Chapter 2. Foundations of Neural Networks and Deep Learning

With your feet in the air and your head on the ground
Try this trick and spin it, yeah
Your head will collapse
But there’s nothing in it
And you’ll ask yourself
Where is my mind
The Pixies, “Where is My Mind?”

Neural Networks

Neural networks are a computational model that shares some properties with the animal brain in which many simple units are working in parallel with no centralized control unit. The weights between the units are the primary means of long-term information storage in neural networks. Updating the weights is the primary way the neural network learns new information.

In Chapter 1 we discussed modeling sets of equations in the form of the equation $Ax = b$. In the context of neural networks, the A matrix is still the input data and the b column vector is still the labels or outcomes for each row in the A matrix. The weights on the neural network connections becomes x (the parameter vector).

The behavior of neural networks is shaped by its network architecture. A network's architecture can be defined (in part) by the following:

- Number of neurons
- Number of layers
- Types of connections between layers

The most well-known and simplest-to-understand neural network is the feed-forward multilayer neural network. It has an input layer, one or many hidden layers, and a single output layer. Each layer can have a different number of neurons and each layer is fully connected to the adjacent layer. The connections between the neurons in the layers form an acyclic graph, as illustrated in Figure 2-1.

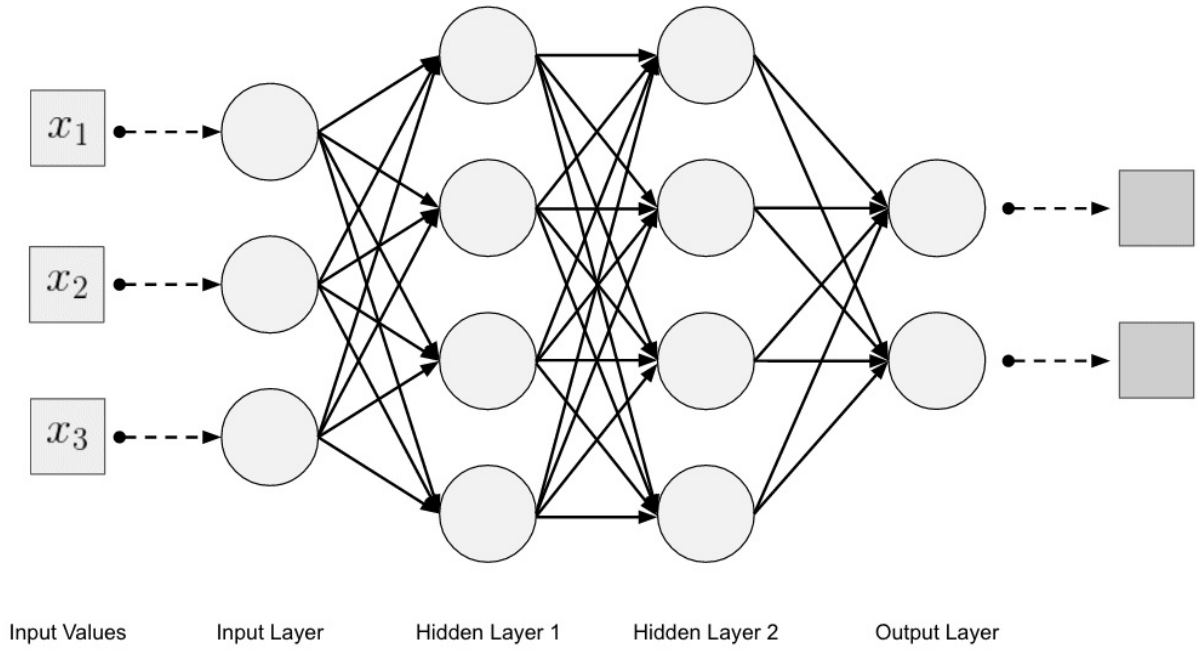

Figure 2-1. Multilayer neural network topology

A feed-forward multilayer neural network can represent any function, given enough artificial neuron units. It is generally trained by a learning algorithm called *backpropagation learning*. Backpropagation uses gradient descent (see Chapter 1) on the weights of the connections in a neural network to minimize the error on the output of the network.

LOCAL MINIMA AND BACKPROPAGATION

Backpropagation can become stuck in local minima, but in practice it generally performs well.

Historically, backpropagation has been considered slow, but recent advances in computational power through parallelism and graphics processing units (GPUs) have renewed interest in neural networks.

Much hubris and many words have been transmitted across the internet and in the literature about the connection of neural networks to the human mind. Let’s separate some signal from the vast noise and begin with the biological inspiration of artificial neural networks.

THE MECHANISTIC VIEW OF THE MIND

Rather than constructing a rigid tree that requires all inputs to be one thing or another, we can construct a model that reflects a world sending us partial, ambiguous information, from which we can draw inferences with relative but not total certainty.

This aspect of neural networks represents a break from the mechanistic view of the mind, dominant in the early twentieth century, which assumed that our brain interlocked with the world in a deterministic way — like two gears meshing — with clear inputs leading to clear outputs. Now, we assume that, based on incomplete and sometimes contradictory information, humans find ways to plunge forward and act. The human brain infers from probabilities and so do neural networks.

We'll provide a brief review of the biological neuron and then take a look at the early precursor to modern neural networks: *the perceptron*. Building on our understanding of the perceptron, we'll then see how it evolved into the more generalized artificial neuron that supports today's modern feed-forward multilayer perceptrons. The culmination of this chapter will give you, the modern neural network practitioner, the fundamentals to delve further into more exotic deep network architectures.

The Biological Neuron

The biological neuron (see Figure 2-2) is a nerve cell that provides the fundamental functional unit for the nervous systems of all animals. Neurons exist to communicate with one another, and pass electro-chemical impulses across synapses, from one cell to the next, as long as the impulse is strong enough to activate the release of chemicals across a synaptic cleft. The strength of the impulse must surpass a minimum threshold or chemicals will not be released.

Figure 2-2 presents the major parts of the nerve cell:

- Soma
- Dendrites
- Axons
- Synapses

The neuron is made up of a nerve cell consisting of a soma (cell body) that has many dendrites but only one axon. The single axon can branch hundreds of times, however. Dendrites are thin structures that arise from the main cell body. Axons are nerve fibers with a special cellular extension that comes from the cell body.

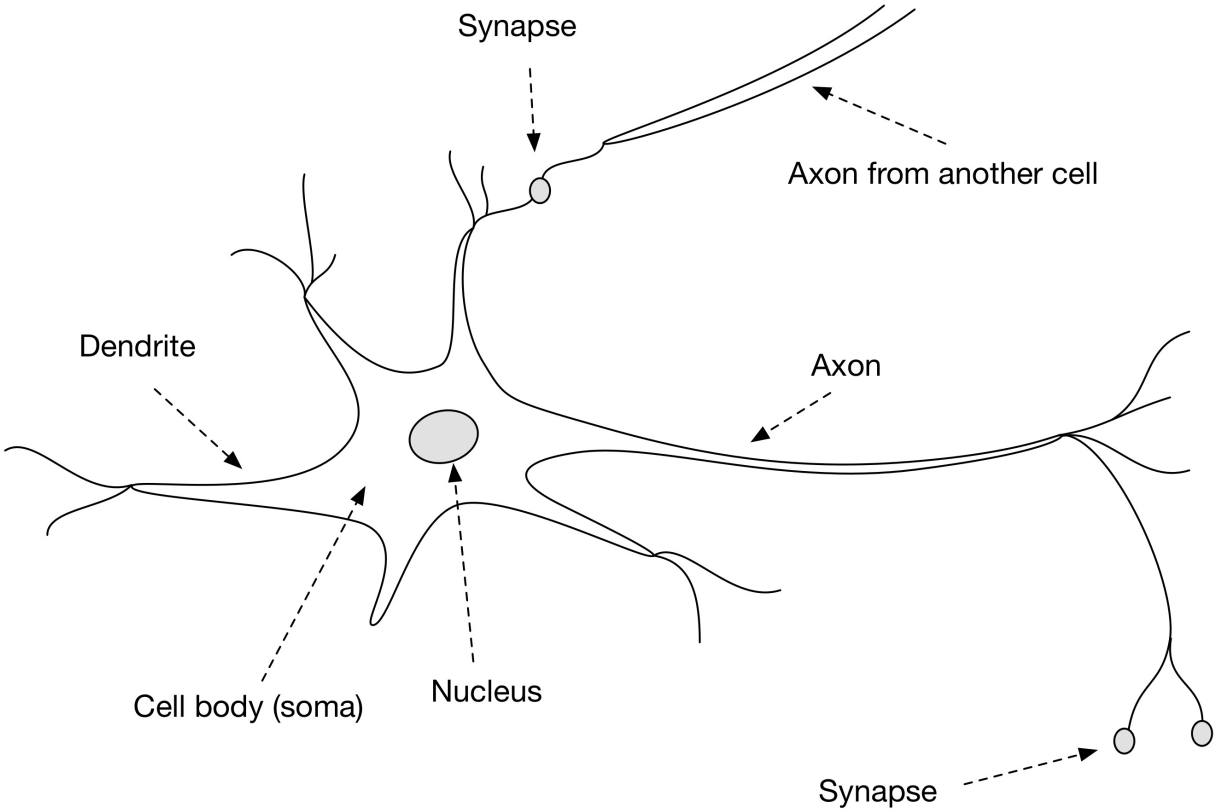

Figure 2-2. The biological neuron

Synapses

Synapses are the connecting junction between axon and dendrites. The majority of synapses send signals from the axon of a neuron to the dendrite of another neuron. The exceptions for this case are when a neuron might lack dendrites, or a neuron lacks an axon, or a synapse, which connects an axon to another axon.

Dendrites

Dendrites have fibers branching out from the soma in a bushy network around the nerve cell. Dendrites allow the cell to receive signals from connected neighboring neurons and each dendrite is able to perform multiplication by that dendrite's weight value. Here multiplication means an increase or decrease in the ratio of synaptic neurotransmitters to signal chemicals introduced into the dendrite.

Axons

Axons are the single, long fibers extending from the main soma. They stretch out longer distances than dendrites and measure generally 1 centimeter in length (100 times the diameter of the soma). Eventually, the axon will branch and connect to other dendrites. Neurons are able to send electrochemical pulses through cross-membrane voltage changes generating *action potential*. This signal travels along the cell's axon and activates synaptic connections with other neurons.

Information flow across the biological neuron

Synapses that increase the potential are considered *excitatory*, and those that decrease the potential are considered *inhibitory*. *Plasticity* refers the long-term changes in strength of connections in response to input stimulus. Neurons also have been shown to form new connections over time and even migrate. These combined mechanics of connection change drive the learning process in the biological brain.

From biological to artificial

The animal brain has been shown to be responsible for the fundamental components of the mind. We can study the basic components of the brain and understand them. Research has shown ways to map out functionality of the brain and track signals as they move through neurons.

CONVOLUTIONAL NEURAL NETWORKS AND THE MAMMALIAN VISION SYSTEM

Later in the book, we take a look at a deep network called a Convolutional Neural Network (CNN). A CNN's image representation at different layers is similar to how the brain processes visual information. Although this research is interesting, it does not mean that a CNN gives us a full approximation of mammalian brain activity.

However, we still do not completely understand how this collection of decentralized functional units provides the foundation for thought and the seat of consciousness.

THE SEAT OF CONSCIOUSNESS

In the eighteenth century, the brain began to be recognized as the “seat of consciousness.” By the late nineteenth century, animal brains began to be mapped out to better understand its functional regions. Previous locations of consciousness included the heart and, oddly enough, the spleen.

