Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- **Part 3.1: Deep Learning and Neural Network Introduction** [Video] [Notebook]
- Part 3.2: Introduction to Tensorflow and Keras [Video] [Notebook]
- Part 3.3: Saving and Loading a Keras Neural Network [Video] [Notebook]
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [Video] [Notebook]
- Part 3.5: Extracting Weights and Manual Calculation [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Part 3.1: 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.

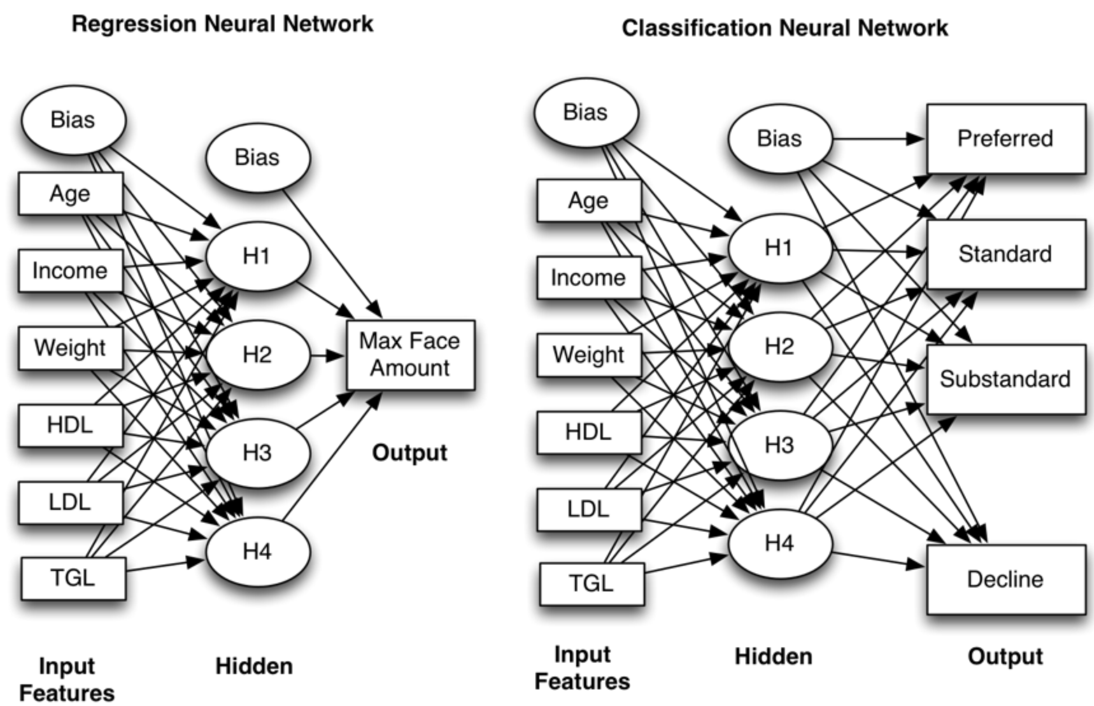
Classification or Regression

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



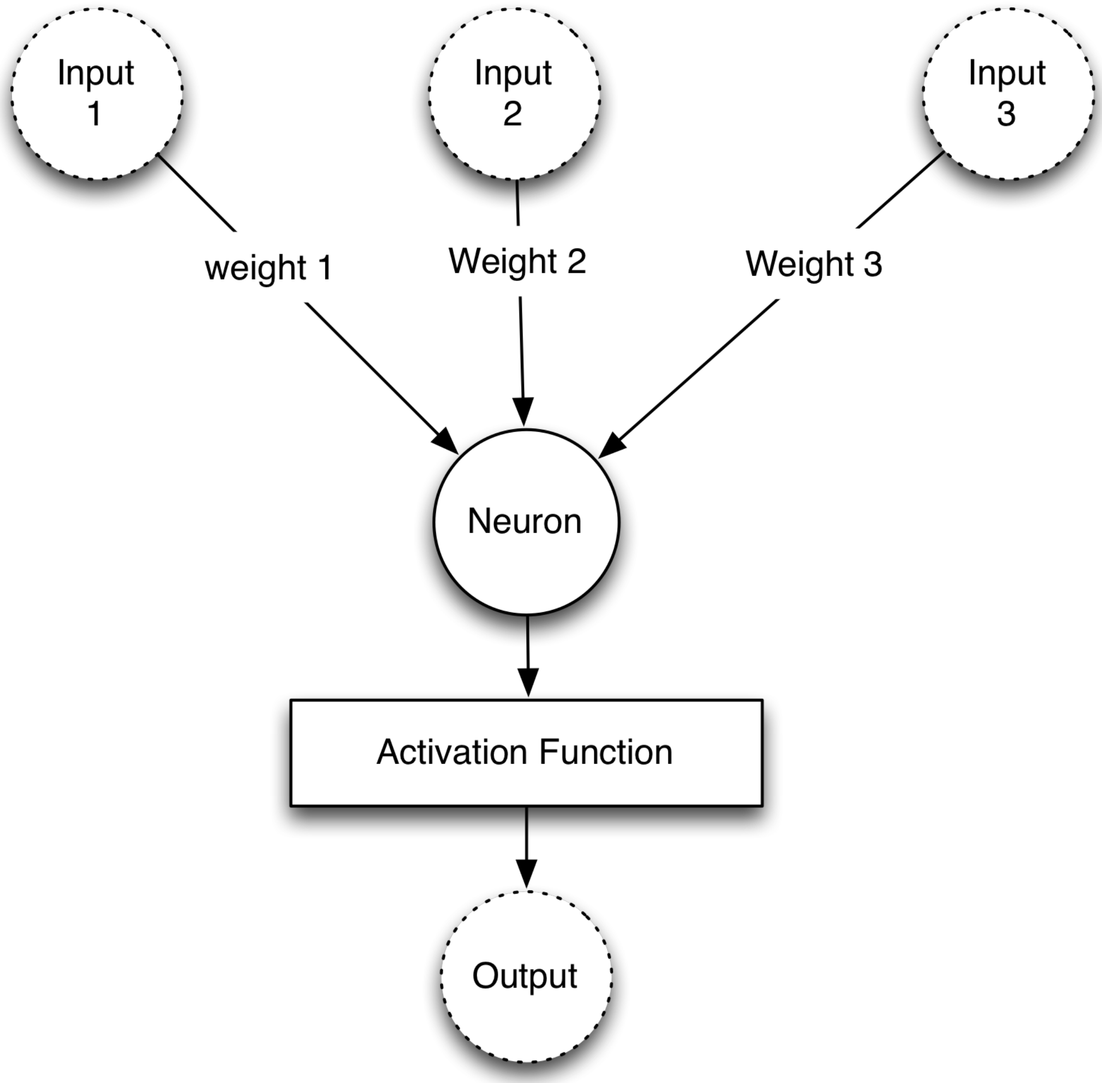
Notice that the output of the regression neural network is numeric, and the classification output is a class. Regression, or two-class classification, networks always have a single output. Classification neural networks have an output neuron for each category.

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:

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

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:

$$[0.1, 0.2, 0.3]$$

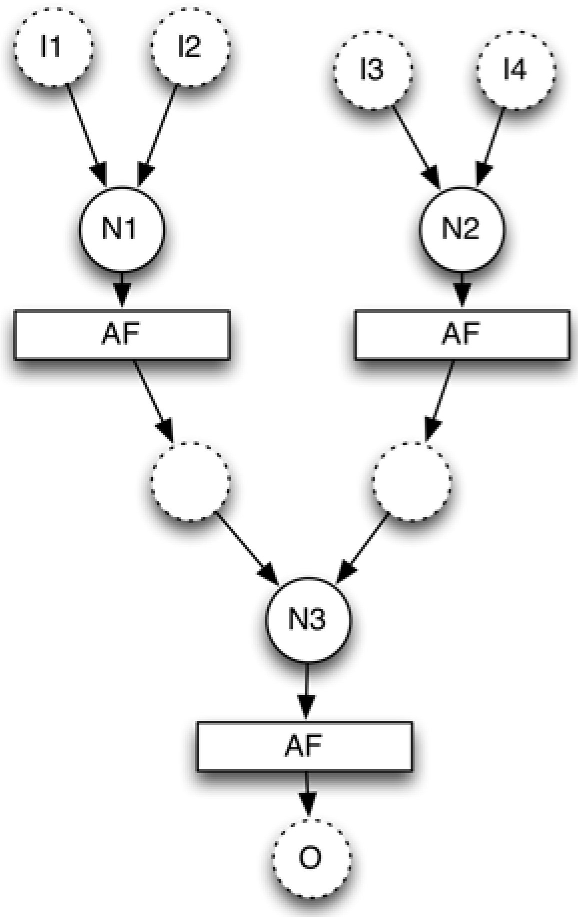
To calculate the summation, perform the following:

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

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:

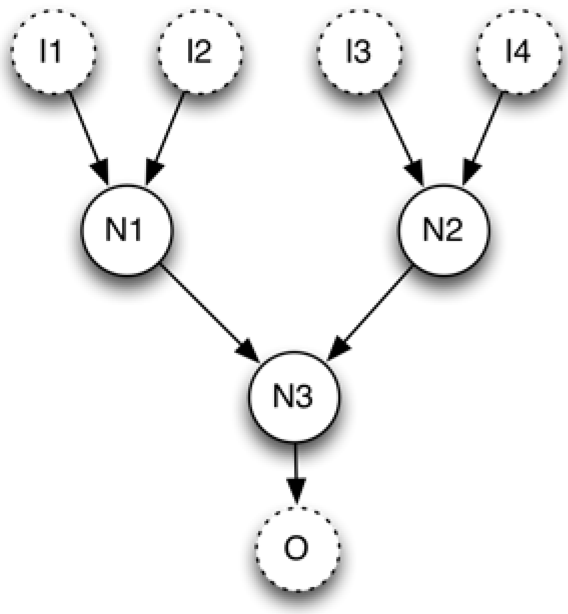
Figure 3.ANN-3: Three Neuron Neural Network



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.

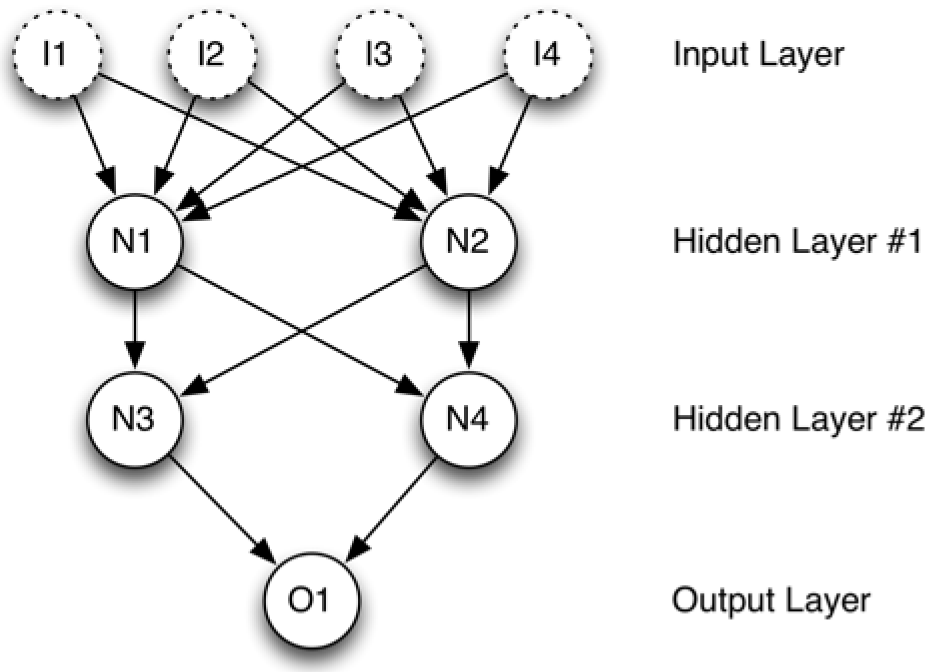
Figure 3.SANN-3: Three Neuron Neural Network



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.

