CHAPTER 7
Neural Networks

A regression and classification technique that has enjoyed a renaissance over the past 10 years is neural networks. In the simplest definition, a *neural network* is a multilayered regression containing layers of weights, biases, and nonlinear functions that reside between input variables and output variables. *Deep learning* is a popular variant of neural networks that utilizes multiple "hidden" (or middle) layers of nodes containing weights and biases. Each node resembles a linear function before being passed to a nonlinear function (called an activation function). Just like linear regression, which we learned about in Chapter 5, optimization techniques like stochastic gradient descent are used to find the optimal weight and bias values to minimize the residuals.

Neural networks offer exciting solutions to problems previously difficult for computers to solve. From identifying objects in images to processing words in audio, neural networks have created tools that affect our everyday lives. This includes virtual assistants and search engines, as well as photo tools in our iPhones.

Given the media hoopla and bold claims dominating news headlines about neural networks, it may be surprising that they have been around since the 1950s. The reason for their sudden popularity after 2010 is due to the growing availability of data and computing power. The ImageNet challenge between 2011 and 2015 was probably the largest driver of the renaissance, boosting performance on classifying one thousand categories on 1.4 million images to an accuracy of 96.4%.

However, like any machine learning technique it only works on narrowly defined problems. Even projects to create "self-driving" cars do not use end-to-end deep learning, and primarily use hand-coded rule systems with convoluted neural networks acting as a "label maker" for identifying objects on the road. We will discuss this later in this chapter to understand where neural networks are actually used. But

first we will build a simple neural network in NumPy, and then use scikit-learn as a library implementation.

When to Use Neural Networks and Deep Learning

Neural networks and deep learning can be used for classification and regression, so how do they size up to linear regression, logistic regression, and other types of machine learning? You might have heard the expression "when all you have is a hammer, everything starts to look like a nail." There are advantages and disadvantages that are situational for each type of algorithm. Linear regression and logistic regression, as well as gradient boosted trees (which we did not cover in this book), do a pretty fantastic job making predictions on structured data. Think of structured data as data that is easily represented as a table, with rows and columns. But perceptual problems like image classification are much less structured, as we are trying to find fuzzy correlations between groups of pixels to identify shapes and patterns, not rows of data in a table. Trying to predict the next four or five words in a sentence being typed, or deciphering the words being said in an audio clip, are also perceptual problems and examples of neural networks being used for natural language processing.

In this chapter, we will primarily focus on simple neural networks with only one hidden layer.

Variants of Neural Networks

Variants of neural networks include convolutional neural networks, which are often used for image recognition. Long short-term memory (LSTM) is used for predicting time series, or forecasting. Recurrent neural networks are often used for text-to-speech applications.

Is Using a Neural Network Overkill?

Using neural networks for the upcoming example is probably overkill, as a logistic regression would probably be more practical. Even a formulaic approach can be used. However, I have always been a fan of understanding complex techniques by applying them to simple problems. You learn about the strengths and limitations of the technique rather than be distracted by large datasets. So with that in mind, try not to use neural networks where simpler models will be more practical. We will break this rule in this chapter for the sake of understanding the technique.

A Simple Neural Network

Here is a simple example to get a feel for neural networks. I want to predict whether a font should be light (1) or dark (0) for a given color background. Here are a few examples of different background colors in Figure 7-1. The top row looks best with light font, and the bottom row looks best with dark font.

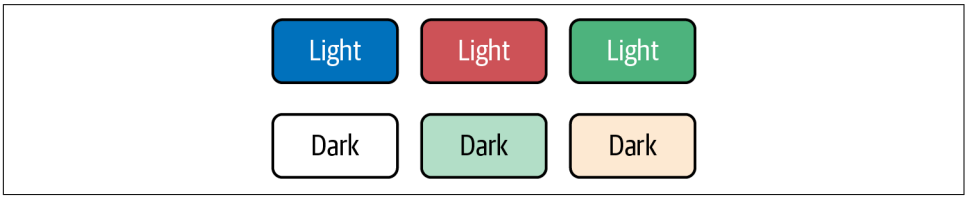

Figure 7-1. Light background colors look best with dark font, and dark background colors look best with light font

In computer science one way to represent a color is with RGB values, or the red, green, and blue values. Each of these values is between 0 and 255 and expresses how these three colors are mixed to create the desired color. For example, if we express the RGB as (red, green, blue), then dark orange would have an RGB of (255,140,0) and pink would be (255,192,203). Black would be (0,0,0) and white would be (255,255,255).