Now that we’ve established the basics of how a biological neuron works, let’s take a look at the first attempts at modeling the neuron with the advent of the perceptron.

The Perceptron

The perceptron is a linear model used for binary classification. In the field of neural networks the perceptron is considered an artificial neuron using the Heaviside step function for the activation function, both of which we'll define further later in the chapter. The precursor to the perceptron was the Threshold Logic Unit (TLU) developed by McCulloch and Pitts in 1943, which could learn the AND and OR logic functions. The perceptron training algorithm is considered a supervised learning algorithm. Both the TLU and the perceptron were inspired by the biological neuron as we'll explore.

History of the perceptron

The perceptron was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It was funded by the US Office of Naval Research and was covered by the *New York Times*:

> [T]he embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

Obviously these predictions were a bit premature — as we've seen many times with the promise of machine learning and AI. Early versions were intended to be implemented as a physical machine rather than a software program. The first software implementation was for the IBM 704, and then later it was implemented in the Mark I Perceptron machine.

It also should be noted that McCulloch and Pitts introduced the basic concept of analyzing neural activity in 1943[1] based on thresholds and weighted sums. These concepts were key in developing a model for later variations like the perceptron.

THE MARK I PERCEPTRON

The Mark I Perceptron was designed for image recognition for military purposes by the US Navy. The Mark I Perceptron had 400 photocells connected to artificial neurons in the machine, and the weights were implemented by potentiometers. Weight updates were physically performed by electric motors.

Definition of the perceptron

The perceptron is a linear-model binary classifier with a simple input–output relationship as depicted in Figure 2-3, which shows we're summing *n* number of inputs times their associated weights and then sending this "net input" to a step function with a defined threshold. Typically with perceptrons, this is a Heaviside step function with a threshold value of 0.5. This function will output a real-valued single binary value (0 or a 1), depending on the input.

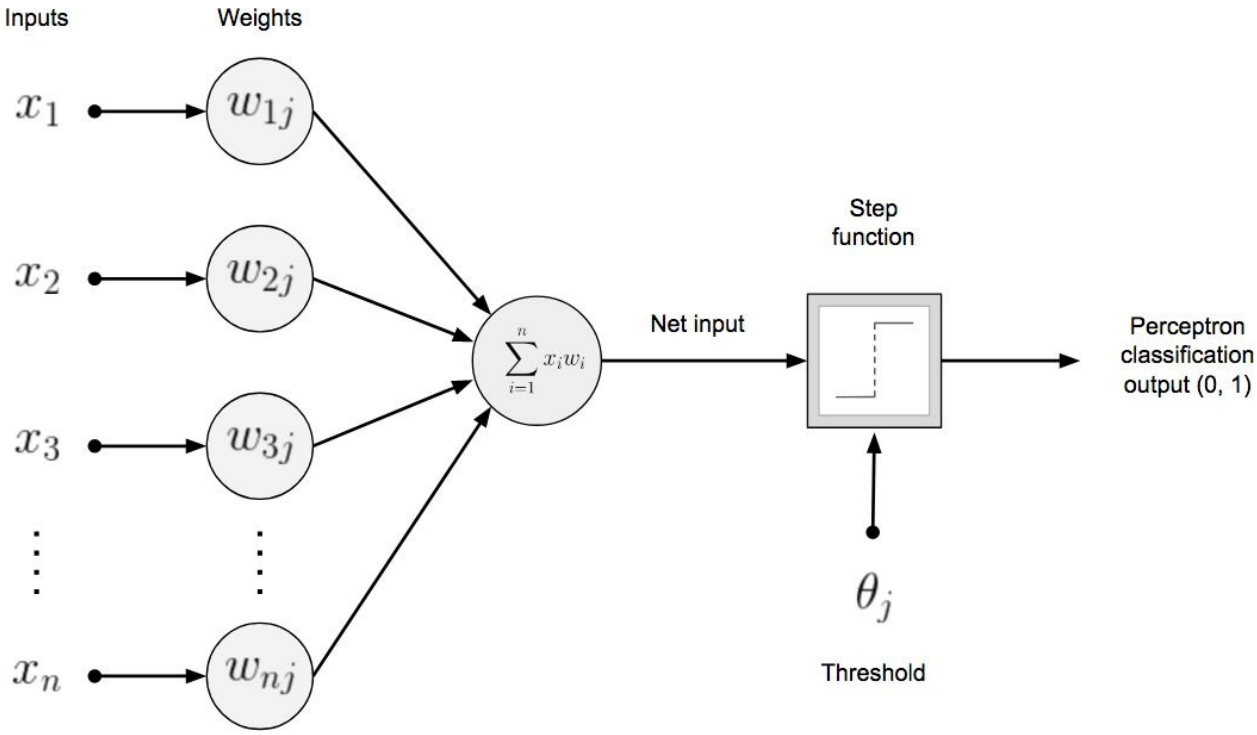

Figure 2-3. Single-layer perceptron

We can model the decision boundary and the classification output in the Heaviside step function equation, as follows:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x >= 0 \end{cases}$$

To produce the net input to the activation function (here, the Heaviside step function) we take the dot product of the input and the connection weights. We see this summation in the left half of Figure 2-3 as the input to the summation function. Table 2-1 provides an explanation of how the summation function is performed as well as notes about the parameters involved in the summation function.

Table 2-1. The summation function parameters

Function parameter	Description
w	Vector of real-valued weights on the connections
w · **x**	Dot product ($\sum_{i=1}^{n} w_i x_i$)
n	Number of inputs to the perceptron
b	The bias term (input value does not affect its value; shifts decision boundary away from origin)

The output of the step function (activation function) is the output for the perceptron and gives us a classification of the input values. If the bias value is negative, it forces the learned weights sum to be a much greater value to get a 1 classification output. The bias term in this capacity moves the decision boundary around for the model. Input values do not affect the bias term, but the bias term is learned through the perceptron learning algorithm.

THE SINGLE-LAYER PERCEPTRON

The perceptron is more widely known as a “single-layer perceptron” in neural network research to distinguish it from its successor the “multilayer perceptron.”

As a basic linear classifier we consider the single-layer perceptron to be the simplest form of the family of feed-forward neural networks.

RELATING THE PERCEPTRON TO THE BIOLOGICAL NEURON

Although we don't have a complete model for how the brain works, we do see how the perceptron was modeled after the biological neuron. In this case, we can see how the perceptron takes input through connections with weights on them in a similar fashion to how synapses pass information to other biological neurons.

The perceptron learning algorithm

The perceptron learning algorithm changes the weights in the perceptron model until all input records are all correctly classified. The algorithm will not terminate if the learning input is not linearly separable. A linearly separable dataset is one for which we can find the values of a hyperplane that will cleanly divide the two classes of the dataset.

The perceptron learning algorithm initializes the weight vector with small random values or 0.0s at the beginning of training. The perceptron learning algorithm takes each input record, as we can see in Figure 2-3, and computes the output classification to check against the actual classification label. To produce the classification, the columns (features) are matched up to weights where *n* is the number of dimensions in both our input and weights. The first input value is the bias input, which is always 1.0 because we don't affect the bias input. The first weight is our bias term in this diagram. The dot product of the input vector and the weight vector gives us the input to our activation function, as we've previously discussed.

If the classification is correct, no weight changes are made. If the classification is incorrect, the weights are adjusted accordingly. Weights are updated between individual training examples in an "online learning" fashion. This loop continues until all of the input examples are correctly classified. If the dataset is not linearly separable, the training algorithm will not terminate. Figure 2-4 demonstrates a dataset that is not linearly separable, the XOR logic function.

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2-4. The XOR function

A basic perceptron (single-layer variant) cannot solve the XOR logic modeling problem, illustrating an early limitation of the perceptron model.

Limitations of the early perceptron

After initial promise, the perceptron was found to be limited in the types of patterns it could recognize. The initial inability to solve nonlinear (e.g., datasets that are not linearly separable) problems was seen as a failure for the field of

neural networks. The 1969 book *Perceptrons* by Minsky and Papert illustrated the limitations of the single-layer perceptron. However, what the general industry did not widely realize was that a multilayer perceptron could indeed solve the XOR problem, among many other nonlinear problems.

AI WINTER I: 1974–1980

The misunderstanding of the multilayer perceptron capabilities was an early public setback that hurt interest in, and funding of, neural networks for the next decade. It wasn’t until the resurgence of neural networks in the mid-1980s that backpropagation became popular (although backpropagation was originally discovered in 1974 by Webos) and neural networks enjoyed a second wave of interest.

Multilayer Feed-Forward Networks

The multilayer feed-forward network is a neural network with an input layer, one or more hidden layers, and an output layer. Each layer has one or more artificial neurons. These artificial neurons are similar to their perceptron precursor yet have a different activation function depending on the layer's specific purpose in the network. We'll look more closely at the layer types in multilayer perceptrons later in the chapter. For now, let's look more closely at this evolved artificial neuron that emerged from the limitations of the single-layer perceptron.

Evolution of the artificial neuron

The artificial neuron of the multilayer perceptron is similar to its predecessor, the perceptron, but it adds flexibility in the type of activation layer we can use. Figure 2-5 shows an updated diagram for the artificial neuron that is based on the perceptron.

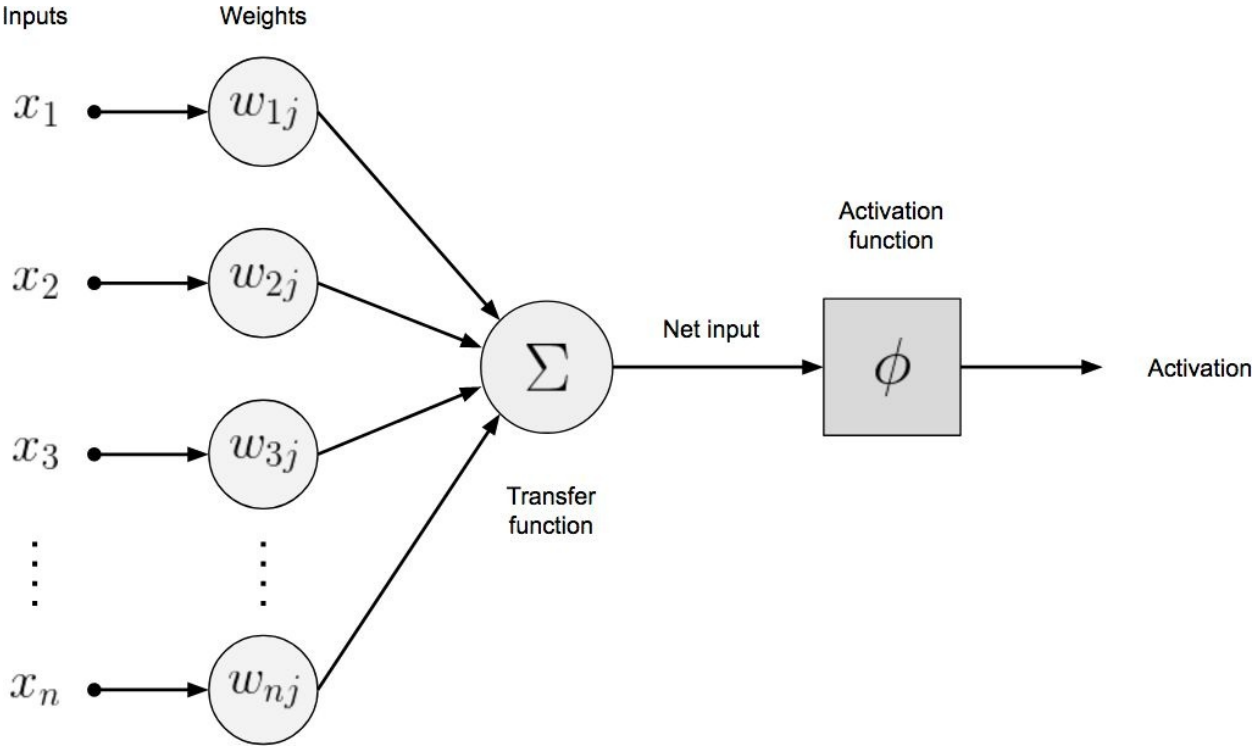

Figure 2-5. Artificial neuron for a multilayer perceptron

This diagram is similar to Figure 2-3 for the single-layer perceptron, yet we

notice a more generalized activation function. We’ll develop this diagram further in a detailed look at the artificial neuron as we proceed.

