Deep Learning and Neural Network Introduction

Neural networks were one of the first machine learning models. Their popularity has fallen twice and is now on its third rise. Deep learning implies the use of neural networks. The "deep" in deep learning refers to a neural network with many hidden layers. Because neural networks have been around for so long, they have quite a bit of baggage. Researchers have created many different training algorithms, activation/transfer functions, and structures. This course is only concerned with the latest, most current state-of-the-art techniques for deep neural networks. I will not spend much time discussing the history of neural networks.

Neural networks accept input and produce output. The input to a neural network is called the feature vector. The size of this vector is always a fixed length. Changing the size of the feature vector usually means recreating the entire neural network. Though the feature vector is called a "vector," this is not always the case. A vector implies a 1D array. Later we will learn about convolutional neural networks (CNNs), which can allow the input size to change without retraining the neural network. Historically the input to a neural network was always 1D. However, with modern neural networks, you might see input data, such as:

- **1D vector** - Classic input to a neural network, similar to rows in a spreadsheet. Common in predictive modeling.
- **2D Matrix** - Grayscale image input to a CNN.
- **3D Matrix** - Color image input to a CNN.
- **nD Matrix** - Higher-order input to a CNN.

Before CNNs, programs either encoded images to an intermediate form or sent the image input to a neural network by merely squashing the image matrix into a long array by placing the image's rows side-by-side. CNNs are different as the matrix passes through the neural network layers.

Initially, this book will focus on 1D input to neural networks. However, later modules will focus more heavily on higher dimension input.

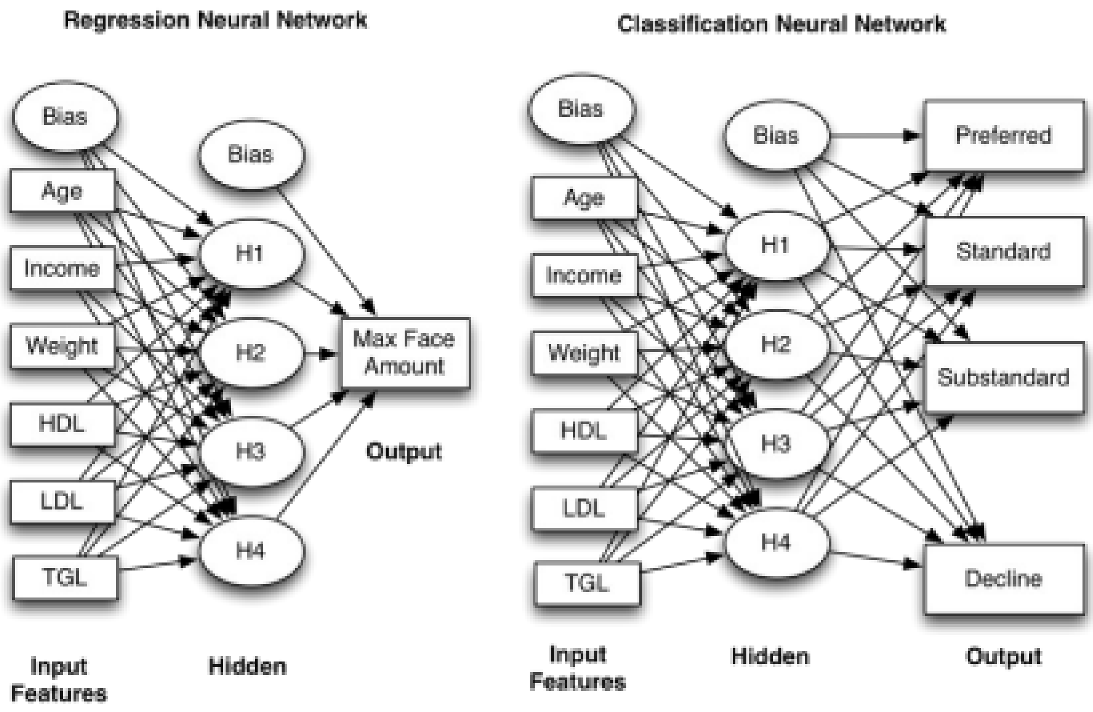
The term dimension can be confusing in neural networks. In the sense of a 1D input vector, dimension refers to how many elements are in that 1D array. For example, a neural network with ten input neurons has ten dimensions. However, now that we have CNNs, the input has dimensions. The input to the neural network will usually have 1, 2, or 3 dimensions. Four or more dimensions are unusual. You might have a 2D input to a neural network with 64x64 pixels. This configuration would result in 4,096 input neurons. This network is either 2D or 4,096D, depending on which dimensions you reference.

Classification or Regression

Like many models, neural networks can function in classification or regression:

- **Regression** - You expect a number as your neural network's prediction.
- **Classification** - You expect a class/category as your neural network's prediction.

A classification and regression neural network is shown by Figure 3.CLS-REG.
Figure 3.CLS-REG: Neural Network Classification and Regression

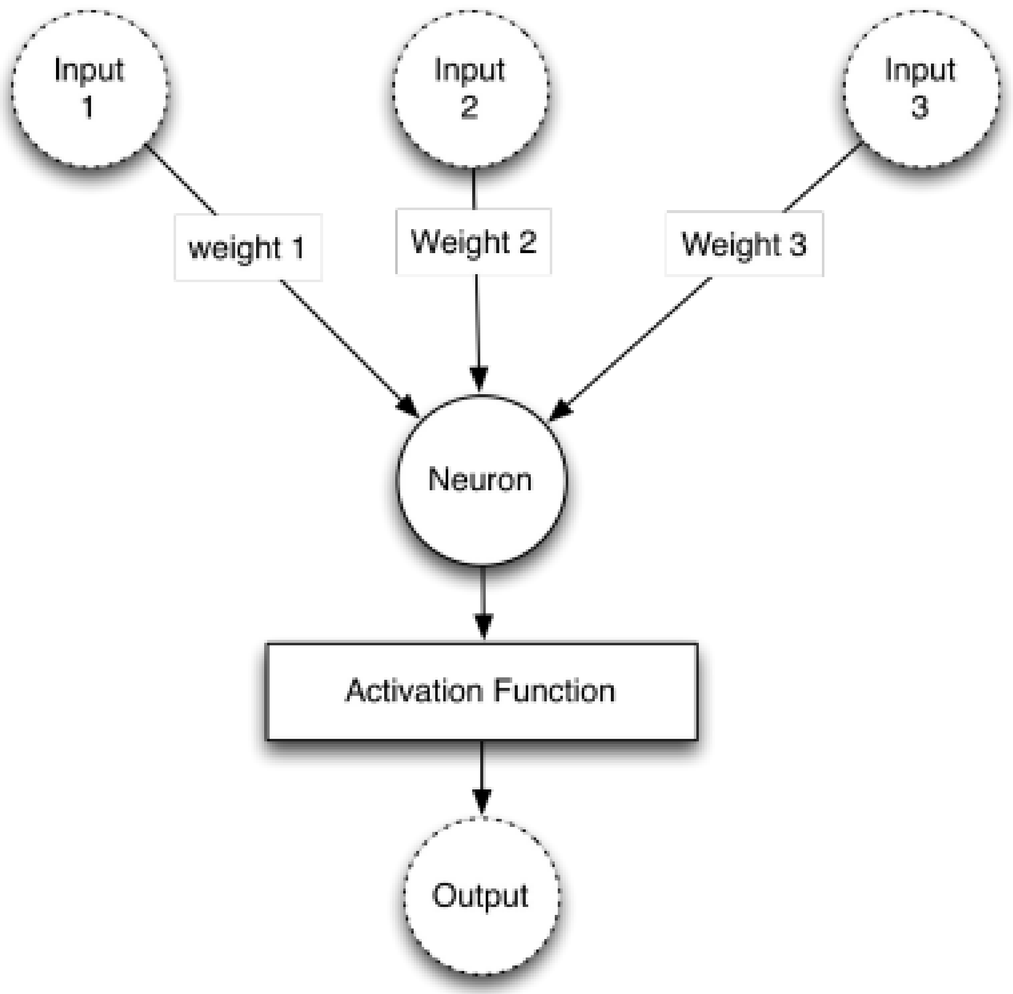


Notice that the output of the regression neural network is numeric, and the classification output is a class. Regression, or two-class classification, networks always have a single output. Classification neural networks have an output neuron for each category.

Neurons and Layers

Most neural network structures use some type of neuron. Many different neural networks exist, and programmers introduce experimental neural network structures. Consequently, it is not possible to cover every neural network architecture. However, there are some commonalities among neural network implementations. A neural network algorithm would typically be composed of individual, interconnected units, even though these units may or may not be called neurons. The name for a neural network processing unit varies among the literature sources. It could be called a node, neuron, or unit.

