2 Deep learning and neural networks

This chapter covers

- Understanding perceptrons and multilayer perceptrons
- Working with the different types of activation functions
- Training networks with feedforward, error functions, and error optimization
- Performing backpropagation

In the last chapter, we discussed the computer vision (CV) pipeline components: the input image, preprocessing, extracting features, and the learning algorithm (classifier). We also discussed that in traditional ML algorithms, we manually extract features that produce a vector of features to be classified by the learning algorithm, whereas in deep learning (DL), neural networks act as both the feature extractor and the classifier. A neural network automatically recognizes patterns and extracts features from the image and classifies them into labels (figure 2.1).

In this chapter, we will take a short pause from the CV context to open the DL algorithm box from figure 2.1. We will dive deeper into how neural networks learn features and make predictions. Then, in the next chapter, we will come

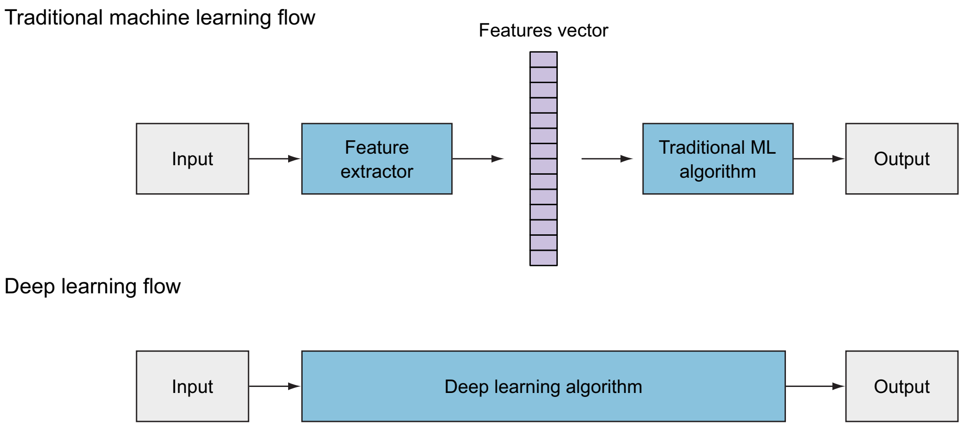


Figure 2.1 Traditional ML algorithms require manual feature extraction. A deep neural network automatically extracts features by passing the input image through its layers.

back to CV applications with one of the most popular DL architectures: convolutional neural networks.

The high-level layout of this chapter is as follows:

- We will begin with the most basic component of the neural network: the *perceptron*, a neural network that contains only one neuron.
- Then we will move on to a more complex neural network architecture that contains hundreds of neurons to solve more complex problems. This network is called a *multilayer perceptron* (MLP), where neurons are stacked in *hidden layers*. Here, you will learn the main components of the neural network architecture: the input layer, hidden layers, weight connections, and output layer.
- You will learn that the network training process consists of three main steps:
  1. Feedforward operation
  2. Calculating the error
  3. Error optimization: using backpropagation and gradient descent to select the most optimum parameters that minimize the error function

We will dive deep into each of these steps. You will see that building a neural network requires making necessary design decisions: choosing an optimizer, cost function, and activation functions, as well as designing the architecture of the network, including how many layers should be connected to each other and how many neurons should be in each layer. Ready? Let's get started!

2.1 Understanding perceptrons

Let's take a look at the artificial neural network (ANN) diagram from chapter 1 (figure 2.2). You can see that ANNs consist of many neurons that are structured in layers to perform some kind of calculations and predict an output. This architecture can be

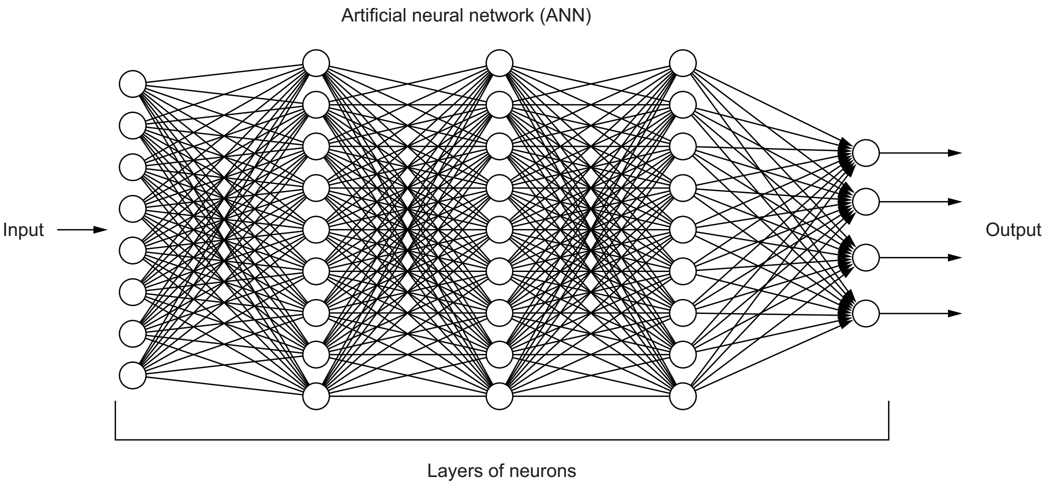


Figure 2.2 An artificial neural network consists of layers of nodes, or neurons connected with edges.

also called a *multilayer perceptron*, which is more intuitive because it implies that the network consists of perceptrons structured in multiple layers. Both terms, MLP and ANN, are used interchangeably to describe this neural network architecture.

In the MLP diagram in figure 2.2, each node is called a *neuron*. We will discuss how MLP networks work soon, but first let's zoom in on the most basic component of the neural network: the perceptron. Once you understand how a single perceptron works, it will become more intuitive to understand how multiple perceptrons work together to learn data features.

2.1.1 What is a perceptron?

The most simple neural network is the perceptron, which consists of a single neuron. Conceptually, the perceptron functions in a manner similar to a biological neuron (figure 2.3). A biological neuron receives electrical signals from its *dendrites*, modulates the electrical signals in various amounts, and then fires an output signal through its *synapses* only when the total strength of the input signals exceeds a certain threshold. The output is then fed to another neuron, and so forth.

To model the biological neuron phenomenon, the artificial neuron performs two consecutive functions: it calculates the *weighted sum* of the inputs to represent the total strength of the input signals, and it applies a *step function* to the result to determine whether to fire the output 1 if the signal exceeds a certain threshold or 0 if the signal doesn't exceed the threshold.

As we discussed in chapter 1, not all input features are equally useful or important. To represent that, each input node is assigned a weight value, called its *connection weight*, to reflect its importance.

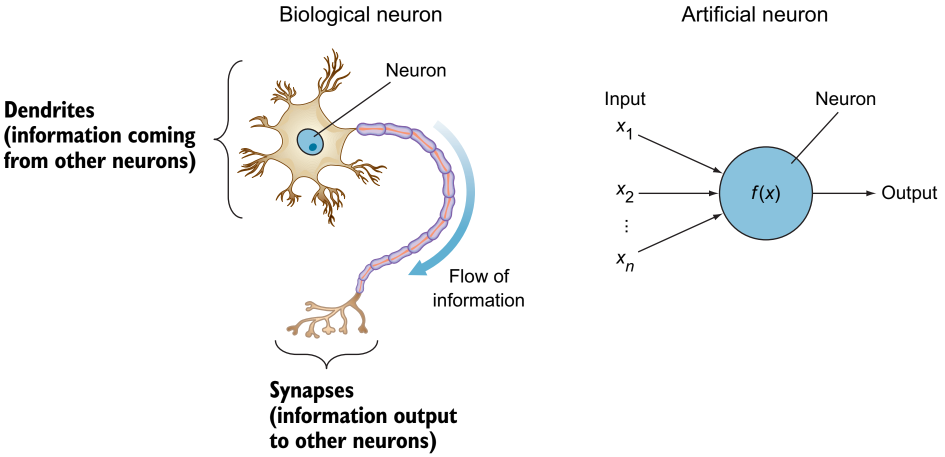


Figure 2.3 Artificial neurons were inspired by biological neurons. Different neurons are connected to each other by synapses that carry information.

Connection weights

Not all input features are equally important (or useful) features. Each input feature (x_1) is assigned its own weight (w_1) that reflects its importance in the decision-making process. Inputs assigned greater weight have a greater effect on the output. If the weight is high, it amplifies the input signal; and if the weight is low, it diminishes the input signal. In common representations of neural networks, the weights are represented by lines or edges from the input node to the perceptron.

For example, if you are predicting a house price based on a set of features like size, neighborhood, and number of rooms, there are three input features (x_1, x_2, and x_3). Each of these inputs will have a different weight value that represents its effect on the final decision. For example, if the size of the house has double the effect on the price compared with the neighborhood, and the neighborhood has double the effect compared with the number of rooms, you will see weights something like 8, 4, and 2, respectively.

How the connection values are assigned and how the learning happens is the core of the neural network training process. This is what we will discuss for the rest of this chapter.

In the perceptron diagram in figure 2.4, you can see the following:

- *Input vector*—The feature vector that is fed to the neuron. It is usually denoted with an uppercase X to represent a vector of inputs (x_1, x_2, . . ., x_n).
- *Weights vector*—Each x_1 is assigned a weight value w_1 that represents its importance to distinguish between different input datapoints.

- *Neuron functions*—The calculations performed within the neuron to modulate the input signals: the weighted sum and step activation function.
- *Output*—Controlled by the type of activation function you choose for your network. There are different activation functions, as we will discuss in detail in this chapter. For a step function, the output is either 0 or 1. Other activation functions produce probability output or float numbers. The output node represents the perceptron prediction.

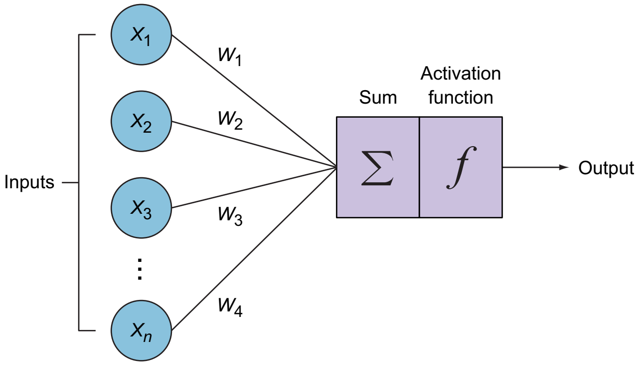


Figure 2.4 Input vectors are fed to the neuron, with weights assigned to represent importance. Calculations performed within the neuron are weighted sum and activation functions.

Let's take a deeper look at the weighted sum and step function calculations that happen inside the neuron.

WEIGHTED SUM FUNCTION

Also known as a *linear combination*, the weighted sum function is the sum of all inputs multiplied by their weights, and then added to a bias term. This function produces a straight line represented in the following equation:

$$z = \sum x_i \cdot w_i + b \text{ (bias)}$$

$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \cdots + x_n \cdot w_n + b$$

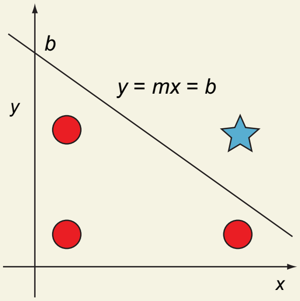
Here is how we implement the weighted sum in Python:

```
z = np.dot(w.T,X) + b
```

X is the input vector (uppercase X), w is the weights vector, and b is the y-intercept.

What is a bias in the perceptron, and why do we add it?

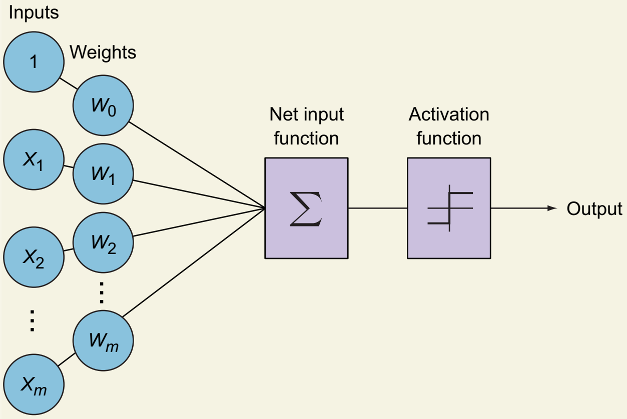
Let's brush up our memory on some linear algebra concepts to help understand what's happening under the hood. Here is the function of the straight line:



The equation of a straight line

The function of a straight line is represented by the equation ($y = mx + b$), where b is the y-intercept. To be able to define a line, you need two things: the slope of the line and a point on the line. The bias is that point on the y-axis. Bias allows you to move the line up and down on the y-axis to better fit the prediction with the data. Without the bias (b), the line always has to go through the origin point (0,0), and you will get a poorer fit. To visualize the importance of bias, look at the graph in the above figure and try to separate the circles from the star using a line that passes through the origin (0,0). It is not possible.

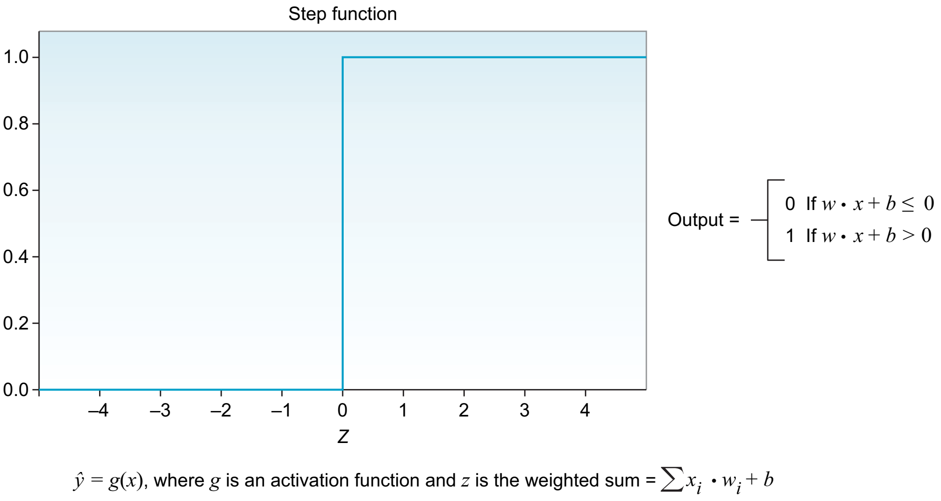
The input layer can be given biases by introducing an extra input node that always has a value of 1, as you can see in the next figure. In neural networks, the value of the bias (b) is treated as an extra weight and is learned and adjusted by the neuron to minimize the cost function, as we will learn in the following sections of this chapter.



The input layer can be given biases by introducing an extra input that always has a value of 1.

STEP ACTIVATION FUNCTION

In both artificial and biological neural networks, a neuron does not just output the bare input it receives. Instead, there is one more step, called an *activation function*; this is the decision-making unit of the brain. In ANNs, the activation function takes the same weighted sum input from before ($z = \Sigma x_i \cdot w_i + b$) and activates (fires) the neuron if the weighted sum is higher than a certain threshold. This activation happens based on the activation function calculations. Later in this chapter, we'll review the different types of activation functions and their general purpose in the broader context of neural networks. The simplest activation function used by the perceptron algorithm is the step function that produces a binary output (0 or 1). It basically says that if the summed input ≥ 0, it "fires" (output = 1); else (summed input < 0), it doesn't fire (output = 0) (figure 2.5).



$$\text{Output} = \begin{cases} 0 & \text{If } w \cdot x + b \leq 0 \\ 1 & \text{If } w \cdot x + b > 0 \end{cases}$$

$\hat{y} = g(x)$, where g is an activation function and z is the weighted sum $= \sum x_i \cdot w_i + b$

Figure 2.5 The step function produces a binary output (0 or 1). If the summed input ≥ 0, it "fires" (output = 1); else (summed input < 0) it doesn't fire (output = 0).

This is how the step function looks in Python:

```
def step_function(z):
    if z <= 0:
         return 0
   else:
    return 1
```

***z* is the weighted sum = $\Sigma x_i \cdot w_i + b$**

2.1.2 How does the perceptron learn?

The perceptron uses trial and error to learn from its mistakes. It uses the weights as knobs by tuning their values up and down until the network is trained (figure 2.6).