A NOTE ABOUT THE TERM "NEURON"

From this point throughout the remainder of the book, when we use the term "neuron" we're referring to the artificial neuron based on Figure 2-5.

The net input to the activation function is still the dot product of the weights and input features yet the flexible activation function allows us to create different types out of output values. This is a major contrast to the earlier perceptron design that used a piecewise linear Heaviside step function as this improvement now allowed the artificial neuron because express more complex activation output.

Artificial neuron input

The artificial neuron (see Figure 2-6) takes input that, based on the weights on the connections, can be ignored (by a 0.0 weight on an input connection) or passed on to the activation function. The activation function also has the ability to filter out data if it does not provide a non-zero activation value as output.

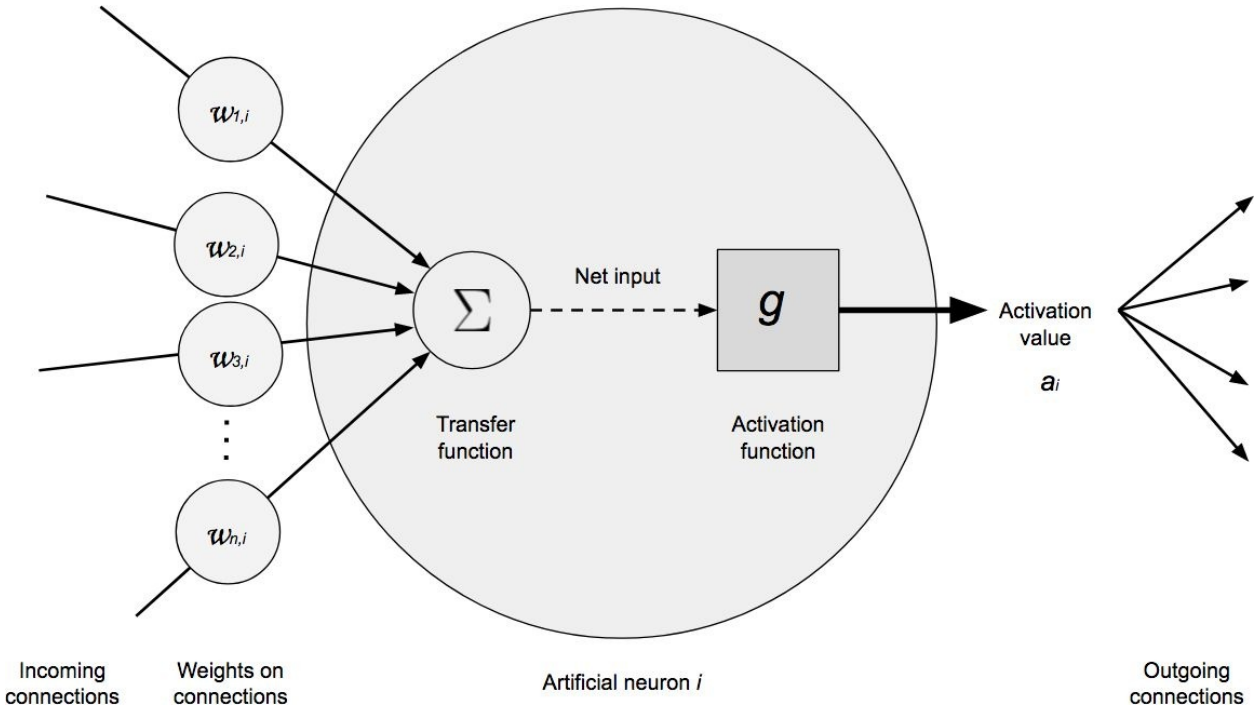

Figure 2-6. Details of an artificial neuron in a multilayer perceptron neural network

We express the net input to a neuron as the weights on connections multiplied by activation incoming on connection, as shown in Figure 2-6. For the input layer, we're just taking the feature at that specific index, and the activation function is linear (it passes on the feature value). For hidden layers, the input is the activation from other neurons. Mathematically, we can express the net input (total weighted input) of the artificial neuron as

- $input\_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i$

where $\mathbf{W}_i$ is the vector of all weights leading into neuron i and $\mathbf{A}_i$ is the vector of activation values for the inputs to neuron i. Let's build on this equation by accounting for the bias term that is added per layer (explained further below):

- $input\_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i + b$

To produce output from the neuron, we'd then wrap this net input with an activation function g, as demonstrated in the following equation:

$$a_i = g(input\_sum_i)$$

We can then expand this function with the definition of $^{input}{}_i$:

$$a_i = g(\mathbf{W}_i \cdot \mathbf{A}_i + b)$$

This activation value for neuron i is the output value passed on to the next layer through connections to other artificial neurons (multiplied by weights on connections) as an input value.

DIFFERENT NOTATIONS

An alternate notation for expressing the output of an artificial neuron we often see in research papers is as follows:

- $h_{w,b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$

This notation is slightly different from the previous equation above. We show this alternate notation to illustrate how some papers will use slightly different notations to explain these concepts, and we want you to be able to recognize the variants.

If our activation function is the sigmoid function, we'll have this:

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

This output will have the range [0, 1], which is the same output as the logistic regression function.

SIGMOID ACTIVATION FUNCTIONS IN PRACTICE

We list the sigmoid activation function here for historical demonstration. As we'll see later on in this book, the sigmoid activation function has largely fallen out of favor.

The inputs are the data from which you want to produce information, and the connection weights and biases are the quantities that govern the activity, activating it or not. As with the perceptron, there is a learning algorithm to change the weights and bias value for each artificial neuron. During the training phase, the weights and biases change as the network learns. We'll cover the learning algorithm for neural networks later in the chapter.

Just as biological neurons don't pass on every electro-chemical impulse they receive, artificial neurons are not just wires or diodes passing on a signal. They are designed to be selective. They filter the data they receive, and aggregate, convert, and transmit only certain information to the next neuron(s) in the network. As these filters and transformations work on data, they convert raw input data to useful information in the context of the larger multilayer perceptron neural network. We illustrate this effect more in the next section.

Artificial neurons can be defined by the kind of input they are able to receive (binary or continuous) and the kind of transform (activation function) they use to produce output. In DL4J, all neurons in a layer have the same activation function.

Connection weights

Weights on connections in a neural network are coefficients that scale (amplify or minimize) the input signal to a given neuron in the network. In common representations of neural networks, these are the lines/arrows going from one point to another, the edges of the mathematical graph. Often, connections are notated as *w* in mathematical representations of neural networks.

Biases

Biases are scalar values added to the input to ensure that at least a few nodes per layer are activated regardless of signal strength. Biases allow learning to happen by giving the network action in the event of low signal. They allow the network

to try new interpretations or behaviors. Biases are generally notated *b*, and, like weights, biases are modified throughout the learning process.

Activation functions

The functions that govern the artificial neuron's behavior are called activation functions. The transmission of that input is known as *forward propagation*. Activation functions transform the combination of inputs, weights, and biases. Products of these transforms are input for the next node layer. Many (but not all) nonlinear transforms used in neural networks transform the data into a convenient range, such as 0 to 1 or –1 to 1. When an artificial neuron passes on a nonzero value to another artificial neuron, it is said to be *activated*.

ACTIVATIONS

Activations are the values passed on to the next layer from each previous layer. These values are the output of the activation function of each artificial neuron.

In “Activation Functions” we’ll review the different types of activation functions and their general function in the broader context of neural networks.

ACTIVATION FUNCTIONS AND THEIR IMPORTANCE

Activation functions and their usage will be a continuing theme throughout almost every chapter for the remainder of this book. The DL4J library uses a layer-based architecture revolving around different types of activation functions.

Comparing the biological neuron and the artificial neuron

If we loop back for a moment and think about the biological neuron on which the artificial neuron is based, we can ask, “How close does the artificial variant match up to the biological version?” The concepts match up with the input connection functionality being performed by dendrites in the biological neuron and the summation functionality being provided by the soma. Finally, we see the activation function being performed by the axon in the biological neuron.

LIMITATIONS OF COMPARISONS

We note (again) that the biological neuron is still more complex than the artificial variant. Research continues toward a better understanding of the biological neuron's function.

Feed-forward neural network architecture

Now that we understand the differences between the artificial neuron and the perceptron, we can better understand the structure of the full multilayer feed-forward neural network. With multilayer feed-forward neural networks, we have artificial neurons arranged into groups called layers. Building on the layer concept, we see that the multilayer neural network has the following:

- A single input layer
- One or many hidden layers, fully connected
- A single output layer

As Figure 2-7 depicts, the neurons in each layer (represented by the circles) are all fully connected to all neurons in all adjacent layers.

The neurons in each layer all use the same type of activation function (most of the time). For the input layer, the input is the raw vector input. The input to neurons of the other layers is the output (activation) of the previous layer's neurons. As data moves through the network in a feed-forward fashion, it is influenced by the connection weights and the activation function type. Let's now take a look at the specifics of each layer type.

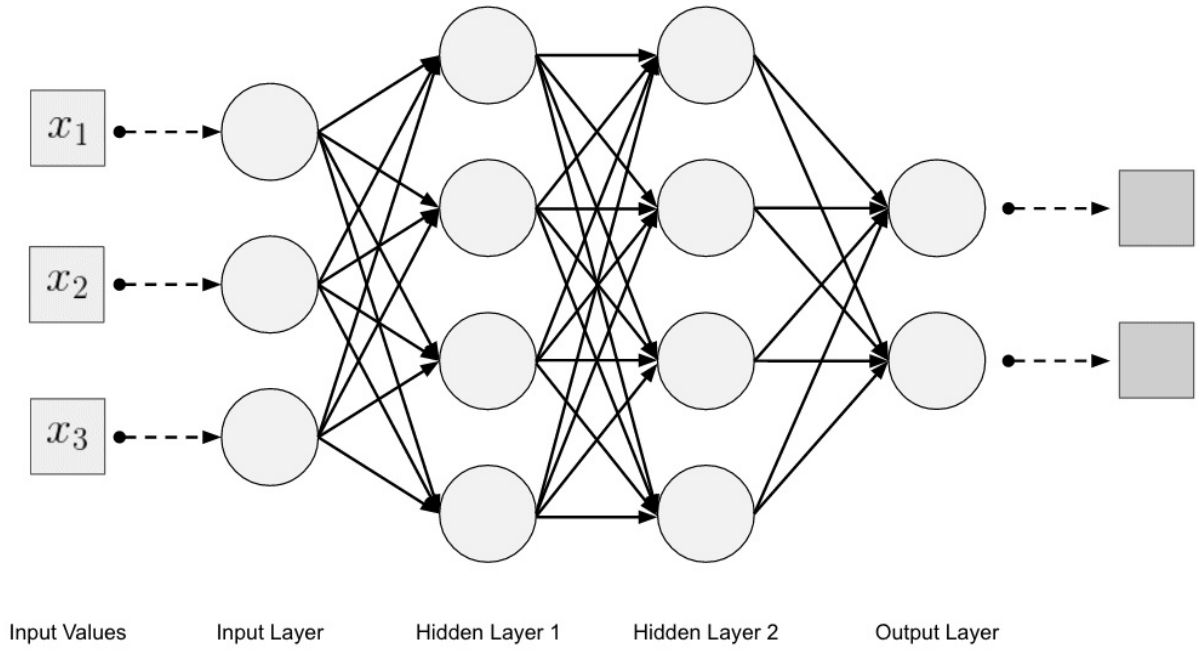

Figure 2-7. Fully connected multilayer feed-forward neural network topology

Input layer

This layer is how we get input data (vectors) fed into our network. The number of neurons in an input layer is typically the same number as the input feature to the network. Input layers are followed by one or more hidden layers (explained in the next section). Input layers in classical feed-forward neural networks are fully connected to the next hidden layer, yet in other network architectures, the input layer might not be fully connected.