A diagram shows the abstract structure of a single artificial neuron in Figure 3.ANN.
Figure 3.ANN: An Artificial Neuron



The artificial neuron receives input from one or more sources that may be other neurons or data fed into the network from a computer program. This input is usually floating-point or binary. Often binary input is encoded to floating-point by representing true or false as 1 or 0. Sometimes the program also depicts the binary information using a bipolar system with true as one and false as -1.

An artificial neuron multiplies each of these inputs by a weight. Then it adds these multiplications and passes this sum to an activation function. Some neural networks do not use an activation function. The following equation summarizes the calculated output of a neuron:

$$f(x, w) = \phi\left(\sum_i (\theta_i \cdot x_i)\right)$$

In the above equation, the variables *x* and *θ* represent the input and weights of the neuron. The variable *i* corresponds to the number of weights and inputs. You must always have the same number of weights as inputs. The neural network multiplies each weight by its respective input and feeds the products of these multiplications into an activation function, denoted by the Greek letter *ϕ* (phi). This process results in a single output from the neuron. The above neuron has two inputs plus the bias as a third. This neuron might accept the following input feature vector:

[1,2)

Because a bias neuron is present, the program should append the value of one as

follows: [1 ,2 ,1)

The weights for a 3-input layer (2 real inputs + bias) will always have additional weight for the bias. A weight vector might be:

[0.1 , 0.2, 0.3)

To calculate the summation, perform the following:

$$0.1*1+0.2*2+0.3*1=0.8$$

The program passes a value of 0.8 to the ϕ (phi) function, representing the activation function.

The above figure shows the structure with just one building block. You can chain together many artificial neurons to build an artificial neural network (ANN). Think of the artificial neurons as building blocks for which the input and output circles are the connectors. Figure 3.ANN-3 shows an artificial neural network composed of three neurons:

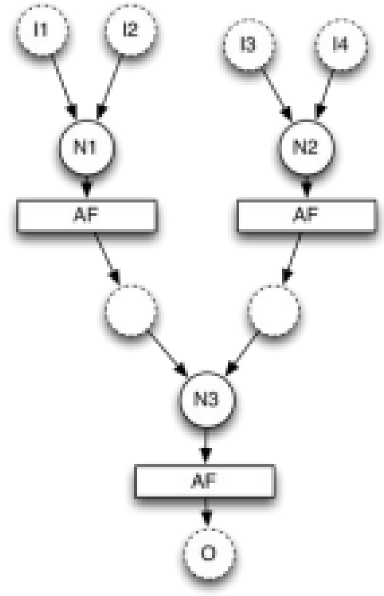


Figure 3.ANN-3: Three Neuron Neural Network

The above diagram shows three interconnected neurons. This representation is essentially this figure, minus a few inputs, repeated three times and then connected. It also has a total of four inputs and a single output. The output of neurons **N1** and **N2** feed **N3** to produce the output **O**.

To calculate the output for this network, we perform the previous equation three times. The first two times calculate **N1** and **N2**, and the third calculation uses the output of **N1** and **N2** to calculate **N3**.

Neural network diagrams do not typically show the detail seen in the previous figure. We can omit the activation functions and intermediate outputs to simplify the chart, resulting in Figure 3.SANN-3.

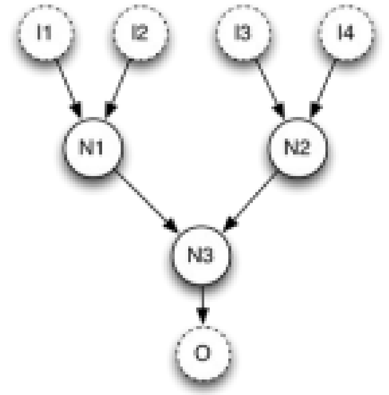
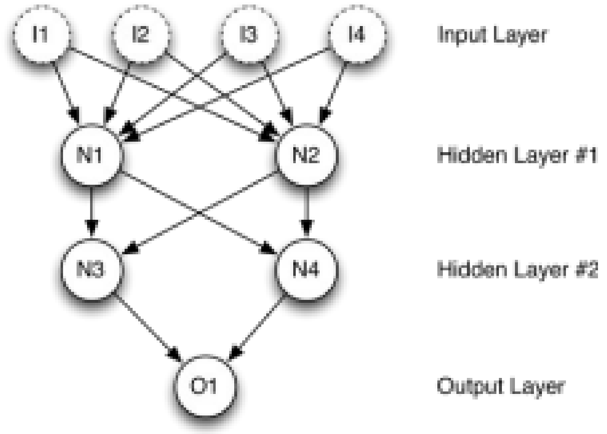


Figure 3.SANN-3: Three Neuron Neural Network

Looking at the previous figure, you can see two additional components of neural networks. First, consider the graph represents the inputs and outputs as abstract dotted line circles. The input and output could be parts of a more extensive neural network. However, the input and output are often a particular type of neuron that accepts data from the computer program using the neural network. The output neurons return a result to the program. This type of neuron is called an input neuron. We will discuss these neurons in the next section. This figure shows the neurons arranged in layers. The input neurons are the first layer, the **N1** and **N2** neurons create the second layer, the third layer contains **N3**, and the fourth layer has **O**. Most neural networks arrange neurons into layers.

The neurons that form a layer share several characteristics. First, every neuron in a layer has the same activation function. However, the activation functions employed by each layer may be different. Each of the layers fully connects to the next layer. In other words, every neuron in one layer has a connection to neurons in the previous layer. The former figure is not fully connected. Several layers are missing connections. For example, **I1** and **N2** do not connect. The next neural network in Figure 3.F-ANN is fully connected and has an additional layer.

Figure 3.F-ANN: Fully Connected Neural Network Diagram



In this figure, you see a fully connected, multilayered neural network. Networks such as this one will always have an input and output layer. The hidden layer structure determines the name of the network architecture. The network in this figure is a two-hidden-layer network. Most networks will have between zero and two hidden layers. Without implementing deep learning strategies, networks with more than two hidden layers are rare.

You might also notice that the arrows always point downward or forward from the input to the output. Later in this course, we will see recurrent neural networks that form inverted loops among the neurons. This type of neural network is called a feedforward neural network.

Types of Neurons

In the last section, we briefly introduced the idea that different types of neurons exist. Not every neural network will use every kind of neuron. It is also possible for a single neuron to fill the role of several different neuron types. Now we will explain all the neuron types described in the course.

There are usually four types of neurons in a neural network:

· **Input Neurons** - We map each input neuron to one element in the feature vector. · **Hidden Neurons** - Hidden neurons allow the neural network to be abstract and process the input into the output.
· **Output Neurons** - Each output neuron calculates one part of the output. · **Bias Neurons** - Work similar to the y-intercept of a linear equation.

We place each neuron into a layer:

· **Input Layer** - The input layer accepts feature vectors from the dataset. Input layers usually have a bias neuron.
· **Output Layer** - The output from the neural network. The output layer does not have a bias neuron.
· **Hidden Layers** - Layers between the input and output layers. Each hidden layer will usually have a bias neuron.

Input and Output Neurons

Nearly every neural network has input and output neurons. The input neurons accept data from the program for the network. The output neuron provides processed data from the network back to the program. The program will group these input and output neurons into separate layers called the input and output layers. The program normally represents the input to a neural network as an array or vector. The number of elements contained in the vector must equal the number of input neurons. For example, a neural network with three input neurons might accept the following input vector:

$$[0.5 , 0.75 , 0.2)$$

Neural networks typically accept floating-point vectors as their input. To be consistent, we will represent the output of a single output neuron network as a single-element vector. Likewise, neural networks will output a vector with a length equal to the number of output neurons. The output will often be a single value from a single output neuron.

Hidden Neurons

Hidden neurons have two essential characteristics. First, hidden neurons only receive input from other neurons, such as input or other hidden neurons. Second, hidden neurons only output to other neurons, such as output or other hidden neurons. Hidden neurons help the neural network understand the input and form the output. Programmers often group hidden neurons into fully connected hidden layers. However, these hidden layers do not directly process the incoming data or the eventual output.