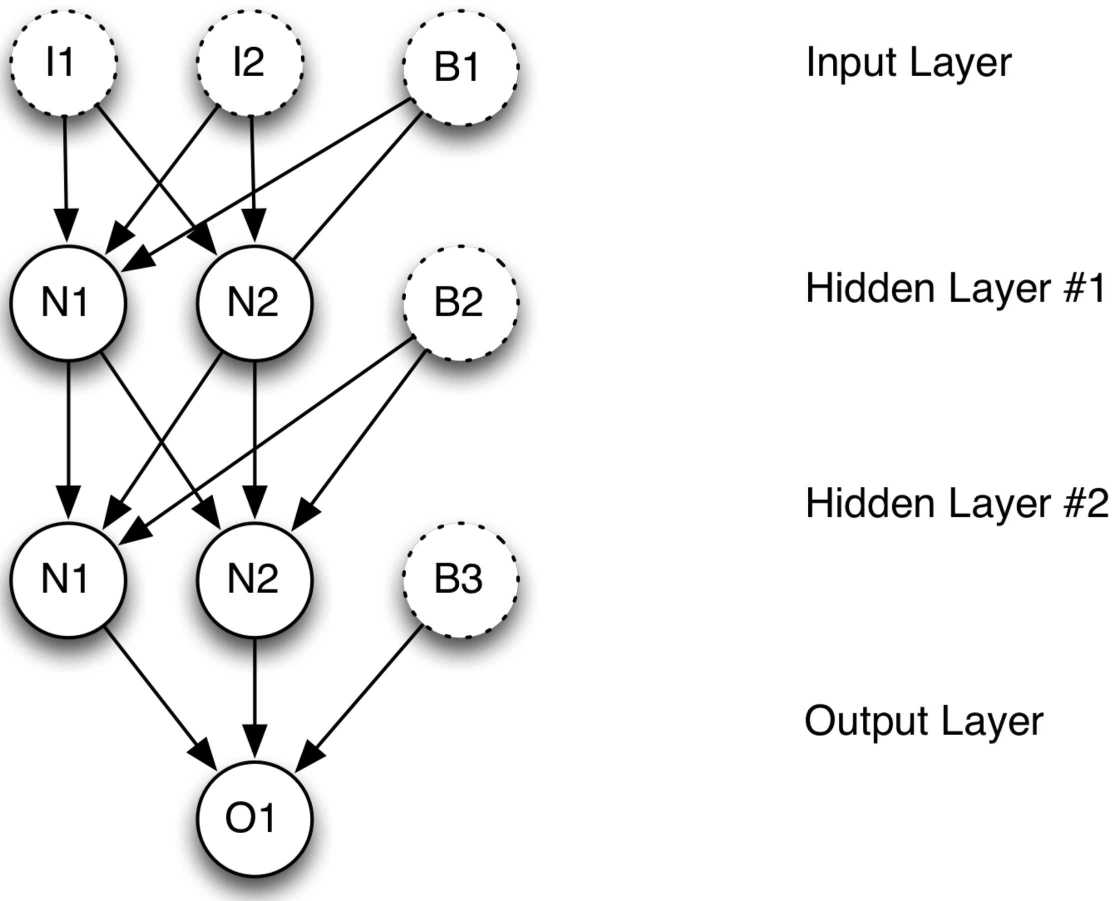
A common question for programmers concerns the number of hidden neurons in a network. Since the answer to this question is complex, more than one section of the course will include a relevant discussion of the number of hidden neurons. Before deep learning, researchers generally suggested that anything 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?

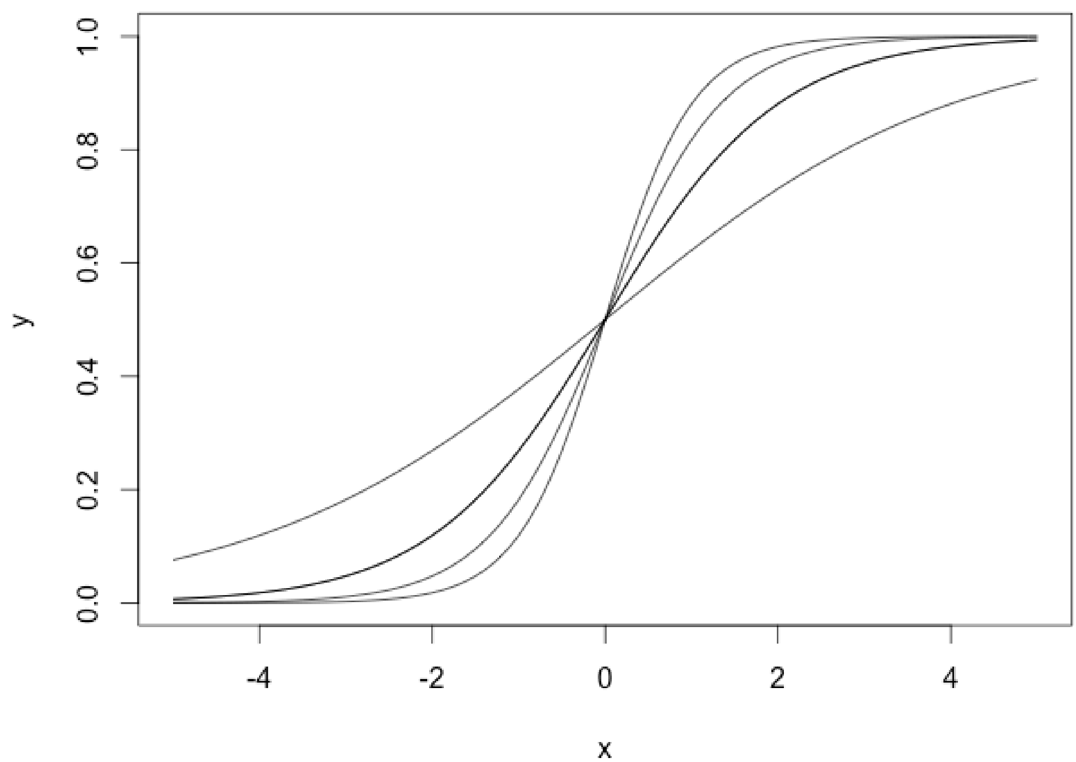
The activation functions from the previous section specify the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider the following equation. It represents a single-input sigmoid activation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

$$f(x, 0.5, 0.0)$$

$$f(x, 1.0, 0.0)$$

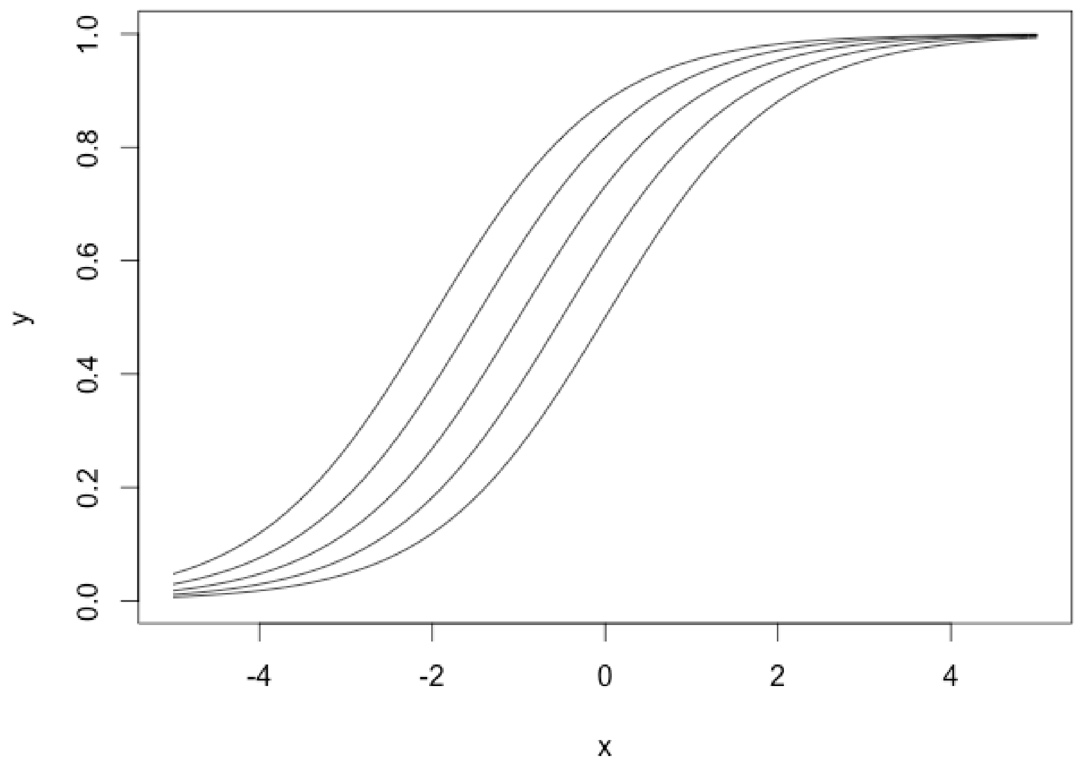
$$f(x, 1.5, 0.0)$$

$$f(x, 2.0, 0.0)$$

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).

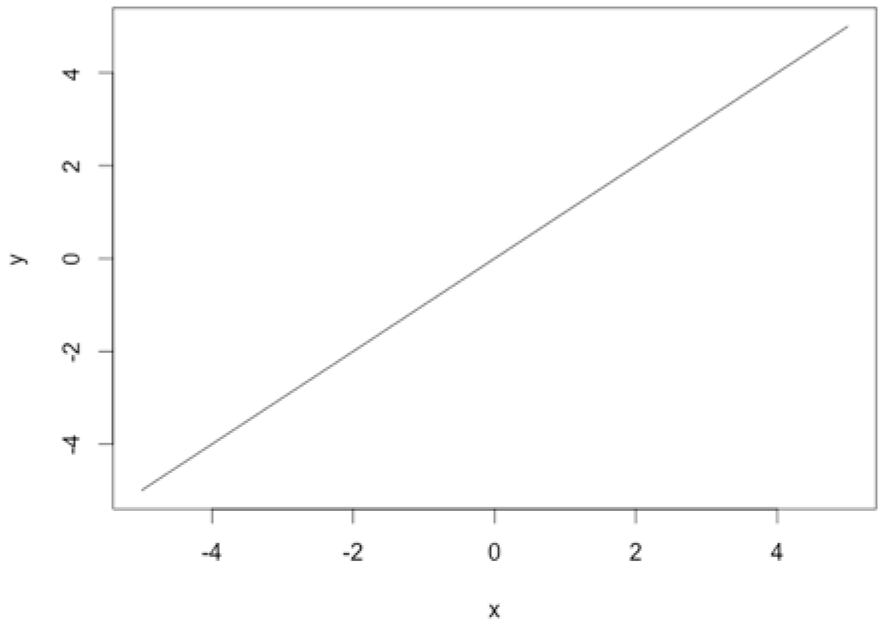
Linear Activation Function

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.

