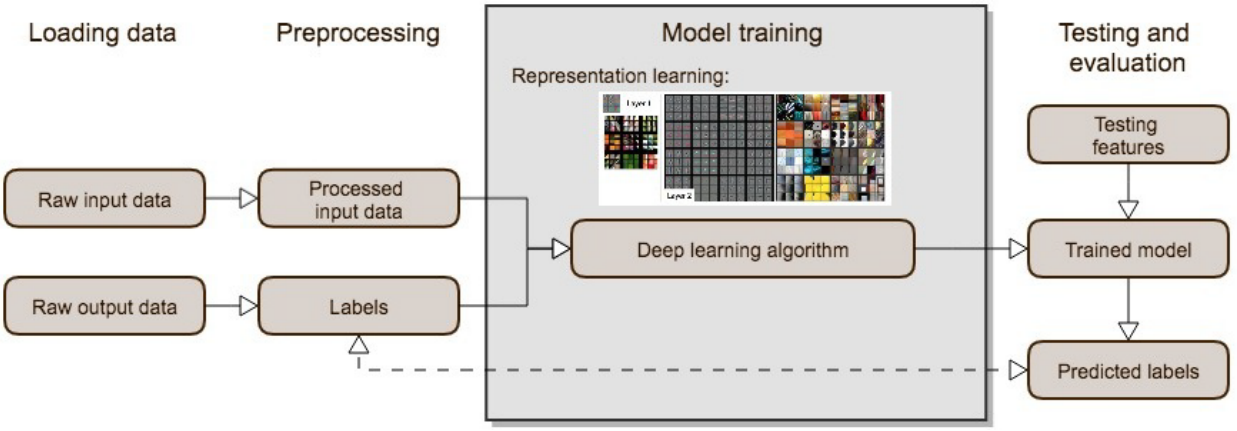
This book is made up of sentences. The sentences are made of words; the words are made of letters; the letters are made up of lines and curves; and ultimately those lines and curves are made up of tiny dots of ink. When teaching a child to read, we start with the smallest parts and work our way up: first letters, then words, then sentences, and finally complete books. (Normally children learn to recognize lines and curves on their own.) This kind of hierarchy is the natural way for people to learn complex concepts. At each level, we ignore some detail and the concepts become more abstract.

Deep learning applies the same idea to machine learning. Deep learning is a subfield of machine learning that uses a specific family of models: sequences of simple functions chained together. These chains of functions are known as *neural*

networks because they were loosely inspired by the structure of natural brains. The core idea of deep learning is that these sequences of functions can analyze a complex concept as hierarchy of simpler ones. The first layer of a deep model can learn to take raw data and organize it in basic ways — like grouping dots into lines. Each successive layer organizes the previous layer into more advanced and abstract concepts.

The amazing thing about deep learning is that you do not need to know what the intermediate concepts are in advance. If you select a model with enough layers, and provide enough training data, the training process will gradually organize the raw data into increasingly high-level concepts. But how does the training algorithm know what concepts to use? It doesn't really; it just organizes the input in any way that helps it match the training examples better. So there is no guarantee this representation matches the way humans would think about the data.

Figure 1.9. Deep learning and representation learning.



Of course, all this power comes with a cost. Deep models have huge numbers of weights to learn. Recall the simple ax + b model we used for our height and weight data set; that model had just two weights to learn. A deep model suitable for our image tagging app could have a million weights. As a result, deep learning demands larger data sets, more computing power, and a more hands-on approach to training. Both techniques have their place. Deep learning is a good choice:

- When your data is in an unstructured form. Images, audio, and written language are good candidates for deep learning. It's possible to apply simple models to that kind of data, but that generally requires sophisticated preprocessing.
- When you have large amounts of data available, or have a plan for acquiring more. In general, the more complex your model is, the more data you need to train it.
- When you have plenty of computing power, or plenty of time. Deep models involve more calculation for both training and evaluation.

Prefer traditional models with fewer parameters:

- When you have structured data. If your inputs look more like database records, you can often apply simple models directly.
- When you want a descriptive model. With simple models, you can look at the final learned function and examine how an individual input affects the output. This can give you insight about how the real-world system you're studying works. In deep models, the connection between a specific piece of the input and the final output is long and winding; it's difficult to interpret the model.

Since deep learning refers to the type of model we use, we can apply deep learning to any of the major machine learning branches. For example, you can do supervised learning with a deep model or a simple model, depending on the type of training data you have.

1.4 What you will learn in this book

This book provides you with a practical introduction to deep learning and reinforcement learning. To get the most of out this book, you should be comfortable reading and writing Python code, and have some familiarity with linear algebra and calculus. In this book, we teach:

- How to design, train, and test neural networks using the Keras deep learning library.
- How to set up supervised deep learning problems.
- How to set up reinforcement learning problems.
- How to integrate deep learning with a useful application.

Throughout the book, we use a concrete and fun example: building an AI that plays Go. Our Go bot combines deep learning with standard computer algorithms. We'll use straightforward Python to enforce the rules of the game, track the game state, and look ahead through possible game sequences. Deep learning will help the bot identify which moves are worth examining and evaluate who is ahead during a game. At each stage, you can play against your bot and watch it improve as we apply more sophisticated techniques.

If you are interested in Go specifically, you can use the bot we will build in the book as a starting point for experimenting with your own ideas. You can also adapt the same techniques to other games. You will also be able to add features powered by deep learning to other applications outside of games.

1.5 Summary

- Machine learning is a family of techniques for generating functions from data instead of writing them directly. You can use machine learning to solve problems that are too ambiguous to solve directly.
- Machine learning generally involves first choosing a *model* — a generic family of

mathematical functions. Next you *train* the model — apply an algorithm to find the best function in that family. Much of the art of machine learning lies in selecting the right model and transforming your particular data set to work with it.

- Three of the major areas of machine learning are supervised learning, unsupervised learning, and reinforcement learning.
- Supervised learning involves learning a function from examples we already know to be correct. When you have examples of human behavior or knowledge available, you can apply supervised learning to imitate them on a computer.
- Unsupervised learning involves extracting structure from data without knowing what the structure is in advance. A common application is splitting a data set into logical groups.
- Reinforcement learning involves learning a function through trial and error. If you can write code to evaluate how well a program achieves a goal, you can apply reinforcement learning to incrementally improve a program over many trials.
- Deep learning is machine learning with a particular type of model that performs well on unstructured inputs, like images or written text. It's one of the most exciting fields in computer science today; it is constantly expanding our ideas about what computers can do.