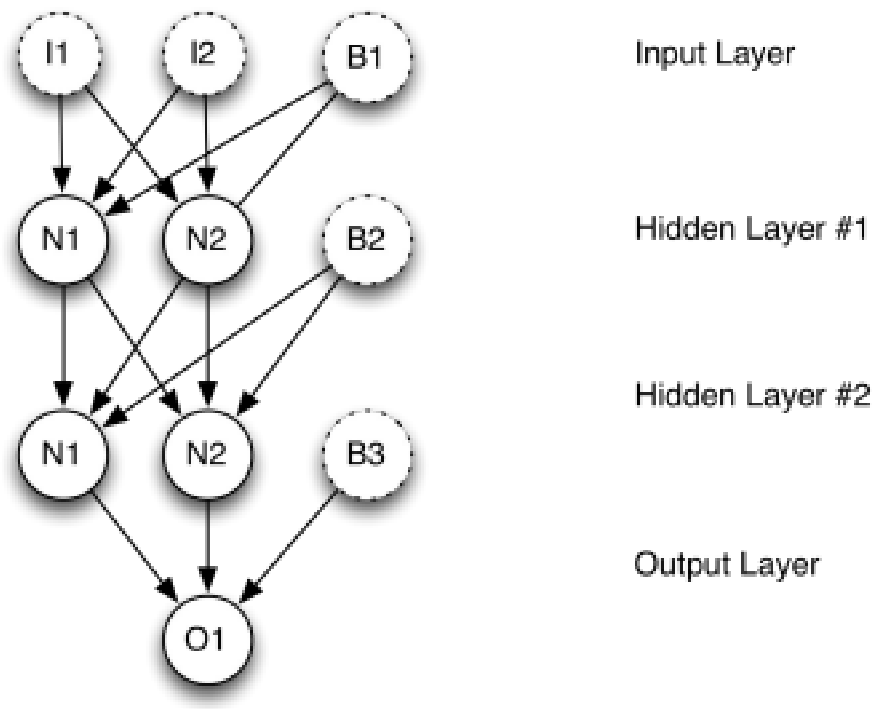
A common question for programmers concerns the number of hidden neurons in a network. Since the answer to this question is complex, more than one section of the course will include a relevant discussion of the number of hidden neurons. Before deep learning, researchers generally suggested that anything more than a single hidden layer is excessive. [Cite:hornik1989multilayer] Researchers have proven that a single-hidden-layer neural network can function as a universal approximator. In other words, this network should be able to learn to produce (or approximate) any output from any input as long as it has enough hidden neurons in a single layer.

Training refers to the process that determines good weight values. Before the advent of deep learning, researchers feared additional layers would lengthen training time or encourage overfitting. Both concerns are true; however, increased hardware speeds and clever techniques can mitigate these concerns. Before researchers introduced deep learning techniques, we did not have an efficient way to train a deep network, which is a neural network with many hidden layers. Although a single-hidden-layer neural network can theoretically learn anything, deep learning facilitates a more complex representation of patterns in the data.

Bias Neurons

Programmers add bias neurons to neural networks to help them learn patterns. Bias neurons function like an input neuron that always produces a value of 1. Because the bias neurons have a constant output of 1, they are not connected to the previous layer. The value of 1, called the bias activation, can be set to values other than 1. However, 1 is the most common bias activation. Not all neural networks have bias neurons. Figure 3.BIAS shows a single-hidden-layer neural network with bias neurons:

Figure 3.BIAS: Neural Network with Bias Neurons



The above network contains three bias neurons. Except for the output layer, every level includes a single bias neuron. Bias neurons allow the program to shift the output of an activation function. We will see precisely how this shifting occurs later in the module when discussing activation functions.

Other Neuron Types

The individual units that comprise a neural network are not always called neurons. Researchers will sometimes refer to these neurons as nodes, units, or summations. You will almost always construct neural networks of weighted connections between these units.

Why are Bias Neurons Needed?

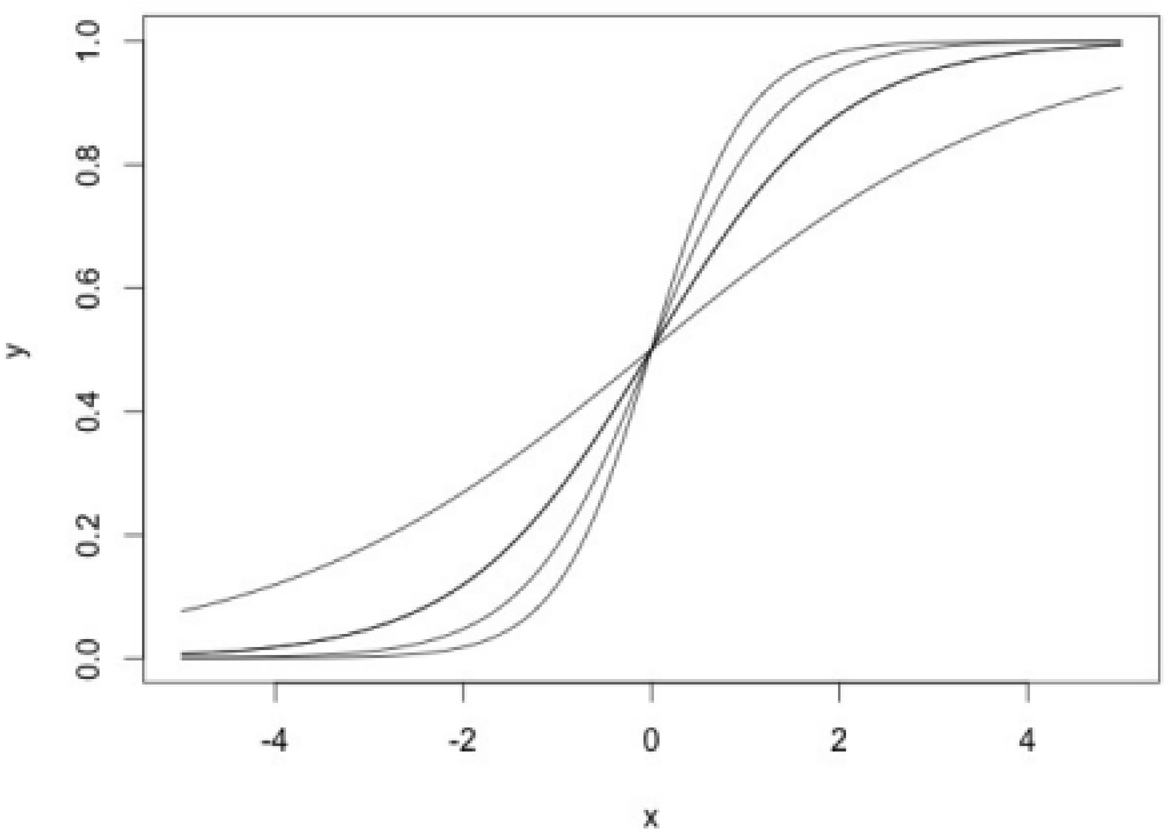
The activation functions from the previous section specify the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider the following equation. It represents a single-input sigmoid activation neural network.

$$f(x, w, b) = \frac{1}{1+e^{-(wx+b)}}$$

The x variable represents the single input to the neural network. The w and b variables specify the weight and bias of the neural network. The above equation combines the weighted sum of the inputs and the sigmoid activation function. For this section, we will consider the sigmoid function because it demonstrates a bias neuron's effect.
The weights of the neuron allow you to adjust the slope or shape of the activation

function. Figure 3.A-WEIGHT shows the effect on the output of the sigmoid activation function if the weight is varied:

Figure 3.A-WEIGHT: Neuron Weight Shifting



The above diagram shows several sigmoid curves using the following parameters: $f(x, 0.5, 0.0)$