From a machine learning and regression perspective, we have three numeric input variables `red`, `green`, and `blue` to capture a given background color. We need to fit a function to these input variables and output whether a light (1) or dark (0) font should be used for that background color.

Representing Colors Through RGB

There are hundreds of color picker palettes online to experiment with RGB values. W3 Schools has one here.

Note this example is not far from how neural networks work recognizing images, as each pixel is often modeled as three numeric RGB values. In this case, we are just focusing on one "pixel" as a background color.

Let's start high level and put all the implementation details aside. We are going to approach this topic like an onion, starting with a higher understanding and peeling away slowly into the details. For now, this is why we simply label as "mystery math" a process that takes inputs and produces outputs. We have three numeric input variables R, G, and B, which are processed by this mystery math. Then it outputs a prediction between 0 and 1 as shown in Figure 7-2.

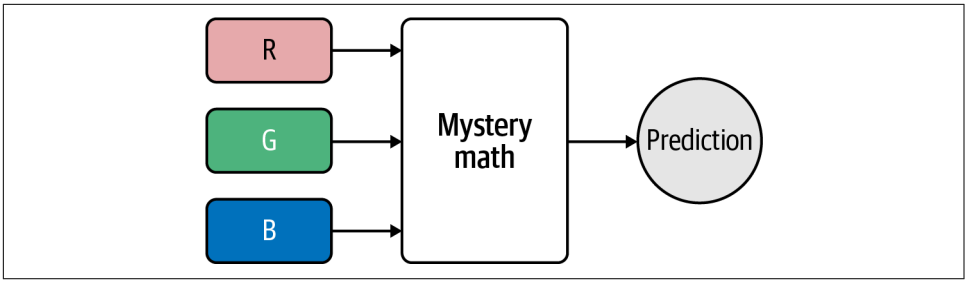

Figure 7-2. We have three numeric RGB values used to make a prediction for a light or dark font

This prediction output expresses a probability. Outputting probabilities is the most common model for classification with neural networks. Once we replace RGB with their numerical values, we see that less than 0.5 will suggest a dark font whereas greater than 0.5 will suggest a light font as demonstrated in Figure 7-3.

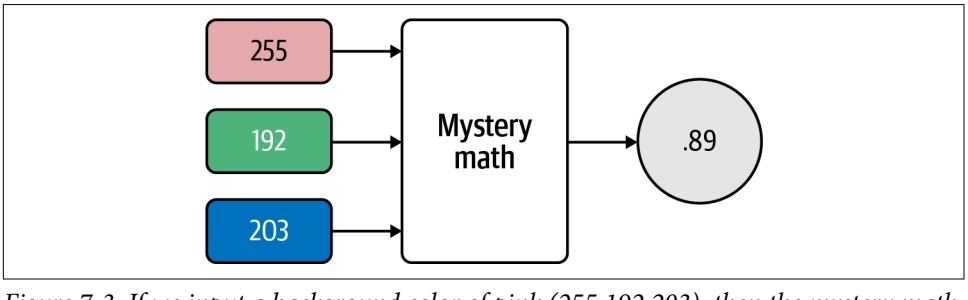

Figure 7-3. If we input a background color of pink (255,192,203), then the mystery math recommends a light font because the output probability 0.89 is greater than 0.5

So what is going on inside that mystery math black box? Let's take a look in Figure 7-4.

We are missing another piece of this neural network, the activation functions, but we will get to that shortly. Let's first understand what's going on here. The first layer on the left is simply an input of the three variables, which in this case are the red, green, and blue values. In the hidden (middle) layer, notice that we produce three *nodes*, or functions of weights and biases, between the inputs and outputs. Each node essentially is a linear function with slopes W_i and intercepts B_i being multiplied and summed with input variables X_i. There is a weight W_i between each input node and hidden node, and another set of weights between each hidden node and output node. Each hidden and output node gets an additional bias B_i added.