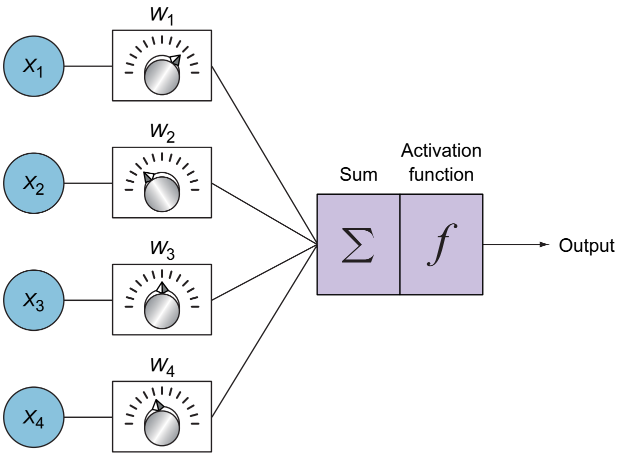


Figure 2.6 Weights are tuned up and down during the learning process to optimize the value of the loss function.

The perceptron's learning logic goes like this:

1. The neuron calculates the weighted sum and applies the activation function to make a prediction $\hat{y}$. This is called the feedforward process:

$$\hat{y} = \text{activation}\left(\sum x_i \cdot w_i + b\right)$$

2. It compares the output prediction with the correct label to calculate the error:

$$\text{error} = y - \hat{y}$$

3. It then updates the weight. If the prediction is too high, it adjusts the weight to make a lower prediction the next time, and vice versa.
4. Repeat!

This process is repeated many times, and the neuron continues to update the weights to improve its predictions until step 2 produces a very small error (close to zero), which means the neuron's prediction is very close to the correct value. At this point, we can stop the training and save the weight values that yielded the best results to apply to future cases where the outcome is unknown.

2.1.3 Is one neuron enough to solve complex problems?

The short answer is no, but let's see why. The perceptron is a linear function. This means the trained neuron will produce a straight line that separates our data.

Suppose we want to train a perceptron to predict whether a player will be accepted into the college squad. We collect all the data from previous years and train the perceptron to predict whether players will be accepted based on only two features (height and weight). The trained perceptron will find the best weights and bias values to produce the *straight line* that best separates the accepted from non-accepted (best fit). The line has this equation:

$$z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$$

After the training is complete on the training data, we can start using the perceptron to predict with new players. When we get a player who is 150 cm in height and 12 years old, we compute the previous equation with the values (150, 12). When plotted in a graph (figure 2.7), you can see that it falls below the line: the neuron is predicting that this player will not be accepted. If it falls above the line, then the player will be accepted.

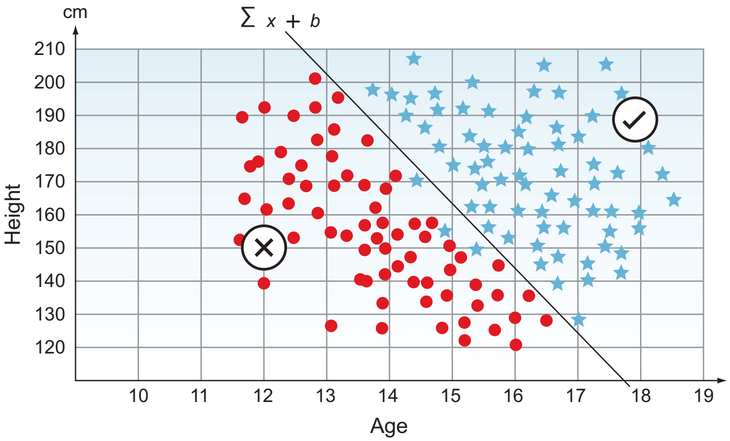


Figure 2.7 Linearly separable data can be separated by a straight line.

In figure 2.7, the single perceptron works fine because our data was *linearly separable*. This means the training data can be separated by a straight line. But life isn't always that simple. What happens when we have a more complex dataset that cannot be separated by a straight line (*nonlinear dataset*)?

As you can see in figure 2.8, a single straight line will not separate our training data. We say that it does not *fit* our data. We need a more complex network for more complex data like this. What if we built a network with two perceptrons? This would produce two lines. Would that help us separate the data better?

Okay, this is definitely better than the straight line. But, I still see some color mispredictions. Can we add more neurons to make the function fit better? Now you are getting it. Conceptually, the more neurons we add, the better the network will fit our

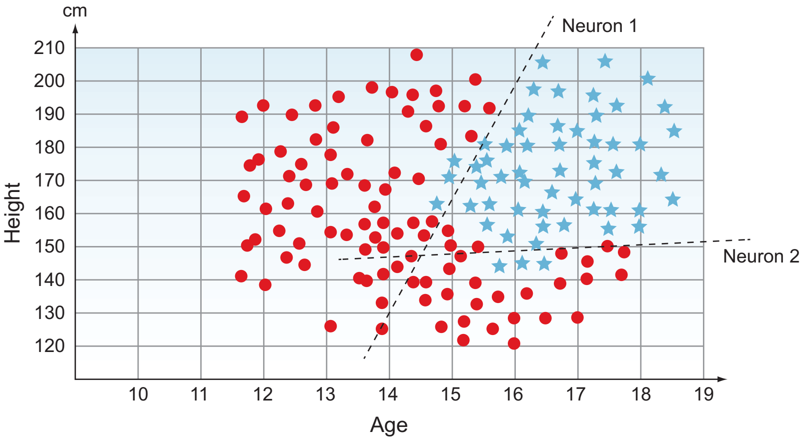


Figure 2.8 In a nonlinear dataset, a single straight line cannot separate the training data. A network with two perceptrons can produce two lines and help separate the data further in this example.

training data. In fact, if we add too many neurons, this will make the network *overfit* the training data (not good). But we will talk about this later. The general rule here is that the more complex our network is, the better it learns the features of our data.

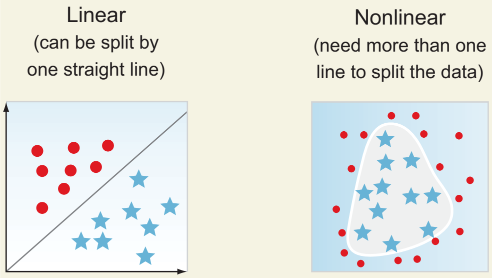
2.2 Multilayer perceptrons

We saw that a single perceptron works great with simple datasets that can be separated by a line. But, as you can imagine, the real world is much more complex than that. This is where neural networks can show their full potential.

Linear vs. nonlinear problems

- *Linear datasets*—The data can be split with a single straight line.
- *Nonlinear datasets*—The data cannot be split with a single straight line. We need more than one line to form a shape that splits the data.

Look at this 2D data. In the linear problem, the stars and dots can be easily classified by drawing a single straight line. In nonlinear data, a single line will not separate both shapes.



Examples of linear data and nonlinear data

To split a nonlinear dataset, we need more than one line. This means we need to come up with an architecture to use tens and hundreds of neurons in our neural network. Let's look at the example in figure 2.9. Remember that a perceptron is a linear function that produces a straight line. So in order to fit this data, we try to create a triangle-like shape that splits the dark dots. It looks like three lines would do the job.

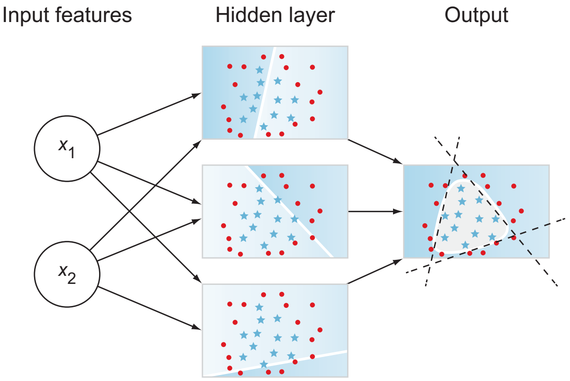


Figure 2.9 A perceptron is a linear function that produces a straight line. So to fit this data, we need three perceptrons to create a triangle-like shape that splits the dark dots.

Figure 2.9 is an example of a small neural network that is used to model nonlinear data. In this network, we used three neurons stacked together in one layer called a *hidden layer*, so called because we don't see the output of these layers during the training process.

2.2.1 Multilayer perceptron architecture

We've seen how a neural network can be designed to have more than one neuron. Let's expand on this idea with a more complex dataset. The diagram in figure 2.10 is from the Tensorflow playground website (https://playground.tensorflow.org). We try to model a spiral dataset to distinguish between two classes. In order to fit this dataset, we need to build a neural network that contains tens of neurons. A very common neural network architecture is to stack the neurons in layers on top of each other, called *hidden layers*. Each layer has *n* number of neurons. Layers are connected to each other by weight connections. This leads to the multilayer perceptron (MLP) architecture in the figure.

The main components of the neural network architecture are as follows:

- *Input layer*—Contains the feature vector.
- *Hidden layers*—The neurons are stacked on top of each other in hidden layers. They are called "hidden" layers because we don't see or control the input going into these layers or the output. All we do is feed the feature vector to the input layer and see the output coming out of the output layer.
- *Weight connections (edges)*—Weights are assigned to each connection between the nodes to reflect the importance of their influence on the final output prediction. In graph network terms, these are called *edges* connecting the *nodes*.

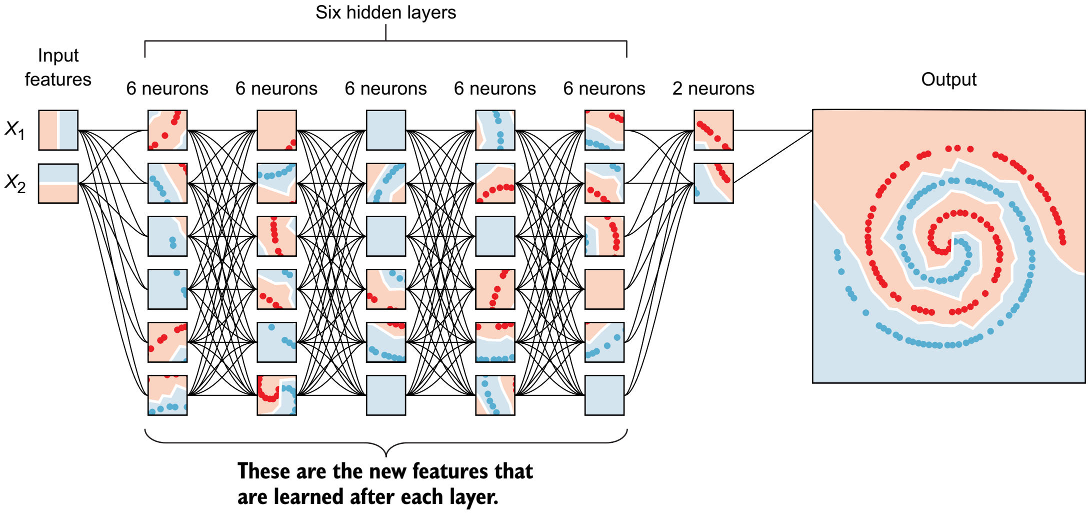


Figure 2.10 Tensorflow playground example representation of the feature learning in a deep neural network

- *Output layer*—We get the answer or prediction from our model from the output layer. Depending on the setup of the neural network, the final output may be a real-valued output (regression problem) or a set of probabilities (classification problem). This is determined by the type of activation function we use in the neurons in the output layer. We'll discuss the different types of activation functions in the next section.

We discussed the input layer, weights, and output layer. The next area of this architecture is the hidden layers.

2.2.2 What are hidden layers?

This is where the core of the feature-learning process takes place. When you look at the hidden layer nodes in figure 2.10, you see that the early layers detect simple patterns to learn low-level features (straight lines). Later layers detect patterns within patterns to learn more complex features and shapes, then patterns within patterns within patterns, and so on. This concept will come in handy when we discuss convolutional networks in later chapters. For now, know that, in neural networks, we stack hidden layers to learn complex features from each other until we fit our data. So when you are designing your neural network, if your network is not fitting the data, the solution could be adding more hidden layers.

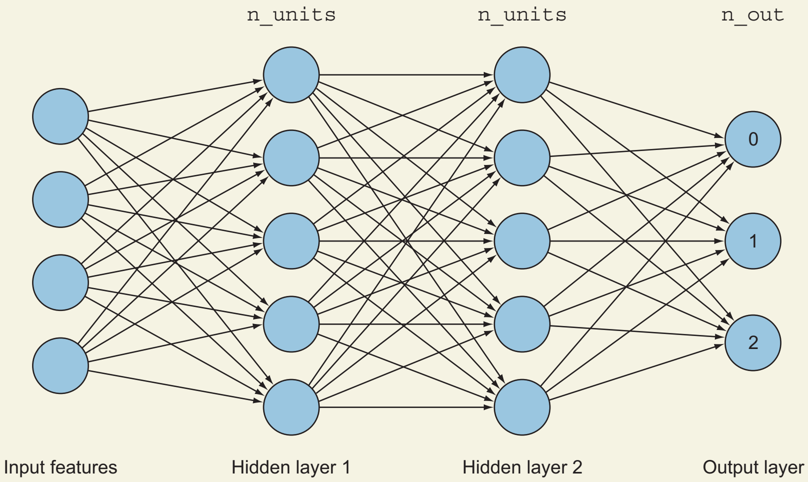
2.2.3 How many layers, and how many nodes in each layer?

As a machine learning engineer, you will mostly be designing your network and tuning its hyperparameters. While there is no single prescribed recipe that fits all models, we will try throughout this book to build your hyperparameter tuning intuition, as well as recommend some starting points. The number of layers and the number of neurons in each layer are among the important hyperparameters you will be designing when working with neural networks.

A network can have one or more hidden layers (technically, as many as you want). Each layer has one or more neurons (again, as many as you want). Your main job, as a machine learning engineer, is to design these layers. Usually, when we have two or more hidden layers, we call this a *deep neural network*. The general rule is this: the deeper your network is, the more it will fit the training data. But too much depth is not a good thing, because the network can fit the training data so much that it fails to generalize when you show it new data (overfitting); also, it becomes more computationally expensive. So your job is to build a network that is not too simple (one neuron) and not too complex for your data. It is recommended that you read about different neural network architectures that are successfully implemented by others to build an intuition about what is too simple for your problem. Start from that point, maybe three to five layers (if you are training on a CPU), and observe the network performance. If it is performing poorly (underfitting), add more layers. If you see signs of overfitting (discussed later), then decrease the number of layers. To build a sense of how neural networks perform when you add more layers, play around with the Tensorflow playground (https://playground.tensorflow.org).

Fully connected layers

It is important to call out that the layers in classical MLP network architectures are fully connected to the next hidden layer. In the following figure, notice that each node in a layer is connected to all nodes in the previous layer. This is called a *fully connected network*. These edges are the weights that represent the importance of this node to the output value.



A fully connected network

In later chapters, we will discuss other variations of neural network architecture (like convolutional and recurrent networks). For now, know that this is the most basic neural network architecture, and it can be referred to by any of these names: ANN, MLP, fully connected network, or feedforward network.

Let's do a quick exercise to find out how many edges we have in our example. Suppose that we designed an MLP network with two hidden layers, and each has five neurons:

- `Weights_0_1`: (4 nodes in the input layer) × (5 nodes in layer 1) + 5 biases [1 bias per neuron] = 25 edges
- `Weights_1_2`: 5 × 5 nodes + 5 biases = 30 edges
- `Weights_2_output`: 5 × 3 nodes + 3 bias = 18 edges
- Total edges (weights) in this network = 73

We have a total of 73 weights in this very simple network. The values of these weights are randomly initialized, and then the network performs feedforward and backpropagation to learn the best values of weights that most fit our model to the training data.