$f(x, 1.0, 0.0)$

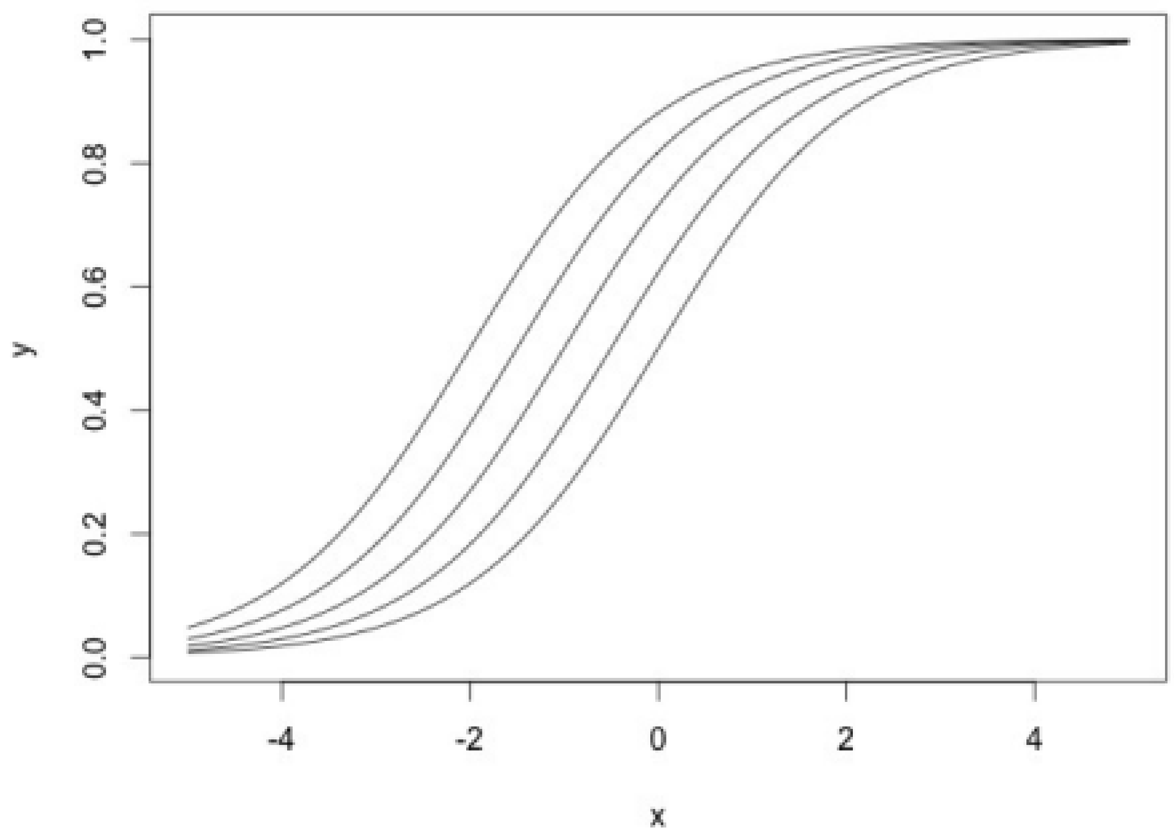
$f(x, 1.5, 0.0)$

$f(x, 2.0, 0.0)$

We did not use bias to produce the curves, which is evident in the third parameter of 0 in each case. Using four weight values yields four different sigmoid curves in the above figure. No matter the weight, we always get the same value of 0.5 when x is 0 because all curves hit the same point when x is 0. We might need the neural network to produce other values when the input is near 0.5.

Bias does shift the sigmoid curve, which allows values other than 0.5 when x is near 0. Figure 3.A-BIAS shows the effect of using a weight of 1.0 with several different biases:

Figure 3.A-BIAS: Neuron Bias Shifting



The above diagram shows several sigmoid curves with the following parameters: $f(x, 1.0, 1.0)$

$f(x, 1.0, 0.5)$

$f(x, 1.0, 1.5)$

$f(x, 1.0, 2.0)$

We used a weight of 1.0 for these curves in all cases. When we utilized several different biases, sigmoid curves shifted to the left or right. Because all the curves merge at the top right or bottom left, it is not a complete shift.

When we put bias and weights together, they produced a curve that created the necessary output. The above curves are the output from only one neuron. In a complete network, the output from many different neurons will combine to produce intricate output patterns.

Modern Activation Functions

Activation functions, also known as transfer functions, are used to calculate the output of each layer of a neural network. Historically neural networks have used a hyperbolic

tangent,
sigmoid/logistic, or linear activation function. However, modern deep neural networks primarily make use of the following activation functions:

- **Rectified Linear Unit (ReLU)** – Used for the output of hidden layers. [Cite:glorot2011deep]
- **Softmax** – Used for the output of classification neural networks.
- **Linear** – Used for the output of regression neural networks (or 2-class classification).

Linear Activation Function

The most basic activation function is the linear function because it does not change the neuron output. The following equation 1.2 shows how the program typically implements a linear activation function:

$$\phi(x)=x$$

As you can observe, this activation function simply returns the value that the neuron inputs passed to it. Figure 3.LIN shows the graph for a linear activation function:

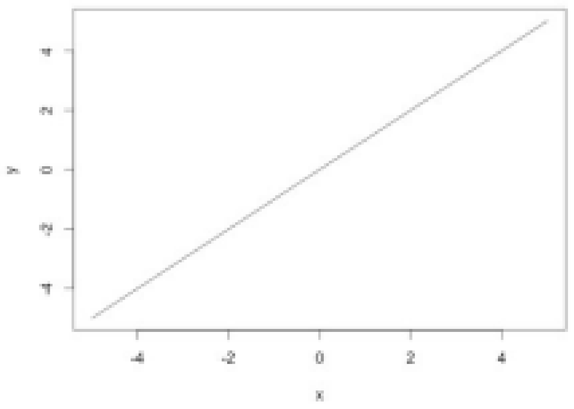


Figure 3.LIN: Linear Activation Function

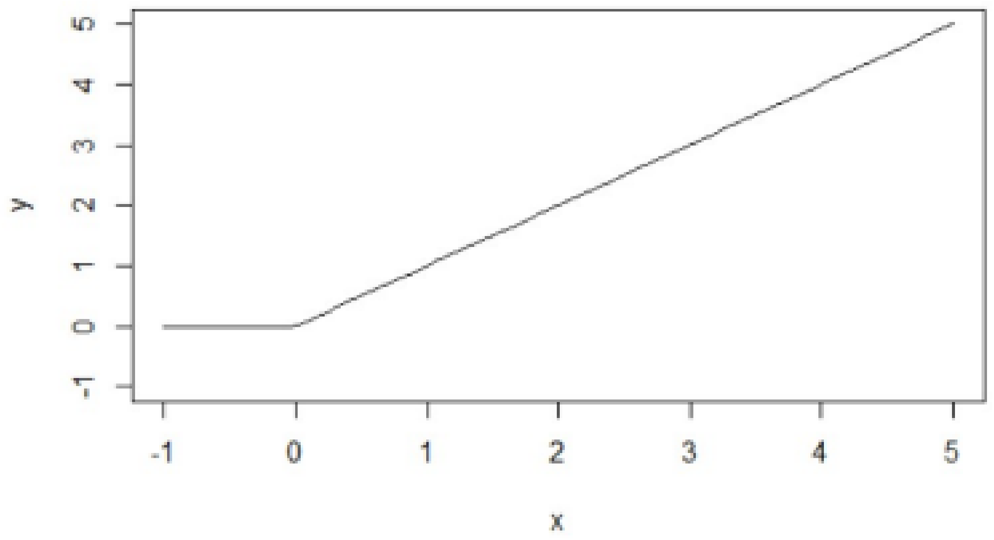
Regression neural networks, which learn to provide numeric values, will usually use a linear activation function on their output layer. Classification neural networks, which determine an appropriate class for their input, will often utilize a softmax activation function for their output layer.

Rectified Linear Units (ReLU)

Since its introduction, researchers have rapidly adopted the rectified linear unit (ReLU). [Cite:nair2010rectified] Before the ReLU activation function, the programmers generally regarded the hyperbolic tangent as the activation function of choice. Most current research now recommends the ReLU due to superior training results. As a result, most neural networks should utilize the ReLU on hidden layers and either softmax or linear on the output layer. The following equation shows the straightforward ReLU function:

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

Figure 3.RELU shows the graph of the ReLU activation function:
Figure 3.RELU: Rectified Linear Units (ReLU)



Most current research states that the hidden layers of your neural network should use the ReLU activation.

Softmax Activation Function

The final activation function that we will examine is the softmax activation function. Along with the linear activation function, you can usually find the softmax function in the output layer of a neural network. Classification neural networks typically employ the softmax function. The neuron with the highest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the neural network's output to represent the probability that the input falls into each of the classes. The neuron's outputs are numeric values without the softmax, with the highest indicating the winning class.

To see how the program uses the softmax activation function, we will look at a typical neural network classification problem. The iris data set contains four measurements for 150 different iris flowers. Each of these flowers belongs to one of three species of iris. When you provide the measurements of a flower, the softmax function allows the neural network to give you the probability that these measurements belong to each of the three species. For example, the neural network might tell you that there is an 80% chance that the iris is setosa, a 15% probability that it is virginica, and only a 5% probability of versicolor. Because these are probabilities, they must add up to 100%. There could not be an 80% probability of setosa, a 75% probability of virginica, and a 20% probability of versicolor—this type of result would be nonsensical.

To classify input data into one of three iris species, you will need one output neuron for each species. The output neurons do not inherently specify the probability of each of the three species. Therefore, it is desirable to provide probabilities that sum to 100%. The neural network will tell you the likelihood of a flower being each of the three species. To get the probability, use the softmax function in the following equation:

$$\phi_i(x) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

In the above equation, *i* represents the index of the output neuron (ϕ) that the program is calculating, and *j* represents the indexes of all neurons in the group/level. The variable *x* designates the array of output neurons. It's important to note that the program calculates the softmax activation differently than the other activation functions in this module. When softmax is the activation function, the output of a single neuron is dependent on the other output neurons.

To see the softmax function in operation, refer to this Softmax example website.