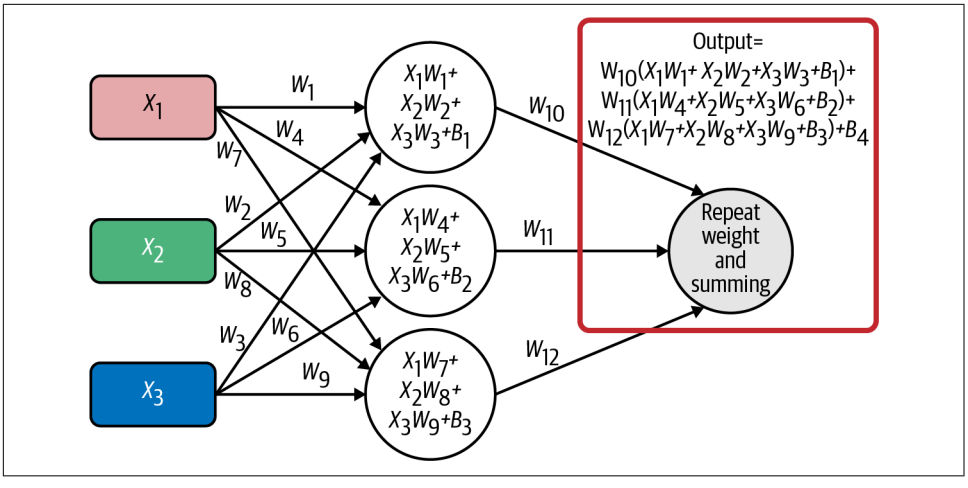

Figure 7-4. The hidden layer of the neural network applies weight and bias values to each input variable, and the output layer applies another set of weights and biases to that output

Notice the output node repeats the same operation, taking the resulting weighted and summed outputs from the hidden layer and making them inputs into the final layer, where another set of weights and biases will be applied.

In a nutshell, this is a regression just like linear or logistic regression, but with many more parameters to solve for. The weight and bias values are analogous to the m and b, or β_1 and β_0, parameters in a linear regression. We do use stochastic gradient descent and minimize loss just like linear regression, but we need an additional tool called backpropagation to untangle the weight W_i and bias B_i values and calculate their partial derivatives using the chain rule. We will get to that later in this chapter, but for now let's assume we have the weight and bias values optimized. We need to talk about activation functions first.

Activation Functions

Let's bring in the activation functions next. An *activation function* is a nonlinear function that transforms or compresses the weighted and summed values in a node, helping the neural network separate the data effectively so it can be classified. Let's take a look at Figure 7-5. If you do not have the activation functions, your hidden layers will not be productive and will perform no better than a linear regression.