Hidden layer

There are one or more hidden layers in a feed-forward neural network. The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data. Hidden layers are the key to allowing neural networks to model nonlinear functions, as we saw from the limitations of the single-layer perceptron networks.

Output layer

We get the answer or prediction from our model from the output layer. Given that we are mapping an input space to an output space with the neural network

model, the output layer gives us an output based on the input from the input layer. Depending on the setup of the neural network, the final output may be a real-valued output (regression) or a set of probabilities (classification). This is controlled by the type of activation function we use on the neurons in the output layer. The output layer typically uses either a *softmax* or *sigmoid activation function* for classification. We'll discuss the difference between these two types of activation functions later in the chapter.

Connections between layers

In a fully connected feed-foward network, the connections between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the next layer. We change these weights progressively as our algorithm finds the best solution it can with the backpropagation learning algorithm. We can understand the weights mathematically by thinking of them as the parameter vector in the earlier linear algebra section describing the machine learning process as optimizing the parameter vector (e.g., "weights" here) to minimize error.

We've now covered the basic structure of feed-forward neural networks. The rest of this chapter provides a more detailed look at the training mechanics of backpropagation as well as specifics of activation functions. The chapter closes with a review of common loss functions and hyperparameters.

Training Neural Networks

A well-trained artificial neural network has weights that amplify the signal and dampen the noise. A bigger weight signifies a tighter correlation between a signal and the network's outcome. Inputs paired with large weights will affect the network's interpretation of the data more than inputs paired with smaller weights.

The process of learning for any learning algorithm using weights is the process of re-adjusting the weights and biases, making some smaller and others larger, thereby allocating significance to certain bits of information and minimizing other bits. This helps our model learn which predictors (or features) are tied to which outcomes, and adjusts the weights and biases accordingly.

In most datasets, certain features are strongly correlated with certain labels (e.g., square footage relates to the sale price of a house). Neural networks learn these relationships blindly by making a guess based on the inputs and weights and then measuring how accurate the results are. The loss functions in optimization algorithms, such as stochastic gradient descent (SGD), reward the network for good guesses and penalize it for bad ones. SGD moves the parameters of the network toward making good predictions and away from bad ones.

Another way to look at the learning process is to view labels as theories and the feature set as evidence. Then, we can make the analogy that the network seeks to establish the correlation between the theory and the evidence. The model attempts to answer the question "which theory does the evidence support?" With these ideas in mind, let's take a look at the learning algorithm most commonly associated with neural networks: *backpropagation learning*.

Backpropagation Learning

Backpropagation is an important part of reducing error in a neural network model. To explain backpropagation, we'll return to our discussion about how information circulates within a feed-foward neural network. Let's look at how this learning algorithm works before we dive deeper into the mathematical notation and pseudocode for backpropagation learning.

ORIGINS OF BACKPROPAGATION LEARNING

Backpropagation learning was first invented by Bryson and Ho in 1969. It was largely ignored in research and practice for neural network training until a resurgence in the mid-1980s.

Algorithm intuition

Backpropagation learning is similar to the perceptron learning algorithm. We want to compute the input example's output with a forward pass through the network. If the output matches the label, we don't do anything. If the output does not match the label, we need to adjust the weights on the connections in the neural network.

To further illustrate general neural network learning, let's take a look at the pseudocode for the algorithm, as shown in Example 2-1.

Example 2-1. General neural network training pseudocode

```
function neural-network-learning( training-records ) returns network
    network <- initialize weights (randomly)
    start loop
        for each example in training-records do
            network-output = neural-network-output( network, example )
            actual-output = observed outcome associated with example
            update weights in network based on
                { example, network-output, actual-output }
        end for
    end loop when all examples correctly predicted or hit stopping conditions
    return network
```

The key is to distribute the blame for the error and divide it between the contributing weights. With the perceptron learning algorithm, it's easy because there is only one weight per input to influence the output value. With feed-forward multilayer networks learning algorithms have a bigger challenge. There are many weights connecting each input to the output, so it becomes more difficult. Each weight contributes to more than one output, so our learning algorithm must be more clever.

Backpropagation is a pragmatic approach to dividing the contribution of error for each weight. It is similar to the perceptron learning algorithm. With backpropagation, we're trying to minimize the error between the label (or "actual") output associated with the training input and the value generated from

the network output. In the next section, we take a look at the mathematical notation that you will see in most literature on neural networks for backpropagation of feed-forward neural networks.

A CAUTION ON LEARNING ALGORITHMS IN MULTILAYER NETWORKS

Learning algorithms for multilayer neural networks are neither guaranteed to converge to a global optimum nor are they absolutely efficient. This is a direct effect of how learning a general function from training examples is considered, in the worst case, an intractable problem. Learning algorithms do, however, work decently in practice with some decent hyperparameter tuning.

A closer look at backpropagation

For most of this book, we don’t intend to throw a lot of math at you. However, for the topic of backpropagation and to better understand a core fundamental concept that much of the book is based on, we feel we should provide a section that illustrates these concepts down to the notation level.

A NOTE ABOUT NOTATION

This notation is similar to what you would see in a conference paper on machine learning or in a well-known machine learning textbook. Hopefully, we can explain the notation in a way that will feel comfortable for you. Ideally, this will serve as a jump-off point for you to explore many more neural network and deep learning papers down the road.

Let’s zoom in on the previous diagram to focus on the input layer and the first hidden layer. Figure 2-8 provides the view.

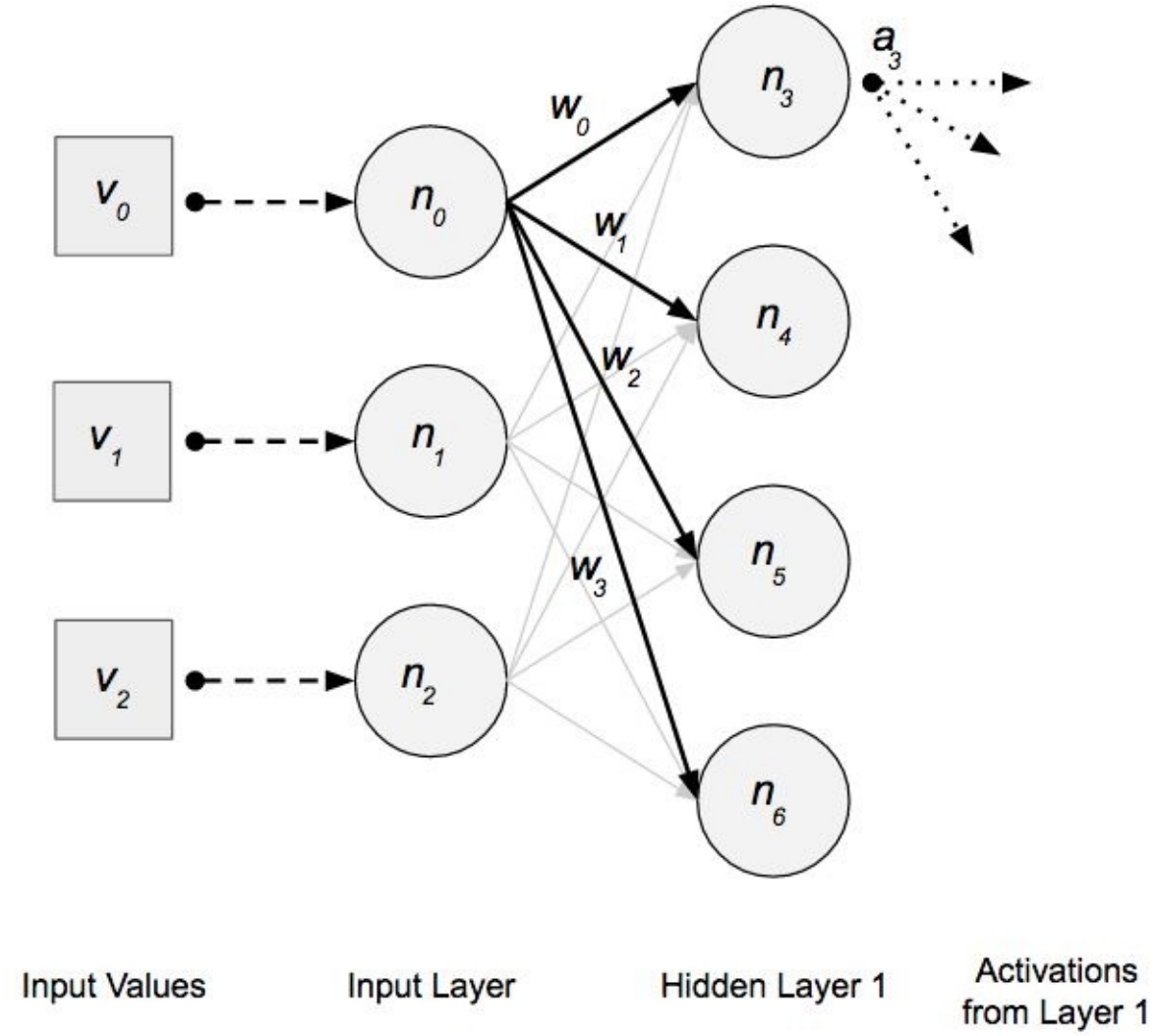

Figure 2-8. Multilayer perceptron network zoomed-in with component labels

In Figure 2-8 we see our previous multilayer neural network diagram zoomed-in and updated with some notation. We'll use this notation for the rest of this section's explanation of backpropagation. Table 2-2 lists the notations in the figure.

Table 2-2. Neural network notation

Notation	Meaning
i	Index of artificial neuron
n_i	Neuron at index *i*
j	Index of neuron in previous layer connecting to neuron *i*
a_i	Activation value of neuron *i* (output of neuron *i*)
$\mathbf{A}_i$	Vector of activation values for the inputs into neuron *i*
g	Activation function
g'	Derivative of the activation function
Err_i	Difference between the network output and the actual output value for the training example
$\mathbf{W}_i$	Vector of weights leading into neuron *i*
$W_{j,i}$	Weight on the incoming connection from previous layer neuron *j* to neuron *i*
$input\_sum_i$	Weighted sum of inputs to neuron *i*
$input\_sum_j$	Weighted sum of inputs for neuron *j* in previous layer (used in backpropagation)
α	Learning rate
Δ_j	Error term for connected neuron *j* in previous layer
Δ_i	Error term for neuron *i* ; $= Err_i \times g'(input\_sum_i)$

To further set the stage for explaining this algorithm, let's take a look at the pseudocode of the backpropagation learning algorithm, as shown in Example 2-2.