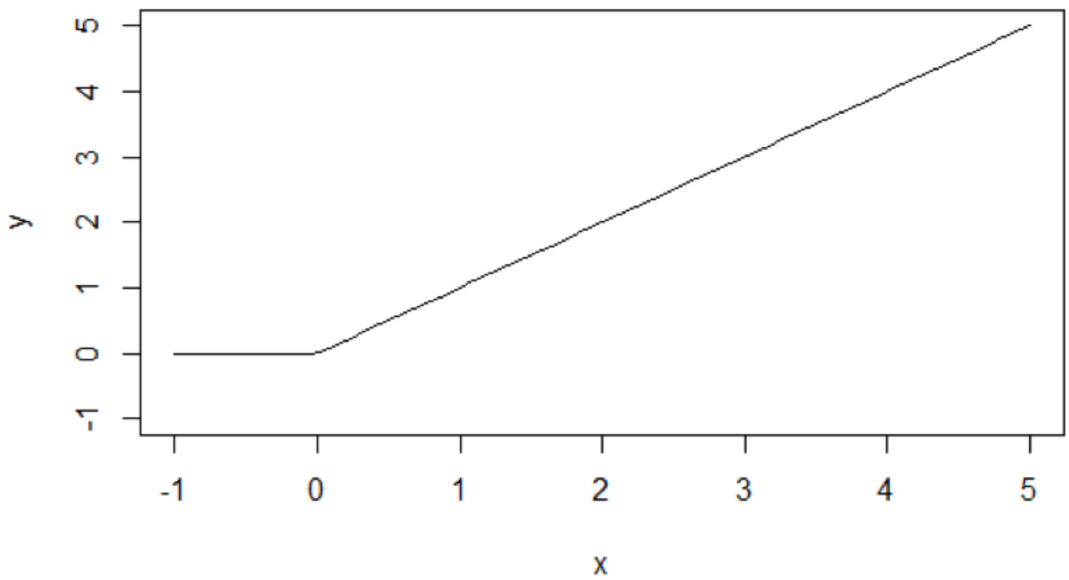
Rectified Linear Units (ReLU)

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:

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

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

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

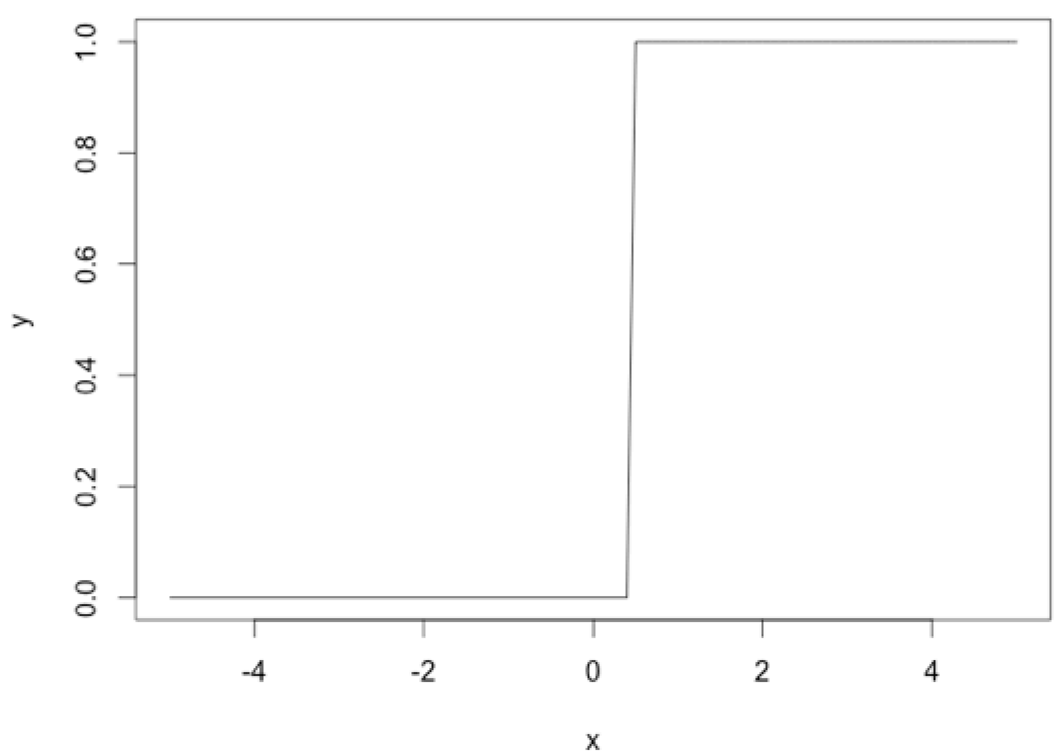
$$j_2 = \exp(0.4)/sum = 0.28839620365112$$

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



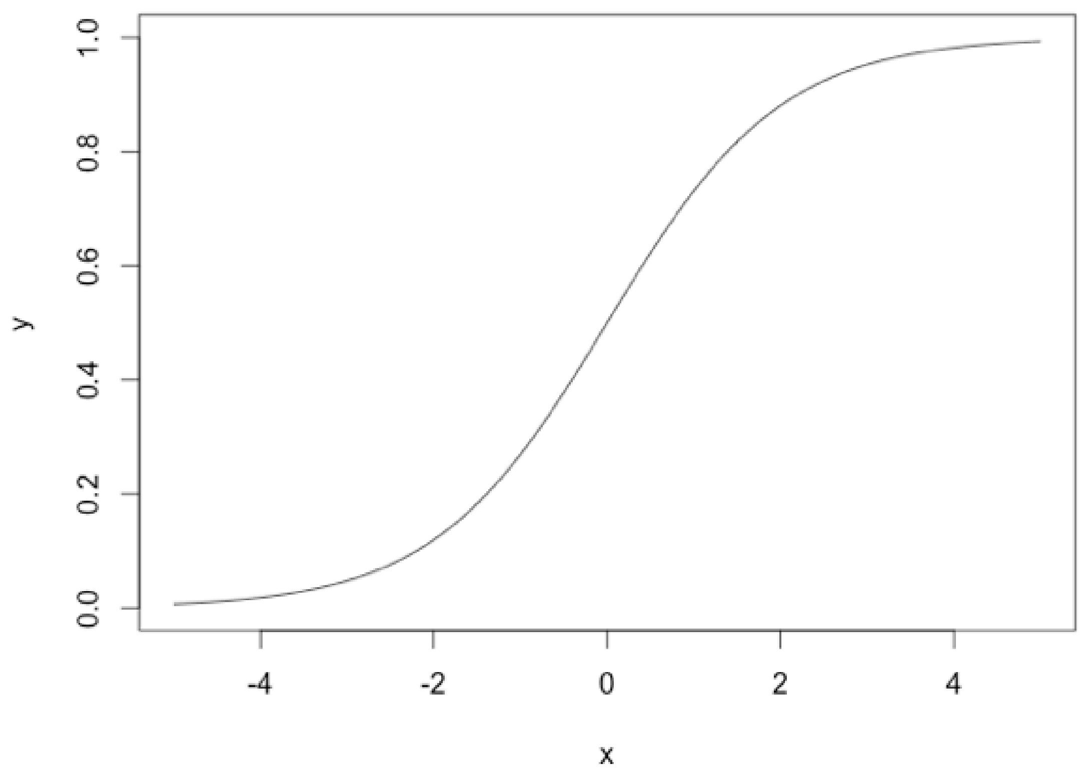
Sigmoid Activation Function

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.

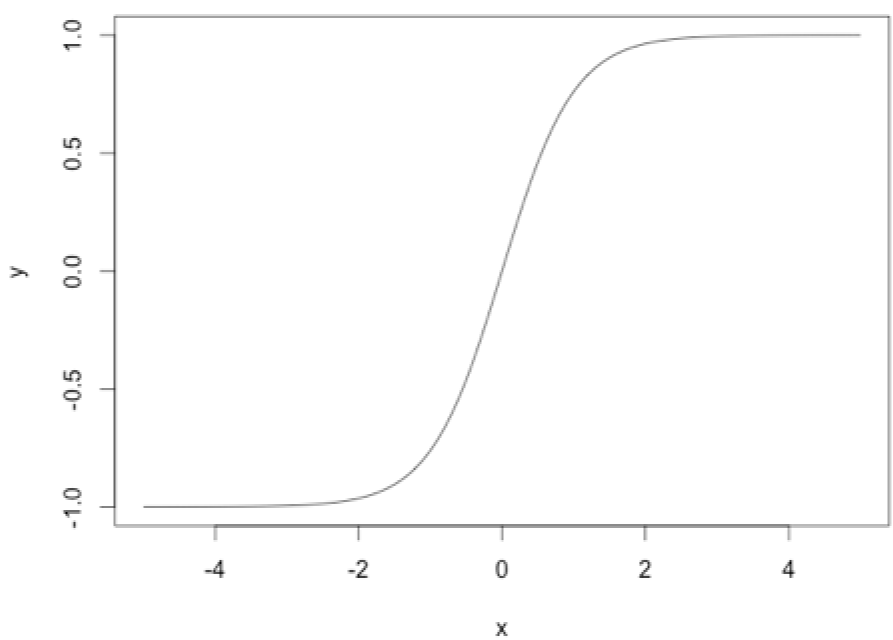
Hyperbolic Tangent Activation Function

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

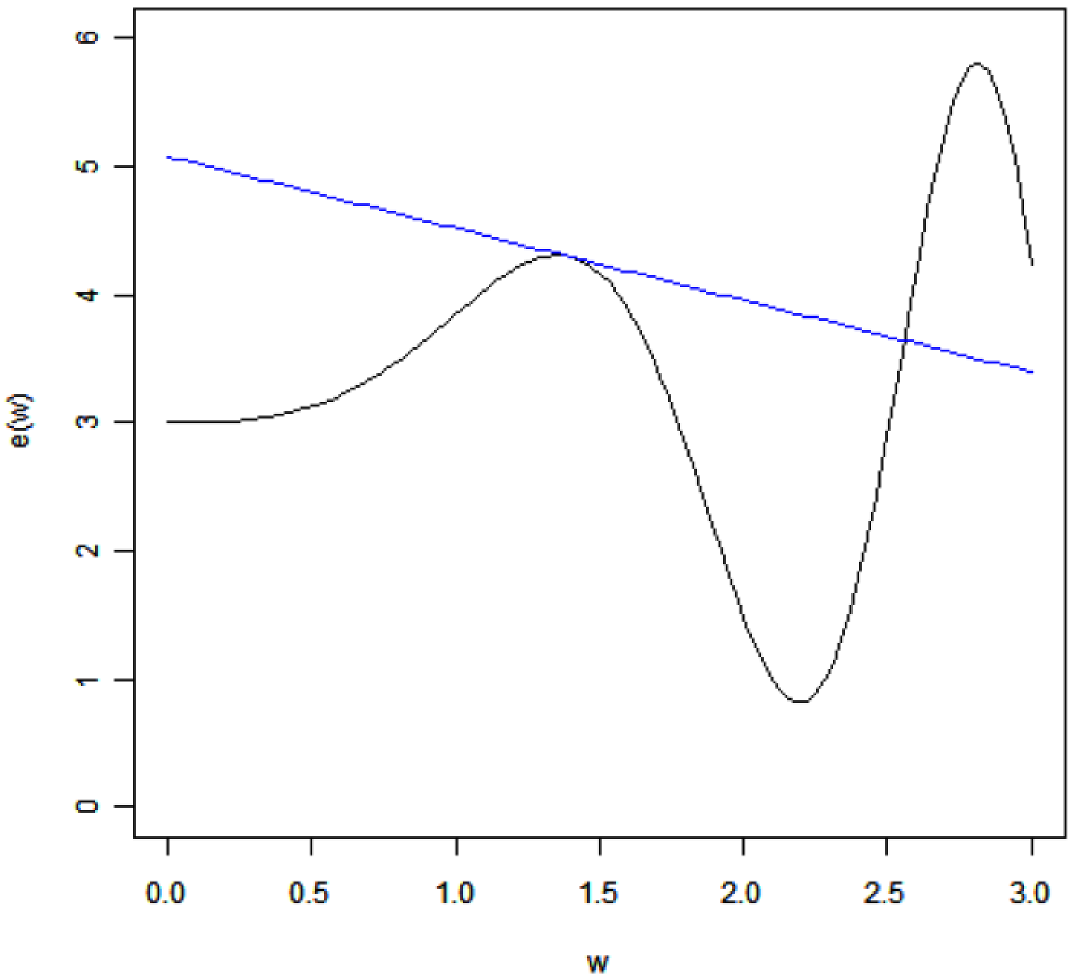


The hyperbolic tangent function has several advantages over the sigmoid activation function.