To see the number of weights in this network, try to build this simple network in Keras as follows:

```
model = Sequential([
    Dense(5, input_dim=4),
    Dense(5),
    Dense(3)
])
```

And print the model summary:

```
model.summary()
```

The output will be as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 5)                 25
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 30
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 18
=================================================================
Total params: 73
Trainable params: 73
Non-trainable params: 0
```

2.2.4 Some takeaways from this section

Let's recap what we've discussed so far:

- We talked about the analogy between biological and artificial neurons: both have inputs and a neuron that does some calculations to modulate the input signals and create output.
- We zoomed in on the artificial neuron's calculations to explore its two main functions: weighted sum and the activation function.
- We saw that the network assigns random weights to all the edges. These weight parameters reflect the usefulness (or importance) of these features on the output prediction.
- Finally, we saw that perceptrons contain a single neuron. They are linear functions that produce a straight line to split linear data. In order to split more complex data (nonlinear), we need to apply more than one neuron in our network to form a multilayer perceptron.
- The MLP architecture contains input features, connection weights, hidden layers, and an output layer.
- We discussed the high-level process of how the perceptron learns. The learning process is a repetition of three main steps: feedforward calculations to produce a prediction (weighted sum and activation), calculating the error, and backpropagating the error and updating the weights to minimize the error.

We should also keep in mind some of the important points about neural network hyperparameters:

- *Number of hidden layers*—You can have as many layers as you want, each with as many neurons as you want. The general idea is that the more neurons you have, the better your network will learn the training data. But if you have too many neurons, this might lead to a phenomenon called *overfitting*: the network learned the training set so much that it memorized it instead of learning its features. Thus, it will fail to generalize. To get the appropriate number of layers, start with a small network, and observe the network performance. Then start adding layers until you get satisfying results.
- *Activation function*—There are many types of activation functions, the most popular being ReLU and softmax. It is recommended that you use ReLU activation in the hidden layers and Softmax for the output layer (you will see how this is implemented in most projects in this book).
- *Error function*—Measures how far the network's prediction is from the true label. Mean square error is common for regression problems, and cross-entropy is common for classification problems.
- *Optimizer*—Optimization algorithms are used to find the optimum weight values that minimize the error. There are several optimizer types to choose from. In this chapter, we discuss batch gradient descent, stochastic gradient descent, and

mini-batch gradient descent. Adam and RMSprop are two other popular optimizers that we don't discuss.

- *Batch size*—Mini-batch size is the number of sub-samples given to the network, after which parameter update happens. Bigger batch sizes learn faster but require more memory space. A good default for batch size might be 32. Also try 64, 128, 256, and so on.
- *Number of epochs*—The number of times the entire training dataset is shown to the network while training. Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing (overfitting).
- *Learning rate*—One of the optimizer's input parameters that we tune. Theoretically, a learning rate that is too small is guaranteed to reach the minimum error (if you train for infinity time). A learning rate that is too big speeds up the learning but is not guaranteed to find the minimum error. The default `lr` value of the optimizer in most DL libraries is a reasonable start to get decent results. From there, go down or up by one order of magnitude. We will discuss the learning rate in detail in chapter 4.

More on hyperparameters

Other hyperparameters that we have not discussed yet include dropout and regularization. We will discuss hyperparameter tuning in detail in chapter 4, after we cover convolutional neural networks in chapter 3.

In general, the best way to tune hyperparameters is by trial and error. By getting your hands dirty with your own projects as well as learning from other existing neural network architectures, you will start to develop intuition about good starting points for your hyperparameters.

Learn to analyze your network's performance and understand which hyperparameter you need to tune for each symptom. And this is what we are going to do in this book. By understanding the reasoning behind these hyperparameters and observing the network performance in the projects at the end of the chapters, you will develop a feel for which hyperparameter to tune for a particular effect. For example, if you see that your error value is not decreasing and keeps oscillating, then you might fix that by reducing the learning rate. Or, if you see that the network is performing poorly in learning the training data, this might mean that the network is underfitting and you need to build a more complex model by adding more neurons and hidden layers.

2.3 Activation functions

When you are building your neural network, one of the design decisions that you will need to make is what activation function to use for your neurons' calculations. Activation functions are also referred to as *transfer functions* or *nonlinearities* because they transform the linear combination of a weighted sum into a nonlinear model. An activation function is placed at the end of each perceptron to decide whether to activate this neuron.

Why use activation functions at all? Why not just calculate the weighted sum of our network and propagate that through the hidden layers to produce an output?

The purpose of the activation function is to introduce nonlinearity into the network. Without it, a multilayer perceptron will perform similarly to a single perceptron no matter how many layers we add. Activation functions are needed to restrict the output value to a certain finite value. Let's revisit the example of predicting whether a player gets accepted (figure 2.11).

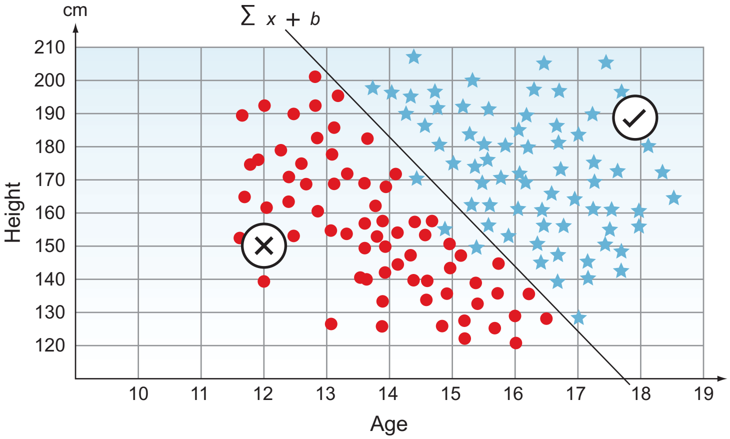


Figure 2.11 This example revisits the prediction of whether a player gets accepted from section 2.1.

First, the model calculates the weighted sum and produces the linear function (z):

$$z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$$

The output of this function has no bound. z could literally be any number. We use an activation function to wrap the prediction values to a finite value. In this example, we use a step function where if $z > 0$, then above the line (accepted) and if $z < 0$, then below the line (rejected). So without the activation function, we just have a linear function that produces a number, but no decision is made in this perceptron. The activation function is what decides whether to fire this perceptron.

There are infinite activation functions. In fact, the last few years have seen a lot of progress in the creation of state-of-the-art activations. However, there are still relatively few activations that account for the vast majority of activation needs. Let's dive deeper into some of the most common types of activation functions.

2.3.1 Linear transfer function

A *linear transfer function*, also called an *identity function*, indicates that the function passes a signal through unchanged. In practical terms, the output will be equal to the input, which means we don't actually have an activation function. So no matter how many layers our neural network has, all it is doing is computing a linear activation function or, at most, scaling the weighted average coming in. But it doesn't transform input into a nonlinear function.

$$\text{activation}(z) = z = wx + b$$

The composition of two linear functions is a linear function, so unless you throw a nonlinear activation function in your neural network, you are not computing any interesting functions no matter how deep you make your network. No learning here!

To understand why, let's calculate the derivative of the activation $z(x) = w \cdot x + b$, where $w = 4$ and $b = 0$. When we plot this function, it looks like figure 2.12. Then the derivative of $z(x) = 4x$ is $z'(x) = 4$ (figure 2.13).

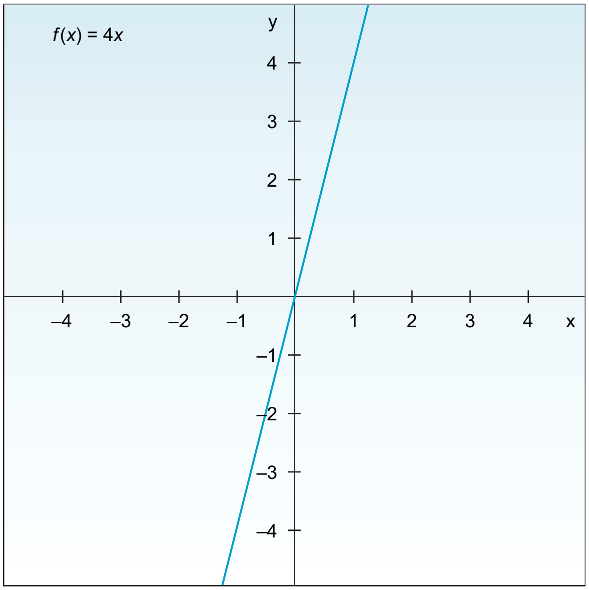


Figure 2.12 The plot for the activation function *f*(*x*) = 4*x*

The derivative of a linear function is constant: it does not depend on the input value x. This means that every time we do a backpropagation, the gradient will be the same. And this is a big problem: we are not really improving the error, since the gradient is pretty much the same. This will be clearer when we discuss backpropagation later in this chapter.

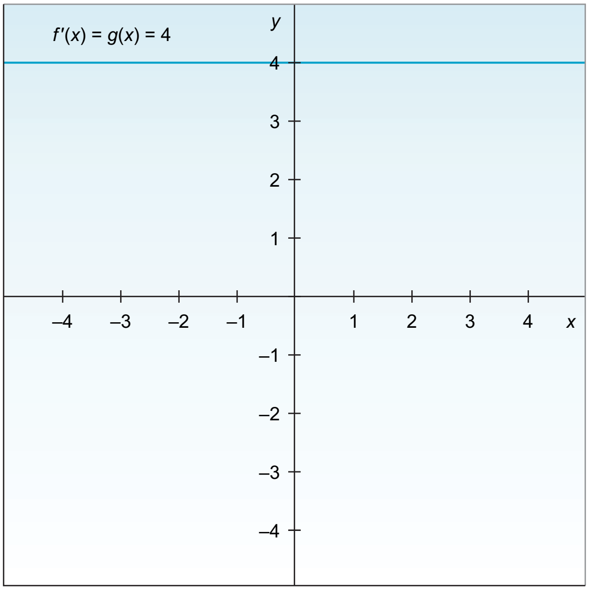


Figure 2.13 The plot for the derivative of $z(x) = 4x$ is $z'(x) = 4$.

2.3.2 Heaviside step function (binary classifier)

The *step function* produces a binary output. It basically says that if the input $x > 0$, it fires (output $y = 1$); else (input < 0), it doesn't fire (output $y = 0$). It is mainly used in binary classification problems like true or false, spam or not spam, and pass or fail (figure 2.14).

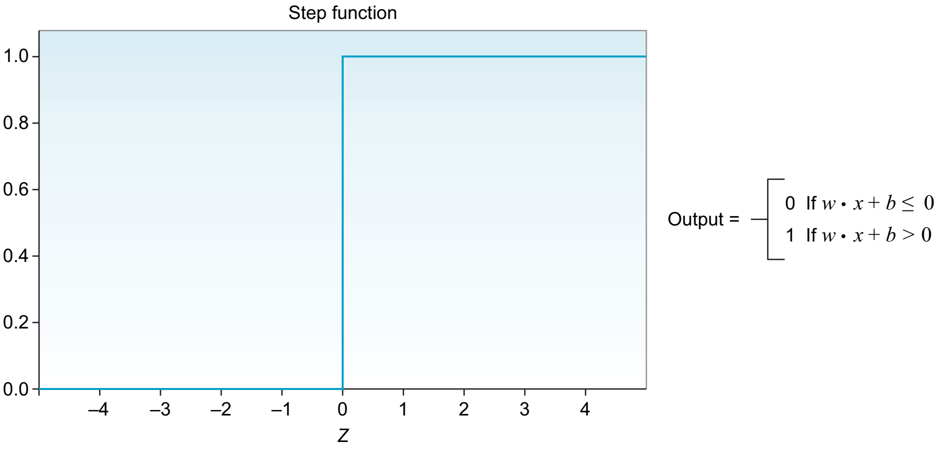


Figure 2.14 Step functions are commonly used in binary classification problems because they transform the input into zero or one.

2.3.3 Sigmoid/logistic function

This is one of the most common activation functions. It is often used in binary classifiers to predict the *probability* of a class when you have two classes. The sigmoid squishes all the values to a probability between 0 and 1, which reduces extreme values or outliers in the data without removing them. Sigmoid or logistic functions convert infinite continuous variables (range between –∞ to +∞) into simple probabilities between 0 and 1. It is also called the *S-shape curve* because when plotted in a graph, it produces an S-shaped curve. While the step function is used to produce a discrete answer (pass or fail), sigmoid is used to produce the probability of passing and probability of failing (figure 2.15):

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

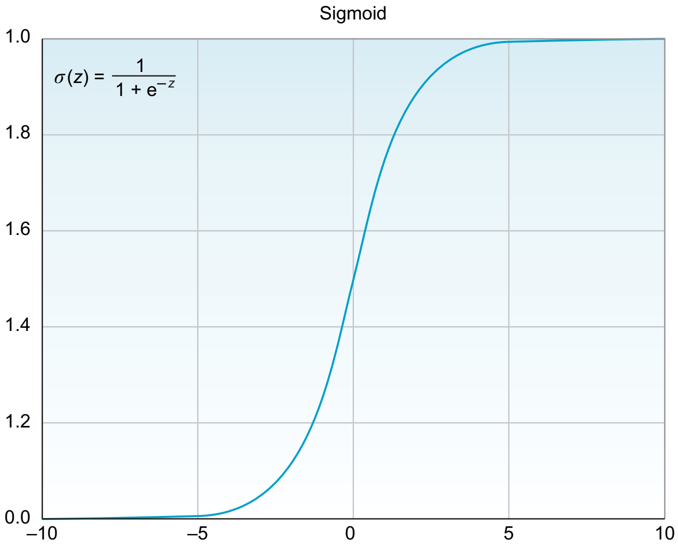


Figure 2.15 While the step function is used to produce a discrete answer (pass or fail), sigmoid is used to produce the probability of passing or failing.

Here is how sigmoid is implemented in Python:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

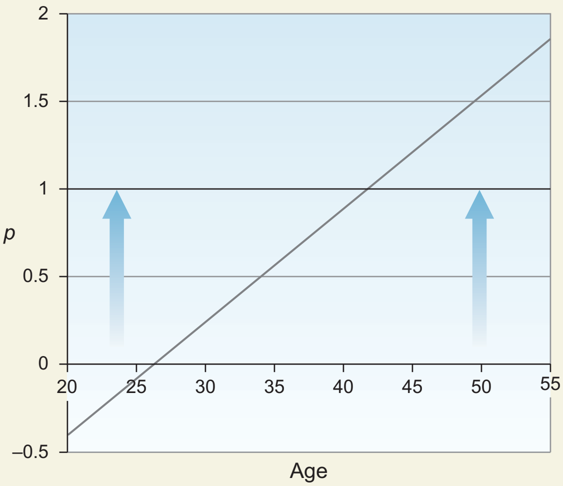
Imports numpy

Sigmoid activation function

Just-in-time linear algebra (optional)

Let's take a deeper dive into the math side of the sigmoid function to understand the problem it helps solve and how the sigmoid function equation is driven. Suppose that we are trying to predict whether patients have diabetes based on only one feature: their age. When we plot the data we have about our patients, we get the linear model shown in the figure:

$$z = \beta_0 + \beta_1 \text{ age}$$



The linear model we get when we plot our data about our patients

In this plot, you can observe the balance of probabilities that should go from 0 to 1. Note that when patients are below the age of 25, the predicted probabilities are negative; meanwhile, they are higher than 1 (100%) when patients are older than 43 years old. This is a clear example of why linear functions do not work in most cases. Now, how do we fix this to give us probabilities within the range of 0 < probability < 1?

First, we need to do something to eliminate all the negative probability values. The exponential function is a great solution for this problem because the exponent of anything (and I mean *anything*) is always going to be positive. So let's apply that to our linear equation to calculate the probability (*p*):

$$p = \exp(z) = \exp(\beta_0 + \beta_1 \text{ age})$$

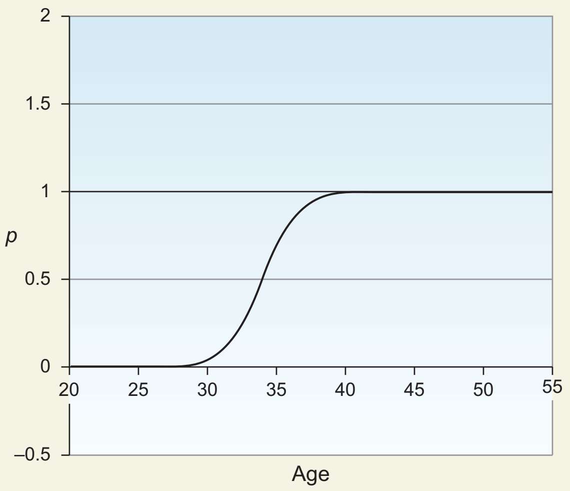
This equation ensures that we always get probabilities greater than 0. Now, what about the values that are higher than 1? We need to do something about them. With proportions, any given number divided by a number that is greater than it will give us a number smaller than 1. Let's do exactly that to the previous equation. We divide the equation by its value plus a small value: either 1 or a (in some cases very small) value—let's call it epsilon (ε):

$$p = \frac{\exp(z)}{\exp(z) + \varepsilon}$$

If you divide the equation by exp(z), you get

$$p = \frac{1}{1 + \exp(-z)}$$

When we plot the probability of this equation, we get the S shape of the sigmoid function, where probability is no longer below 0 or above 1. In fact, as patients' ages grow, the probability asymptotically gets closer to 1; and as the weights move down, the function asymptotically gets closer to 0 but is never outside the $0 < p < 1$ range. This is the plot of the sigmoid function and logistic regression.



As patients get older, the probability asymptotically gets closer to 1. This is the plot of the sigmoid function and logistic regression.

2.3.4 Softmax function

The softmax function is a generalization of the sigmoid function. It is used to obtain classification probabilities when we have more than two classes. It forces the outputs of a neural network to sum to 1 (for example, 0 < output < 1). A very common use case in deep learning problems is to predict a single class out of many options (more than two).

The softmax equation is as follows:

$$\sigma(x_j) = \frac{e^{x_j}}{\Sigma_i \, e^{x_i}}$$

Figure 2.16 shows an example of the softmax function.

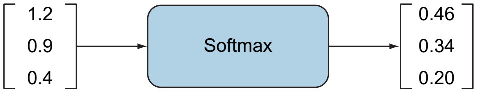


Figure 2.16 The softmax function transforms the input values to probability values between 0 and 1.

TIP Softmax is the go-to function that you will often use at the output layer of a classifier when you are working on a problem where you need to predict a class between more than two classes. Softmax works fine if you are classifying two classes, as well. It will basically work like a sigmoid function. By the end of this section, I'll tell you my recommendations about when to use each activation function.

2.3.5 Hyperbolic tangent function (tanh)

The hyperbolic tangent function is a shifted version of the sigmoid version. Instead of squeezing the signal values between 0 and 1, tanh squishes all values to the range –1 to 1. Tanh almost always works better than the sigmoid function in hidden layers because it has the effect of centering your data so that the mean of the data is close to zero rather than 0.5, which makes learning for the next layer a little bit easier:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

One of the downsides of both sigmoid and tanh functions is that if (z) is very large or very small, then the gradient (or derivative or slope) of this function becomes very small (close to zero), which will slow down gradient descent (figure 2.17). This is when the ReLU activation function (explained next) provides a solution.

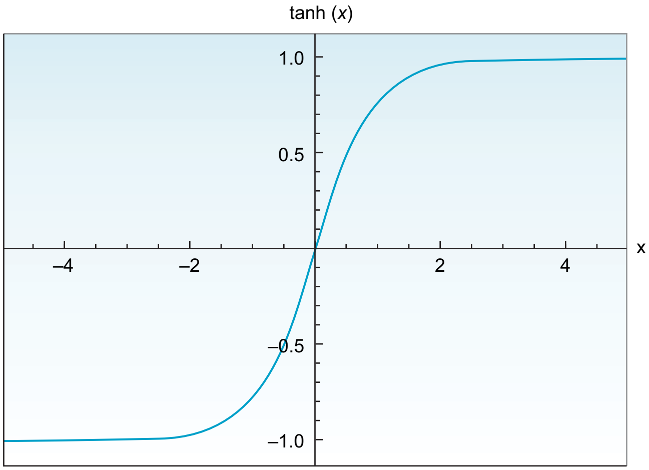


Figure 2.17 If (z) is very large or very small, then the gradient (or derivative or slope) of this function becomes very small (close to zero).

2.3.6 Rectified linear unit

The rectified linear unit (ReLU) activation function activates a node only if the input is above zero. If the input is below zero, the output is always zero. But when the input is higher than zero, it has a linear relationship with the output variable. The ReLU function is represented as follows:

$$f(x) = \max(0, x)$$

At the time of writing, ReLU is considered the state-of-the-art activation function because it works well in many different situations, and it tends to train better than sigmoid and tanh in hidden layers (figure 2.18).

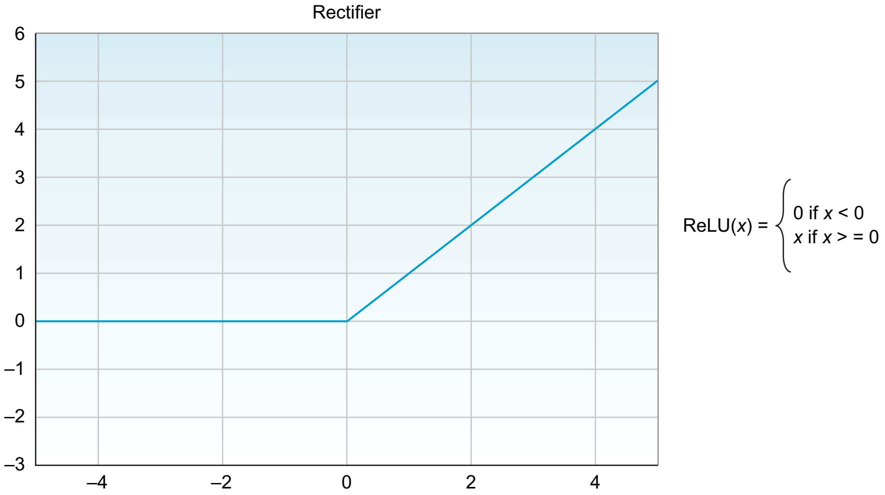


Figure 2.18 The ReLU function eliminates all negative values of the input by transforming them into zeros.

Here is how ReLU is implemented in Python:

```
def relu(x):
    if x < 0:
      return 0
    else:
return x
```

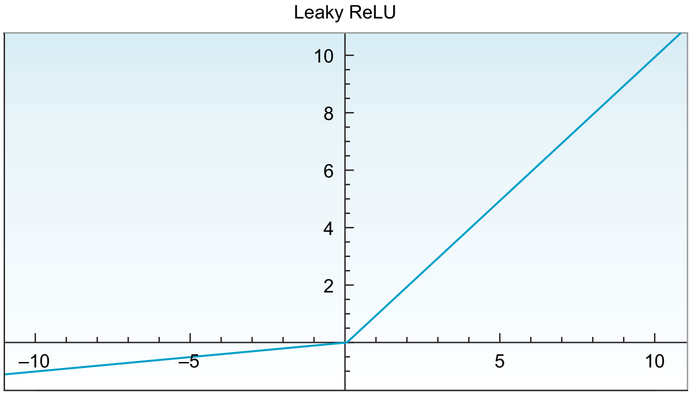
ReLU activation function

2.3.7 Leaky ReLU

One disadvantage of ReLU activation is that the derivative is equal to zero when (x) is negative. Leaky ReLU is a ReLU variation that tries to mitigate this issue. Instead of having the function be zero when $x < 0$, leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative. It usually works better than the ReLU function, although it's not used as much in practice. Take a look at the leaky ReLU graph in figure 2.19; can you see the leak?

$$f(x) = \max(0.01x, x)$$

Why 0.01? Some people like to use this as another hyperparameter to tune, but that would be overkill, since you already have other, bigger problems to worry about. Feel free to try different values (0.1, 0.01, 0.002) in your model and see how they work.



$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x = > 0 \end{cases}$$

Figure 2.19 Instead of having the function be zero when *x* < 0, leaky ReLU introduces a small negative slope (around 0.01) when (*x*) is negative.

Here is how Leaky ReLU is implemented in Python:

```
def leaky_relu(x):
    if x < 0:
      return x * 0.01
    else:
return x
```

Leaky ReLU activation function with a 0.01 leak

Table 2.1 summarizes the various activation functions we've discussed in this section.

Table 2.1 A cheat sheet of the most common activation functions

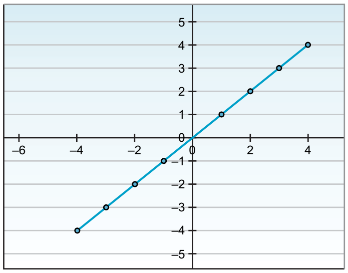
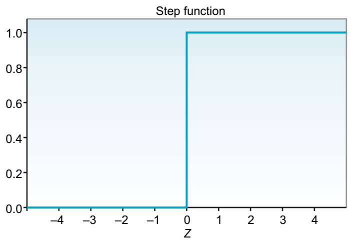
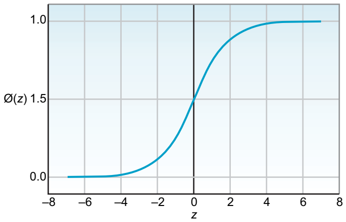
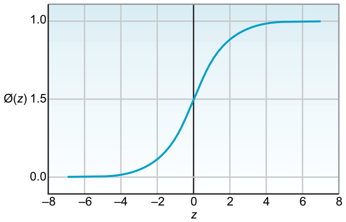
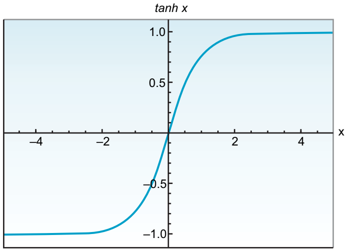
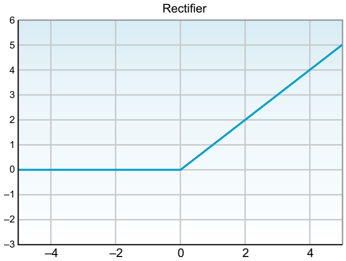
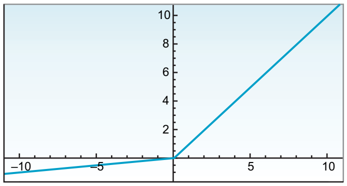
Activation function	Description	Plot	Equation
Linear transfer function (identity function)	The signal passes through it unchanged. It remains a linear function. Almost never used.		$f(x) = x$
Heaviside step function (binary classifier)	Produces a binary output of 0 or 1. Mainly used in binary classification to give a discrete value.		$\text{output} = \begin{cases} 0 \text{ if } w \cdot x + b \leq 0 \\ 1 \text{ if } w \cdot x + b > 0 \end{cases}$

Table 2.1 A cheat sheet of the most common activation functions

Activation function	Description	Plot	Equation
Sigmoid/ logistic function	Squishes all the values to a probability between 0 and 1, which reduces extreme values or outliers in the data. Usually used to classify two classes.		$\sigma(z) = \frac{1}{1 + e^{-z}}$
Softmax function	A generalization of the sigmoid function. Used to obtain classification probabilities when we have more than two classes.		$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$
Hyperbolic tangent function (tanh)	Squishes all values to the range of –1 to 1. Tanh almost always works better than the sigmoid function in hidden layers.		$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$ $= \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
Rectified linear unit (ReLU)	Activates a node only if the input is above zero. Always recommended for hidden layers. Better than tanh.		$f(x) = \max(0, x)$
Leaky ReLU	Instead of having the function be zero when $x < 0$, leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative.		$f(x) = \max(0.01x, x)$

Hyperparameter alert

Due to the number of activation functions, it may appear to be an overwhelming task to select the appropriate activation function for your network. While it is important to select a good activation function, I promise this is not going to be a challenging task when you design your network. There are some rules of thumb that you can start with, and then you can tune the model as needed. If you are not sure what to use, here are my two cents about choosing an activation function:

- *For hidden layers*—In most cases, you can use the ReLU activation function (or leaky ReLU) in hidden layers, as you will see in the projects that we will build throughout this book. It is increasingly becoming the default choice because it is a bit faster to compute than other activation functions. More importantly, it reduces the likelihood of the gradient vanishing because it does not saturate for large input values—as opposed to the sigmoid and tanh activation functions, which saturate at ~ 1. Remember, the gradient is the slope. When the function plateaus, this will lead to no slope; hence, the gradient starts to vanish. This makes it harder to *descend* to the minimum error (we will talk more about this phenomenon, called *vanishing/exploding gradients*, in later chapters).
- *For the output layer*—The softmax activation function is generally a good choice for most classification problems when the classes are mutually exclusive. The sigmoid function serves the same purpose when you are doing binary classification. For regression problems, you can simply use no activation function at all, since the weighted sum node produces the continuous output that you need: for example, if you want to predict house pricing based on the prices of other houses in the same neighborhood.

2.4 The feedforward process

Now that you understand how to stack perceptrons in layers, connect them with weights/edges, perform a weighted sum function, and apply activation functions, let's implement the complete forward-pass calculations to produce a prediction output. The process of computing the linear combination and applying the activation function is called *feedforward*. We briefly discussed feedforward several times in the previous sections; let's take a deeper look at what happens in this process.

The term *feedforward* is used to imply the forward direction in which the information flows from the input layer through the hidden layers, all the way to the output layer. This process happens through the implementation of two consecutive functions: the weighted sum and the activation function. In short, the forward pass is the calculations through the layers to make a prediction.

Let's take a look at the simple three-layer neural network in figure 2.20 and explore each of its components:

- *Layers*—This network consists of an input layer with three input features, and three hidden layers with 3, 4, 1 neurons in each layer.

- *Weights and biases (w, b)*—The edges between nodes are assigned random weights denoted as $W_{ab}^{(n)}$, where (n) indicates the layer number and (ab) indicates the weighted edge connecting the ath neuron in layer (n) to the bth neuron in the previous layer $(n - 1)$. For example, $W_{23}^{(2)}$ is the weight that connects the second node in layer 2 to the third node in layer 1 (a_2^2 to a_1^3). (Note that you can see different denotations of $W_{ab}^{(n)}$ in other DL literature, which is fine as long as you follow one convention for your entire network.)

  The biases are treated similarly to weights because they are randomly initialized, and their values are learned during the training process. So, for convenience, from this point forward we are going to represent the basis with the same notation that we gave for the weights (w). In DL literature, you will mostly find all weights and biases represented as (w) for simplicity.
- *Activation functions* $\sigma(x)$—In this example, we are using the sigmoid function $\sigma(x)$ as an activation function.
- *Node values (a)*—We will calculate the weighted sum, apply the activation function, and assign this value to the node a_m^n, where n is the layer number and m is the node index in the layer. For example, a_2^3 means node number 2 in layer 3.

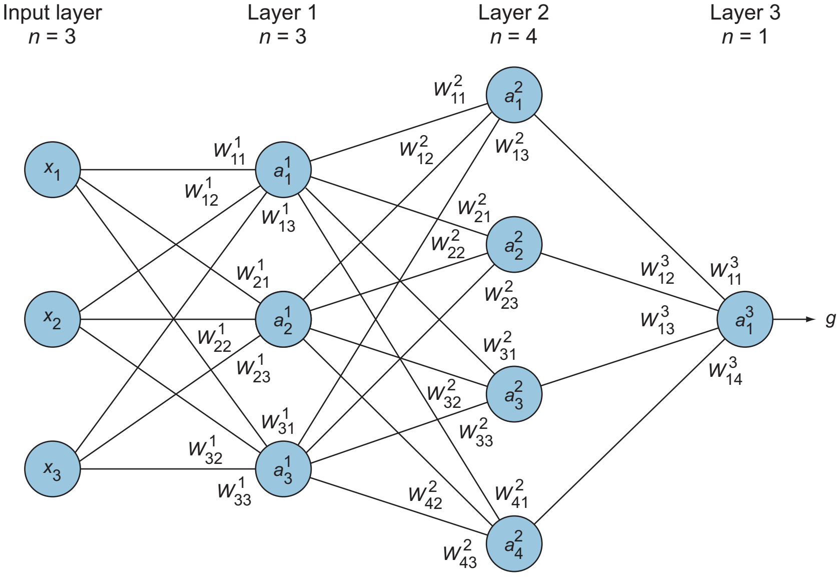


Figure 2.20 A simple three-layer neural network

2.4.1 Feedforward calculations

We have all we need to start the feedforward calculations:

$$a_1^{(1)} = \sigma(w_{11}^{(1)}\, x_1 + w_{21}^{(1)}\, x_2 + w_{31}^{(1)}\, x_3)$$

$$a_2^{(1)} = \sigma(w_{12}^{(1)}\, x_1 + w_{22}^{(1)}\, x_2 + w_{32}^{(1)}\, x_3)$$

$$a_3^{(1)} = \sigma(w_{13}^{(1)}\, x_1 + w_{23}^{(1)}\, x_2 + w_{33}^{(1)}\, x_3)$$

Then we do the same calculations for layer 2

$$a_1^{(2)}, a_2^{(2)}, a_3^{(2)}, \text{ and } a_4^{(2)}$$

all the way to the output prediction in layer 3:

$$\hat{y} = a_1^{(2)} = \sigma\ (w_{11}^{(3)}\, a_1^{(2)} + w_{12}^{(3)}\, a_2^{(2)} + w_{13}^{(3)}\, a_3^{(2)} + w_{14}^{(3)}\, a_4^{(2)}\ )$$

And there you have it! You just calculated the feedforward of a two-layer neural network. Let's take a moment to reflect on what we just did. Take a look at how many equations we need to solve for such a small network. What happens when we have a more complex problem with hundreds of nodes in the input layer and hundreds more in the hidden layers? It is more efficient to use matrices to pass through multiple inputs at once. Doing this allows for big computational speedups, especially when using tools like NumPy, where we can implement this with one line of code.

Let's see how the matrices computation looks (figure 2.21). All we did here is simply stack the inputs and weights in matrices and multiply them together. The intuitive way to read this equation is from the right to the left. Start at the far right and follow with me:

- We stack all the inputs together in one vector (row, column), in this case (3, 1).
- We multiply the input vector by the weights matrix from layer 1 ($W^{(1)}$) and then apply the sigmoid function.
- We multiply the result for layer 2 ⇒ $\sigma \cdot W^{(2)}$ and layer 3 ⇒ $\sigma \cdot W^{(3)}$.
- If we have a fourth layer, you multiply the result from step 3 by $\sigma \cdot W^{(4)}$, and so on, until we get the final prediction output $\hat{y}$!

Here is a simplified representation of this matrices formula:

$$\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)} \cdot (x)$$

$$\hat{y} = \underbrace{\sigma \overset{W^{(3)}}{\begin{bmatrix} W^3_{11} & W^3_{12} & W^3_{13} & W^3_{14} \end{bmatrix}}}_{\text{Layer 3}} \cdot \sigma \underbrace{\overset{W^{(2)}}{\begin{bmatrix} W^2_{11} & W^2_{12} & W^2_{13} \\ W^2_{21} & W^2_{22} & W^2_{23} \\ W^2_{31} & W^2_{32} & W^2_{33} \\ W^2_{41} & W^2_{42} & W^2_{43} \end{bmatrix}}}_{\text{Layer 2}} \cdot \sigma \underbrace{\overset{W^{(1)}}{\begin{bmatrix} W^1_{11} & W^1_{12} & W^1_{13} \\ W^1_{21} & W^1_{22} & W^1_{23} \\ W^1_{31} & W^1_{32} & W^1_{33} \end{bmatrix}}}_{\text{Layer 1}} \underbrace{\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}}_{\text{Input vector}}$$

Figure 2.21 Reading from left to right, we stack the inputs together in one vector, multiply the input vector by the weights matrix from layer 1, apply the sigmoid function, and multiply the result.

2.4.2 Feature learning

The nodes in the hidden layers (a_i) are the new features that are learned after each layer. For example, if you look at figure 2.20, you see that we have three feature inputs (x_1, x_2, and x_3). After computing the forward pass in the first layer, the network learns patterns, and these features are transformed to three new features with different values ($a_1^{(1)}, a_2^{(1)}, a_3^{(1)}$). Then, in the next layer, the network learns patterns within the patterns and produces new features ($a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$, and $a_4^{(2)}$, and so forth). The produced features after each layer are not totally understood, and we don't see them, nor do we have much control over them. It is part of the neural network magic. That's why they are called *hidden* layers. What we do is this: we look at the final output prediction and keep tuning some parameters until we are satisfied by the network's performance.

To reiterate, let's see this in a small example. In figure 2.22, you see a small neural network to estimate the price of a house based on three features: how many bedrooms it has, how big it is, and which neighborhood it is in. You can see that the original input feature values 3, 2000, and 1 were transformed into new feature values after performing the feedforward process in the first layer ($a_1^{(2)}, a_2^{(2)}, a_3^{(2)}, a_4^{(2)}$). Then they were transformed again to a prediction output value ($\hat{y}$). When training a neural network, we see the prediction output and compare it with the true price to calculate the error and repeat the process until we get the minimum error.

To help visualize the feature-learning process, let's take another look at figure 2.9 (repeated here in figure 2.23) from the Tensorflow playground. You can see that the first layer learns basic features like lines and edges. The second layer begins to learn more complex features like corners. The process continues until the last layers of the network learn even more complex feature shapes like circles and spirals that fit the dataset.

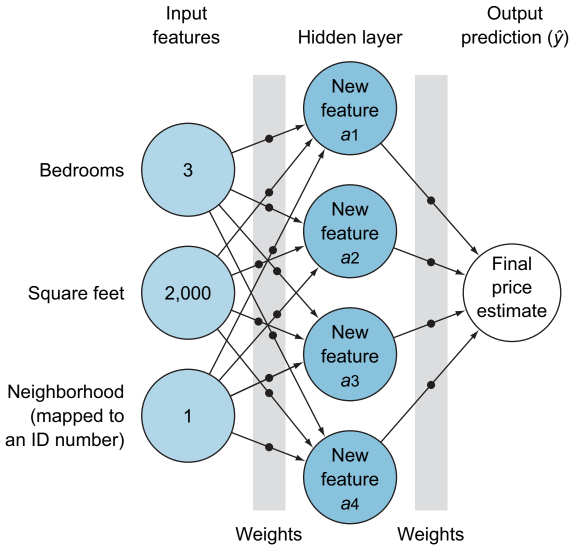


Figure 2.22 A small neural network to estimate the price of a house based on three features: how many bedrooms it has, how big it is, and which neighborhood it is in

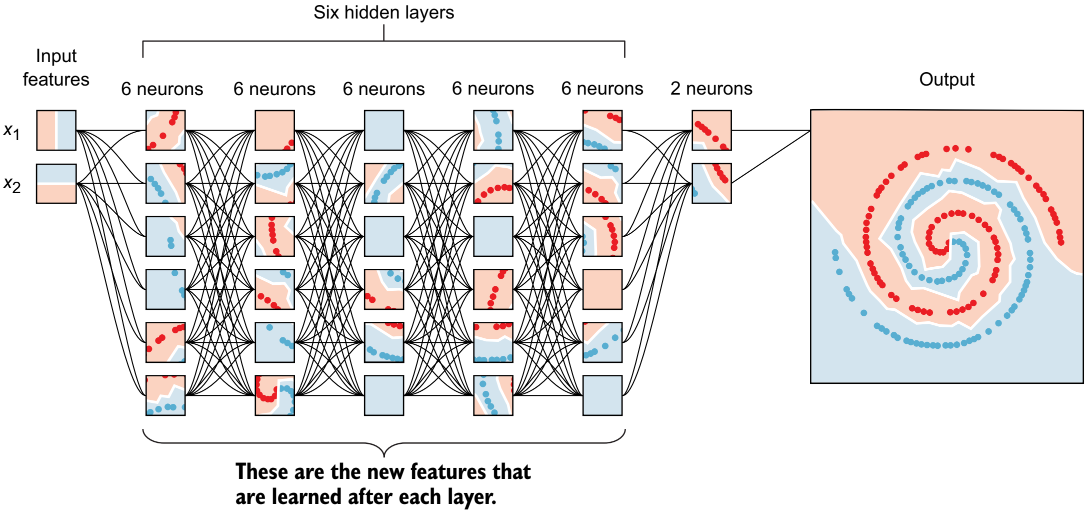


Figure 2.23 Learning features in multiple hidden layers

That is how a neural network learns new features: via the network's hidden layers. First, they recognize patterns in the data. Then, they recognize patterns within patterns; then patterns within patterns within patterns, and so on. The deeper the network is, the more it learns about the training data.

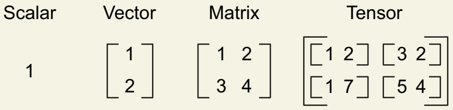
Vectors and matrices refresher

If you understood the matrix calculations we just did in the feedforward discussion, feel free to skip this sidebar. If you are still not convinced, hang tight: this sidebar is for you.

The feedforward calculations are a set of matrix multiplications. While you will not do these calculations by hand, because there are a lot of great DL libraries that do them for you with just one line of code, it is valuable to understand the mathematics that happens under the hood so you can debug your network. Especially because this is very trivial and interesting, let's quickly review matrix calculations.

Let's start with some basic definitions of matrix dimensions:

- A *scalar* is a single number.
- A *vector* is an array of numbers.
- A *matrix* is a 2D array.
- A *tensor* is an *n*-dimensional array with $n > 2$.



Matrix dimensions: a scalar is a single number, a vector is an array of numbers, a matrix is a 2D array, and a tensor is an *n*-dimensional array.

We will follow the conventions used in most mathematical literature:

- Scalars are written in lowercase and italics: for instance, *n*.
- Vectors are written in lowercase, italics, and bold type: for instance, ***x***.
- Matrices are written in uppercase, italics, and bold: for instance, ***X***.
- Matrix dimensions are written as follows: (row × column).

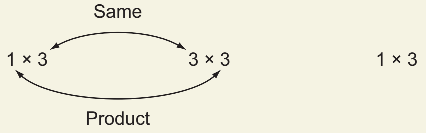
Multiplication:

- *Scalar multiplication*—Simply multiply the scalar number by all the numbers in the matrix. Note that scalar multiplications don't change the matrix dimensions:

$$2 \cdot \begin{bmatrix} 10 & 6 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 10 & 2 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 3 \end{bmatrix}$$

- *Matrix multiplication*—When multiplying two matrices, such as in the case of (row_1 × column_1) × (row_2 × column_2), column_1 and row_2 must be equal to each other, and the product will have the dimensions (row_1 × column_2). For example,

$$\begin{bmatrix} 3 & 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 13 & 9 & 7 \\ 8 & 7 & 4 \\ 6 & 4 & 0 \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix}$$



where $x = 3 \cdot 13 + 4 \cdot 8 + 2 \cdot 6 = 83$, and the same for $y = 63$ and $z = 37$.

Now that you know the matrices multiplication rules, pull out a piece of paper and work through the dimensions of matrices in the earlier neural network example. The following figure shows the matrix equation again for your convenience.

$$\hat{y} = \sigma \underbrace{\overset{W^{(3)}}{\begin{bmatrix} W^3_{11} & W^3_{12} & W^3_{13} & W^3_{14} \end{bmatrix}}}_{\text{Layer 3}} \cdot \sigma \underbrace{\overset{W^{(2)}}{\begin{bmatrix} W^2_{11} & W^2_{12} & W^2_{13} \\ W^2_{21} & W^2_{22} & W^2_{23} \\ W^2_{31} & W^2_{32} & W^2_{33} \\ W^2_{41} & W^2_{42} & W^2_{43} \end{bmatrix}}}_{\text{Layer 2}} \cdot \sigma \underbrace{\overset{W^{(1)}}{\begin{bmatrix} W^1_{11} & W^1_{12} & W^1_{13} \\ W^1_{21} & W^1_{22} & W^1_{23} \\ W^1_{31} & W^1_{32} & W^1_{33} \end{bmatrix}}}_{\text{Layer 1}} \underbrace{\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}}_{\text{Input vector}}$$

The matrix equation from the main text. Use it to work through matrix dimensions.

The last thing I want you to understand about matrices is *transposition*. With transposition, you can convert a row vector to a column vector and vice versa, where the shape ($m \times n$) is inverted and becomes ($n \times m$). The superscript (A^T) is used for transposed matrices:

$$A = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \quad \Rightarrow A^T = [2 \;\; 8]$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} \mathbf{1} & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 4 \\ 1 & -1 \end{bmatrix} \quad \Rightarrow A^T = \begin{bmatrix} 0 & 2 & 1 \\ \mathbf{1} & 4 & -1 \end{bmatrix}$$

2.5 Error functions

So far, you have learned how to implement the forward pass in neural networks to produce a prediction that consists of the weighted sum plus activation operations. Now, how do we evaluate the prediction that the network just produced? More importantly, how do we know how far this prediction is from the correct answer (the label)? The answer is this: measure the error. The selection of an error function is another important aspect of the design of a neural network. Error functions can also be referred to as *cost functions* or *loss functions*, and these terms are used interchangeably in DL literature.

2.5.1 What is the error function?

The *error function* is a measure of how "wrong" the neural network prediction is with respect to the expected output (the label). It quantifies how far we are from the correct solution. For example, if we have a high loss, then our model is not doing a good job. The smaller the loss, the better the job the model is doing. The larger the loss, the more our model needs to be trained to increase its accuracy.

2.5.2 Why do we need an error function?

Calculating error is an optimization problem, something all machine learning engineers love (mathematicians, too). Optimization problems focus on defining an error function and trying to optimize its parameters to get the minimum error (more on optimization in the next section). But for now, know that, in general, when we are working on an optimization problem, if we are able to define the error function for the problem, we have a very good shot at solving it by running optimization algorithms to minimize the error function.

In optimization problems, our ultimate goal is to find the optimum variables (weights) that would minimize the error function as much as we can. If we don't know how far from the target we are, how will we know what to change in the next iteration? The process of minimizing this error is called *error function optimization*. We will review several optimization methods in the next section. But for now, all we need to know from the error function is how far we are from the correct prediction, or how much we missed the desired degree of performance.

2.5.3 Error is always positive

Consider this scenario: suppose we have two data points that we are trying to get our network to predict correctly. If the first gives an error of 10 and the second gives an error of –10, then our average error is zero! This is misleading because "error = 0" means our network is producing perfect predictions, when, in fact, it missed by 10 twice. We don't want that. We want the error of each prediction to be positive, so the errors don't cancel each other when we take the average error. Think of an archer aiming at a target and missing by 1 inch. We are not really concerned about which direction they missed; all we need to know is how far each shot is from the target.

A visualization of loss functions of two separate models plotted over time is shown in figure 2.24. You can see that model #1 is doing a better job of minimizing error, whereas model #2 starts off better until epoch 6 and then plateaus.

Different loss functions will give different errors for the same prediction, and thus have a considerable effect on the performance of the model. A thorough discussion of loss functions is outside the scope of this book. Instead, we will focus on the two most commonly used loss functions: mean squared error (and its variations), usually used for regression problems, and cross-entropy, used for classification problems.

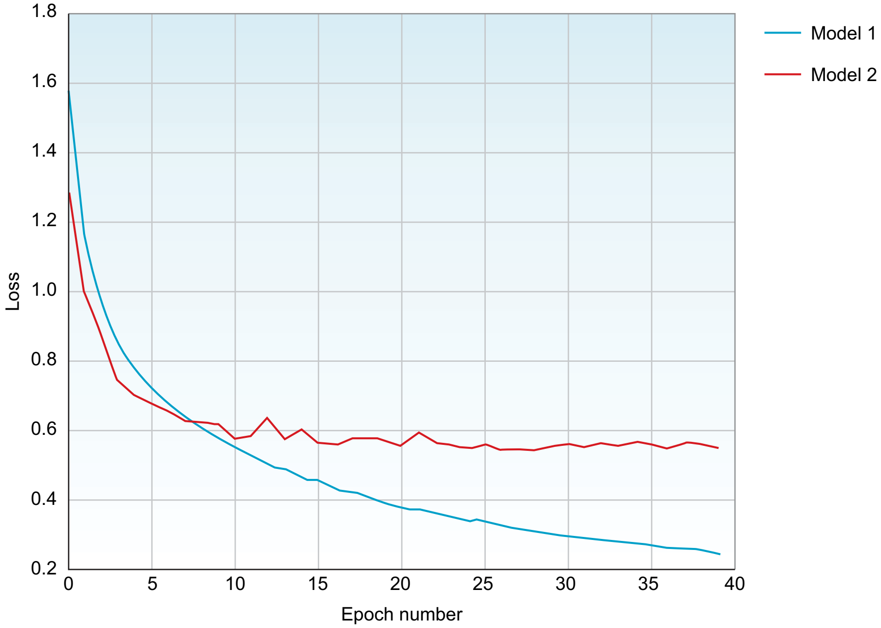


Figure 2.24 A visualization of the loss functions of two separate models plotted over time

2.5.4 Mean square error

Mean squared error (MSE) is commonly used in regression problems that require the output to be a real value (like house pricing). Instead of just comparing the prediction output with the label ($\hat{y}_i - y_i$), the error is squared and averaged over the number of data points, as you see in this equation:

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

MSE is a good choice for a few reasons. The square ensures the error is always positive, and larger errors are penalized more than smaller errors. Also, it makes the math nice, which is always a plus. The notations in the formula are listed in table 2.2.

MSE is quite sensitive to outliers, since it squares the error value. This might not be an issue for the specific problem that you are solving. In fact, this sensitivity to outliers might be beneficial in some cases. For example, if you are predicting a stock price, you would want to take outliers into account, and sensitivity to outliers would be a good thing. In other scenarios, you wouldn't want to build a model that is skewed by outliers, such as predicting a house price in a city. In that case, you are more interested in the median and less in the mean. A variation error function of MSE called

Table 2.2 Meanings of notation used in regression problems

Notation	Meaning
$E(W, b)$	The loss function. Is also annotated as $J(W, b)$ in other literature.
W	Weights matrix. In some literature, the weights are denoted by the theta sign (θ).
b	Biases vector.
N	Number of training examples.
$\hat{y}_i$	Prediction output. Also notated as $h_{w,b}(X)$ in some DL literature.
y_i	The correct output (the label).
$(\hat{y}_i - y_i)$	Usually called the *residual*.

mean absolute error (MAE) was developed just for this purpose. It averages the absolute error over the entire dataset without taking the square of the error:

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} |\hat{y}_i - y_i|$$

2.5.5 Cross-entropy

Cross-entropy is commonly used in classification problems because it quantifies the difference between two probability distributions. For example, suppose that for a specific training instance, we are trying to classify a dog image out of three possible classes (dogs, cats, fish). The true distribution for this training instance is as follows:

```
Probability(cat)      P(dog)          P(fish)
     0.0                1.0             0.0
```

We can interpret this “true” distribution to mean that the training instance has 0% probability of being class A, 100% probability of being class B, and 0% probability of being class C. Now, suppose our machine learning algorithm predicts the following probability distribution:

```
Probability(cat)      P(dog)          P(fish)
     0.2                 0.3              0.5
```

How close is the predicted distribution to the true distribution? That is what the cross-entropy loss function determines. We can use this formula:

$$E(W, b) = -\sum_{i=1}^{m} \hat{y}_i \log(p_i)$$

where (y) is the target probability, (p) is the predicted probability, and (m) is the number of classes. The sum is over the three classes: cat, dog, and fish. In this case, the loss is 1.2:

```
E = - (0.0 * log(0.2) + 1.0 * log(0.3) + 0.0 * log(0.5)) = 1.2
```

So that is how “wrong” or “far away” our prediction is from the true distribution.

Let’s do this one more time, just to show how the loss changes when the network makes better predictions. In the previous example, we showed the network an image of a dog, and it predicted that the image was 30% likely to be a dog, which was very far from the target prediction. In later iterations, the network learns some patterns and gets the predictions a little better, up to 50%:

```
Probability(cat)      P(dog)         P(fish)
     0.3               0.5              0.2
```

Then we calculate the loss again:

```
E = - (0.0*log(0.3) + 1.0*log(0.5) + 0.0*log(0.2)) = 0.69
```

You see that when the network makes a better prediction (dog is up to 50% from 30%), the loss decreases from 1.2 to 0.69. In the ideal case, when the network predicts that the image is 100% likely to be a dog, the cross-entropy loss will be 0 (feel free to try the math).

To calculate the cross-entropy error across all the training examples (n), we use this general formula:

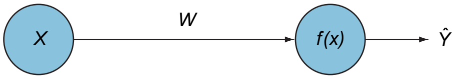
$$E(W, b) = -\sum_{i=1}^{n}\sum_{i=1}^{m} \hat{y}_{ij} \log(p_{ij})$$

NOTE It is important to note that you will not be doing these calculations by hand. Understanding how things work under the hood gives you better intuition when you are designing your neural network. In DL projects, we usually use libraries like Tensorflow, PyTorch, and Keras where the error function is generally a parameter choice.

2.5.6 A final note on errors and weights

As mentioned before, in order for the neural network to learn, it needs to minimize the error function as much as possible (0 is ideal). The lower the errors, the higher the accuracy of the model in predicting values. How do we minimize the error?

Let’s look at the following perceptron example with a single input to understand the relationship between the weight and the error:



Suppose the input $x = 0.3$, and its label (goal prediction) $y = 0.8$. The prediction output ($\hat{y}$) of this perception is calculated as follows:

$$\hat{y}_i = w \cdot x = w \cdot 0.3$$

And the error, in its simplest form, is calculated by comparing the prediction $\hat{y}$ and the label y:

$$\begin{aligned} \text{error} &= |\hat{y} - y| \\ &= |(w \cdot x) - y| \\ &= |w \cdot 0.3 - 0.8| \end{aligned}$$

If you look at this error function, you will notice that the input (x) and the goal prediction (y) are fixed values. They will never change for these specific data points. The only two variables that we can change in this equation are the error and the weight. Now, if we want to get to the minimum error, which variable can we play with? Correct: the weight! The weight acts as a knob that the network needs to adjust up and down until it gets the minimum error. This is how the network learns: by adjusting weight. When we plot the error function with respect to the weight, we get the graph shown in figure 2.25.

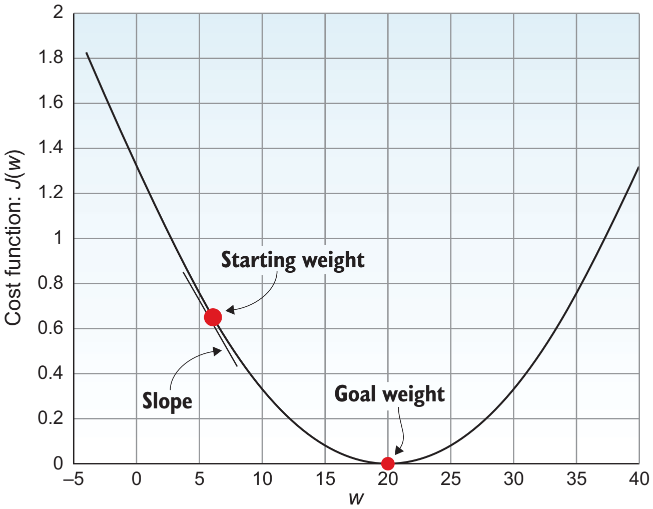


Figure 2.25 The network learns by adjusting weight. When we plot the error function with respect to weight, we get this type of graph.

As mentioned before, we initialize the network with random weights. The weight lies somewhere on this curve, and our mission is to make it *descend* this curve to its optimal value with the minimum error. The process of finding the goal weights of the neural network happens by adjusting the weight values in an iterative process using an *optimization algorithm*.

2.6 Optimization algorithms

Training a neural network involves showing the network many examples (a training dataset); the network makes predictions through feedforward calculations and compares them with the correct labels to calculate the error. Finally, the neural network needs to *adjust the weights* (on all edges) until it gets the minimum error value, which means maximum accuracy. Now, all we need to do is build algorithms that can find the optimum weights for us.

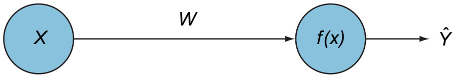
2.6.1 What is optimization?

Ahh, optimization! A topic that is dear to my heart, and dear to every machine learning engineer (mathematicians too). Optimization is a way of framing a problem to maximize or minimize some value. The best thing about computing an error function is that we turn the neural network into an optimization problem where our goal is to *minimize the error*.

Suppose you want to optimize your commute from home to work. First, you need to define the metric that you are optimizing (the error function). Maybe you want to optimize the cost of the commute, or the time, or the distance. Then, based on that specific loss function, you work on minimizing its value by changing some parameters. Changing the parameters to minimize (or maximize) a value is called *optimization*. If you choose the loss function to be the cost, maybe you will choose a longer commute that will take two hours, or (hypothetically) you might walk for five hours to minimize the cost. On the other hand, if you want to optimize the time spent commuting, maybe you will spend $50 to take a cab that will decrease the commute time to 20 minutes. Based on the loss function you defined, you can start changing your parameters to get the results you want.

TIP In neural networks, optimizing the error function means updating the weights and biases until we find the *optimal weights*, or the best values for the weights to produce the minimum error.

Let's look at the space that we are trying to optimize:



In a neural network of the simplest form, a perceptron with one input, we have only one weight. We can easily plot the error (that we are trying to minimize) with respect to this weight, represented by the 2D curve in figure 2.26 (repeated from earlier).

But what if we have two weights? If we graph all the possible values of the two weights, we get a 3D plane of the error (figure 2.27).

What about more than two weights? Your network will probably have hundreds or thousands of weights (because each edge in your network has its own weight value).

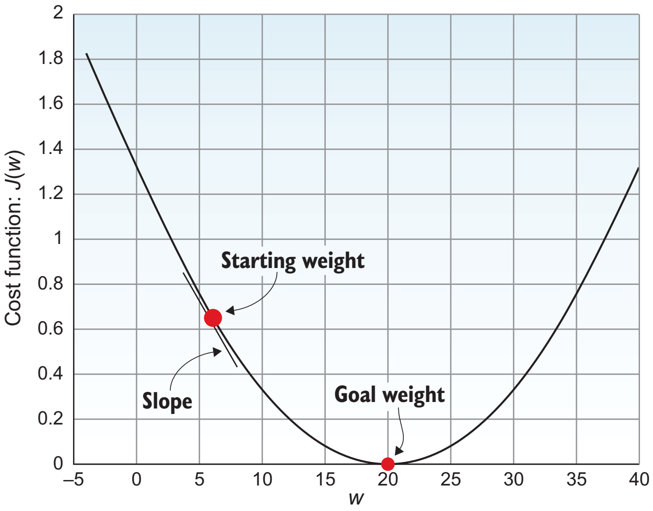


Figure 2.26 The error function with respect to its weight for a single perceptron is a 2D curve.

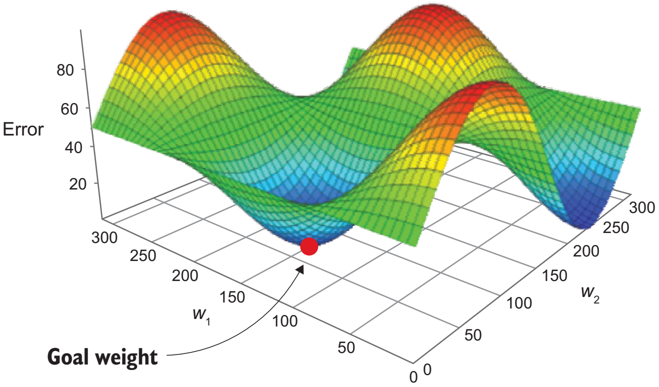


Figure 2.27 Graphing all possible values of two weights gives a 3D error plane.

Since we humans are only equipped to understand a maximum of 3 dimensions, it is impossible for us to visualize error graphs when we have 10 weights, not to mention hundreds or thousands of weight parameters. So, from this point on, we will study the error function using the 2D or 3D plane of the error. In order to optimize the model, our goal is to search this space to find the best weights that will achieve the lowest possible error.

Why do we need an optimization algorithm? Can't we just brute-force through a lot of weight values until we get the minimum error?

Suppose we used a brute-force approach where we just tried a lot of different possible weights (say 1,000 values) and found the weight that produced the minimum error. Could that work? Well, theoretically, yes. This approach might work when we have very few inputs and only one or two neurons in our network. Let me try to convince you that this approach wouldn't scale. Let's take a look at a scenario where we have a very simple neural network. Suppose we want to predict house prices based on only four features (inputs) and one hidden layer of five neurons (see figure 2.28).

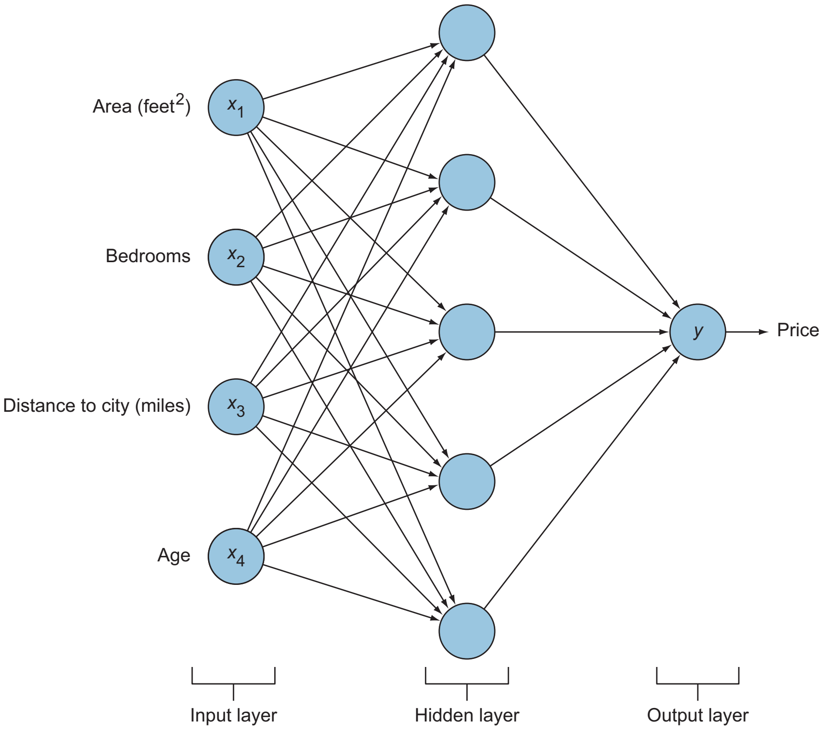


Figure 2.28 If we want to predict house prices based on only four features (inputs) and one hidden layer of five neurons, we'll have 20 edges (weights) from the input to the hidden layer, plus 5 weights from the hidden layer to the output prediction.

As you can see, we have 20 edges (weights) from the input to the hidden layer, plus 5 weights from the hidden layer to the output prediction, totaling 25 weight variables that need to be adjusted for optimum values. To brute-force our way through a simple neural network of this size, if we are trying 1,000 different values for each weight, then we will have a total of 10^{75} combinations:

$$1{,}000 \times 1{,}000 \times \ldots \times 1{,}000 = 1{,}000^{25} = 10^{75} \text{ combinations}$$

Let's say we were able to get our hands on the fastest supercomputer in the world: Sunway TaihuLight, which operates at a speed of 93 petaflops ⇒ 93×10^{15} floating-point operations per second (FLOPs). In the best-case scenario, this supercomputer would need

$$\frac{10^{75}}{93 \times 10^{15}} = 1.08 \times 10^{58} \text{ seconds} = 3.42 \times 10^{50} \text{ years}$$

That is a huge number: it's longer than the universe has existed. Who has that kind of time to wait for the network to train? Remember that this is a very simple neural network that usually takes a few minutes to train using smart optimization algorithms. In the real world, you will be building more complex networks that have thousands of inputs and tens of hidden layers, and you will be required to train them in a matter of hours (or days, or sometimes weeks). So we have to come up with a different approach to find the optimal weights.

Hopefully I have convinced you that brute-forcing through the optimization process is not the answer. Now, let's study the most popular optimization algorithm for neural networks: gradient descent. Gradient descent has several variations: batch gradient descent (BGD), stochastic gradient descent (SGD), and mini-batch GD (MB-GD).

2.6.2 Batch gradient descent

The general definition of a *gradient* (also known as a *derivative*) is that it is the function that tells you the slope or rate of change of the line that is tangent to the curve at any given point. It is just a fancy term for the slope or steepness of the curve (figure 2.29).

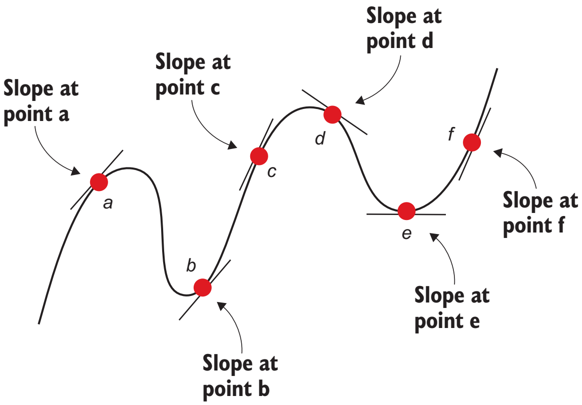


Figure 2.29 A gradient is the function that describes the rate of change of the line that is tangent to a curve at any given point.

Gradient descent simply means updating the weights iteratively to descend the slope of the error curve until we get to the point with minimum error. Let's take a look at the error function that we introduced earlier with respect to the weights. At the initial weight point, we calculate the derivative of the error function to get the slope (direction) of the next step. We keep repeating this process to take steps down the curve until we reach the minimum error (figure 2.30).

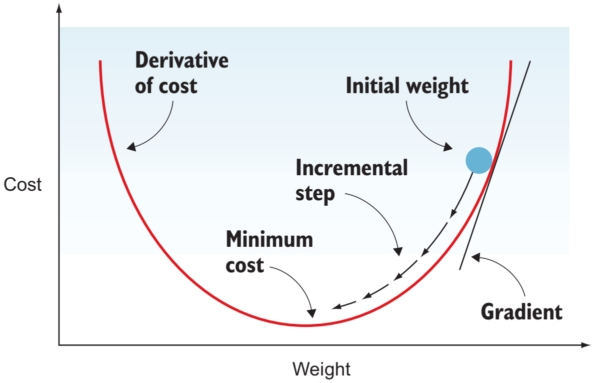


Figure 2.30 Gradient descent takes incremental steps to descend the error function.

How does gradient descent work?

To visualize how gradient descent works, let's plot the error function in a 3D graph (figure 2.31) and go through the process step by step. The random initial weight (starting weight) is at point A, and our goal is to descend this error mountain to the goal w_1 and w_2 weight values, which produce the minimum error value. The way we do that is by taking a series of steps *down* the curve until we get the minimum error. In order to descend the error mountain, we need to determine two things for each step:

- The step direction (gradient)
- The step size (learning rate)

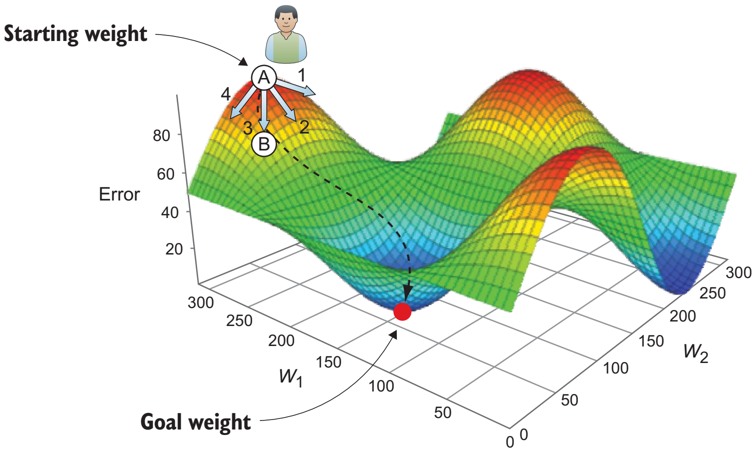


Figure 2.31 The random initial weight (starting weight) is at point A. We descend the error mountain to the w_1 and w_2 weight values that produce the minimum error value.

The direction (gradient)

Suppose you are standing on the top of the error mountain at point A. To get to the bottom, you need to determine the step direction that results in the deepest descent (has the steepest slope). And what is the slope, again? It is the derivative of the curve. So if you are standing on top of that mountain, you need to look at all the directions around you and find out which direction will result in the deepest descent (1, 2, 3, or 4, for example). Let's say it is direction 3; we choose that way. This brings us to point B, and we restart the process (calculate feedforward and error) and find the direction of deepest descent, and so forth, until we get to the bottom of the mountain.

This process is called *gradient descent*. By taking the derivative of the error with respect to the weight ($\frac{dE}{dw}$), we get the direction that we should take. Now there's one thing left. The gradient only determines the direction. How large should the size of the step be? It could be a 1-foot step or a 100-foot jump. This is what we need to determine next.

The step size (learning rate α)

The *learning rate* is the size of each step the network takes when it descends the error mountain, and it is usually denoted by the Greek letter alpha (α). It is one of the most important hyperparameters that you tune when you train your neural network (more on that later). A larger learning rate means the network will learn faster (since it is descending the mountain with larger steps), and smaller steps mean slower learning. Well, this sounds simple enough. Let's use large learning rates and complete the neural network training in minutes instead of waiting for hours. Right? Not quite. Let's take a look at what could happen if we set a very large learning rate value.

In figure 2.32, you are starting at point A. When you take a large step in the direction of the arrow, instead of descending the error mountain, you end up at point B, on the other side. Then another large step takes you to C, and so forth. The error will keep *oscillating* and will never descend. We will talk more later about tuning the learning rate and how to determine if the error is oscillating. But for now, you need to know this: if you use a very small learning rate, the network will eventually descend the

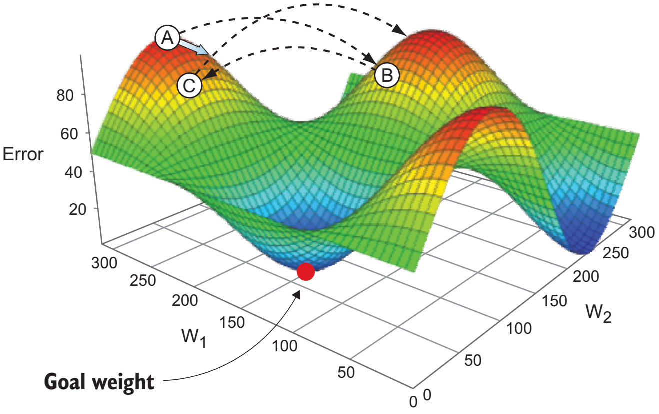


Figure 2.32 Setting a very large learning rate causes the error to oscillate and never descend.

mountain and will get to the minimum error. But this training will take longer (maybe weeks or months). On the other hand, if you use a very large learning rate, the network might keep oscillating and never train. So we usually initialize the learning rate value to 0.1 or 0.01 and see how the network performs, and then tune it further.

PUTTING DIRECTION AND STEP TOGETHER

By multiplying the direction (derivative) by the step size (learning rate), we get the change of the weight for each step:

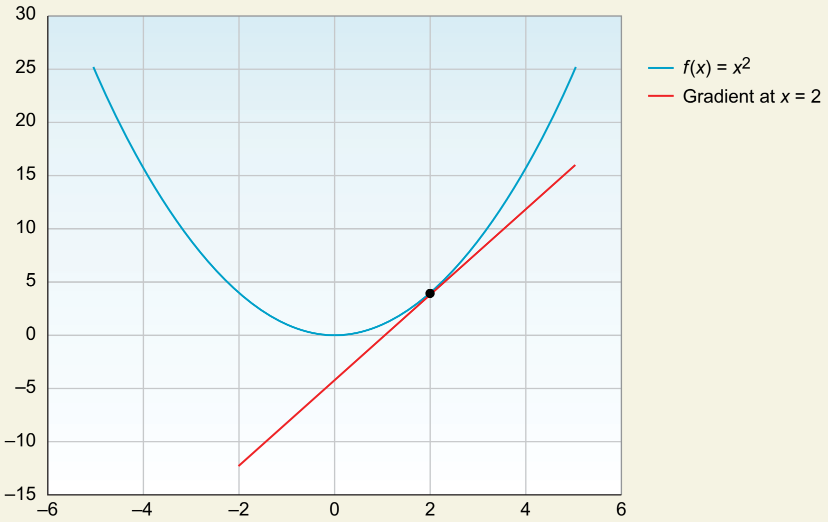
$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

We add the minus sign because the derivative always calculates the slope in the upward direction. Since we need to descend the mountain, we go in the opposite direction of the slope:

$$w_{\text{next-step}} = w_{\text{current}} + \Delta w$$

Calculus refresher: Calculating the partial derivative

The derivative is the study of change. It measures the steepness of a curve at a particular point on a graph.



We want to find the steepness of the curve at the exact weight point.

It looks like mathematics has given us just what we are looking for. On the error graph, we want to find the steepness of the curve at the exact weight point. Thank you, math!

Other terms for *derivative* are *slope* and *rate of change*. If the error function is denoted as $E(x)$, then the derivative of the error function *with respect to the weight* is denoted as

$$\frac{d}{dw}E(x) \quad \text{or just} \quad \frac{dE(x)}{dw}$$

This formula shows how much the total error will change when we change the weight.

Luckily, mathematicians created some rules for us to calculate the derivative. Since this is not a mathematics book, we will not discuss the proof of the rules. Instead, we will start applying these rules at this point to calculate our gradient. Here are the basic derivative rules:

Constant Rule: $\frac{d}{dx}(c) = 0$	Difference Rule: $\frac{d}{dx}[f(x) - g(x)] - f'(x) - g'(x)$
Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$	Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Power Rule: $\frac{d}{dx}(x^n) = x^{n-1}$	Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Sum Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$	Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

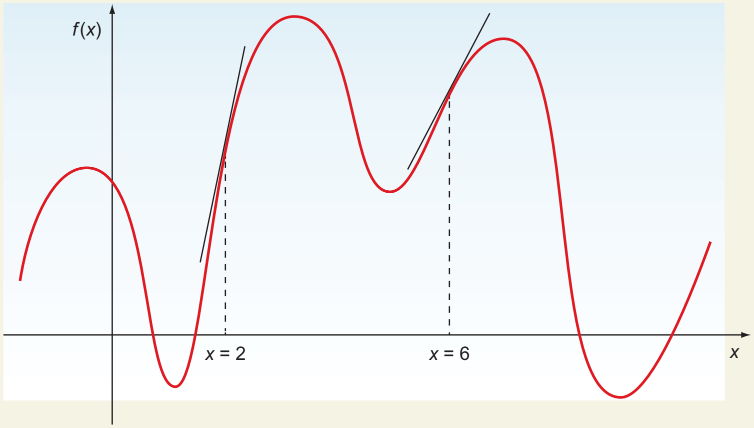
Let's take a look at a simple function to apply the derivative rules:

$$f(x) = 10x^5 + 4x^7 + 12x$$

We can apply the power, constant, and sum rules to get $\frac{df}{dx}$ also denoted as $f'(x)$:

$$\text{then, } f'(x) = 50x^4 + 28x^6 + 12$$

To get an intuition of what this means, let's plot $f(x)$:



Using a simple function to apply derivative rules. To get the slope at any point, we can compute $f'(x)$ at that point.

(continued)

If we want to get the slope at any point, we can compute $f'(x)$ at that point. So $f'(2)$ gives us the slope of the line on the left, and $f'(6)$ gives the slope of the second line. Get it?

For a last example of derivatives, let's apply the power rule to calculate the derivative of the sigmoid function:

$$
\begin{aligned}
\frac{d}{dx}\,\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\,(1+e^{-x}) \quad \text{power rule} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^{2}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \sigma(x)\cdot(1-\sigma(x))
\end{aligned}
$$

If you want to write out the derivative of the sigmoid activation function in code, it will look like this:

```
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Note that you don't need to memorize the derivative rules, nor do you need to calculate the derivatives of the functions yourself. Thanks to the awesome DL community, we have great libraries that will compute these functions for you in just one line of code. But it is valuable to understand how things are happening under the hood.

PITFALLS OF BATCH GRADIENT DESCENT

Gradient descent is a very powerful algorithm to get to the minimum error. But it has two major pitfalls.

First, not all cost functions look like the simple bowls we saw earlier. There may be holes, ridges, and all sorts of irregular terrain that make reaching the minimum error very difficult. Consider figure 2.33, where the error function is a little more complex and has ups and downs.

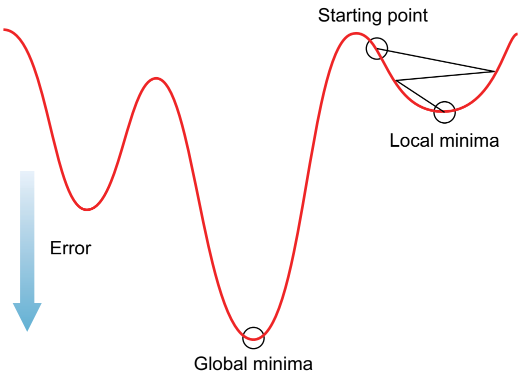


Figure 2.33 Complex error functions are represented by more complex curves with many local minima values. Our goal is to reach the global minimum value.

Remember that during weight initialization, the starting point is randomly selected. What if the starting point of the gradient descent algorithm is as shown in this figure? The error will start descending the small mountain on the right and will indeed reach a minimum value. But this minimum value, called the *local minima*, is not the lowest possible error value for this error function. It is the minimum value for the local mountain where the algorithm randomly started. Instead, we want to get to the lowest possible error value, the *global minima*.

Second, batch gradient descent uses the entire training set to compute the gradients at every step. Remember this loss function?

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

This means that if your training set (N) has 100,000,000 (100 million) records, the algorithm needs to sum over 100 million records just to take *one step*. That is computationally very expensive and slow. And this is why this algorithm is also called *batch gradient descent*—because it uses the entire training data in one batch.

One possible approach to solving these two problems is stochastic gradient descent. We'll take a look at SGD in the next section.

2.6.3 Stochastic gradient descent

In stochastic gradient descent, the algorithm randomly selects data points and goes through the gradient descent one data point at a time (figure 2.34). This provides many different weight starting points and descends all the mountains to calculate their local minimas. Then the minimum value of all these local minimas is the global minima. This sounds very intuitive; that is the concept behind the SGD algorithm.

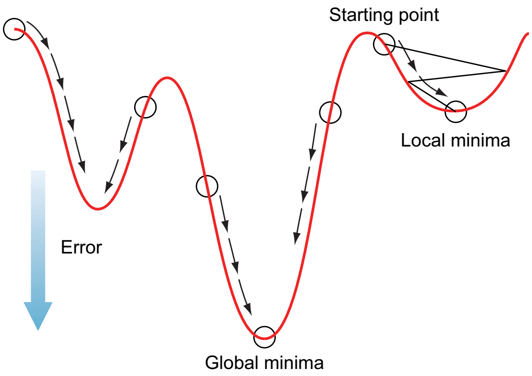
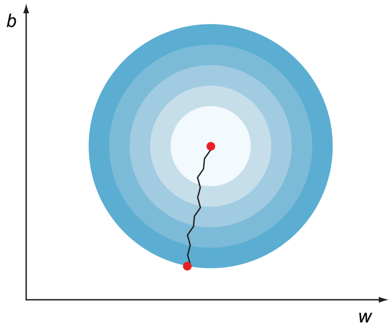
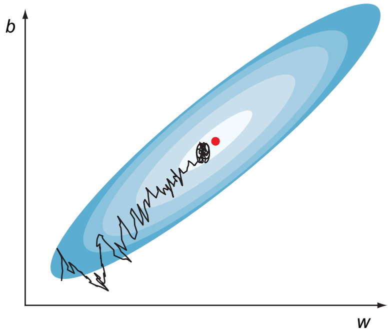


Figure 2.34 The stochastic gradient descent algorithm randomly selects data points across the curve and descends all of them to find the local minima.