Example 2-2. Backpropagation algorithm for updating weights pseudocode

```
function backpropagation-algorithm
    ( network, training-records, learning-rate ) returns network
    network <- initialize weights (randomly)
```

```
start loop
    for each example in training-records do

        // compute the output for this input example
        network-output <- neural-network-output( network, example )

        // compute the error and the [delta] for neurons in the output layer
        example_err <- target-output - network-output

        // update the weights leading to the output layer
```

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(input\_sum_i)$$

```
        for each subsequent-layer in network do

            // compute the error at each node
```

$$\Delta_j \leftarrow g'(input\_sum_j)\Sigma_i W_{j,i}\Delta_i$$

```
            // update the weights leading into the layer
```

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

```
        end for

    end for
end loop when network has converged
return network
```

A NOTE ABOUT LOSS FUNCTIONS IN THE PSEUDOCODE

The loss function (described later in this chapter) is not explicitly called out in the pseudocode in Example 2-2. We've presented backpropagation in algorithmic form here for readability purposes because it's most approachable to the general practioner. However, to give consideration to the mathematics-minded practioner, we also have a version of backpropagation explained mathematically in Appendix C.

In this case, the Err_i term is dependent on the derivative of the loss function. We're using mean squared error (MSE), so the derivative works out to be the difference.

Understanding backpropagation pseudocode

Example 2-2 has the following inputs:

- Network: a multilayer feed-forward neural network
- Training records: a set of training vectors with associated outputs
- Learning rate: the learning rate (sometimes denoted by the Greek alpha symbol)

To start the algorithm, we initialize our neural network and begin looping through the input examples (until we encounter a terminating condition or a maximum number of epochs). First, we compute the output of the current network for the current input example. We compare this output to the actual output associated with the input and compute the error (*example_err*).

Now we're ready to compute the weight updates leading to the output layer.

Updating the output layer weights

The output layer weights update is calculated by using the following:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(input\_sum_i)$$

This is the weight update rule for the connection between neuron *j* and neuron *i* for all connections in the neural network. We can see these connections leading into the output layer highlighted in Figure 2-9.

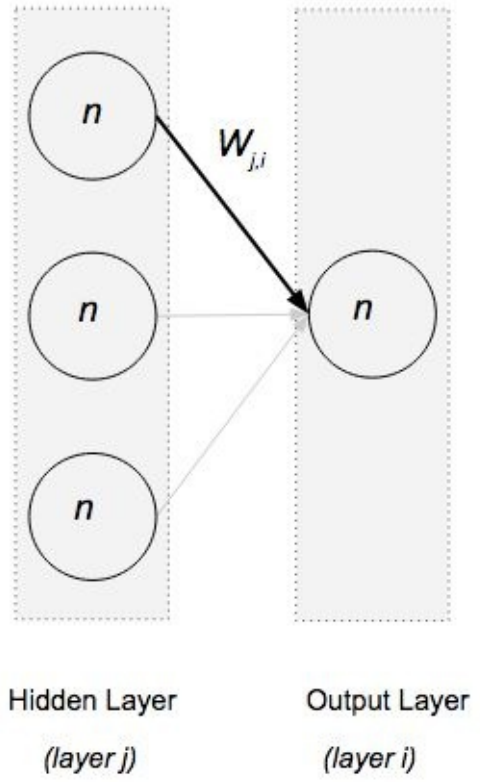

Figure 2-9. Updating connections into the output layer

Breaking this diagram down further, we can see we are working with the neuron *j* weight from the previous hidden layer connecting to the current neuron *i*. We're multiplying the learning rate α (discussed later in the chapter) by the incoming activation from the *j* neuron. This input is calculated by getting the net input to the *j* neuron and then computing the activation of neuron *j*.

Computing the total weight input to the activation function for neuron *i*, we compute the dot product of the incoming weight vector $\mathbf{W}_j$ and the activation vector $\mathbf{A}_j$ and then add in the bias term value:

$input\_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i + b$

Here is the equation for computing the activation value of neuron *j*:

$$a_j = g(input\_sum_j)$$

The error term for example *e* at neuron *i* is denoted as Err_i. We denote the derivative of the activation function as *g'(x)*, which is applied to the net input of neuron *i* with the term:

$$g'(input\_sum_i)$$

This update rule is similar to how we'd update a perceptron except we're using the activations of the previous layers as opposed to their raw input values. This rule also contains a term for the derivative of the activation function to get the gradient of the activation function.

Further expressing the error term

We saw in the previous section that *g'(z)* gave us the derivative of the activation function. The error term is commonly expressed as Δ_i, giving us:

$$\Delta_i = Err_i \times g'(input\_sum_i)$$

This gives us a more condensed weight update function, expressed as the following:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

We see this update expression used in the inner loop of the pseudocode, as we'll now discuss.

GRADIENT DESCENT IN WEIGHT SPACE

We consider backpropagation to be doing gradient descent in weight space where the gradient is on the error surface. This error surface describes the error of the input features as a function of the weight values in the neural network.

The new propagation rule for the error value

The propagation rule for the delta values now becomes the following:

$$\Delta_j \leftarrow g'(input\_sum_j)\Sigma_i W_{j,i}\Delta_i$$

This gives us a new update rule for weights between the inputs and the hidden layer.

Updating the hidden layers

With the backpropagation algorithm, we walk back across the hidden layers, updating the connection between each one until we reach the input layer. Figure 2-10 illustrates a connections between the two hidden layers highlighted.

To update these connections we take the input from the fractional error value computed previously and multiply it by the activation (input) from the connection from the previous layer and the learning rate.

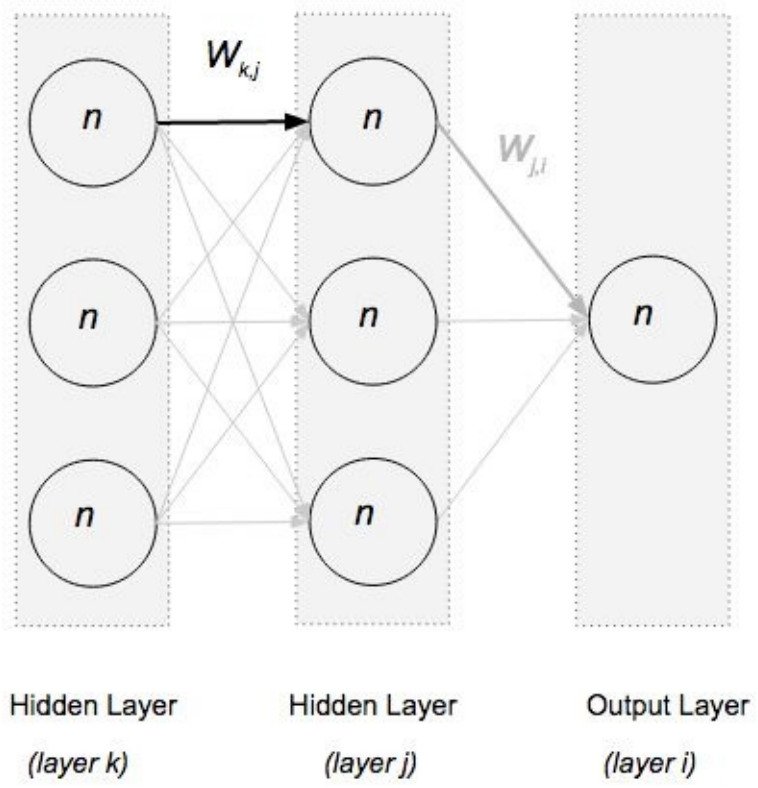

Figure 2-10. Highlighting connections between hidden layers

We then add this to the previous value for the weight, and this is our updated value:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

Weights and biases that have been assigned the blame for error are reduced to limit their signal. Weights and biases that pass along signals supporting correct answers are strengthened. The adjustment of weights to an optimal state is an iterative process, taken one step at a time.

BACKPROPAGATION AND FRACTIONAL ERROR RESPONSIBILITY

With backpropagation is how we distribute the “blame” backward through the network. Each hidden node sending input to the current node is somewhat “responsible” for some portion of the error in each of the neurons to which it has a forward connection.

The first hidden layer uses the input from the raw feature vector as input. All subsequent layers use the activation value of the previous layer's neurons as input. However, for hidden layers before the output layer we must divide up this error appropriately. With error backpropagation, we divide the Δ_i values according to the connection weight between the hidden node and the output node.

To calculate Δ_j in the preceding equation we use the equation:

$$\Delta_j = g'(input\_sum_j)\Sigma_i W_{j,i}\Delta_i$$

This equation takes a look at each node i in the current layer and takes the current error value Δ_i and multiplies it by the weight of the incoming connection times the derivative of the activation function. This produces the fractional error value for the node in the previous layer, which is used as we update the incoming connections to that layer. We progressively perform this algorithm layer by layer until we've updated every layer in the network.

The length of these learning steps, or the amount the weights that are changed with each iteration, is known as the *learning rate*. The learning rate is a parameter we define (as opposed to being a measurement of the network's performance). We discuss learning rate later in this chapter when we talk more about hyperparameters in general.

BACKPROPAGATION AND MINI-BATCH STOCHASTIC GRADIENT DESCENT

In Chapter 1, we learned about a variant of SGD called mini-batch, in which we train the model on multiple examples at once as opposed to a single example at a time. We see mini-batch used with backpropagation and SGD in neural networks as well to improve training.

Under the hood, we're computing the average of the gradient across all the the examples inside the mini-batch. Specifically, we compute the forward pass for all of the examples to get their output scores as a batch linear algebra matrix operation. During the backward pass for each layer, we are computing the average of the gradient (for the layer). By doing backpropagation this way, we're able to get a better gradient approximation and use our hardware more efficiently at the same time.

AI WINTER II: EARLY 1990S

In the late 1980s and early 1990s, we saw an overpromotion of technologies such as expert systems and LISP machines, both of which failed to live up to expectations. The Strategic Computing Initiative cancelled new spending at the end of this cycle. The fifth-generation computer failed its goals.

Activation Functions

We use activation functions to propagate the output of one layer's nodes forward to the next layer (up to and including the output layer). Activation functions are a scalar-to-scalar function, yielding the neuron's activation. We use activation functions for hidden neurons in a neural network to introduce nonlinearity into the network's modeling capabilities. Many activation functions belong to a logistic class of transforms that (when graphed) resemble an S. This class of function is called *sigmoidal*. The sigmoid family of functions contains several variations, one of which is known as the Sigmoid function. Let's now take a look at some useful activation functions in neural networks.

Linear

A linear transform (see Figure 2-11) is basically the identity function, and $f(x) = Wx$, where the dependent variable has a direct, proportional relationship with the independent variable. In practical terms, it means the function passes the signal through unchanged.