Consider a trained neural network that classifies data into three categories: the three iris species. In this case, you would use one output neuron for each of the target classes. Consider if the neural network were to output the following:

- **Neuron 1**: setosa: 0.9
- **Neuron 2**: versicolour: 0.2
- **Neuron 3**: virginica: 0.4

The above output shows that the neural network considers the data to represent a setosa iris. However, these numbers are not probabilities. The 0.9 value does not represent a 90% likelihood of the data representing a setosa. These values sum to 1.5. For the program to treat them as probabilities, they must sum to 1.0. The output vector for this neural network is the following:

$$[0.9, 0.2, 0.4)$$

If you provide this vector to the softmax function it will return the following

vector: $[0.47548495534876745, 0.2361188410001125, 0.28839620365112)$

The above three values do sum to 1.0 and can be treated as probabilities. The likelihood of the data representing a setosa iris is 48% because the first value in the vector rounds to 0.48 (48%). You can calculate this value in the following manner:

$$sum = \exp(0.9) + \exp(0.2) + \exp(0.4) = 5.17283056695839$$

$$j_0 = \exp(0.9) / sum = 0.47548495534876745$$

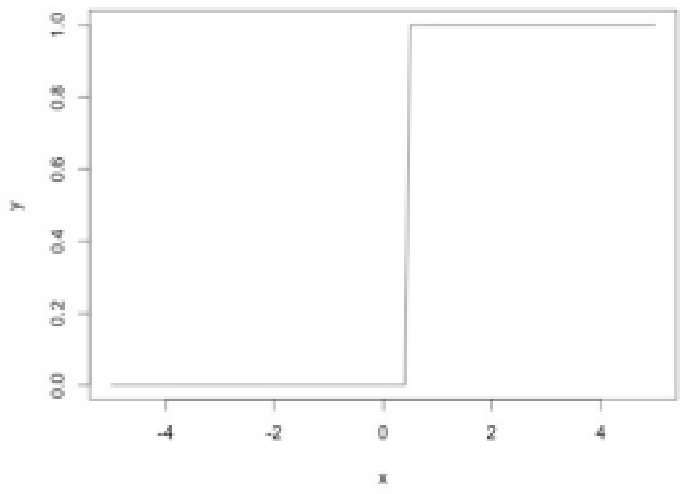
$$j_1 = \exp(0.2) / sum = 0.2361188410001125$$

$$j_2 = \exp(0.4) / sum = 0.28839620365112$$

Step Activation Function

The step or threshold activation function is another simple activation function. Neural networks were initially called perceptrons. McCulloch & Pitts (1943) introduced the original perceptron and used a step activation function like the following equation:[Cite:mcculloch1943logical] The step activation is 1 if x>=0.5, and 0 otherwise. This equation outputs a value of 1.0 for incoming values of 0.5 or higher and 0 for all other values. Step functions, also known as threshold functions, only return 1 (true) for values above the specified threshold, as seen in Figure 3.STEP.

Figure 3.STEP: Step Activation Function

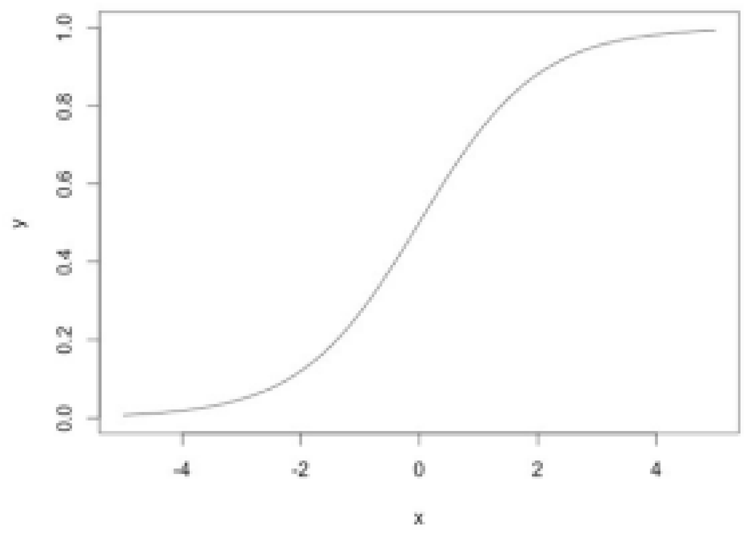


Sigmoid Activation Function

The sigmoid or logistic activation function is a common choice for feedforward neural networks that need to output only positive numbers. Despite its widespread use, the hyperbolic tangent or the rectified linear unit (ReLU) activation function is usually a more suitable choice. We introduce
the ReLU activation function later in this module. The following equation shows the sigmoid activation function:

$$\phi(x) = \frac{1}{1+e^{-x}}$$

Use the sigmoid function to ensure that values stay within a relatively small range, as seen in Figure 3.SIGMOID:
Figure 3.SIGMOID: Sigmoid Activation Function



As you can see from the above graph, we can force values to a range. Here, the function compressed values above or below 0 to the approximate range between 0 and 1.

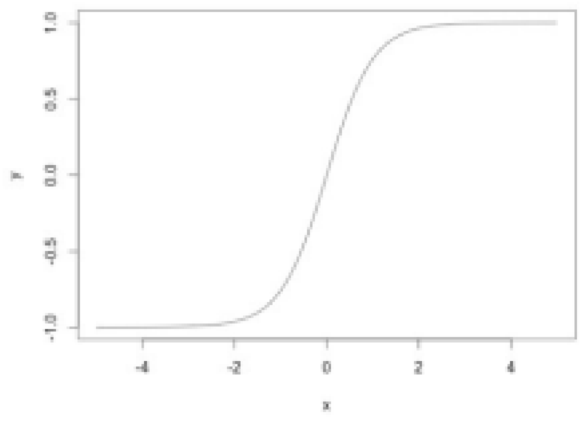
Hyperbolic Tangent Activation Function

The hyperbolic tangent function is also a prevalent activation function for neural networks that must output values between −1 and 1. This activation function is simply the hyperbolic tangent (tanh) function, as shown in the following equation:

$$\phi(x) = \tanh(x)$$

The graph of the hyperbolic tangent function has a similar shape to the sigmoid activation function, as seen in Figure 3.HTAN.

Figure 3.HTAN: Hyperbolic Tangent Activation Function

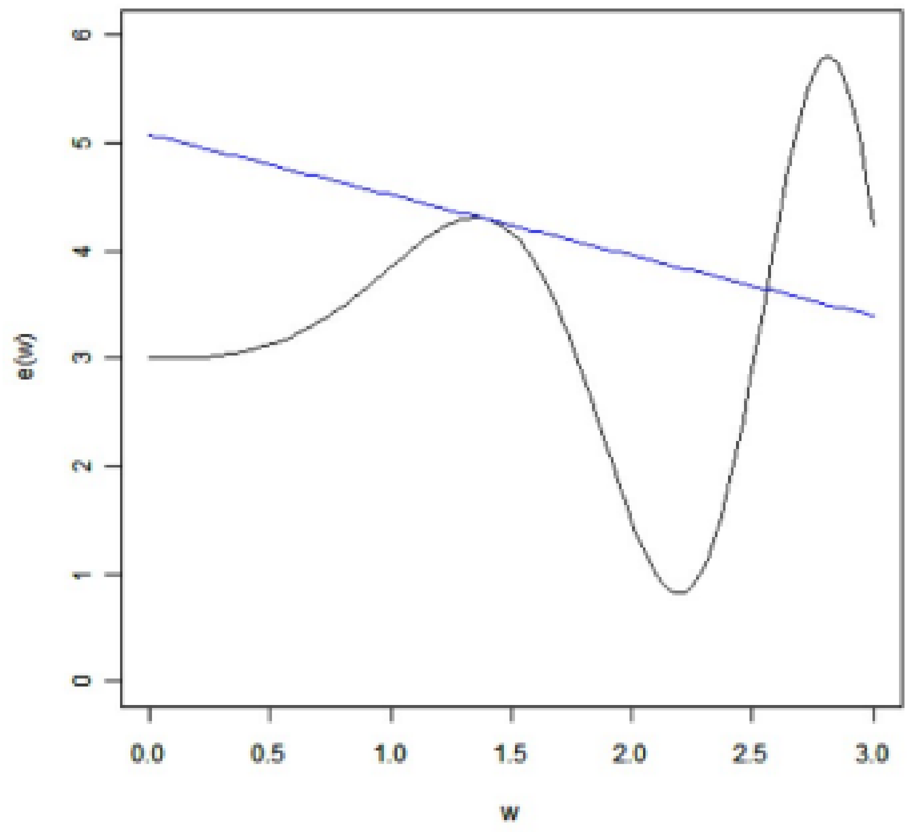


The hyperbolic tangent function has several advantages over the sigmoid activation function.

Why ReLU?