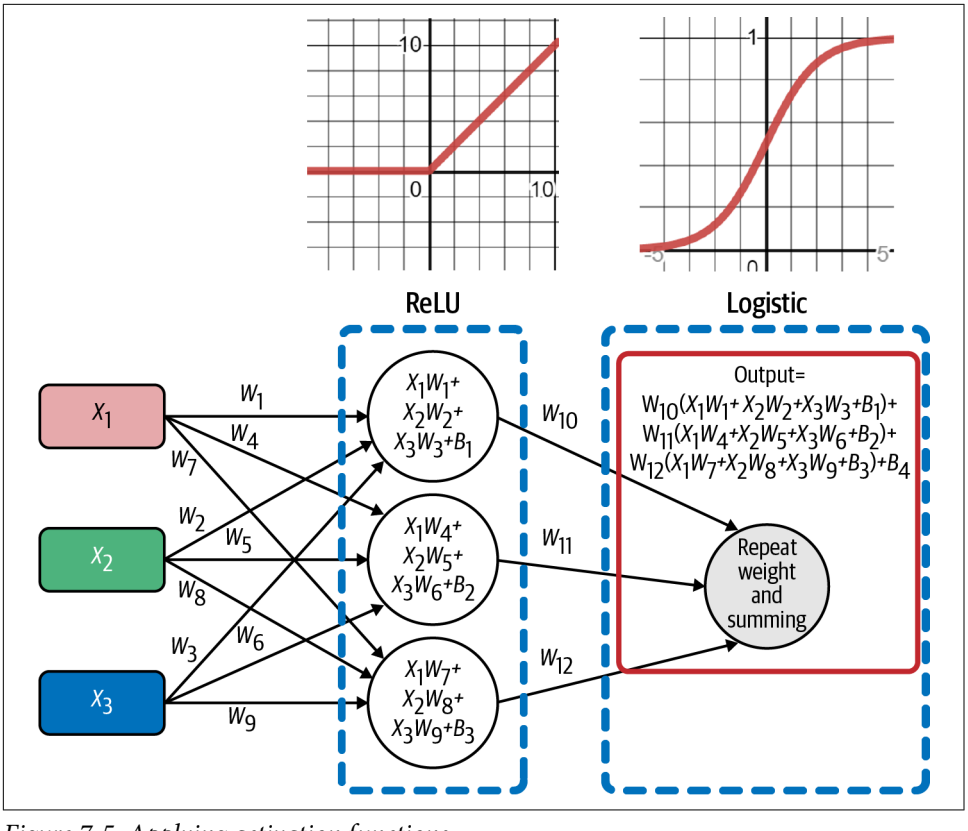

Figure 7-5. Applying activation functions

The *ReLU activation function* will zero out any negative outputs from the hidden nodes. If the weights, biases, and inputs multiply and sum to a negative number, it will be converted to 0. Otherwise the output is left alone. Here is the graph for ReLU (Figure 7-6) using SymPy (Example 7-1).

Example 7-1. Plotting the ReLU function

```
from sympy import *

# plot relu
x = symbols('x')
relu = Max(0, x)
plot(relu)
```

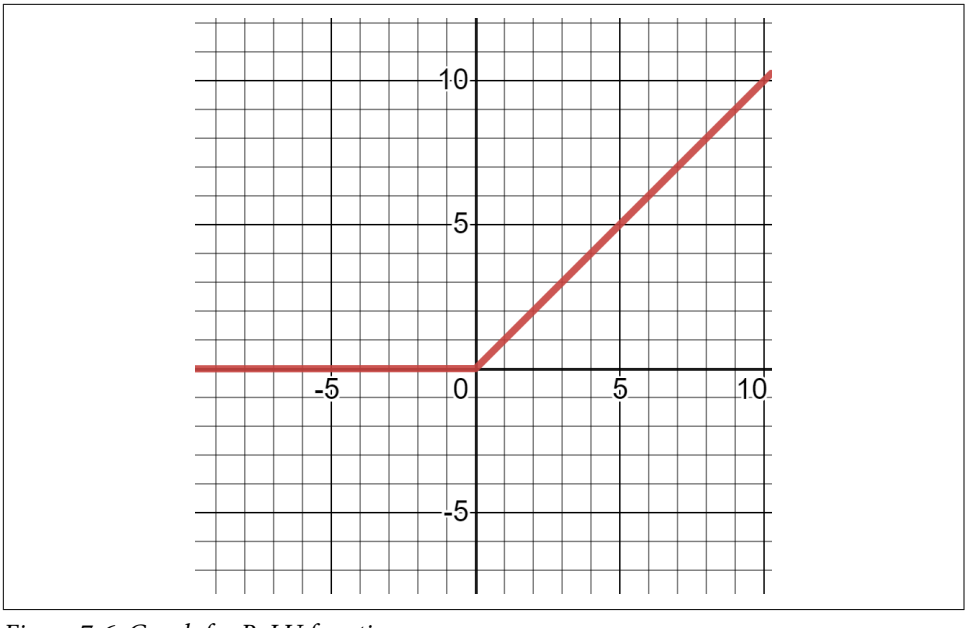

Figure 7-6. Graph for ReLU function

ReLU is short for "rectified linear unit," but that is just a fancy way of saying "turn negative values into 0." ReLU has gotten popular for hidden layers in neural networks and deep learning because of its speed and mitigation of the vanishing gradient problem. Vanishing gradients occur when the partial derivative slopes get so small they prematurely approach 0 and bring training to a screeching halt.

The output layer has an important job: it takes the piles of math from the hidden layers of the neural network and turns them into an interpretable result, such as presenting classification predictions. The output layer for this particular neural network uses the *logistic activation function*, which is a simple sigmoid curve. If you read Chapter 6, the logistic (or sigmoid) function should be familiar, and it demonstrates that logistic regression is acting as a layer in our neural network. The output node weights, biases, and sums each of the incoming values from the hidden layer. After that, it passes the resulting value through the logistic function so it outputs a number between 0 and 1. Much like logistic regression in Chapter 6, this represents a probability that the given color input into the neural network recommends a light font. If it is greater than or equal to 0.5, the neural network is suggesting a light font, but less than that will advise a dark font.