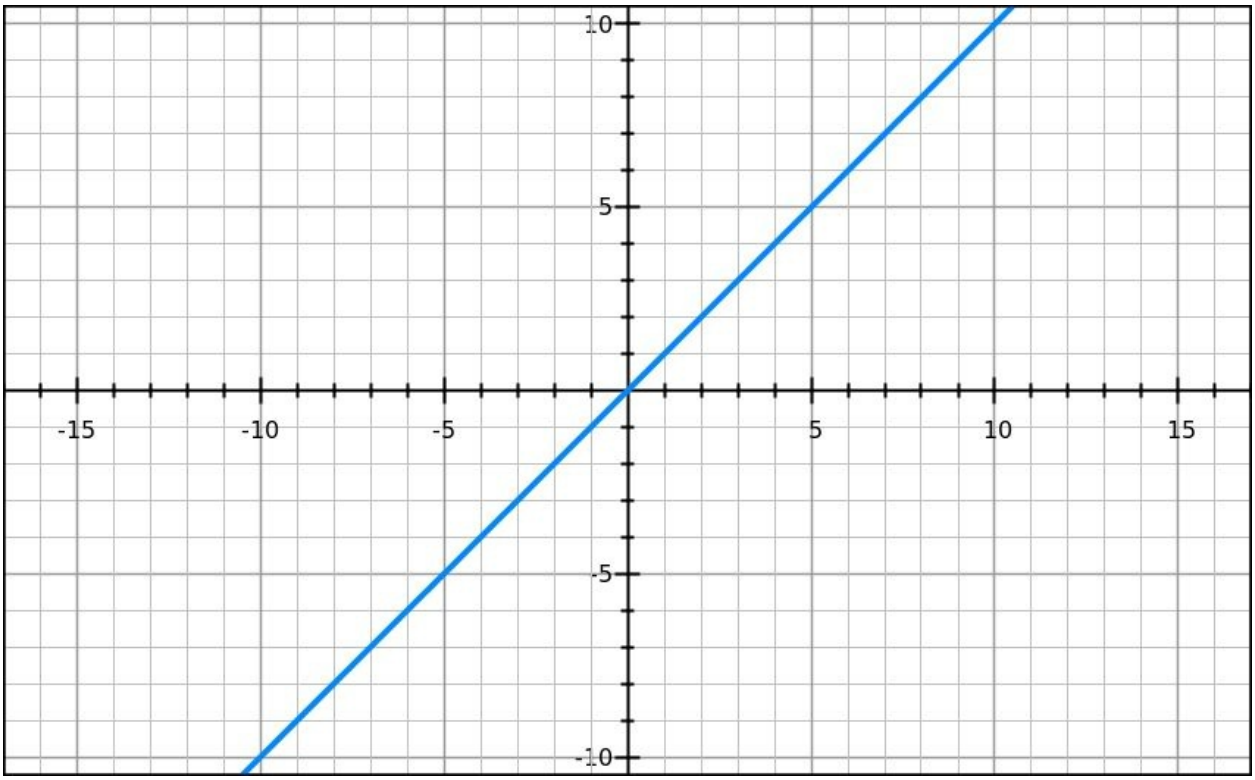

Figure 2-11. Linear activation function

We see this activation function used in the input layer of neural networks.

Sigmoid

Like all logistic transforms, sigmoids can reduce extreme values or outliers in data without removing them. The vertical line in Figure 2-12 is the decision boundary.

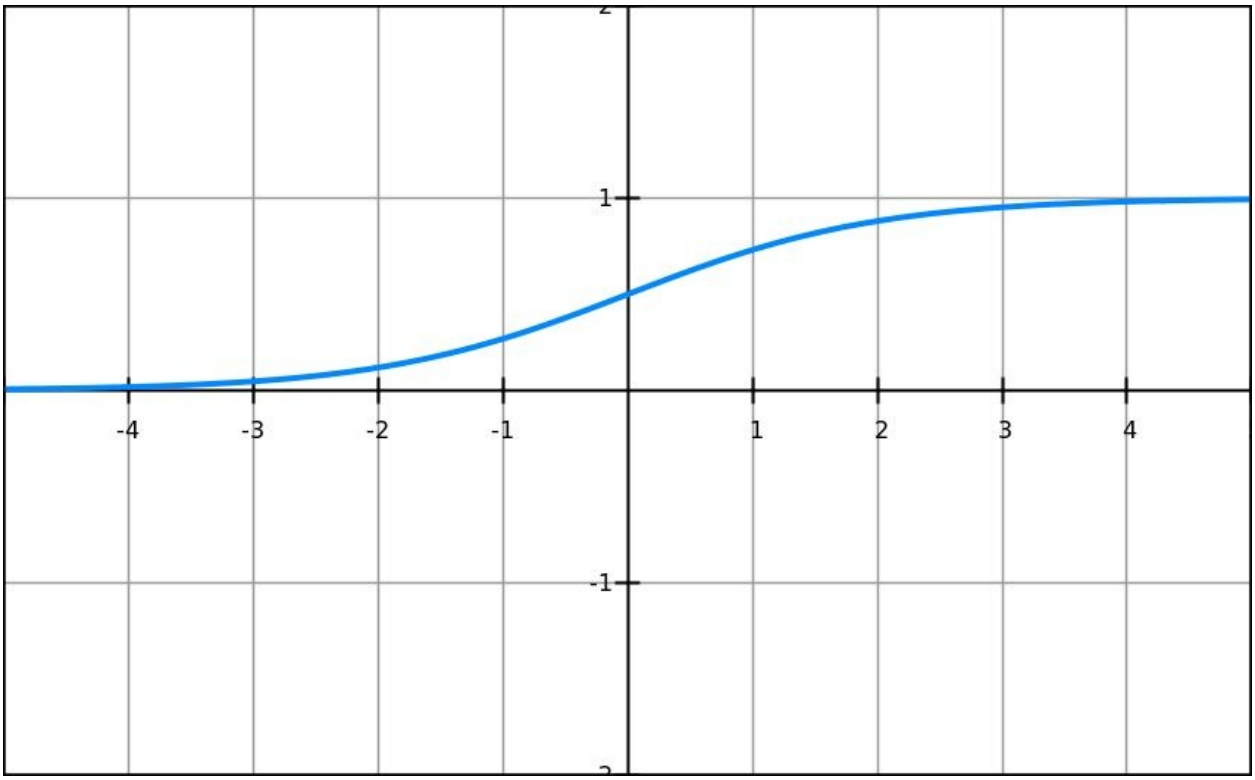

Figure 2-12. Sigmoid activation function

A sigmoid function is a machine that converts independent variables of near infinite range into simple probabilities between 0 and 1, and most of its output will be very close to 0 or 1.

UNDERSTANDING SIGMOID OUTPUT

A sigmoid activation function outputs an independent probability for each class.

Tanh

Pronounced "*tanch*," tanh is a hyperbolic trigonometric function (see Figure 2-13). Just as the tangent represents a ratio between the opposite and adjacent sides of a right triangle, tanh represents the ratio of the hyperbolic sine to the hyperbolic cosine: tanh(x) = sinh(x) / cosh(x). Unlike the Sigmoid function, the normalized range of tanh is –1 to 1. The advantage of tanh is that it can deal more easily with negative numbers.

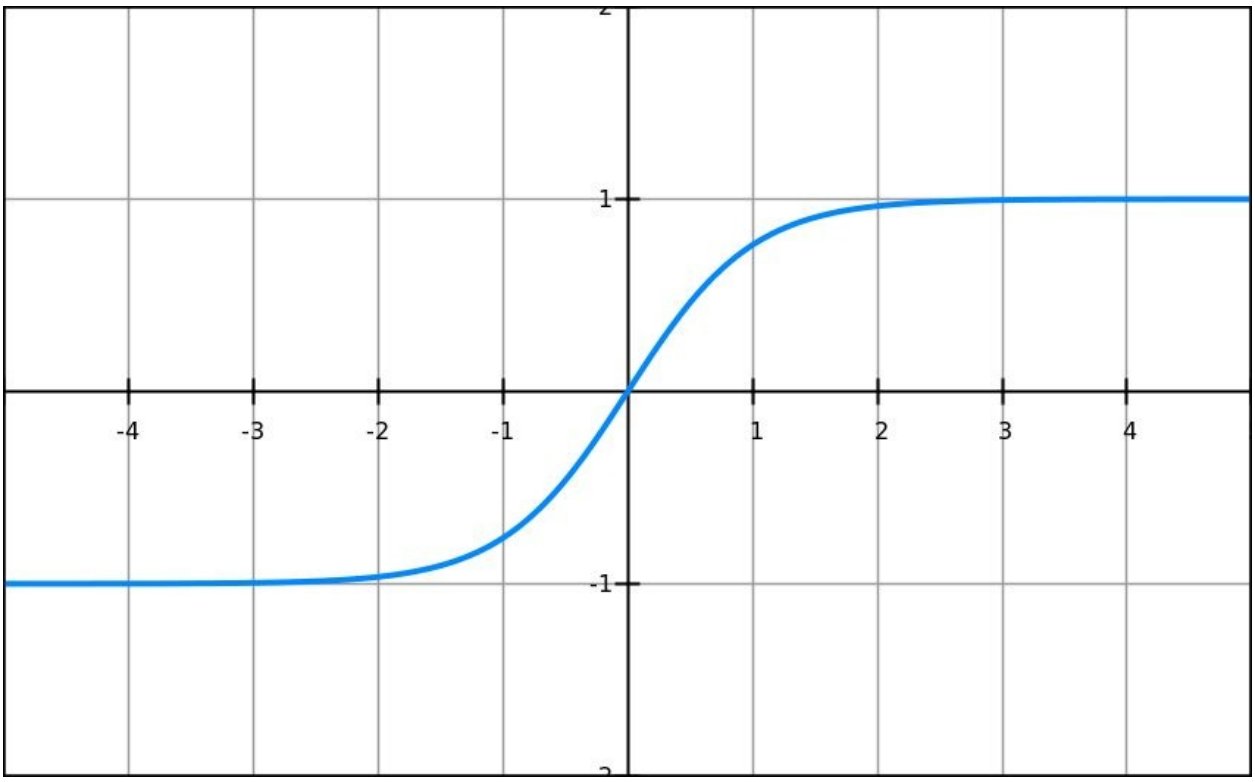

Figure 2-13. Tanh activation function

Hard Tanh

Similar to tanh, hard tanh simply applies hard caps to the normalized range. Anything more than 1 is made into 1, and anything less than –1 is made into –1. This allows for a more robust activation function that allows for a limited decision boundary.

Softmax

Softmax is a generalization of logistic regression inasmuch as it can be applied to continuous data (rather than classifying binary) and can contain multiple decision boundaries. It handles multinomial labeling systems. Softmax is the function you will often find at the output layer of a classifier.

UNDERSTANDING SOFTMAX OUTPUT

The softmax activation function returns the probability distribution over mutually exclusive output classes.

To further illustrate the idea of the softmax output layer and how to use it, let’s consider two use cases. If we have a multiclass modeling problem yet we care only about the best score across these classes, we’d use a softmax output layer with an `argmax()` function to get the highest score of all the classes.

DEALING WITH MULTIPLE CLASSIFICATIONS

If we want to get multiple classifications per output (e.g., “person + car”), we do not want softmax as an output layer. Instead, we’d use the sigmoid output layer giving us a probability for every class independently.

For the case in which we have a large set of labels (e.g., thousands of labels), we’d use the variant of the softmax activation function called the *hierarchical softmax* activation function. This variant decomposes the labels into a tree structure, and the softmax classifier is trained at each node of the tree to direct the branching for classification.

Rectified Linear

Rectified linear is a more interesting transform that activates a node only if the input is above a certain quantity. While the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable $f(x) = max(0, x)$, as demonstrated in Figure 2-14.