Why is the ReLU activation function so popular? One of the critical improvements to neural networks makes deep learning work. [Cite:nair2010rectified] Before deep learning, the sigmoid activation function was prevalent. We covered the sigmoid activation function earlier in this module. Frameworks like Keras often train neural networks with gradient descent. For the neural network to use gradient descent, it is necessary to take the derivative of the activation function. The program must derive partial derivatives of each of the weights for the error function. Figure 3.DERV shows a derivative, the instantaneous rate of change.

Figure 3.DERV: Derivative

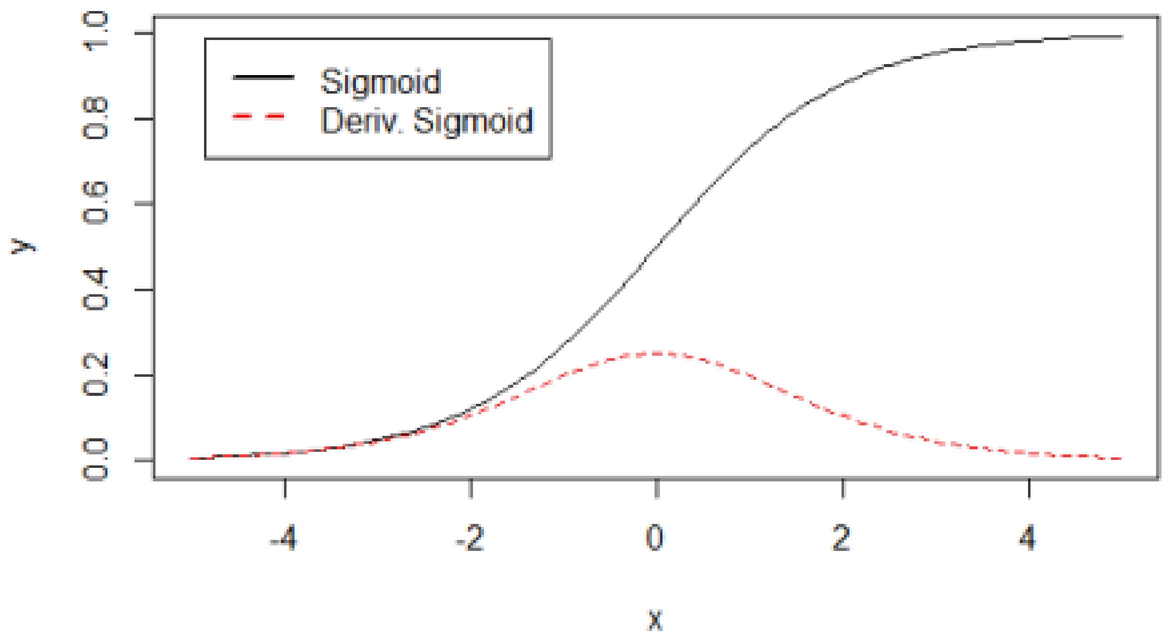


The derivative of the sigmoid function is given here:

$$\phi'(x)=\phi(x)(1-\phi(x))$$

Textbooks often give this derivative in other forms. We use the above form for computational efficiency. To see how we determined this derivative, refer to the following article.

We present the graph of the sigmoid derivative in Figure 3.SDERV.
Figure 3.SDERV: Sigmoid Derivative



The derivative quickly saturates to zero as *x* moves from zero. This is not a problem for the derivative of the ReLU, which is given here:

$$\phi'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Early Stopping in Keras to Prevent Overfitting

It can be difficult to determine how many epochs to cycle through to train a neural network. Overfitting will occur if you train the neural network for too many epochs, and the neural network will not perform well on new data, despite attaining a good accuracy on the training set. Overfitting occurs when a neural network is trained to the point that it begins to memorize rather than generalize, as demonstrated in Figure 3.OVER.

Figure 3.OVER: Training vs. Validation Error for Overfitting

It is important to segment the original dataset into several datasets:

- **Training Set**
- **Validation Set**
- **Holdout Set**

You can construct these sets in several different ways. The following programs demonstrate some of these.

The first method is a training and validation set. We use the training data to train the neural network until the validation set no longer improves. This attempts to stop at a near-optimal training point. This method will only give accurate "out of sample" predictions for the validation set; this is usually 20% of the data. The predictions for the training data will be overly optimistic, as these were the data that we used to train the neural network. Figure 3.VAL demonstrates how we divide the dataset.

Figure 3.VAL: Training with a Validation Set

There are a number of parameters that are specified to the **EarlyStopping** object.

- **min_delta** This value should be kept small. It simply means the minimum change in error to be registered as an improvement. Setting it even smaller will not likely have a great deal of impact.
- **patience** How long should the training wait for the validation error to improve?
- **verbose** How much progress information do you want?
- **mode** In general, always set this to "auto". This allows you to specify if the error should be minimized or maximized. Consider accuracy, where higher numbers are desired vs log-loss/RMSE where lower numbers are desired.
- **restore_best_weights** This should always be set to true. This restores the weights to the values they were at when the validation set is the highest. Unless you are manually tracking the weights yourself (we do not use this technique in this course), you should have Keras perform this step for you.

As you can see from above, the entire number of requested epochs were not used. The neural network training stopped once the validation set no longer improved.

Extracting Weights and Manual Network Calculation

Weight Initialization

The weights of a neural network determine the output for the neural network. The training process can adjust these weights, so the neural network produces useful output. Most neural network training algorithms begin by initializing the weights to a random state. Training then progresses through iterations that continuously improve the weights to produce better output.

The random weights of a neural network impact how well that neural network can be trained. If a neural network fails to train, you can remedy the problem by simply restarting with a new set of random weights. However, this solution can be frustrating when you are experimenting with
the architecture of a neural network and trying different combinations of hidden layers and neurons. If you add a new layer, and the network's performance improves, you must ask

yourself
if this improvement resulted from the new layer or from a new set of weights. Because of this uncertainty, we look for two key attributes in a weight initialization algorithm:

- How consistently does this algorithm provide good weights?
- How much of an advantage do the weights of the algorithm provide?

One of the most common yet least practical approaches to weight initialization is to set the weights to random values within a specific range. Numbers between −1 and +1 or −5 and +5 are often the choice. If you want to ensure that you get the same set of random weights each time, you should use a seed. The seed specifies a set of predefined random weights to use. For example, a seed of 1000 might produce random weights of 0.5, 0.75, and 0.2. These values are still random; you cannot predict them, yet you will always get these values when you choose a seed of 1000. Not all seeds are created equal. One problem with random weight initialization is that the random weights created by some seeds are much more difficult to train than others. The weights can be so bad that training is impossible. If you cannot train a neural network with a particular weight set, you should generate a new set of weights using a different seed.

Because weight initialization is a problem, considerable research has been around it. By default, Keras uses the Xavier weight initialization algorithm, introduced in 2006 by Glorot & Bengio[Cite:glorot2010understanding], produces good weights with reasonable consistency. This relatively simple algorithm uses normally distributed random numbers.

To use the Xavier weight initialization, it is necessary to understand that normally distributed random numbers are not the typical random numbers between 0 and 1 that most programming languages generate. Normally distributed random numbers are centered on a mean (μ, mu) that is typically 0. If 0 is the center (mean), then you will get an equal number of random numbers above and below 0. The next question is how far these random numbers will venture from 0. In theory, you could end up with both positive and negative numbers close to the maximum positive and negative ranges supported by your computer. However, the reality is that you will more likely see random numbers that are between 0 and three standard deviations from the center.

The standard deviation (σ, sigma) parameter specifies the size of this standard deviation. For example, if you specified a standard deviation of 10, you would mainly see random numbers between −30 and +30, and the numbers nearer to 0 have a much higher probability of being selected.

The above figure illustrates that the center, which in this case is 0, will be generated with a 0.4 (40%) probability. Additionally, the probability decreases very quickly beyond −2 or +2 standard deviations. By defining the center and how large the standard deviations are, you can control the range of random numbers that you will receive.

The Xavier weight initialization sets all weights to normally distributed random numbers. These weights are always centered at 0; however, their standard deviation varies depending on how many connections are present for the current layer of weights. Specifically, Equation 4.2 can determine the standard deviation:

$$Var(W)=\frac{2}{n_{in}+n_{out}}$$

The above equation shows how to obtain the variance for all weights. The square root of the variance is the standard deviation. Most random number generators accept a standard deviation rather than a variance. As a result, you usually need to take the square root of the above equation. Figure 3.XAVIER shows how this algorithm might initialize one layer.

Figure 3.XAVIER: Xavier Weight Initialization Xavier Weight

Initialization We complete this process for each layer in the neural

network.