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



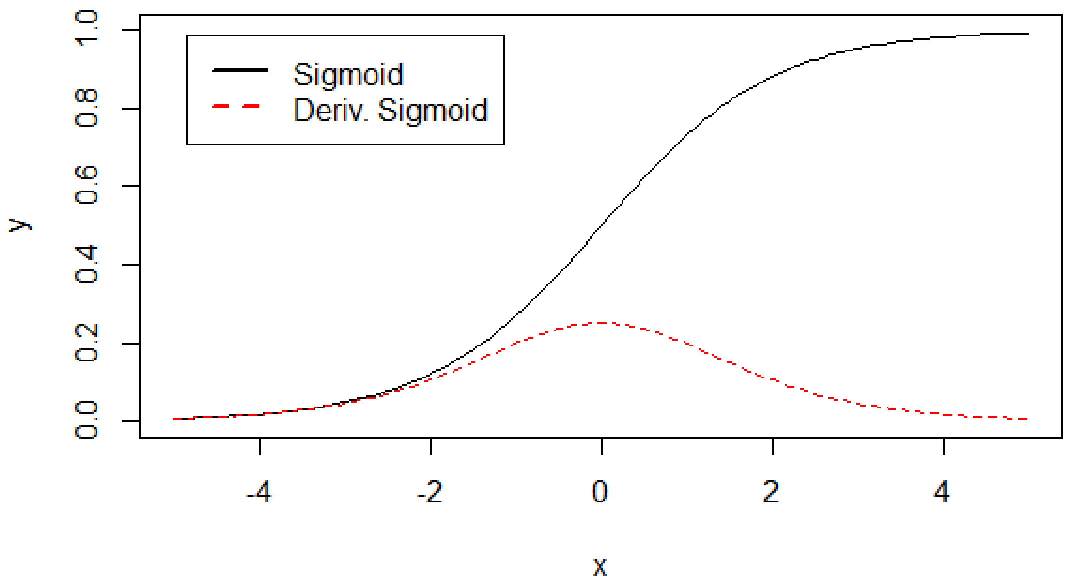
The derivative of the sigmoid function is given here:

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

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:

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

Module 3 Assignment

You can find the first assignment here: assignment 3

```
In [ ]:
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [Video] [Notebook]
- **Part 3.2: Introduction to Tensorflow and Keras** [Video] [Notebook]
- Part 3.3: Saving and Loading a Keras Neural Network [Video] [Notebook]
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [Video] [Notebook]
- Part 3.5: Extracting Weights and Manual Calculation [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [28]:

```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 3.2: Introduction to Tensorflow and Keras

TensorFlow [Cite:GoogleTensorFlow] is an open-source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for research and production by different teams in many commercial Google products, such as speech recognition, Gmail, Google Photos, and search, many of which had previously used its predecessor DistBelief. TensorFlow was originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license on November 9, 2015.

- TensorFlow Homepage
- TensorFlow GitHib
- TensorFlow Google Groups Support
- TensorFlow Google Groups Developer Discussion
- TensorFlow FAQ

Why TensorFlow

- Supported by Google
- Works well on Windows, Linux, and Mac
- Excellent GPU support
- Python is an easy to learn programming language
- Python is extremely popular in the data science community

Deep Learning Tools

TensorFlow is not the only game in town. The biggest competitor to TensorFlow/Keras is PyTorch. Listed below are some of the deep learning toolkits actively being supported:

- **TensorFlow** - Google's deep learning API. The focus of this class, along with Keras.
- **Keras** - Acts as a higher-level to Tensorflow.
- **PyTorch** - PyTorch is an open-source machine learning library based on the Torch library, used for computer vision and natural language applications processing. Facebook's AI Research lab primarily develops PyTorch.

Other deep learning tools:

- **Deeplearning4J** - Java-based. Supports all major platforms. GPU support in Java!
- **H2O** - Java-based.

In my opinion, the two primary Python libraries for deep learning are PyTorch and Keras. Generally, PyTorch requires more lines of code to perform the deep learning applications presented in this course. This trait of PyTorch gives Keras an easier learning curve than PyTorch. However, if you are creating entirely new neural network structures in a research setting, PyTorch can make for easier access to some of the low-level internals of deep learning.

Using TensorFlow Directly

Most of the time in the course, we will communicate with TensorFlow using Keras [Cite:franccois2017deep], which allows you to specify the number of hidden layers and create the neural network. TensorFlow is a low-level mathematics API, similar to Numpy. However, unlike Numpy, TensorFlow is built for deep learning. TensorFlow compiles these compute graphs into highly efficient C++/CUDA code.

TensorFlow Linear Algebra Examples

TensorFlow is a library for linear algebra. Keras is a higher-level abstraction for neural networks that you build upon TensorFlow. In this section, I will demonstrate some basic linear algebra that directly employs TensorFlow and does not use Keras. First, we will see how to multiply a row and column matrix.

In [29]:

```
import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)

print(product)
print(float(product))
```

```
tf.Tensor([[12.]], shape=(1, 1), dtype=float32)
12.0
```

This example multiplied two TensorFlow constant tensors. Next, we will see how to subtract a constant from a variable.

In [30]:

```
import tensorflow as tf

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])
```

```python
# Add an op to subtract 'a' from 'x'.  Run it and print the result
sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
# ==> [-2. -1.]
```

```
tf.Tensor([-2. -1.], shape=(2,), dtype=float32)
[-2. -1.]
```

Of course, variables are only useful if their values can be changed. The program can accomplish this change in value by calling the assign function.

In [31]:
```python
x.assign([4.0, 6.0])
```

Out[31]:
```
<tf.Variable 'UnreadVariable' shape=(2,) dtype=float32, numpy=array([4.,
6.], dtype=float32)>
```

The program can now perform the subtraction with this new value.

In [32]:
```python
sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
```

```
tf.Tensor([1. 3.], shape=(2,), dtype=float32)
[1. 3.]
```

In the next section, we will see a TensorFlow example that has nothing to do with neural networks.

TensorFlow Mandelbrot Set Example

Next, we examine another example where we use TensorFlow directly. To demonstrate that TensorFlow is mathematical and does not only provide neural networks, we will also first use it for a non-machine learning rendering task. The code presented here can render a Mandelbrot set.

In [33]:
```python
import tensorflow as tf
import numpy as np

import PIL.Image
from io import BytesIO
from IPython.display import Image, display

def render(a):
  a_cyclic = (a*0.3).reshape(list(a.shape)+[1])
  img = np.concatenate([10+20*np.cos(a_cyclic),
                        30+50*np.sin(a_cyclic),
                        155-80*np.cos(a_cyclic)], 2)
  img[a==a.max()] = 0
  a = img
  a = np.uint8(np.clip(a, 0, 255))
  f = BytesIO()
  return PIL.Image.fromarray(a)
```

```
#@tf.function
def mandelbrot_helper(grid_c, current_values, counts,cycles):

  for i in range(cycles):
    temp = current_values*current_values + grid_c
    not_diverged = tf.abs(temp) < 4
    current_values.assign(temp),
    counts.assign_add(tf.cast(not_diverged, tf.float32))

def mandelbrot(render_size,center,zoom,cycles):
  f = zoom/render_size[0]
  real_start = center[0]-(render_size[0]/2)*f
  real_end = real_start + render_size[0]*f
  imag_start = center[1]-(render_size[1]/2)*f
  imag_end = imag_start + render_size[1]*f

  real_range = tf.range(real_start,real_end,f,dtype=tf.float64)
  imag_range = tf.range(imag_start,imag_end,f,dtype=tf.float64)
  real, imag = tf.meshgrid(real_range,imag_range)
  grid_c = tf.constant(tf.complex(real, imag))
  current_values = tf.Variable(grid_c)
  counts = tf.Variable(tf.zeros_like(grid_c, tf.float32))

  mandelbrot_helper(grid_c, current_values,counts,cycles)
  return counts.numpy()
```

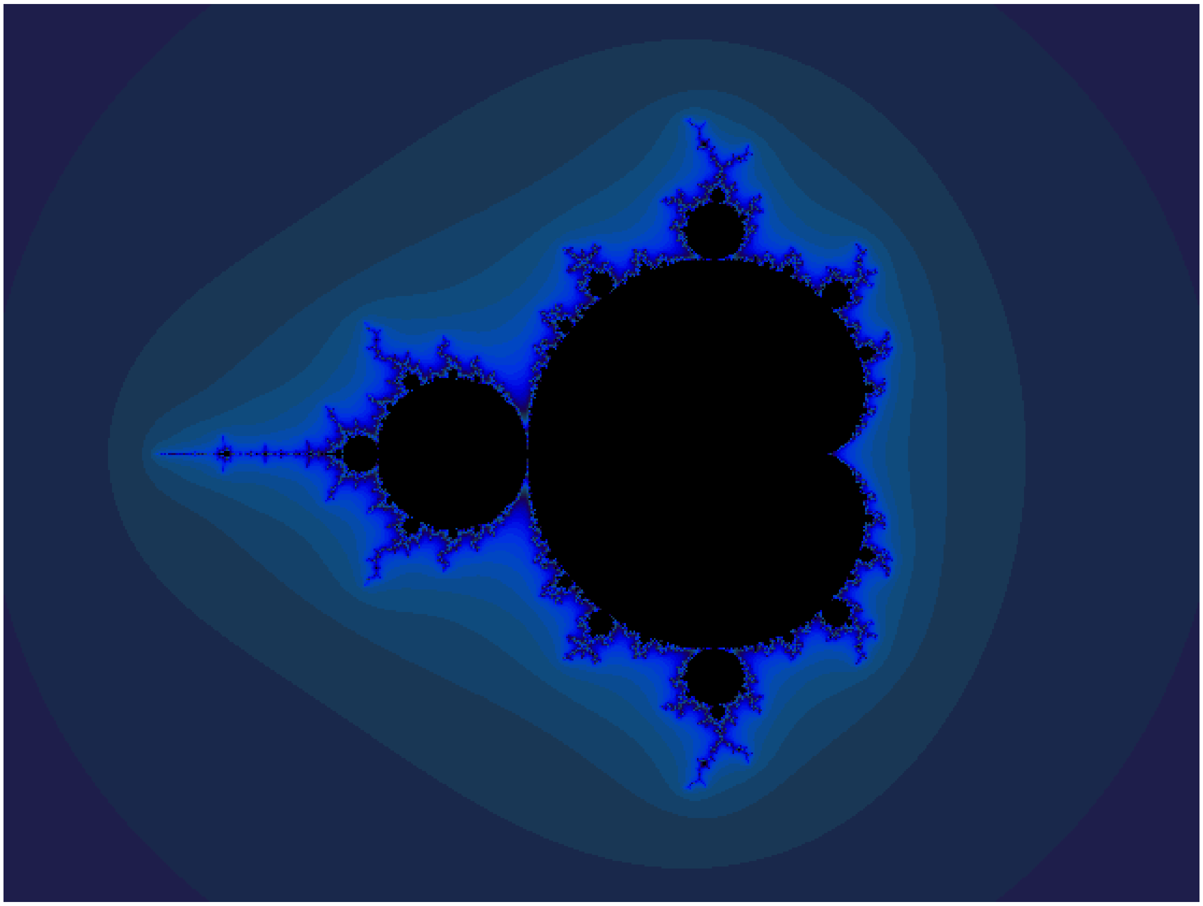
With the above code defined, we can now calculate and render a Mandlebrot plot.

In [34]:
```
counts = mandelbrot(
    #render_size=(3840,2160), # 4K
    #render_size=(1920,1080), # HD
    render_size=(640,480),
    center=(-0.5,0),
    zoom=4,
    cycles=200
)
img = render(counts)
print(img.size)
img

#img.save("test.png")
```

(640, 480)

Out[34]:



Mandlebrot rendering programs are both simple and infinitely complex at the same time. This view shows the entire Mandlebrot universe simultaneously, as a view completely zoomed out. However, if you zoom in on any non-black portion of the plot, you will find infinite hidden complexity.

Introduction to Keras

Keras is a layer on top of Tensorflow that makes it much easier to create neural networks. Rather than define the graphs, as you see above, you set the individual layers of the network with a much more high-level API. Unless you are researching entirely new structures of deep neural networks, it is unlikely that you need to program TensorFlow directly.

For this class, we will usually use TensorFlow through Keras, rather than direct TensorFlow

Simple TensorFlow Regression: MPG

This example shows how to encode the MPG dataset for regression and predict values. We will see if we can predict the miles per gallon (MPG) for a car based on the car's weight, cylinders, engine size, and other features.