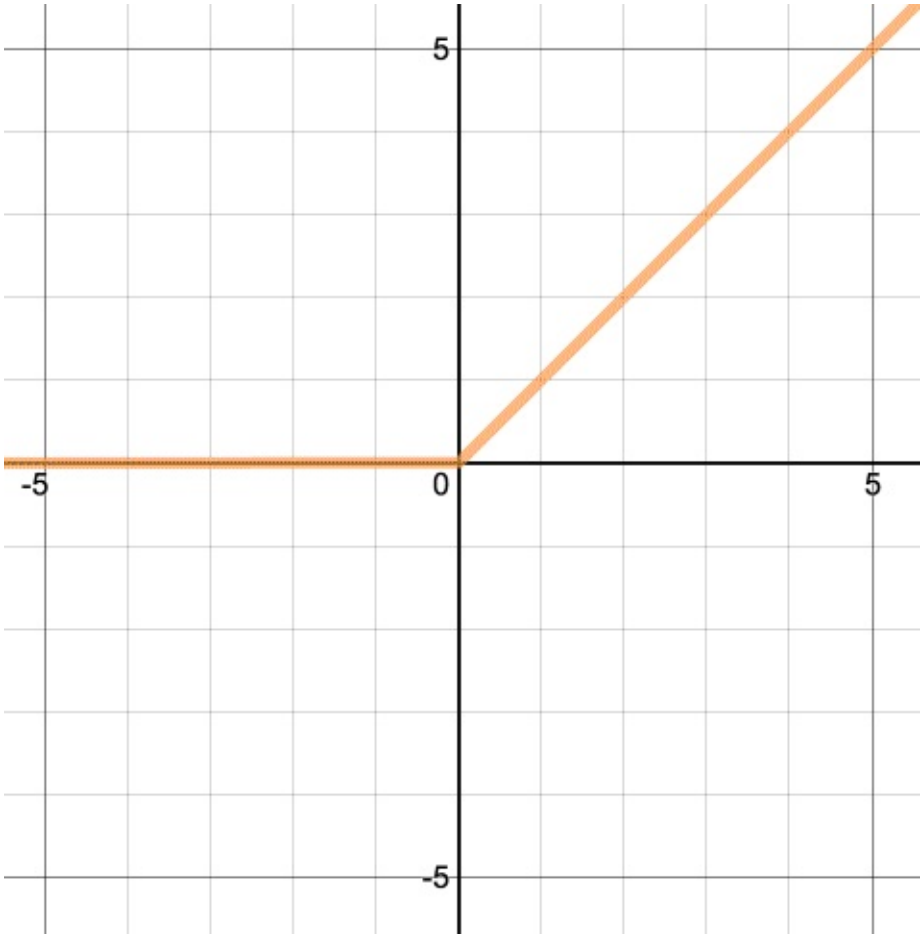

Figure 2-14. Rectified linear activation function

Rectified linear units (ReLU) are the current state of the art because they have proven to work in many different situations. Because the gradient of a ReLU is either zero or a constant, it is possible to reign in the vanishing exploding gradient issue. ReLU activation functions have shown to train better in practice than sigmoid activation functions.

THE UNREASONABLE EFFECTIVENESS OF RELU ACTIVATION FUNCTIONS

Compared to the sigmoid and tanh activation functions, the ReLU activation function does not suffer from vanishing gradient issues. If we use hard max as the activation function, we can induce sparsity in the activation output from the layer. Research has shown deep networks using ReLU activation functions to train well without using pretraining techniques.

Leaky ReLU

Leaky ReLUs are a strategy to mitigate the "dying ReLU" issue.[2] As opposed to having the function being zero when x < 0, the leaky ReLU will instead have a small negative slope (e.g., "around 0.01"). Some success has been seen in practice with this ReLU variation but results are not always consistent. The equation is given here:

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

Softplus

This activation function is considered to be the "smooth version of the ReLU," as is illustrated in Figure 2-15. Compare this plot to the ReLU in Figure 2-14.

Figure 2-15 shows that the softplus activation function ($f(x) = ln[\ 1 + exp(x)\ ]$) has a similar shape to the ReLU. We also notice the differentiability and nonzero derivative of the softplus everywhere on the graph, in contrast to the ReLU.

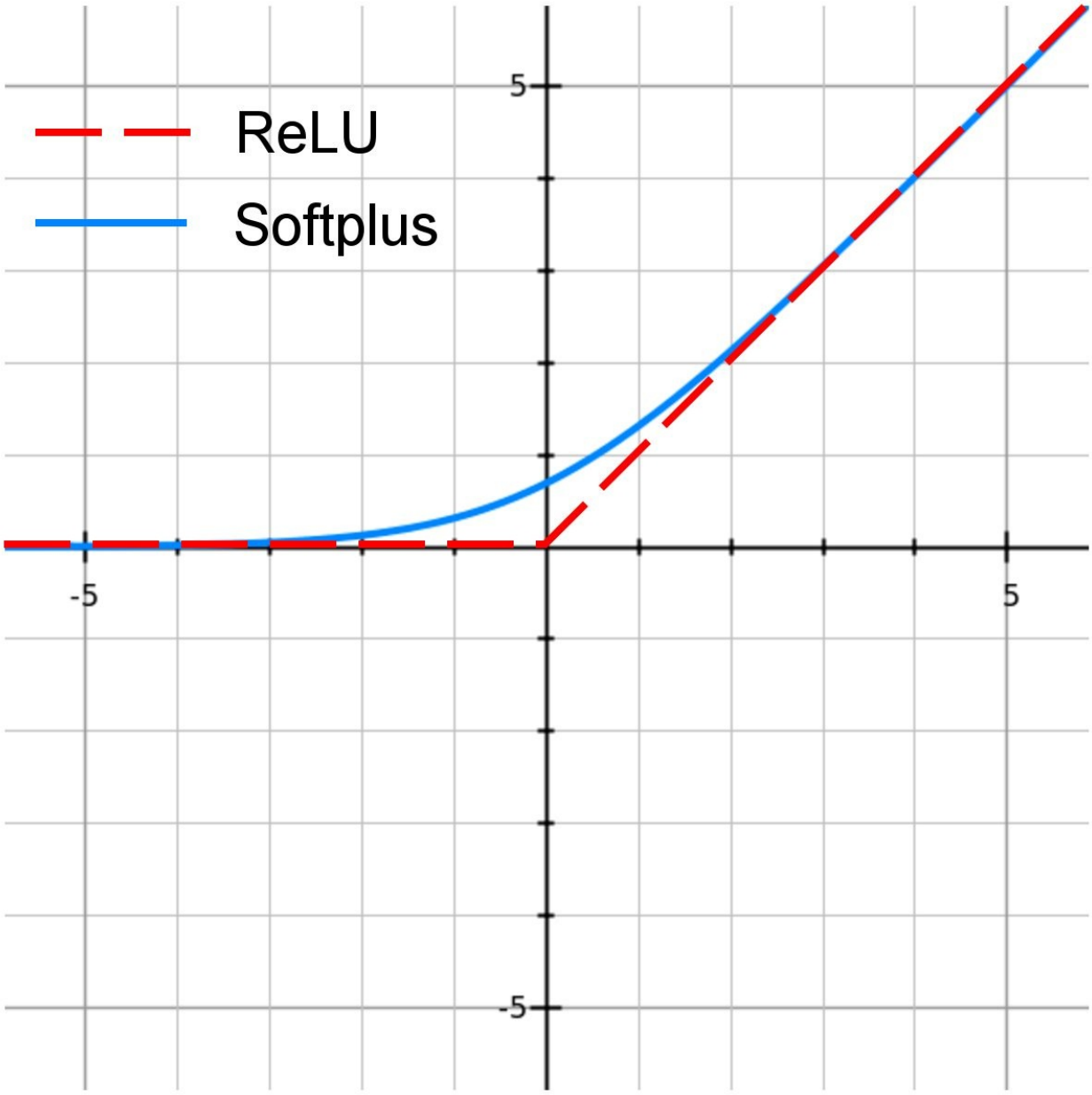

Figure 2-15. Visualizing the ReLU and softplus activation functions

Loss Functions

Loss functions quantify how close a given neural network is to the ideal toward which it is training. The idea is simple. We calculate a metric based on the error we observe in the network's predictions. We then aggregate these errors over the entire dataset and average them and now we have a single number representative of how close the neural network is to its ideal.

Looking for this ideal state is equivalent to finding the parameters (weights and biases) that will minimize the "loss" incurred from the errors. In this way, loss functions help reframe training neural networks as an optimization problem. In most cases, these parameters cannot be solved for analytically, but, more often than not, they can be approximated well with iterative optimization algorithms like gradient descent. The following section provides an overview on commonly seen loss functions, linking them back to their origins in machine learning, as necessary.

Loss Function Notation

Equations in this section will use the notation described here:

- Consider the dataset gathered to train a neural net. Let "*N*" denote the number of samples (set of inputs with corresponding outcomes) that have been gathered.

- Consider the nature of the input and output collected. Each data point records some set of unique input features and output features. Let "*P*" denote the number of input features gathered and "*M*" denote the number of output features that have been observed.

- We will use (*X,Y*) to denote the input and output data we collected. Note that there will be *N* such pairs where the input is a collection of *P* values and the output *Y* is a collection of *M* values. We will denote the *ith* pair in the dataset as X_i and Y_i.

- We will use $\hat{Y}$ to denote the output of the neural net. Of course, $\hat{Y}$ is the network's guess at *Y* and therefore it will also have *M* features.

- We will use the notation $h(X_i) = \hat{Y}_i$ to denote the neural network transforming the input X_i to give the output $\hat{Y}_i$. We will alter this notation a little later to emphasize its dependence on weights and biases.

- When referring to *jth* output feature, we will use it as a subscript firmly linking our notation to a matrix where the rows are different data points and the columns are the different unique features. Thus $y_{i,j}$ refers to the *jth* feature observed in the *ith* sample collected.

- We will represent the loss function by *L(W,b)*.

Given the data available, the loss function notation indicates that its value depends only on *W* and *b*, the weights and the biases of the neural network. This cannot be emphasized enough. In the universe of a given neural network with a set number of layers, configurations, and so on that will be trained on a given set of data, the value of the loss function depends exclusively on the state of the

network, as defined by the weights and biases. Wiggle those and our losses wiggle. Wiggle those for a given input and our output wiggles.

So, our notation $h(X) = \hat{Y}$ should be conditioned on a set of weights and biases; thus, we will amend our notation to say $h_{w,b}(X) = \hat{Y}$. We are now ready to tackle loss functions.

Loss Functions for Regression

In this section, we cover loss functions appropriate for regression models.

Mean squared error loss

When working on a regression model that requires a real valued output, we use the squared loss function, much like the case of ordinary least squares in linear regression. Consider the case in which we have to predict only one output feature ($M = 1$). The error in a prediction is squared and is averaged over the number of data points, plain and simple, as we see in the following equation for MSE loss:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{Y}_i - Y_i\right)^2$$

What if M is greater than one and we are looking to predict multiple output features for a given set of input features? In this case, the desired and predicted entities, Y and $\hat{Y}$, respectively, are an ordered list of numbers, or, in other words, vectors.

A NOTE ABOUT LOSS FUNCTIONS

The loss function boils-down the difference between desired and predicted, be that they are vectors, into a single number.

Let's now look at another variation of the MSE loss function:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(\hat{y}_{ij} - y_{ij})^2$$

If you're familiar with linear algebra, you'll recognize the inner sigma in the preceding equation as the square of the Euclidean distance. In fact, the MSE is sometimes referred to by these terms. Note that *N*, the size of your dataset, and *M*, the number of features the network has to predict, are constants. So, consider these as simple scaling factors that you can account for in other ways (like by scaling the learning rate). In a lot of use cases (including DL4J), the *M* is dropped and a division by two is added for mathematical convenience (which will become clearer in the context of its gradient in backpropagation). In the following equation, we see the version of MSE that the DL4J library uses for regression:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\hat{y}_{ij} - y_{ij})^2$$

IS MSE REALLY A CONVEX LOSS FUNCTION?