Manual Neural Network Calculation

This section will build a neural network and analyze it down the individual weights. We will train a simple neural network that learns the XOR function. It is not hard to hand-code the neurons to provide an XOR function; however, we will allow Keras for simplicity to train this network for us. The neural network is small, with two inputs, two hidden neurons, and a single output. We will
use 100K epochs on the ADAM optimizer. This approach is overkill, but it gets the result, and our focus here is not on tuning.

Multiclass Classification with ROC and AUC

The output of modern neural networks can be of many different forms. However, classically, neural network output has typically been one of the following:

· **Binary Classification** - Classification between two possibilities (positive and negative). Common in medical testing, does the person has the disease (positive) or not (negative). · **Classification** - Classification between more than 2. The iris dataset (3-way classification).
· **Regression** - Numeric prediction. How many MPG does a car get? (covered in next

video) We will look at some visualizations for all three in this section.

It is important to evaluate the false positives and negatives in the results produced by a neural network. We will now look at assessing error for both classification and regression neural networks.

Binary Classification and ROC Charts

Binary classification occurs when a neural network must choose between two options: true/false, yes/no, correct/incorrect, or buy/sell. To see how to use binary classification, we will consider a classification system for a credit card company. This system will either "issue a credit card" or "decline a credit card." This classification system must decide how to respond to a new potential customer.

When you have only two classes that you can consider, the objective function's score is the number of false-positive predictions versus the number of false negatives. False negatives and false positives are both types of errors, and it is essential to understand the difference. For the previous example, issuing a credit card would be positive. A false positive occurs when a model decides to issue a credit card to someone who will not make payments as agreed. A false negative happens when a model denies a credit card to someone who would have made payments as agreed.

Because only two options exist, we can choose the mistake that is the more serious type of error, a false positive or a false negative. For most banks issuing credit cards, a false positive is worse than a false negative. Declining a potentially good credit card holder is better than accepting a credit card holder who would cause the bank to undertake expensive collection activities.

ROC curves can be a bit confusing. However, they are prevalent in analytics. It is essential to know how to read them. Even their name is confusing. Do not worry about their name; the receiver operating characteristic curve (ROC) comes from electrical engineering (EE).

Binary classification is common in medical testing. Often you want to diagnose if someone has a disease. This diagnosis can lead to two types of errors, known as false positives and false negatives:

- **False Positive** - Your test (neural network) indicated that the patient had the disease; however, the patient did not.
- **False Negative** - Your test (neural network) indicated that the patient did not have the disease; however, the patient did have the disease.
- **True Positive** - Your test (neural network) correctly identified that the patient had the disease.
- **True Negative** - Your test (neural network) correctly identified that the patient did not have the disease.

Figure 4.ETYP shows you these types of errors.

Figure 4.ETYP: Type of Error

Neural networks classify in terms of the probability of it being positive. However, at what possibility do you give a positive result? Is the cutoff 50%? 90%? Where you set, this cutoff is called the threshold. Anything above the cutoff is positive; anything below is negative. Setting this cutoff allows the model to be more sensitive or specific:

More info on Sensitivity vs. Specificity: Khan Academy

Multiclass Classification Error Metrics

If you want to predict more than one outcome, you will need more than one output neuron. Because a single neuron can predict two results, a neural network with two output neurons is somewhat rare. If there are three or more outcomes, there will be three or more output neurons. The following sections will examine several metrics for evaluating classification error. We will assess the following classification neural network.

Calculating Classification Accuracy

Accuracy is the number of rows where the neural network correctly predicted the target class. Accuracy is only used for classification, not regression.

$$accuracy=\frac{c}{N}$$

Where c is the number correct and N is the size of the evaluated set (training or validation). Higher accuracy numbers are desired.

As we just saw, by default, Keras will return the percent probability for each class. We can change these prediction probabilities into the actual iris predicted with **argmax**.

Calculating Classification Log Loss

Accuracy is like a final exam with no partial credit. However, neural networks can predict a probability of each of the target classes. Neural networks will give high probabilities to predictions that are more likely. Log loss is an error metric that penalizes confidence in wrong answers. Lower log loss values are desired.

Log loss is calculated as follows:

$$\text{log loss}=-\frac{1}{N}\sum_{i=1}^{N}\left(y_i\log(\hat{y}_i)+(1-y_i)\log(1-\hat{y}_i)\right)$$

You should use this equation only as an objective function for classifications that have two outcomes. The variable y-hat is the neural network's prediction, and the variable y is the known correct answer. In this case, y will always be 0 or 1. The training data have no probabilities. The neural network classifies it either into one class (1) or the other (0).

The variable N represents the number of elements in the training set the number of questions in the test. We divide by N because this process is customary for an average. We also begin the equation with a negative because the log function is always negative over the domain 0 to 1. This negation allows a positive score for the training to minimize.

You will notice two terms are separated by the addition (+). Each contains a log function. Because y will be either 0 or 1, then one of these two terms will cancel out to 0. If y is 0, then the first term will reduce to 0. If y is 1, then the second term will be 0.

If your prediction for the first class of a two-class prediction is y-hat, then your prediction for the second class is 1 minus y-hat. Essentially, if your prediction for class A is 70% (0.7), then your prediction for class B is 30% (0.3). Your score will increase by the log of your prediction for the correct class. If the neural network had predicted 1.0 for class A, and the correct answer was A, your score would increase by log (1), which is 0. For log loss, we seek a low score, so a correct answer results in 0. Some of these log values for a neural network's probability estimate for the correct class:

- -log(1.0) = 0
- -log(0.95) = 0.02
- -log(0.9) = 0.05
- -log(0.8) = 0.1
- -log(0.5) = 0.3
- -log(0.1) = 1
- -log(0.01) = 2
- -log(1.0e-12) = 12
- -log(0.0) = negative infinity

As you can see, giving a low confidence to the correct answer affects the score the most. Because log (0) is negative infinity, we typically impose a minimum value. Of course, the above log values are for a single training set element. We will average the log values for the entire training set.

The log function is useful to penalizing wrong answers.

Confusion Matrix

A confusion matrix shows which predicted classes are often confused for the other classes. The vertical axis (y) represents the true labels and the horizontal axis (x) represents the predicted labels. When the true label and predicted label are the same, the highest values occur down the diagonal extending from the upper left to the lower right. The other values, outside the diagonal, represent incorrect predictions. For example, in the confusion matrix below, the value in row 2, column 1 shows how often the predicted value A occurred when it should have been B.

Mean Square Error

The mean square error (MSE) is the sum of the squared differences between the prediction ($\hat{y}$) and the expected (y). MSE values are not of a particular unit. If an MSE value has decreased for a model, that is good. However, beyond this, there is not much more you can determine. We seek to achieve low MSE values. The following equation demonstrates how to calculate MSE.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The following code calculates the MSE on the predictions from the neural

network.

Root Mean Square Error

The root mean square (RMSE) is essentially the square root of the MSE. Because of this, the RMSE error is in the same units as the training data outcome. We desire Low RMSE values. The following equation calculates RMSE.

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

Training Neural Networks

Backpropagation [Cite:rumelhart1986learning] is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams introduced backpropagation, and it remains
popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Researchers have extended classic backpropagation and modified to give rise to many different training algorithms. This section will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and end the chapter with stochastic gradient descent (SGD).

Backpropagation is the primary means of determining a neural network's weights during training. Backpropagation works by calculating a weight change amount (v_t) for every weight(θ, theta) in the neural network. This value is subtracted from every weight by the following equation:

$$\theta_t = \theta_{t-1} - v_t$$

We repeat this process for every iteration(t). The training algorithm determines how we calculate the weight change. Classic backpropagation calculates a gradient (∇, nabla) for every weight in the neural network for the neural network's error function (J). We scale the gradient by a learning rate (η, eta).

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1})$$

The learning rate is an important concept for backpropagation training. Setting the learning rate can be complex:

- Too low a learning rate will usually converge to a reasonable solution; however, the process will be prolonged.
- Too high of a learning rate will either fail outright or converge to a higher error than a better learning rate.

Common values for learning rate are: 0.1, 0.01, 0.001, etc.