Here is the graph for the logistic function (Figure 7-7) using SymPy (Example 7-2).

Example 7-2. Logistic activation function in SymPy

```
from sympy import *

# plot logistic
x = symbols('x')
logistic = 1 / (1 + exp(-x))
plot(logistic)
```

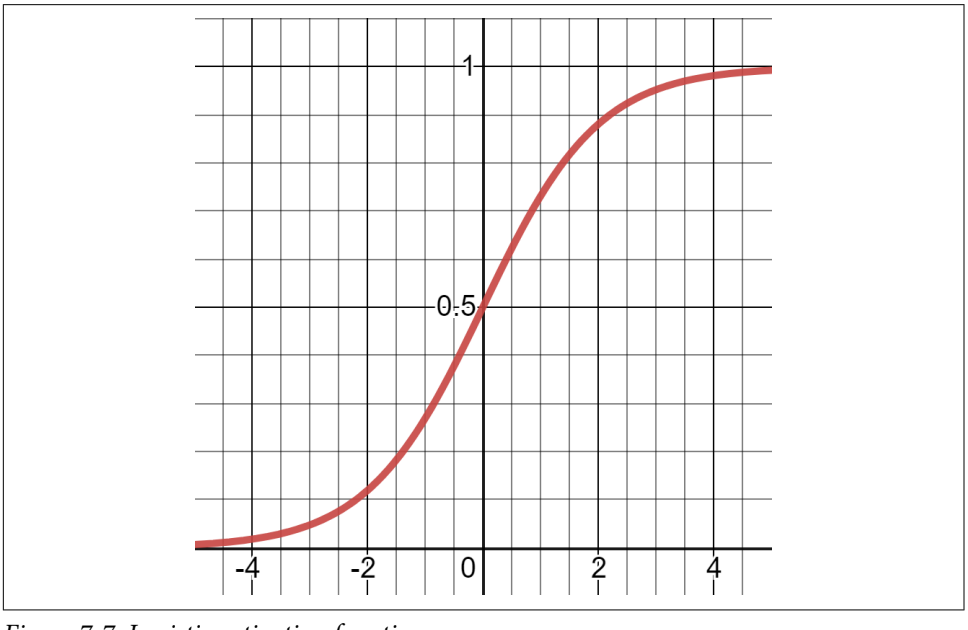

Figure 7-7. Logistic activation function

Note that when we pass a node's weighted, biased, and summed value through an activation function, we now call that an *activated output*, meaning it has been filtered through the activation function. When the activated output leaves the hidden layer, the signal is ready to be fed into the next layer. The activation function could have strengthened, weakened, or left the signal as is. This is where the brain and synapse metaphor for neural networks comes from.

Given the potential for complexity, you might be wondering if there are other activation functions. Some common ones are shown in Table 7-1.

Table 7-1. Common activation functions

Name	Typical layer used	Description	Notes
Linear	Output	Leaves values as is	Not commonly used
Logistic	Output	S-shaped sigmoid curve	Compresses values between 0 and 1, often assists binary classification
Tangent Hyperbolic	Hidden	tanh, S-shaped sigmoid curve between -1 and 1	Assists in "centering" data by bringing mean close to 0
ReLU	Hidden	Turns negative values to 0	Popular activation faster than sigmoid and tanh, mitigates vanishing gradient problems and computationally cheap
Leaky ReLU	Hidden	Multiplies negative values by 0.01	Controversial variant of ReLU that marginalizes rather than eliminates negative values
Softmax	Output	Ensures all output nodes add up to 1.0	Useful for multiple classifications and rescaling outputs so they add up to 1.0

This is not a comprehensive list of activation functions, and in theory any function could be an activation function in a neural network.