In a technical sense, the MSE is a convex loss function. However, when dealing with hidden layers in neural networks, the convex property no longer holds true, because we could have multiple parameter sets of values resulting in the same loss value.

OPTIMIZING MSE

Optimizing the MSE is equivalent to optimizing for the mean.

Other loss functions for regression

Although the MSE is used widely, it is quite sensitive to outliers, and this is something that you should consider when picking a loss function. When picking a stock to invest in, we want to take the outliers into account. But perhaps when buying a house we don't. In this case, what is of most interest is what most people would pay for it. In which case, we are more interested in the median and less so in the mean.

Mean absolute error loss

In a similar vein, an alternative to the MSE loss is the mean absolute error (MAE) loss, as shown in the following equation:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M} | \hat{y}_{ij} - y_{ij} |$$

This simply averages the absolute error over the entire dataset.

Mean squared log error loss

Another loss function used for regression is the mean squared log error (MSLE):

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\log \hat{y}_{ij} - \log y_{ij})^2$$

Mean absolute percentage error loss

Finally, we have mean absolute percentage error (MAPE) loss:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100 \times | \hat{y}_{ij} - y_{ij} |}{y_{ij}}$$

Regression loss function discussion

These are all valid choices, and there certainly is no single loss function that will outperform all other loss functions for every scenario. The MSE is very widely used and is a safe bet in most cases. So is the MAE. The MSLE and the MAPE are worth taking into consideration if our network is predicting outputs that vary largely in range. Suppose that a network is to predict two output variables: one in the range of [0, 10] and the other in the range of [0, 100]. In this case, the MAE and the MSE will penalize the error in the second output more significantly than the first. The MAPE makes it a relative error and therefore doesn’t discriminate based on the range. The MSLE squishes the range of all the outputs down, simply how 10 and 100 translate to 1 and 2 (in log base 10).

COMMON PRACTICE FOR REGRESSION IN NEURAL NETWORKS

Although MSLE and MAPE are approaches to handling large ranges, common practice with neural networks is to normalize inputs to a suitable range and use the MSE or MAE to optimize for either the mean or the median.

Loss Functions for Classification

We can build neural networks to bin data points into different categories; for example, fraud|not fraud. However, when building neural networks for classification problems, the focus is often on attaching probabilities to these classifications (30 percent fraud|70 percent not fraud). These differing scenarios require different loss functions.

Hinge loss

Hinge loss is the most commonly used loss function when the network must be optimized for a hard classification. For example, 0 = no fraud and 1 = fraud, which by convention is called a 0-1 classifier. The 0,1 choice is somewhat arbitrary and –1, 1 is also seen in lieu of 0–1. Hinge loss is also seen in a class of models called maximum-margin classification models (e.g., support vector machines, a somewhat distant cousin to neural networks).

Following is the equation for hinge loss when data points must be categorized as –1 or 1:

$$L(W,\ b) = \frac{1}{N}\sum_{i=1}^{N} max(0,\ 1 - y_{ij} \times \hat{y}_{ij})$$

The hinge loss is mostly used for binary classifications. There are extensions for multiclass classification (e.g., “one versus all,” “one versus one”) for the hinge loss that are not covered here.

HINGE LOSS IS A CONVEX FUNCTION

Note that like the MSE, the hinge loss is known to be a convex function.

Logistic loss

Logistic loss functions are used when probabilities are of greater interest than hard classifications. Great examples of these would be flagging potential fraud, with a human-in-the-loop solution or predicting the "probability of someone clicking on an ad," which can then be linked to a currency number.

Predicting valid probabilities means generating numbers between 0 and 1. Predicting valid probabilities also means making sure the probability of mutually exclusive outcomes should sum to one. For this reason, it is essential that the very last layer of a neural network used in classification is a softmax. Note that the sigmoid activation function also will give valid values between 0 and 1. However, you cannot use it for scenarios in which the outputs are mutually exclusive, because it does not model the dependencies between the output values.

Now that we have made sure our neural network will produce valid probabilities for the classes we have, we can dive headlong into the loss function and into the idea of what we should be optimizing here. We want to optimize for what is formally called the "maximum likelihood." In other words, we want to maximize the probability we predict for the correct class AND we want to do so for every single sample we have.

Let us consider the case in which our network predicts a probability for two classes, like the fraud and not fraud 0–1 classifier. Based on the notation described earlier, we can express our output for a given input X_i as $h(X_i)$ and $1 - h(X_i)$, given a set of weights and biases, *W* and *b*, expressing the probability of 1 and 0, respectively, as shown here:

$$P(y_i = 1 \mid X_i; \mathbf{W}, \mathbf{b}) = h_{\mathbf{W},\mathbf{b}}(X_i)$$

$$P(y_i = 0 \mid X_i; \mathbf{W}, \mathbf{b}) = 1 - h_{\mathbf{W},\mathbf{b}}(X_i)$$

We can combine these equations and express them as follows:

$$P(y_i \mid X_i; \mathbf{W}, \mathbf{b}) = (h_{\mathbf{W},\mathbf{b}}(X_i))^{y_i} \times (1 - h_{\mathbf{W},\mathbf{b}}(X_i))^{1-y_i}$$

The word “AND” from our colloquial definition of maximum likelihood in the context of probability should immediately ring a bell. The “AND” across all available samples translates to the product of the probabilities in question, as shown here:

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

Let’s now move on and take a look at negative log likelihood.

Negative log likelihood

For the sake of mathematical convenience, when dealing with the product of probabilities, it is customary to convert them to the log of the probabilities; hence, the product of the probabilities transforms to the sum of the log of the probabilities.

NEGATIVE LOG LIKELIHOOD AND MAXIMIZING PROBABILITY

The logarithm is a monotonically increasing function. Thus, minimizing the negative log likelihood is equivalent to maximizing the probability.

We also negate the expression so that the equation now corresponds to a "loss." So, the loss function in question becomes the following, commonly referred to as the negative log likelihood:

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

Extending the loss function from two classes to *M* classes gives us the following equation for logistic loss:

$$L(W, b) = -\sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \times \log \hat{y}_{i,j}$$

This is mathematically equivalent to what is called the cross-entropy between two probability distributions — in this case, what we predict and what we have observed under the same criteria. We will dive into this a bit more in the section that follows.

UNIFYING VIEWS ON LOSS FUNCTIONS

Note that we want to minimize our loss function, which we do by maximizing the likelihood, which we in turn do by minimizing the negative log likelihood. A mouthful? Yes, indeed.

Cross-entropy has its origin in information theory, whereas negative log likelihood for classification has its origin in statistical modeling. These two methods are mathematically the same, so although it might not matter which is used, it tends to confuse people.

Loss Functions for Reconstruction

This set of loss functions relates to what is called *reconstruction*. The idea is simple. A neural network is trained to recreate its input as closely as possible. So, why is this any different from memorizing the entire dataset? The key here is to tweak the scenario so that the network is forced to learn commonalities and features across the dataset.

In one approach, the number of parameters in the network is constrained such that the network is forced to compress the data and then re-create it. Another often-used approach is to corrupt the input with meaningless “noise” and train the network to ignore the noise and learn the data. Examples of these kinds of neural nets are restricted Boltzmann machines, autoencoders, and so on. These neural networks all use loss functions that are rooted in information theory.

Following is the equation for KL divergence:

$$D_{KL}(Y \mid\mid \hat{Y}) = -\sum_{i=1}^{N} Y_i \times \log\left(\frac{Y_i}{\hat{Y}_i}\right)$$

Although we briefly touched on the subject of cross-entropy a moment ago and justified taking the log of probabilities to turn our sum into product, we did not mention that taking the log of the probability puts us squarely in the field of informational theory and the concept of entropy.

DIFFERENT APPROACHES

Even though the negative log likelihood as framed above is mathematically equivalent to the cross-entropy, they are, however, grounded in different theoretical approaches.

Hyperparameters

In machine learning, we have both model parameters and parameters we tune to make networks train better and faster. These tuning parameters are called *hyperparameters*, and they deal with controlling optimization functions and model selection during training with our learning algorithm. In DL4J, we also refer to the optimization algorithms as updaters, because updates are synonymous with the steps the algorithm takes across the weight space to minimize error.

Hyperparameter selection focuses on ensuring that the model neither underfits nor overfits the training dataset, while learning the structure of the data as quickly as possible.

Learning Rate

The learning rate affects the amount by which you adjust parameters during optimization in order to minimize the error of neural network's guesses. It is a coefficient that scales the size of the steps (updates) a neural network takes to its parameter vector *x* as it crosses the loss function space.

During backpropagation we multiply the error gradient by the learning rate, and then update a connection weight's last iteration with the product to reach a new weight. The learning rate determines how much of the gradient we want to use for the algorithm's next step. A large error and steep gradient combine with the learning rate to produce a large step. As we approach minimal error and the gradient flattens out, the step size tends to shorten.

A large learning rate coefficient (e.g., 1) will make your parameters take leaps, and small ones (e.g., 0.00001) will make it inch along slowly. Large leaps will save time initially, but they can be disastrous if they lead us to overshoot our minimum. A learning rate too large oversteps the nadir, making the algorithm bounce back and forth on either side of the minimum without ever coming to rest.

In contrast, small learning rates should lead you eventually to an error minimum (it might be a local minimum rather than a global one), but they can take a very long time and add to the burden of an already computationally intensive process. Time matters when neural network training can take weeks on large datasets. If you can't wait another week for the results, choose a moderate learning rate (e.g., 0.1) and experiment with several others in the same ballpark to get the best speed and accuracy at once. Beyond setting a static learning rate, we'll look at ways to vary the learning rate over time to get the best of both worlds later in the book.

Regularization

Regularization helps with the effects of out-of-control parameters by using different methods to minimize parameter size over time.

CONTROLLING OVERFITTING IN MACHINE LEARNING

Regularization’s main purpose is to control overfitting in machine learning.

In mathematical notation, we see regularization represented by the coefficient lambda, controlling the trade-off between finding a good fit and keeping the value of certain feature weights low as the exponents on features increase.

Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller. Smaller-valued weights lead to simpler hypotheses, and simpler hypotheses are the most generalizable. Unregularized weights with several higher-order polynomials in the feature set tend to overfit the training set.

As the input training set size grows, the effect of regularization decreases and the parameters tend to increase in magnitude. This is appropriate, because an excess of features relative to training set examples leads to overfitting in the first place. Bigger data is the ultimate regularizer.

Momentum

Momentum helps the learning algorithm get out of spots in the search space where it would otherwise become stuck. In the errorscape, it helps the updater find the gulleys that lead toward the minima. Momentum is to the learning rate what the learning rate is to weights, and it helps us produce better quality models. We'll see the momentum hyperparameter in action in many later chapters.

Sparsity

The sparsity hyperparameter recognizes that for some inputs only a few features are relevant. For example, let's assume that a network can classify a million images. Any one of those images will be indicated by a limited number of features. But to effectively classify millions of images a network must be able to recognize considerably more features, many of which don't appear most of the time. An example of this would be how photos of sea urchins don't contain noses and hooves. This contrasts to how in submarine images the nose and hoof features will be 0.

The features that indicate sea urchins will be few and far between, in the vastness of the neural network's layers. That's a problem, because sparse features can limit the number of nodes that activate and impede a network's ability to learn. In response to sparsity, biases force neurons to activate and the activations stay around a mean that keeps the network from becoming stuck.

---

1 McCulloch and Pitts. 1943. "A logical calculus of the ideas immanent in nervous activity."

2 Li, Karpathy, "CS231n: Convolutional Neural Networks for Visual Recognition" (Course Notes). See *http://cs231n.stanford.edu* and *http://cs231n.github.io*