In [35]:

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

```
Train on 398 samples
Epoch 1/100
398/398 - 0s - loss: 359076.9421
Epoch 2/100
398/398 - 0s - loss: 74176.6153
Epoch 3/100
398/398 - 0s - loss: 9767.8802
Epoch 4/100
398/398 - 0s - loss: 1427.8838
Epoch 5/100
398/398 - 0s - loss: 2057.3394
Epoch 6/100
398/398 - 0s - loss: 1513.2297
Epoch 7/100
398/398 - 0s - loss: 1295.3801
Epoch 8/100
398/398 - 0s - loss: 1272.4230
Epoch 9/100
398/398 - 0s - loss: 1239.1273
Epoch 10/100
398/398 - 0s - loss: 1207.5261
Epoch 11/100
398/398 - 0s - loss: 1183.4229
Epoch 12/100
398/398 - 0s - loss: 1154.0081
Epoch 13/100
398/398 - 0s - loss: 1124.9607
Epoch 14/100
398/398 - 0s - loss: 1099.9529
Epoch 15/100
398/398 - 0s - loss: 1076.2990
Epoch 16/100
398/398 - 0s - loss: 1060.9813
Epoch 17/100
398/398 - 0s - loss: 1047.1127
Epoch 18/100
398/398 - 0s - loss: 1035.2276
Epoch 19/100
398/398 - 0s - loss: 1023.7794
Epoch 20/100
398/398 - 0s - loss: 1012.4624
Epoch 21/100
398/398 - 0s - loss: 1001.8168
Epoch 22/100
398/398 - 0s - loss: 992.2781
Epoch 23/100
398/398 - 0s - loss: 979.8584
Epoch 24/100
398/398 - 0s - loss: 969.5121
Epoch 25/100
398/398 - 0s - loss: 958.6515
Epoch 26/100
398/398 - 0s - loss: 950.3542
Epoch 27/100
398/398 - 0s - loss: 941.1478
Epoch 28/100
```

```
398/398 - 0s - loss: 926.8202
Epoch 29/100
398/398 - 0s - loss: 917.2789
Epoch 30/100
398/398 - 0s - loss: 905.2999
Epoch 31/100
398/398 - 0s - loss: 892.8540
Epoch 32/100
398/398 - 0s - loss: 884.2463
Epoch 33/100
398/398 - 0s - loss: 871.6899
Epoch 34/100
398/398 - 0s - loss: 864.9260
Epoch 35/100
398/398 - 0s - loss: 849.8424
Epoch 36/100
398/398 - 0s - loss: 840.7186
Epoch 37/100
398/398 - 0s - loss: 836.8501
Epoch 38/100
398/398 - 0s - loss: 824.8931
Epoch 39/100
398/398 - 0s - loss: 810.3899
Epoch 40/100
398/398 - 0s - loss: 800.1687
Epoch 41/100
398/398 - 0s - loss: 784.4148
Epoch 42/100
398/398 - 0s - loss: 774.2326
Epoch 43/100
398/398 - 0s - loss: 762.1020
Epoch 44/100
398/398 - 0s - loss: 751.2068
Epoch 45/100
398/398 - 0s - loss: 740.0900
Epoch 46/100
398/398 - 0s - loss: 728.2087
Epoch 47/100
398/398 - 0s - loss: 719.4002
Epoch 48/100
398/398 - 0s - loss: 710.2463
Epoch 49/100
398/398 - 0s - loss: 695.5551
Epoch 50/100
398/398 - 0s - loss: 686.5956
Epoch 51/100
398/398 - 0s - loss: 676.1623
Epoch 52/100
398/398 - 0s - loss: 661.9155
Epoch 53/100
398/398 - 0s - loss: 652.7001
Epoch 54/100
398/398 - 0s - loss: 649.1527
Epoch 55/100
398/398 - 0s - loss: 626.8102
Epoch 56/100
```

```
398/398 - 0s - loss: 617.8689
Epoch 57/100
398/398 - 0s - loss: 610.6475
Epoch 58/100
398/398 - 0s - loss: 592.5715
Epoch 59/100
398/398 - 0s - loss: 582.9929
Epoch 60/100
398/398 - 0s - loss: 571.0975
Epoch 61/100
398/398 - 0s - loss: 560.9386
Epoch 62/100
398/398 - 0s - loss: 549.7743
Epoch 63/100
398/398 - 0s - loss: 542.2976
Epoch 64/100
398/398 - 0s - loss: 524.7271
Epoch 65/100
398/398 - 0s - loss: 512.6835
Epoch 66/100
398/398 - 0s - loss: 501.3199
Epoch 67/100
398/398 - 0s - loss: 489.1912
Epoch 68/100
398/398 - 0s - loss: 476.9358
Epoch 69/100
398/398 - 0s - loss: 465.7011
Epoch 70/100
398/398 - 0s - loss: 454.4821
Epoch 71/100
398/398 - 0s - loss: 444.3988
Epoch 72/100
398/398 - 0s - loss: 432.2656
Epoch 73/100
398/398 - 0s - loss: 420.4270
Epoch 74/100
398/398 - 0s - loss: 408.0120
Epoch 75/100
398/398 - 0s - loss: 400.3801
Epoch 76/100
398/398 - 0s - loss: 384.8745
Epoch 77/100
398/398 - 0s - loss: 373.1397
Epoch 78/100
398/398 - 0s - loss: 364.7359
Epoch 79/100
398/398 - 0s - loss: 349.1083
Epoch 80/100
398/398 - 0s - loss: 336.7873
Epoch 81/100
398/398 - 0s - loss: 329.1152
Epoch 82/100
398/398 - 0s - loss: 315.6647
Epoch 83/100
398/398 - 0s - loss: 302.7605
Epoch 84/100
```

```
398/398 - 0s - loss: 292.5489
Epoch 85/100
398/398 - 0s - loss: 280.1445
Epoch 86/100
398/398 - 0s - loss: 268.6837
Epoch 87/100
398/398 - 0s - loss: 260.4212
Epoch 88/100
398/398 - 0s - loss: 250.5896
Epoch 89/100
398/398 - 0s - loss: 247.6127
Epoch 90/100
398/398 - 0s - loss: 230.2208
Epoch 91/100
398/398 - 0s - loss: 218.4122
Epoch 92/100
398/398 - 0s - loss: 208.7282
Epoch 93/100
398/398 - 0s - loss: 200.0094
Epoch 94/100
398/398 - 0s - loss: 189.9866
Epoch 95/100
398/398 - 0s - loss: 182.8754
Epoch 96/100
398/398 - 0s - loss: 175.6958
Epoch 97/100
398/398 - 0s - loss: 172.0255
Epoch 98/100
398/398 - 0s - loss: 156.4483
Epoch 99/100
398/398 - 0s - loss: 149.1817
Epoch 100/100
398/398 - 0s - loss: 142.9415
```

Out[35]:
```
<tensorflow.python.keras.callbacks.History at 0x7f3ee44468d0>
```

Introduction to Neural Network Hyperparameters

If you look at the above code, you will see that the neural network contains four layers. The first layer is the input layer because it contains the **input_dim** parameter that the programmer sets to be the number of inputs the dataset has. The network needs one input neuron for every column in the data set (including dummy variables).

There are also several hidden layers, with 25 and 10 neurons each. You might be wondering how the programmer chose these numbers. Selecting a hidden neuron structure is one of the most common questions about neural networks. Unfortunately, there is no right answer. These are hyperparameters. They are settings that can affect neural network performance, yet there are no clearly defined means of setting them.

In general, more hidden neurons mean more capability to fit complex problems. However, too many neurons can lead to overfitting and lengthy training times. Too few can lead to underfitting the problem and will sacrifice accuracy. Also, how many layers you have is another hyperparameter. In general, more layers allow the neural network to perform more of its feature engineering and data preprocessing. But this also comes at the expense of training times and the risk of overfitting. In general, you will see that neuron counts start larger near the input layer and tend to shrink towards the output layer in a triangular fashion.

Some techniques use machine learning to optimize these values. These will be discussed in Module 8.3.

Controlling the Amount of Output

The program produces one line of output for each training epoch. You can eliminate this output by setting the verbose setting of the fit command:

- **verbose=0** - No progress output (use with Jupyter if you do not want output).
- **verbose=1** - Display progress bar, does not work well with Jupyter.
- **verbose=2** - Summary progress output (use with Jupyter if you want to know the loss at each epoch).

Regression Prediction

Next, we will perform actual predictions. The program assigns these predictions to the **pred** variable. These are all MPG predictions from the neural network. Notice that this is a 2D array? You can always see the dimensions of what Keras returns by printing out **pred.shape**. Neural networks can return multiple values, so the result is always an array. Here the neural network only returns one value per prediction (there are 398 cars, so 398 predictions). However, a 2D range is needed because the neural network has the potential of returning more than one value.

In [36]:
```
pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

```
Shape: (398, 1)
[[22.639828]
 [20.882801]
 [19.801853]
 [20.337807]
 [21.1946  ]
 [23.72337 ]
 [21.285397]
 [21.545208]
 [21.873882]
 [19.303974]]
```

We would like to see how good these predictions are. We know the correct MPG for each car so we can measure how close the neural network was.

In [37]:
```
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Final score (RMSE): {score}")
```

```
Final score (RMSE): 11.862759295790802
```

The number printed above is the average number of predictions above or below the expected output. We can also print out the first ten cars with predictions and actual MPG.

In [38]:
```
# Sample predictions
for i in range(10):
    print(f"{i+1}. Car name: {cars[i]}, MPG: {y[i]}, "
          + f"predicted MPG: {pred[i]}")
```

```
1. Car name: chevrolet chevelle malibu, MPG: 18.0, predicted MPG: [22.63982
8]
2. Car name: buick skylark 320, MPG: 15.0, predicted MPG: [20.882801]
3. Car name: plymouth satellite, MPG: 18.0, predicted MPG: [19.801853]
4. Car name: amc rebel sst, MPG: 16.0, predicted MPG: [20.337807]
5. Car name: ford torino, MPG: 17.0, predicted MPG: [21.1946]
6. Car name: ford galaxie 500, MPG: 15.0, predicted MPG: [23.72337]
7. Car name: chevrolet impala, MPG: 14.0, predicted MPG: [21.285397]
8. Car name: plymouth fury iii, MPG: 14.0, predicted MPG: [21.545208]
9. Car name: pontiac catalina, MPG: 14.0, predicted MPG: [21.873882]
10. Car name: amc ambassador dpl, MPG: 15.0, predicted MPG: [19.303974]
```

Simple TensorFlow Classification: Iris

Classification is how a neural network attempts to classify the input into one or more classes. The simplest way of evaluating a classification network is to track the percentage of training set items classified incorrectly. We typically score human results in this manner. For example, you might have taken multiple-choice exams in school in which you had to shade in a bubble for choices A, B, C, or D. If you chose the wrong letter on a 10-question exam, you would earn a 90%. In the same way, we can grade computers; however, most classification algorithms do not merely choose A, B, C, or D. Computers typically report a classification as their percent confidence in each class. Figure 3.EXAM shows how a computer and a human might respond to question number 1 on an exam.

Figure 3.EXAM: Classification Neural Network Output
Classification Neural Network Output

As you can see, the human test taker marked the first question as "B." However, the computer test taker had an 80% (0.8) confidence in "B" and was also somewhat sure with 10% (0.1) on "A." The computer then distributed the remaining points to the other two. In the simplest sense, the machine would get 80% of the score for this question if the correct answer were "B." The computer would get only 5% (0.05) of the points if the correct answer were "D."

We previously saw how to train a neural network to predict the MPG of a car. Based on four measurements, we will now see how to predict a class, such as the type of iris flower. The code to classify iris flowers is similar to MPG; however, there are several important differences:

- The output neuron count matches the number of classes (in the case of Iris, 3).
- The Softmax transfer function is utilized by the output layer.* The loss function is cross entropy.

In [39]:

```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values


# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

```
Train on 150 samples
Epoch 1/100
150/150 - 0s - loss: 1.4756
Epoch 2/100
150/150 - 0s - loss: 1.3098
Epoch 3/100
150/150 - 0s - loss: 1.1715
Epoch 4/100
150/150 - 0s - loss: 1.0646
Epoch 5/100
150/150 - 0s - loss: 0.9959
Epoch 6/100
150/150 - 0s - loss: 0.9439
Epoch 7/100
150/150 - 0s - loss: 0.8898
Epoch 8/100
150/150 - 0s - loss: 0.8446
Epoch 9/100
150/150 - 0s - loss: 0.8176
Epoch 10/100
150/150 - 0s - loss: 0.7961
Epoch 11/100
150/150 - 0s - loss: 0.7736
Epoch 12/100
150/150 - 0s - loss: 0.7527
Epoch 13/100
150/150 - 0s - loss: 0.7327
Epoch 14/100
150/150 - 0s - loss: 0.7109
Epoch 15/100
150/150 - 0s - loss: 0.6913
Epoch 16/100
150/150 - 0s - loss: 0.6720
Epoch 17/100
150/150 - 0s - loss: 0.6527
Epoch 18/100
150/150 - 0s - loss: 0.6311
Epoch 19/100
150/150 - 0s - loss: 0.6117
Epoch 20/100
150/150 - 0s - loss: 0.5933
Epoch 21/100
150/150 - 0s - loss: 0.5761
Epoch 22/100
150/150 - 0s - loss: 0.5568
Epoch 23/100
150/150 - 0s - loss: 0.5384
Epoch 24/100
150/150 - 0s - loss: 0.5222
Epoch 25/100
150/150 - 0s - loss: 0.5063
Epoch 26/100
150/150 - 0s - loss: 0.4945
Epoch 27/100
150/150 - 0s - loss: 0.4753
Epoch 28/100
```

```
150/150 - 0s - loss: 0.4608
Epoch 29/100
150/150 - 0s - loss: 0.4478
Epoch 30/100
150/150 - 0s - loss: 0.4333
Epoch 31/100
150/150 - 0s - loss: 0.4209
Epoch 32/100
150/150 - 0s - loss: 0.4114
Epoch 33/100
150/150 - 0s - loss: 0.3964
Epoch 34/100
150/150 - 0s - loss: 0.3886
Epoch 35/100
150/150 - 0s - loss: 0.3799
Epoch 36/100
150/150 - 0s - loss: 0.3674
Epoch 37/100
150/150 - 0s - loss: 0.3569
Epoch 38/100
150/150 - 0s - loss: 0.3477
Epoch 39/100
150/150 - 0s - loss: 0.3386
Epoch 40/100
150/150 - 0s - loss: 0.3297
Epoch 41/100
150/150 - 0s - loss: 0.3243
Epoch 42/100
150/150 - 0s - loss: 0.3121
Epoch 43/100
150/150 - 0s - loss: 0.3051
Epoch 44/100
150/150 - 0s - loss: 0.2995
Epoch 45/100
150/150 - 0s - loss: 0.2886
Epoch 46/100
150/150 - 0s - loss: 0.2836
Epoch 47/100
150/150 - 0s - loss: 0.2746
Epoch 48/100
150/150 - 0s - loss: 0.2683
Epoch 49/100
150/150 - 0s - loss: 0.2608
Epoch 50/100
150/150 - 0s - loss: 0.2545
Epoch 51/100
150/150 - 0s - loss: 0.2483
Epoch 52/100
150/150 - 0s - loss: 0.2426
Epoch 53/100
150/150 - 0s - loss: 0.2349
Epoch 54/100
150/150 - 0s - loss: 0.2310
Epoch 55/100
150/150 - 0s - loss: 0.2238
Epoch 56/100
```

```
150/150 - 0s - loss: 0.2193
Epoch 57/100
150/150 - 0s - loss: 0.2144
Epoch 58/100
150/150 - 0s - loss: 0.2101
Epoch 59/100
150/150 - 0s - loss: 0.2043
Epoch 60/100
150/150 - 0s - loss: 0.1996
Epoch 61/100
150/150 - 0s - loss: 0.1954
Epoch 62/100
150/150 - 0s - loss: 0.1909
Epoch 63/100
150/150 - 0s - loss: 0.1858
Epoch 64/100
150/150 - 0s - loss: 0.1823
Epoch 65/100
150/150 - 0s - loss: 0.1789
Epoch 66/100
150/150 - 0s - loss: 0.1747
Epoch 67/100
150/150 - 0s - loss: 0.1722
Epoch 68/100
150/150 - 0s - loss: 0.1684
Epoch 69/100
150/150 - 0s - loss: 0.1633
Epoch 70/100
150/150 - 0s - loss: 0.1623
Epoch 71/100
150/150 - 0s - loss: 0.1579
Epoch 72/100
150/150 - 0s - loss: 0.1563
Epoch 73/100
150/150 - 0s - loss: 0.1551
Epoch 74/100
150/150 - 0s - loss: 0.1513
Epoch 75/100
150/150 - 0s - loss: 0.1484
Epoch 76/100
150/150 - 0s - loss: 0.1435
Epoch 77/100
150/150 - 0s - loss: 0.1425
Epoch 78/100
150/150 - 0s - loss: 0.1396
Epoch 79/100
150/150 - 0s - loss: 0.1378
Epoch 80/100
150/150 - 0s - loss: 0.1351
Epoch 81/100
150/150 - 0s - loss: 0.1357
Epoch 82/100
150/150 - 0s - loss: 0.1308
Epoch 83/100
150/150 - 0s - loss: 0.1284
Epoch 84/100
```

```
150/150 - 0s - loss: 0.1278
Epoch 85/100
150/150 - 0s - loss: 0.1248
Epoch 86/100
150/150 - 0s - loss: 0.1237
Epoch 87/100
150/150 - 0s - loss: 0.1212
Epoch 88/100
150/150 - 0s - loss: 0.1214
Epoch 89/100
150/150 - 0s - loss: 0.1183
Epoch 90/100
150/150 - 0s - loss: 0.1199
Epoch 91/100
150/150 - 0s - loss: 0.1155
Epoch 92/100
150/150 - 0s - loss: 0.1159
Epoch 93/100
150/150 - 0s - loss: 0.1112
Epoch 94/100
150/150 - 0s - loss: 0.1141
Epoch 95/100
150/150 - 0s - loss: 0.1104
Epoch 96/100
150/150 - 0s - loss: 0.1089
Epoch 97/100
150/150 - 0s - loss: 0.1089
Epoch 98/100
150/150 - 0s - loss: 0.1089
Epoch 99/100
150/150 - 0s - loss: 0.1197
Epoch 100/100
150/150 - 0s - loss: 0.1076
```

Out[39]:
```
<tensorflow.python.keras.callbacks.History at 0x7f3ee4332410>
```

In [40]:
```
# Print out number of species found:

print(species)
```

```
Index(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype='object')
```

Now that you have a neural network trained, we would like to be able to use it. The following code makes use of our neural network. Exactly like before, we will generate predictions. Notice that three values come back for each of the 150 iris flowers. There were three types of iris (Iris-setosa, Iris-versicolor, and Iris-virginica).

In [41]:
```
pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

```
Shape: (150, 3)
[[0.99278367 0.00704507 0.00017127]
 [0.98646706 0.01317458 0.0003583 ]
 [0.98927665 0.01040124 0.00032212]
 [0.98444796 0.01508906 0.00046296]
 [0.99318063 0.00664989 0.0001695 ]
 [0.9944395  0.00544237 0.00011823]
 [0.99035525 0.00932539 0.0003193 ]
 [0.99134773 0.00843632 0.00021587]
 [0.980791   0.01855918 0.00064985]
 [0.98711026 0.01257468 0.00031499]]
```

If you would like to turn of scientific notation, the following line can be used:

In [42]:
```
np.set_printoptions(suppress=True)
```

Now we see these values rounded up.

In [43]:
```
print(y[0:10])
```

```
[[1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]]
```

Usually, the program considers the column with the highest prediction to be the prediction of the neural network. It is easy to convert the predictions to the expected iris species. The argmax function finds the index of the maximum prediction for each row.

In [44]:
```
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
print(f"Predictions: {predict_classes}")
print(f"Expected: {expected_classes}")
```

```
Predictions: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1
 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
Expected: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Of course, it is straightforward to turn these indexes back into iris species. We use the species list that we created earlier.

In [45]:
```
print(species[predict_classes[1:10]])
```

```
Index(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa'],
      dtype='object')
```

Accuracy might be a more easily understood error metric. It is essentially a test score. For all of the iris predictions, what percent were correct? The downside is it does not consider how confident the neural network was in each prediction.

In [46]:
```
from sklearn.metrics import accuracy_score

correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 0.9733333333333334
```

The code below performs two ad hoc predictions. The first prediction is a single iris flower, and the second predicts two iris flowers. Notice that the **argmax** in the second prediction requires **axis=1**? Since we have a 2D array now, we must specify which axis to take the **argmax** over. The value **axis=1** specifies we want the max column index for each row.

In [47]:
```
sample_flower = np.array( [[5.0,3.0,4.0,2.0]], dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred)
print(f"Predict that {sample_flower} is: {species[pred]}")
```

```
[[0.00402835 0.25205988 0.74391174]]
Predict that [[5. 3. 4. 2.]] is: Iris-virginica
```

You can also predict two sample flowers.