Backpropagation is a gradient descent type, and many texts will use these two terms interchangeably. Gradient descent refers to calculating a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will indicate how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the expected output. We can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

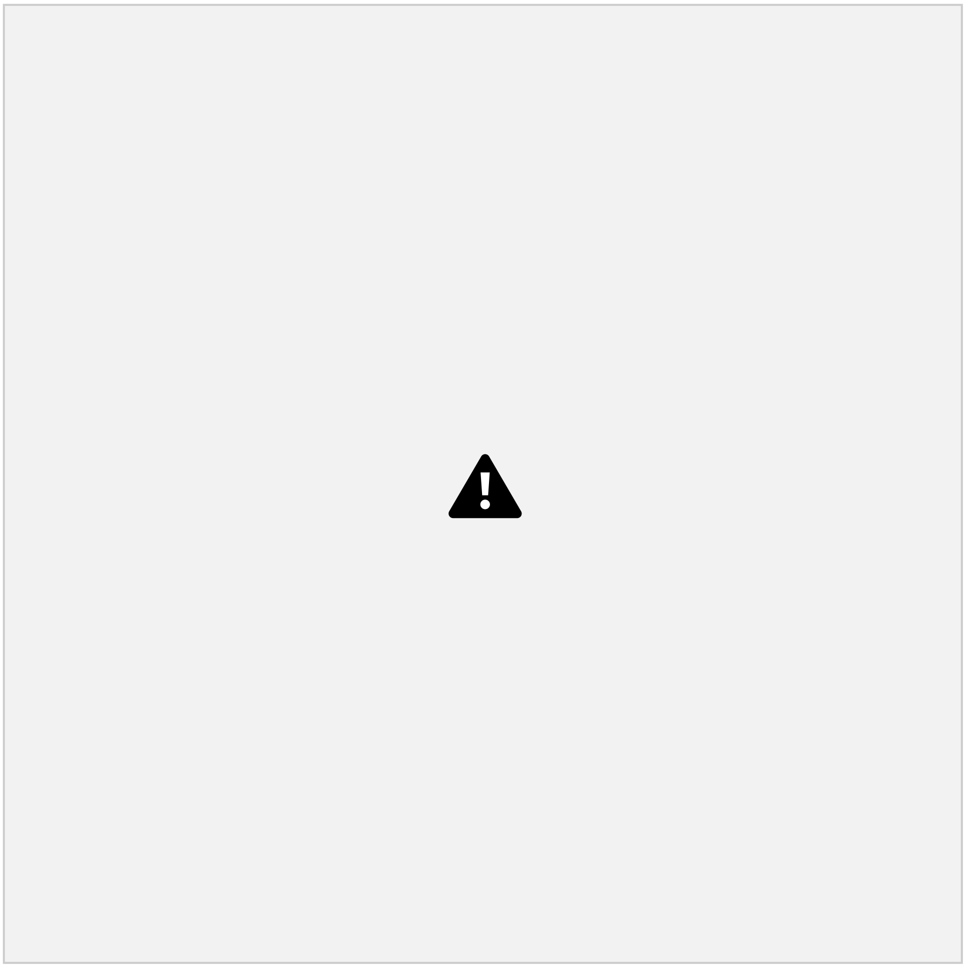
The gradient is the partial derivative of each weight in the neural network concerning the error function. Each weight has a gradient that is the slope of the error function. Weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between
the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

· Zero gradient - The weight does not contribute to the neural network's error. · Negative gradient - The algorithm should increase the weight to lower error. · Positive gradient - The algorithm should decrease the weight to lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process. First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had infinite computation resources, we would try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some shortcuts to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 4.DRV is a graph that demonstrates the error for a single weight:
Figure 4.DRV: Derivative



Looking at this chart, you can easily see that the optimal weight is where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. However, we can determine the slope of the error curve at a particular weight. In the above chart, we see the
slope of the error curve at 1.5. The straight line barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error. The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given weight. Derivatives are one of the most fundamental concepts in calculus. For this book, you need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to adjust the weight for a lower error. Using our working definition of the gradient, we will show how to calculate it.

Momentum Backpropagation

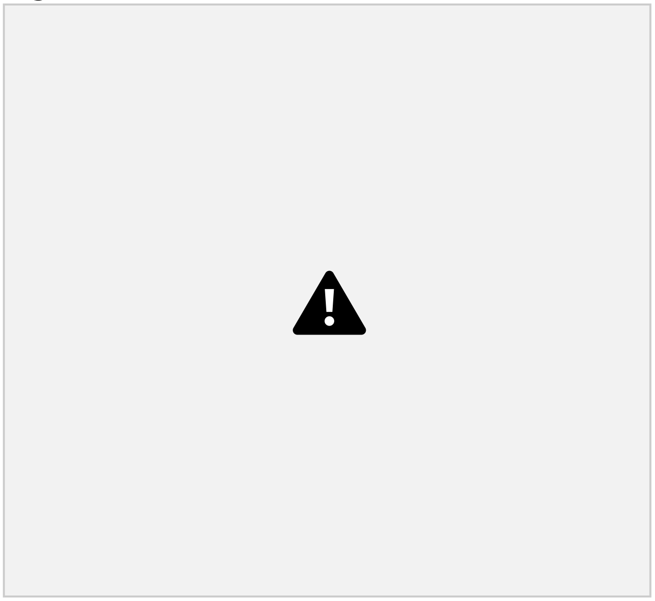
Momentum adds another term to the calculation of v_t:

$$v_t=\eta\nabla_{\theta_{t-1}}J\,(\theta_{t-1})+\lambda\, v_{t-1}$$

Like the learning rate, momentum adds another training parameter that scales the effect of momentum. Momentum backpropagation has two training parameters: learning rate (η, eta) and momentum (λ, lambda). Momentum adds the scaled value of the previous weight change amount (v_{t-1}) to the current weight change amount(v_t).

This technique has the effect of adding additional force behind the direction a weight is moving. Figure 4.MTM shows how this might allow the weight to escape local minima.

Figure 4.MTM: Momentum



A typical value for momentum is 0.9.

Batch and Online Backpropagation

How often should the weights of a neural network be updated? We can calculate gradients for a training set element. These gradients can also be summed together into batches, and the weights updated once per batch.

- **Online Training** – Update the weights based on gradients calculated from a single training set element.
- **Batch Training** – Update the weights based on the sum of the gradients over all training set elements.
- **Batch Size** – Update the weights based on the sum of some batch size of training set elements.
- **Mini-Batch Training** – The same as batch size, but with minimal batch size. Mini-batches are very popular, often in the 32-64 element range.

Because the batch size is smaller than the full training set size, it may take several batches to make it completely through the training set.

- **Step/Iteration** – The number of processed batches.

- **Epoch** – The number of times the algorithm processed the complete training set.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is currently one of the most popular neural network training algorithms. It works very similarly to Batch/Mini-Batch training, except that the batches are made up of a random set of training elements.

This technique leads to a very irregular convergence in error during training, as shown in Figure 4.SGD.

Figure 4.SGD: SGD Error

Image from Wikipedia

Because the neural network is trained on a random sample of the complete training set each time, the error does not make a smooth transition downward. However, the error usually does go down.

Advantages to SGD include:

- Computationally efficient. Each training step can be relatively fast, even with a huge training set.

- Decreases overfitting by focusing on only a portion of the training set each step.

Other Techniques

One problem with simple backpropagation training algorithms is that they are susceptible to learning rate and momentum. This technique is difficult because:

- Learning rate must be adjusted to a small enough level to train an accurate neural network.
- Momentum must be large enough to overcome local minima yet small enough not to destabilize the training.
- A single learning rate/momentum is often not good enough for the entire training process. It is often helpful to automatically decrease the learning rate as the training progresses.
- All weights share a single learning rate/momentum.

Other training techniques:

- **Resilient Propagation** – Use only the magnitude of the gradient and allow each neuron to learn at its rate. There is no need for learning rate/momentum; however, it only works in full batch mode.
- **Nesterov accelerated gradient** – Helps mitigate the risk of choosing a bad mini-batch. · **Adagrad** – Allows an automatically decaying per-weight learning rate and momentum concept.
- **Adadelta** – Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- **Non-Gradient Methods** – Non-gradient methods can sometimes be useful, though rarely outperform gradient-based backpropagation methods. These include: simulated annealing, genetic algorithms, particle swarm optimization, Nelder Mead, and many more.

ADAM Update

ADAM is the first training algorithm you should try. It is very effective. Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates. [Cite:kingma2014adam] Adam estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with an exponentially decaying average of past gradients (m):

$$m_t=\beta_1 m_{t-1}+(1-\beta_1)\, g_t$$

This average accomplishes a similar goal as classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$$v_t=\beta_2\, v_{t-1}+(1-\beta_2)\, g_t^2$$

The values m_t and v_t are estimates of the gradients' first moment (the mean) and the second moment (the uncentered variance). However, they will be strongly biased towards zero in the initial training cycles. The first moment's bias is corrected as follows.

$$\hat{m}_t = \frac{m_t}{1-\beta_{1t}}$$

Similarly, the second moment is also corrected:

$$\hat{v}_t = \frac{v_t}{1-\beta_{2t}}$$

These bias-corrected first and second moment estimates are applied to the ultimate Adam update rule, as follows:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \eta \hat{m}_t}$$

Adam is very tolerant to initial learning rate (\alpha) and other training parameters. Kingma and Ba (2014) propose default values of 0.9 for β_1, 0.999 for β_2, and 10−8 for η.