Stochastic is just a fancy word for *random*. Stochastic gradient descent is probably the most-used optimization algorithm for machine learning in general and for deep learning in particular. While gradient descent measures the loss and gradient over the full training set to take one step toward the minimum, SGD *randomly* picks *one instance* in the training set for each one step and calculates the gradient based only on that single instance. Let's take a look at the pseudocode of both GD and SGD to get a better understanding of the differences between these algorithms:

GD	Stochastic GD
1 Take all the data. 2 Compute the gradient. 3 Update the weights and take a step down.  4 Repeat for *n* number of epochs (iterations). A smooth path for the GD down the error curve	1 Randomly shuffle samples in the training set. 2 Pick one data instance. 3 Compute the gradient. 4 Update the weights and take a step down. 5 Pick another one data instance. 6 Repeat for *n* number of epochs (training iterations).  An oscillated path for SGD down the error curve

Because we take a step after we compute the gradient for the entire training data in batch GD, you can see that the path down the error is smooth and almost a straight line. In contrast, due to the stochastic (random) nature of SGD, you see the path toward the global cost minimum is not direct but may zigzag if we visualize the cost surface in a 2D space. That is because in SGD, every iteration tries to better fit just a single training example, which makes it a lot faster but does not guarantee that every step takes us a step down the curve. It will arrive close to the global minimum and, once it gets there, it will continue to bounce around, never settling down. In practice, this isn't a problem because ending up very close to the global minimum is good enough for most practical purposes. SGD almost always performs better and faster than batch GD.

2.6.4 Mini-batch gradient descent

Mini-batch gradient descent (MB-GD) is a compromise between BGD and SGD. Instead of computing the gradient from one sample (SGD) or all samples (BGD), we divide the training sample into *mini-batches* from which to compute the gradient (a common mini-batch size is $k = 256$). MB-GD converges in fewer iterations than BGD because we update the weights more frequently; however, MB-GD lets us use vectorized operations, which typically result in a computational performance gain over SGD.

2.6.5 Gradient descent takeaways

There is a lot going on here, so let's sum it up, shall we? Here is how gradient descent is summarized in my head:

- Three types: batch, stochastic, and mini-batch.
- All follow the same concept:
  - Find the direction of the steepest slope: the derivative of the error with respect to the weight $\frac{dE}{dw_i}$.
  - Set the learning rate (or step size). The algorithm will compute the slope, but you will set the learning rate as a hyperparameter that you will tune by trial and error.
  - Start the learning rate at 0.01, and then go down to 0.001, 0.0001, 0.00001. The lower you set your learning rate, the more guaranteed you are to descend to the minimum error (if you train for an infinite time). Since we don't have infinite time, 0.01 is a reasonable start, and then we go down from there.
- Batch GD updates the weights after computing the gradient of *all* the training data. This can be computationally very expensive when the data is huge. It doesn't scale well.
- Stochastic GD updates the weights after computing the gradient of a single instance of the training data. SGD is faster than BGD and usually reaches very close to the global minimum.
- Mini-batch GD is a compromise between batch and stochastic, using neither all the data nor a single instance. Instead, it takes a group of training instances (called a mini-batch), computes the gradient on them and updates the weights, and then repeats until it processes all the training data. In most cases, MB-GD is a good starting point.
  - `batch_size` is a hyperparameter that you will tune. This will come up again in the hyperparameter-tuning section in chapter 4. But typically, you can start experimenting with batch_size = 32, 64, 128, 256.
  - Don't get *batch_size* confused with *epochs*. An *epoch* is the full cycle over all the training data. The batch is the number of training samples in the group for which we are computing the gradient. For example, if we have 1,000 samples in our training data and set batch_size = 256, then epoch 1 = batch 1 of 256 samples plus batch 2 (256 samples) plus batch 3 (256 samples) plus batch 4 (232 samples).

Finally, you need to know that a lot of variations to gradient descent have been used over the years, and this is a very active area of research. Some of the most popular enhancements are

- Nesterov accelerated gradient
- RMSprop

- Adam
- Adagrad

Don't worry about these optimizers now. In chapter 4, we will discuss tuning techniques to improve your optimizers in more detail.

I know that was a lot, but stay with me. These are the main things I want to you remember from this section:

- How gradient descent works (slope plus step size)
- The difference between batch, stochastic, and mini-batch GD
- The GD hyperparameters that you will tune: learning rate and batch_size

If you've got this covered, you are good to move to the next section. And don't worry a lot about hyperparameter tuning. I'll cover network tuning in more detail in coming chapters and in almost all the projects in this book.

2.7 Backpropagation

Backpropagation is the core of how neural networks learn. Up until this point, you learned that training a neural network typically happens by the repetition of the following three steps:

- Feedforward: get the linear combination (weighted sum), and apply the activation function to get the output prediction ($\hat{y}$):

$$\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)} \cdot (x)$$

- Compare the prediction with the label to calculate the error or loss function:

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} |\hat{y}_i - y_i|$$

- Use a gradient descent optimization algorithm to compute the Δw that optimizes the error function:

$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

Backpropagate the Δw through the network to update the weights:

$$\underbrace{W_{new}}_{\textbf{New weight}} = \overbrace{W_{old}}^{\textbf{Old weight}} - \underbrace{\alpha}_{\textbf{Learning rate}} \left(\overbrace{\frac{\partial Error}{\partial W_x}}^{\textbf{Derivative of error with respect to weight}} \right)$$

In this section, we will dive deeper into the final step: backpropagation.

2.7.1 What is backpropagation?

Backpropagation, or *backward pass*, means propagating derivatives of the error with respect to each specific weight

$$\frac{dE}{dw_i}$$

from the last layer (output) back to the first layer (inputs) to adjust weights. By propagating the Δw backward from the prediction node ($\hat{y}$) all the way through the hidden layers and back to the input layer, the weights get updated:

$$(w_{\text{next–step}} = w_{\text{current}} + \Delta w)$$

This will take the error one step down the error mountain. Then the cycle starts again (steps 1 to 3) to update the weights and take the error another step down, until we get to the minimum error.

Backpropagation might sound clearer when we have only one weight. We simply adjust the weight by adding the Δw to the old weight $w_{\text{new}} = w - \alpha \frac{dE}{dw_i}$.

But it gets complicated when we have a multilayer perceptron (MLP) network with many weight variables. To make this clearer, consider the scenario in figure 2.35.

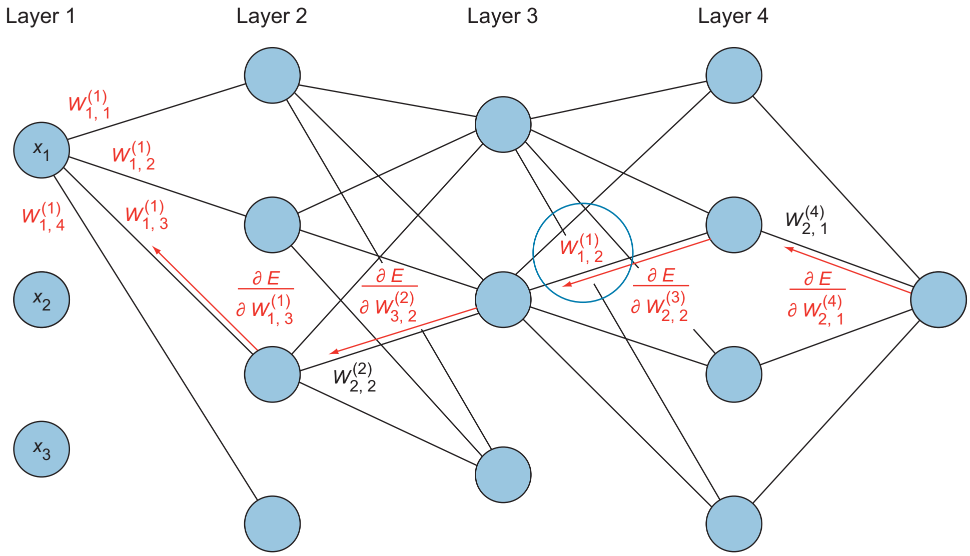

Layer 1
Layer 2
Layer 3
Layer 4
$W^{(1)}_{1,1}$
x_1
$W^{(1)}_{1,2}$
$W^{(1)}_{1,4}$
$W^{(1)}_{1,3}$
x_2
x_3
$W^{(4)}_{2,1}$
$W^{(1)}_{1,2}$
$\frac{\partial E}{\partial W^{(1)}_{1,3}}$
$\frac{\partial E}{\partial W^{(2)}_{3,2}}$
$\frac{\partial E}{\partial W^{(3)}_{2,2}}$
$\frac{\partial E}{\partial W^{(4)}_{2,1}}$
$W^{(2)}_{2,2}$

Figure 2.35 Backpropagation becomes complicated when we have a multilayer perceptron (MLP) network with many weight variables.

How do we compute the change of the total error with respect to $w_{13} \frac{dE}{dw_{13}}$? Remember that $\frac{dE}{dw_{13}}$ basically says, "How much will the total error change when we change the parameter w_{13}?"

We learned how to compute $\frac{dE}{dw_{21}}$ by applying the derivative rules on the error function. That is straightforward because w_{21} is directly connected to the error function. But to compute the derivatives of the total error with respect to the weights all the way back to the input, we need a calculus rule called *the chain rule*.

Calculus refresher: Chain rule in derivatives

Back again to calculus. Remember the derivative rules that we listed earlier? One of the most important rules is the chain rule. Let's dive deep into it to see how it is implemented in backpropagation:

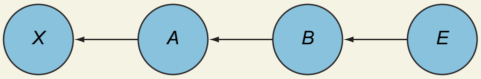
Chain Rule: $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$

The chain rule is a formula for calculating the derivatives of functions that are composed of functions inside other functions. It is also called the *outside-inside rule*. Look at this:

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx} \text{ outside function} \times \frac{d}{dx} \text{ inside function}$$
$$= \frac{d}{dx} f(g(x)) \times \frac{d}{dx} g(x)$$

The chain rule says, "When composing functions, the derivatives just multiply." That is going to be very useful for us when implementing backpropagation, because feedforwarding is just composing a bunch of functions, and backpropagation is taking the derivative at each piece of this function.

To implement the chain rule in backpropagation, all we are going to do is multiply a bunch of partial derivatives to get the effect of errors all the way back to the input. Here is how it works—but first, remember that our goal is to propagate the error backward all the way to the input layer. So in the following example, we want to calculate $\frac{dE}{dx}$, which is the effect of total error on input (x):



$$\frac{dE}{dx} = \frac{dE}{dB} \cdot \frac{dB}{dA} \cdot \frac{dA}{dx}$$

All we do here is multiply the upstream gradient by the local gradient all the way until we get to the target value.

Figure 2.36 shows how backpropagation uses the chain rule to flow the gradients in the backward direction through the network. Let's apply the chain rule to calculate the derivative of the error with respect to the third weight on the first input $w_{1,3}^{(1)}$, where the (1) means layer 1, and $w_{1,3}$ means node number 1 and weight number 3:

$$\frac{dE}{dw_{1,3}^{(1)}} = \frac{dE}{dw_{2,1}^{(4)}} \times \frac{dw_{2,1}^{(4)}}{dw_{2,2}^{(3)}} \times \frac{dw_{2,2}^{(3)}}{dw_{3,2}^{(2)}} \times \frac{dw_{3,2}^{(2)}}{dw_{1,3}^{(1)}}$$

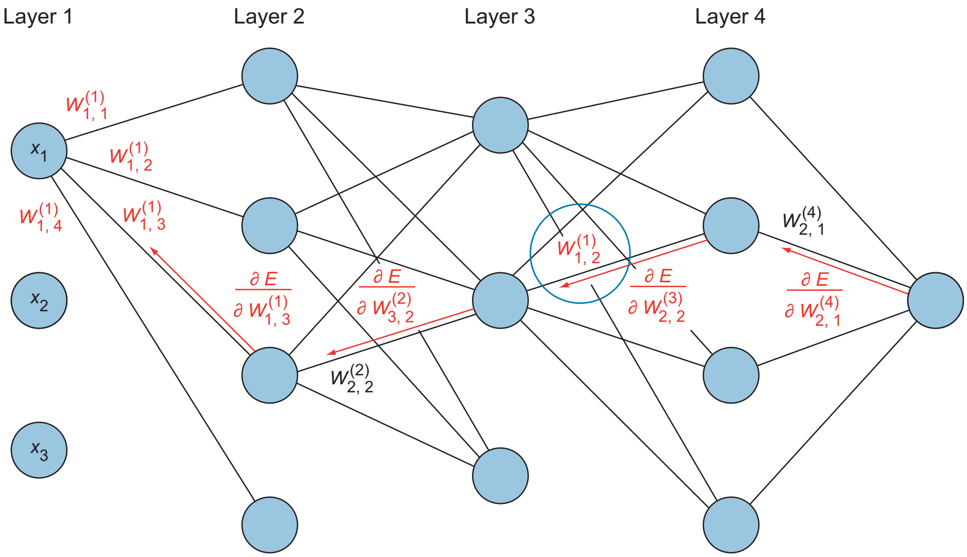

Layer 1
Layer 2
Layer 3
Layer 4
x_1
x_2
x_3
$W_{1,1}^{(1)}$
$W_{1,2}^{(1)}$
$W_{1,3}^{(1)}$
$W_{1,4}^{(1)}$
$\frac{\partial E}{\partial W_{1,3}^{(1)}}$
$\frac{\partial E}{\partial W_{3,2}^{(2)}}$
$W_{2,2}^{(2)}$
$W_{1,2}^{(1)}$
$\frac{\partial E}{\partial W_{2,2}^{(3)}}$
$W_{2,1}^{(4)}$
$\frac{\partial E}{\partial W_{2,1}^{(4)}}$

Figure 2.36 Backpropagation uses the chain rule to flow gradients back through the network.

The equation might look complex at the beginning, but all we are doing really is multiplying the partial derivative of the edges starting from the output node all the way backward to the input node. All the notations are what makes this look complex, but once you understand how to read $w_{1,3}^{(1)}$, the backward-pass equation looks like this:

The error backpropagated to the edge $w_{1,3}^{(1)}$ = effect of error on edge 4 · effect on edge 3 · effect on edge 2 · effect on target edge

There you have it. That is the backpropagation technique used by neural networks to update the weights to best fit our problem.

2.7.2 Backpropagation takeaways

- Backpropagation is a learning procedure for neurons.
- Backpropagation repeatedly adjusts weights of the connections (weights) in the network to minimize the cost function (the difference between the actual output vector and the desired output vector).
- As a result of the weight adjustments, hidden layers come to represent important features other than the features represented in the input layer.
- For each layer, the goal is to find a set of weights that ensures that for each input vector, the output vector produced is the same as (or close to) the desired output vector. The difference in values between the produced and desired outputs is called the error function.
- The backward pass (backpropagation; figure 2.37) starts at the end of the network, backpropagates or feeds the errors back, recursively applies the chain rule to compute gradients all the way to the inputs of the network, and then updates the weights.
- To reiterate, the goal of a typical neural network problem is to discover a model that best fits our data Ultimately, we want to minimize the cost or loss function by choosing the best set of weight parameters.

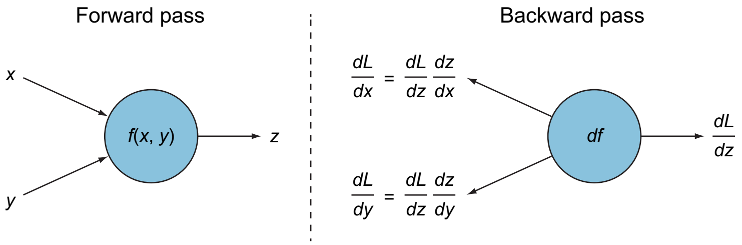

Forward pass
x
y
f(x, y)
z
Backward pass
$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx}$
$\frac{dL}{dy} = \frac{dL}{dz}\frac{dz}{dy}$
df
$\frac{dL}{dz}$

Figure 2.37 The forward pass calculates the output prediction (left). The backward pass passes the derivative of the error backward to update its weights (right).

Summary

- Perceptrons work fine for datasets that can be separated by one straight line (linear operation).
- Nonlinear datasets that cannot be modeled by a straight line need a more complex neural network that contains many neurons. Stacking neurons in layers creates a multilayer perceptron.
- The network learns by the repetition of three main steps: feedforward, calculate error, and optimize weights.

- Parameters are variables that are updated by the network during the training process, like weights and biases. These are tuned automatically by the model during training.
- Hyperparameters are variables that you tune, such as number of layers, activation functions, loss functions, optimizers, early stopping, and learning rate. We tune these before training the model.