In [48]:
```
sample_flower = np.array( [[5.0,3.0,4.0,2.0],[5.2,3.5,1.5,0.8]],\
        dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred,axis=1)
print(f"Predict that these two flowers {sample_flower} ")
print(f"are: {species[pred]}")
```

```
[[0.00402835 0.25205988 0.74391174]
 [0.98642194 0.01315794 0.0004201 ]]
Predict that these two flowers [[5.  3.  4.  2. ]
 [5.2 3.5 1.5 0.8]]
are: Index(['Iris-virginica', 'Iris-setosa'], dtype='object')
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [Video] [Notebook]
- Part 3.2: Introduction to Tensorflow and Keras [Video] [Notebook]
- **Part 3.3: Saving and Loading a Keras Neural Network** [Video] [Notebook]
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [Video] [Notebook]
- Part 3.5: Extracting Weights and Manual Calculation [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow. Running the following code will map your GDrive to `/content/drive`.

In [1]:
```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Mounted at /content/drive
Note: using Google CoLab
```

Part 3.3: Saving and Loading a Keras Neural Network

Complex neural networks will take a long time to fit/train. It is helpful to be able to save these neural networks so that you can reload them later. A reloaded neural network will not require retraining. Keras provides three formats for neural network saving.

- **JSON** - Stores the neural network structure (no weights) in the JSON file format.
- **HDF5** - Stores the complete neural network (with weights) in the HDF5 file format. Do not confuse HDF5 with HDFS. They are different. We do not use HDFS in this class.

Usually, you will want to save in HDF5.

In [2]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)

# Predict
pred = model.predict(x)

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Before save score (RMSE): {score}")
```

```
# save neural network structure to JSON (no weights)
model_json = model.to_json()
with open(os.path.join(save_path,"network.json"), "w") as json_file:
    json_file.write(model_json)

# save entire network to HDF5 (save everything, suggested)
model.save(os.path.join(save_path,"network.h5"))
```

```
Epoch 1/100
13/13 - 1s - loss: 223035.4531 - 729ms/epoch - 56ms/step
Epoch 2/100
13/13 - 0s - loss: 94454.9609 - 26ms/epoch - 2ms/step
Epoch 3/100
13/13 - 0s - loss: 31163.9199 - 26ms/epoch - 2ms/step
Epoch 4/100
13/13 - 0s - loss: 6590.2344 - 24ms/epoch - 2ms/step
Epoch 5/100
13/13 - 0s - loss: 692.0208 - 23ms/epoch - 2ms/step
Epoch 6/100
13/13 - 0s - loss: 110.7149 - 23ms/epoch - 2ms/step
Epoch 7/100
13/13 - 0s - loss: 154.5042 - 22ms/epoch - 2ms/step
Epoch 8/100
13/13 - 0s - loss: 118.5529 - 29ms/epoch - 2ms/step
Epoch 9/100
13/13 - 0s - loss: 91.5691 - 27ms/epoch - 2ms/step
Epoch 10/100
13/13 - 0s - loss: 85.7397 - 24ms/epoch - 2ms/step
Epoch 11/100
13/13 - 0s - loss: 85.6981 - 23ms/epoch - 2ms/step
Epoch 12/100
13/13 - 0s - loss: 85.2837 - 24ms/epoch - 2ms/step
Epoch 13/100
13/13 - 0s - loss: 84.9037 - 23ms/epoch - 2ms/step
Epoch 14/100
13/13 - 0s - loss: 84.6506 - 30ms/epoch - 2ms/step
Epoch 15/100
13/13 - 0s - loss: 84.4048 - 26ms/epoch - 2ms/step
Epoch 16/100
13/13 - 0s - loss: 84.1072 - 24ms/epoch - 2ms/step
Epoch 17/100
13/13 - 0s - loss: 83.9168 - 23ms/epoch - 2ms/step
Epoch 18/100
13/13 - 0s - loss: 83.7391 - 24ms/epoch - 2ms/step
Epoch 19/100
13/13 - 0s - loss: 83.1922 - 21ms/epoch - 2ms/step
Epoch 20/100
13/13 - 0s - loss: 82.9178 - 27ms/epoch - 2ms/step
Epoch 21/100
13/13 - 0s - loss: 82.5835 - 28ms/epoch - 2ms/step
Epoch 22/100
13/13 - 0s - loss: 82.2728 - 24ms/epoch - 2ms/step
Epoch 23/100
13/13 - 0s - loss: 81.9899 - 24ms/epoch - 2ms/step
Epoch 24/100
13/13 - 0s - loss: 81.7262 - 23ms/epoch - 2ms/step
Epoch 25/100
13/13 - 0s - loss: 81.2958 - 26ms/epoch - 2ms/step
Epoch 26/100
13/13 - 0s - loss: 80.9488 - 30ms/epoch - 2ms/step
Epoch 27/100
13/13 - 0s - loss: 80.5811 - 33ms/epoch - 3ms/step
Epoch 28/100
13/13 - 0s - loss: 80.3213 - 25ms/epoch - 2ms/step
```

```
Epoch 29/100
13/13 - 0s - loss: 79.8659 - 27ms/epoch - 2ms/step
Epoch 30/100
13/13 - 0s - loss: 79.5628 - 24ms/epoch - 2ms/step
Epoch 31/100
13/13 - 0s - loss: 79.2613 - 24ms/epoch - 2ms/step
Epoch 32/100
13/13 - 0s - loss: 78.8549 - 23ms/epoch - 2ms/step
Epoch 33/100
13/13 - 0s - loss: 78.3649 - 23ms/epoch - 2ms/step
Epoch 34/100
13/13 - 0s - loss: 78.0478 - 23ms/epoch - 2ms/step
Epoch 35/100
13/13 - 0s - loss: 77.6581 - 30ms/epoch - 2ms/step
Epoch 36/100
13/13 - 0s - loss: 77.1970 - 24ms/epoch - 2ms/step
Epoch 37/100
13/13 - 0s - loss: 76.8659 - 23ms/epoch - 2ms/step
Epoch 38/100
13/13 - 0s - loss: 76.6319 - 23ms/epoch - 2ms/step
Epoch 39/100
13/13 - 0s - loss: 76.0007 - 24ms/epoch - 2ms/step
Epoch 40/100
13/13 - 0s - loss: 75.5929 - 25ms/epoch - 2ms/step
Epoch 41/100
13/13 - 0s - loss: 75.2667 - 26ms/epoch - 2ms/step
Epoch 42/100
13/13 - 0s - loss: 75.3607 - 24ms/epoch - 2ms/step
Epoch 43/100
13/13 - 0s - loss: 74.5779 - 27ms/epoch - 2ms/step
Epoch 44/100
13/13 - 0s - loss: 73.9867 - 21ms/epoch - 2ms/step
Epoch 45/100
13/13 - 0s - loss: 73.7650 - 25ms/epoch - 2ms/step
Epoch 46/100
13/13 - 0s - loss: 73.0263 - 24ms/epoch - 2ms/step
Epoch 47/100
13/13 - 0s - loss: 72.7102 - 23ms/epoch - 2ms/step
Epoch 48/100
13/13 - 0s - loss: 72.2177 - 28ms/epoch - 2ms/step
Epoch 49/100
13/13 - 0s - loss: 71.8469 - 22ms/epoch - 2ms/step
Epoch 50/100
13/13 - 0s - loss: 71.4904 - 28ms/epoch - 2ms/step
Epoch 51/100
13/13 - 0s - loss: 71.1223 - 25ms/epoch - 2ms/step
Epoch 52/100
13/13 - 0s - loss: 70.5943 - 25ms/epoch - 2ms/step
Epoch 53/100
13/13 - 0s - loss: 70.1748 - 21ms/epoch - 2ms/step
Epoch 54/100
13/13 - 0s - loss: 69.8101 - 25ms/epoch - 2ms/step
Epoch 55/100
13/13 - 0s - loss: 69.3219 - 23ms/epoch - 2ms/step
Epoch 56/100
13/13 - 0s - loss: 68.7525 - 22ms/epoch - 2ms/step
```

```
Epoch 57/100
13/13 - 0s - loss: 68.4256 - 22ms/epoch - 2ms/step
Epoch 58/100
13/13 - 0s - loss: 67.8394 - 23ms/epoch - 2ms/step
Epoch 59/100
13/13 - 0s - loss: 67.4138 - 22ms/epoch - 2ms/step
Epoch 60/100
13/13 - 0s - loss: 66.9941 - 33ms/epoch - 3ms/step
Epoch 61/100
13/13 - 0s - loss: 66.6573 - 29ms/epoch - 2ms/step
Epoch 62/100
13/13 - 0s - loss: 66.1712 - 22ms/epoch - 2ms/step
Epoch 63/100
13/13 - 0s - loss: 65.8375 - 29ms/epoch - 2ms/step
Epoch 64/100
13/13 - 0s - loss: 65.3441 - 23ms/epoch - 2ms/step
Epoch 65/100
13/13 - 0s - loss: 64.9143 - 22ms/epoch - 2ms/step
Epoch 66/100
13/13 - 0s - loss: 64.7354 - 24ms/epoch - 2ms/step
Epoch 67/100
13/13 - 0s - loss: 63.8731 - 30ms/epoch - 2ms/step
Epoch 68/100
13/13 - 0s - loss: 63.5211 - 26ms/epoch - 2ms/step
Epoch 69/100
13/13 - 0s - loss: 62.9679 - 22ms/epoch - 2ms/step
Epoch 70/100
13/13 - 0s - loss: 62.6917 - 21ms/epoch - 2ms/step
Epoch 71/100
13/13 - 0s - loss: 62.1212 - 22ms/epoch - 2ms/step
Epoch 72/100
13/13 - 0s - loss: 62.1577 - 32ms/epoch - 2ms/step
Epoch 73/100
13/13 - 0s - loss: 61.1758 - 22ms/epoch - 2ms/step
Epoch 74/100
13/13 - 0s - loss: 61.0303 - 24ms/epoch - 2ms/step
Epoch 75/100
13/13 - 0s - loss: 60.5673 - 23ms/epoch - 2ms/step
Epoch 76/100
13/13 - 0s - loss: 60.0197 - 24ms/epoch - 2ms/step
Epoch 77/100
13/13 - 0s - loss: 59.7046 - 24ms/epoch - 2ms/step
Epoch 78/100
13/13 - 0s - loss: 59.0460 - 25ms/epoch - 2ms/step
Epoch 79/100
13/13 - 0s - loss: 58.6879 - 27ms/epoch - 2ms/step
Epoch 80/100
13/13 - 0s - loss: 58.2086 - 28ms/epoch - 2ms/step
Epoch 81/100
13/13 - 0s - loss: 58.1870 - 40ms/epoch - 3ms/step
Epoch 82/100
13/13 - 0s - loss: 57.3580 - 35ms/epoch - 3ms/step
Epoch 83/100
13/13 - 0s - loss: 57.0140 - 24ms/epoch - 2ms/step
Epoch 84/100
13/13 - 0s - loss: 56.5466 - 36ms/epoch - 3ms/step
```

```
Epoch 85/100
13/13 - 0s - loss: 56.2083 - 30ms/epoch - 2ms/step
Epoch 86/100
13/13 - 0s - loss: 55.7131 - 24ms/epoch - 2ms/step
Epoch 87/100
13/13 - 0s - loss: 55.2924 - 28ms/epoch - 2ms/step
Epoch 88/100
13/13 - 0s - loss: 54.9157 - 26ms/epoch - 2ms/step
Epoch 89/100
13/13 - 0s - loss: 54.8022 - 27ms/epoch - 2ms/step
Epoch 90/100
13/13 - 0s - loss: 53.9416 - 25ms/epoch - 2ms/step
Epoch 91/100
13/13 - 0s - loss: 53.6013 - 30ms/epoch - 2ms/step
Epoch 92/100
13/13 - 0s - loss: 53.3547 - 25ms/epoch - 2ms/step
Epoch 93/100
13/13 - 0s - loss: 52.7261 - 39ms/epoch - 3ms/step
Epoch 94/100
13/13 - 0s - loss: 52.3562 - 25ms/epoch - 2ms/step
Epoch 95/100
13/13 - 0s - loss: 51.9567 - 23ms/epoch - 2ms/step
Epoch 96/100
13/13 - 0s - loss: 51.4552 - 29ms/epoch - 2ms/step
Epoch 97/100
13/13 - 0s - loss: 51.4597 - 23ms/epoch - 2ms/step
Epoch 98/100
13/13 - 0s - loss: 50.6219 - 24ms/epoch - 2ms/step
Epoch 99/100
13/13 - 0s - loss: 50.2118 - 25ms/epoch - 2ms/step
Epoch 100/100
13/13 - 0s - loss: 49.8828 - 25ms/epoch - 2ms/step
Before save score (RMSE): 7.044431690300903
```

The code below sets up a neural network and reads the data (for predictions), but it does not clear the model directory or fit the neural network. The code loads the weights from the previous fit. Now we reload the network and perform another prediction. The RMSE should match the previous one exactly if we saved and reloaded the neural network correctly.

In [3]:

```
from tensorflow.keras.models import load_model
model2 = load_model(os.path.join(save_path,"network.h5"))
pred = model2.predict(x)
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"After load score (RMSE): {score}")
```

```
After load score (RMSE): 7.044431690300903
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [Video] [Notebook]
- Part 3.2: Introduction to Tensorflow and Keras [Video] [Notebook]
- Part 3.3: Saving and Loading a Keras Neural Network [Video] [Notebook]
- **Part 3.4: Early Stopping in Keras to Prevent Overfitting** [Video] [Notebook]
- Part 3.5: Extracting Weights and Manual Calculation [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:

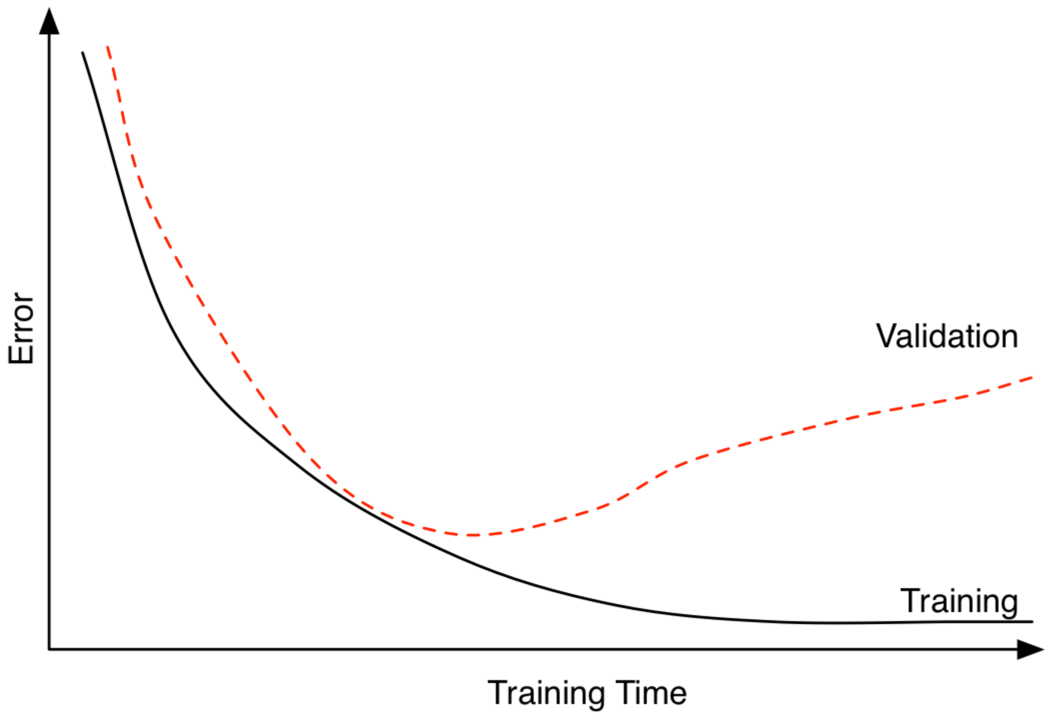
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 3.4: 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



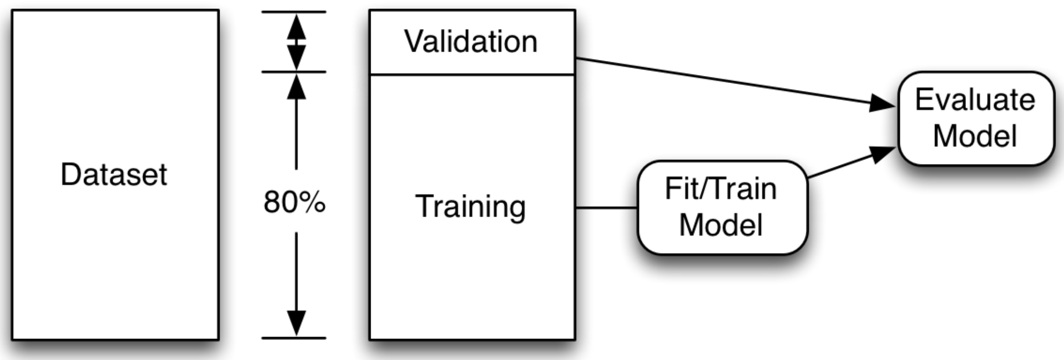
It is important to segment the original dataset into several datasets:

- **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



Early Stopping with Classification

We will now see an example of classification training with early stopping. We will train the neural network until the error no longer improves on the validation set.

In [2]:
```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
        verbose=1, mode='auto', restore_best_weights=True)
```