While this neural network seemingly supports two classes (light or dark font), it actually is modeled to one class: whether or not a font should be light (1) or not (0). If you wanted to support multiple classes, you could add more output nodes for each class. For instance, if you are trying to recognize handwritten digits 0–9, there would be 10 output nodes representing the probability a given image is each of those numbers. You might consider using softmax as the output activation when you have multiple classes as well. Figure 7-8 shows an example of taking a pixellated image of a digit, where the pixels are broken up as individual neural network inputs and then passed through two middle layers, and then an output layer with 10 nodes representing probabilities for 10 classes (for the digits 0–9).

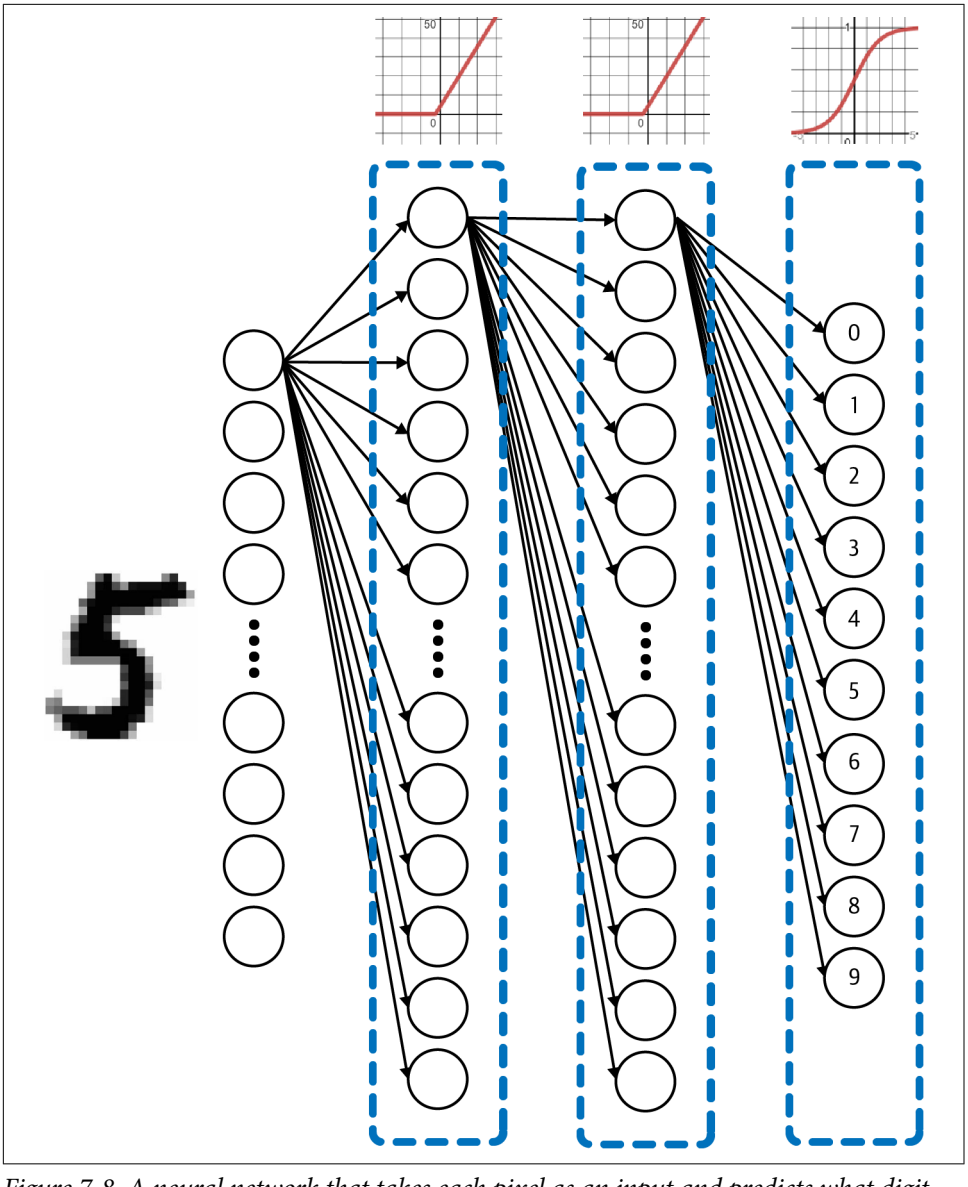

Figure 7-8. A neural network that takes each pixel as an input and predicts what digit the image contains

An example of using the MNIST dataset on a neural network can be found in Appendix A.