```python
model.fit(x_train,y_train,validation_data=(x_test,y_test),
        callbacks=[monitor],verbose=2,epochs=1000)
```

```
Train on 112 samples, validate on 38 samples
Epoch 1/1000
112/112 - 0s - loss: 1.1940 - val_loss: 1.1126
Epoch 2/1000
112/112 - 0s - loss: 1.0545 - val_loss: 0.9984
Epoch 3/1000
112/112 - 0s - loss: 0.9533 - val_loss: 0.9130
Epoch 4/1000
112/112 - 0s - loss: 0.8823 - val_loss: 0.8365
Epoch 5/1000
112/112 - 0s - loss: 0.8243 - val_loss: 0.7619
Epoch 6/1000
112/112 - 0s - loss: 0.7592 - val_loss: 0.7059
Epoch 7/1000
112/112 - 0s - loss: 0.7142 - val_loss: 0.6644
Epoch 8/1000
112/112 - 0s - loss: 0.6788 - val_loss: 0.6302
Epoch 9/1000
112/112 - 0s - loss: 0.6481 - val_loss: 0.5979
Epoch 10/1000
112/112 - 0s - loss: 0.6198 - val_loss: 0.5698
Epoch 11/1000
112/112 - 0s - loss: 0.5957 - val_loss: 0.5434
Epoch 12/1000
112/112 - 0s - loss: 0.5738 - val_loss: 0.5189
Epoch 13/1000
112/112 - 0s - loss: 0.5539 - val_loss: 0.4964
Epoch 14/1000
112/112 - 0s - loss: 0.5344 - val_loss: 0.4771
Epoch 15/1000
112/112 - 0s - loss: 0.5177 - val_loss: 0.4601
Epoch 16/1000
112/112 - 0s - loss: 0.5022 - val_loss: 0.4455
Epoch 17/1000
112/112 - 0s - loss: 0.4869 - val_loss: 0.4334
Epoch 18/1000
112/112 - 0s - loss: 0.4786 - val_loss: 0.4236
Epoch 19/1000
112/112 - 0s - loss: 0.4634 - val_loss: 0.4096
Epoch 20/1000
112/112 - 0s - loss: 0.4521 - val_loss: 0.3980
Epoch 21/1000
112/112 - 0s - loss: 0.4409 - val_loss: 0.3872
Epoch 22/1000
112/112 - 0s - loss: 0.4296 - val_loss: 0.3776
Epoch 23/1000
112/112 - 0s - loss: 0.4204 - val_loss: 0.3688
Epoch 24/1000
112/112 - 0s - loss: 0.4113 - val_loss: 0.3598
Epoch 25/1000
112/112 - 0s - loss: 0.4025 - val_loss: 0.3519
Epoch 26/1000
112/112 - 0s - loss: 0.3970 - val_loss: 0.3478
Epoch 27/1000
112/112 - 0s - loss: 0.3860 - val_loss: 0.3382
Epoch 28/1000
```

```
112/112 - 0s - loss: 0.3763 - val_loss: 0.3297
Epoch 29/1000
112/112 - 0s - loss: 0.3678 - val_loss: 0.3213
Epoch 30/1000
112/112 - 0s - loss: 0.3600 - val_loss: 0.3137
Epoch 31/1000
112/112 - 0s - loss: 0.3535 - val_loss: 0.3062
Epoch 32/1000
112/112 - 0s - loss: 0.3451 - val_loss: 0.2995
Epoch 33/1000
112/112 - 0s - loss: 0.3380 - val_loss: 0.2940
Epoch 34/1000
112/112 - 0s - loss: 0.3301 - val_loss: 0.2860
Epoch 35/1000
112/112 - 0s - loss: 0.3228 - val_loss: 0.2791
Epoch 36/1000
112/112 - 0s - loss: 0.3152 - val_loss: 0.2726
Epoch 37/1000
112/112 - 0s - loss: 0.3084 - val_loss: 0.2668
Epoch 38/1000
112/112 - 0s - loss: 0.3009 - val_loss: 0.2608
Epoch 39/1000
112/112 - 0s - loss: 0.2945 - val_loss: 0.2558
Epoch 40/1000
112/112 - 0s - loss: 0.2874 - val_loss: 0.2516
Epoch 41/1000
112/112 - 0s - loss: 0.2818 - val_loss: 0.2437
Epoch 42/1000
112/112 - 0s - loss: 0.2744 - val_loss: 0.2364
Epoch 43/1000
112/112 - 0s - loss: 0.2689 - val_loss: 0.2313
Epoch 44/1000
112/112 - 0s - loss: 0.2612 - val_loss: 0.2268
Epoch 45/1000
112/112 - 0s - loss: 0.2556 - val_loss: 0.2219
Epoch 46/1000
112/112 - 0s - loss: 0.2498 - val_loss: 0.2179
Epoch 47/1000
112/112 - 0s - loss: 0.2443 - val_loss: 0.2111
Epoch 48/1000
112/112 - 0s - loss: 0.2381 - val_loss: 0.2053
Epoch 49/1000
112/112 - 0s - loss: 0.2331 - val_loss: 0.2008
Epoch 50/1000
112/112 - 0s - loss: 0.2273 - val_loss: 0.1956
Epoch 51/1000
112/112 - 0s - loss: 0.2249 - val_loss: 0.1906
Epoch 52/1000
112/112 - 0s - loss: 0.2172 - val_loss: 0.1909
Epoch 53/1000
112/112 - 0s - loss: 0.2170 - val_loss: 0.1943
Epoch 54/1000
112/112 - 0s - loss: 0.2099 - val_loss: 0.1791
Epoch 55/1000
112/112 - 0s - loss: 0.2073 - val_loss: 0.1758
Epoch 56/1000
```

```
112/112 - 0s - loss: 0.2031 - val_loss: 0.1712
Epoch 57/1000
112/112 - 0s - loss: 0.1970 - val_loss: 0.1717
Epoch 58/1000
112/112 - 0s - loss: 0.1907 - val_loss: 0.1648
Epoch 59/1000
112/112 - 0s - loss: 0.1862 - val_loss: 0.1606
Epoch 60/1000
112/112 - 0s - loss: 0.1831 - val_loss: 0.1572
Epoch 61/1000
112/112 - 0s - loss: 0.1840 - val_loss: 0.1590
Epoch 62/1000
112/112 - 0s - loss: 0.1753 - val_loss: 0.1518
Epoch 63/1000
112/112 - 0s - loss: 0.1721 - val_loss: 0.1470
Epoch 64/1000
112/112 - 0s - loss: 0.1706 - val_loss: 0.1443
Epoch 65/1000
112/112 - 0s - loss: 0.1660 - val_loss: 0.1488
Epoch 66/1000
112/112 - 0s - loss: 0.1643 - val_loss: 0.1441
Epoch 67/1000
112/112 - 0s - loss: 0.1598 - val_loss: 0.1390
Epoch 68/1000
112/112 - 0s - loss: 0.1566 - val_loss: 0.1334
Epoch 69/1000
112/112 - 0s - loss: 0.1554 - val_loss: 0.1316
Epoch 70/1000
112/112 - 0s - loss: 0.1519 - val_loss: 0.1315
Epoch 71/1000
112/112 - 0s - loss: 0.1483 - val_loss: 0.1396
Epoch 72/1000
112/112 - 0s - loss: 0.1502 - val_loss: 0.1327
Epoch 73/1000
112/112 - 0s - loss: 0.1441 - val_loss: 0.1229
Epoch 74/1000
112/112 - 0s - loss: 0.1417 - val_loss: 0.1198
Epoch 75/1000
112/112 - 0s - loss: 0.1411 - val_loss: 0.1189
Epoch 76/1000
112/112 - 0s - loss: 0.1365 - val_loss: 0.1207
Epoch 77/1000
112/112 - 0s - loss: 0.1350 - val_loss: 0.1229
Epoch 78/1000
112/112 - 0s - loss: 0.1355 - val_loss: 0.1182
Epoch 79/1000
112/112 - 0s - loss: 0.1320 - val_loss: 0.1152
Epoch 80/1000
112/112 - 0s - loss: 0.1300 - val_loss: 0.1092
Epoch 81/1000
112/112 - 0s - loss: 0.1285 - val_loss: 0.1091
Epoch 82/1000
112/112 - 0s - loss: 0.1258 - val_loss: 0.1140
Epoch 83/1000
112/112 - 0s - loss: 0.1308 - val_loss: 0.1144
Epoch 84/1000
```

```
112/112 - 0s - loss: 0.1259 - val_loss: 0.1027
Epoch 85/1000
112/112 - 0s - loss: 0.1237 - val_loss: 0.1022
Epoch 86/1000
112/112 - 0s - loss: 0.1202 - val_loss: 0.1022
Epoch 87/1000
112/112 - 0s - loss: 0.1180 - val_loss: 0.1049
Epoch 88/1000
112/112 - 0s - loss: 0.1174 - val_loss: 0.1028
Epoch 89/1000
112/112 - 0s - loss: 0.1153 - val_loss: 0.0974
Epoch 90/1000
112/112 - 0s - loss: 0.1167 - val_loss: 0.0946
Epoch 91/1000
112/112 - 0s - loss: 0.1149 - val_loss: 0.0966
Epoch 92/1000
112/112 - 0s - loss: 0.1157 - val_loss: 0.1050
Epoch 93/1000
112/112 - 0s - loss: 0.1122 - val_loss: 0.0930
Epoch 94/1000
112/112 - 0s - loss: 0.1136 - val_loss: 0.0905
Epoch 95/1000
112/112 - 0s - loss: 0.1086 - val_loss: 0.1000
Epoch 96/1000
112/112 - 0s - loss: 0.1118 - val_loss: 0.1087
Epoch 97/1000
112/112 - 0s - loss: 0.1095 - val_loss: 0.0923
Epoch 98/1000
112/112 - 0s - loss: 0.1096 - val_loss: 0.0864
Epoch 99/1000
112/112 - 0s - loss: 0.1138 - val_loss: 0.0856
Epoch 100/1000
112/112 - 0s - loss: 0.1096 - val_loss: 0.1144
Epoch 101/1000
112/112 - 0s - loss: 0.1197 - val_loss: 0.1026
Epoch 102/1000
112/112 - 0s - loss: 0.1064 - val_loss: 0.0827
Epoch 103/1000
112/112 - 0s - loss: 0.1069 - val_loss: 0.0823
Epoch 104/1000
112/112 - 0s - loss: 0.1022 - val_loss: 0.0863
Epoch 105/1000
112/112 - 0s - loss: 0.0992 - val_loss: 0.0933
Epoch 106/1000
112/112 - 0s - loss: 0.1017 - val_loss: 0.0926
Epoch 107/1000
Restoring model weights from the end of the best epoch.
112/112 - 0s - loss: 0.1001 - val_loss: 0.0869
Epoch 00107: early stopping
```

```
Out[2]: <tensorflow.python.keras.callbacks.History at 0x22a9ad34708>
```

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.

In [3]:
```
from sklearn.metrics import accuracy_score

pred = model.predict(x_test)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y_test,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 1.0
```

Early Stopping with Regression

The following code demonstrates how we can apply early stopping to a regression problem. The technique is similar to the early stopping for classification code that we just saw.

In [4]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']
```

```python
# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=5, verbose=1, mode='auto',
        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
        callbacks=[monitor], verbose=2,epochs=1000)
```

```
Train on 298 samples, validate on 100 samples
Epoch 1/1000
298/298 - 0s - loss: 254618.1117 - val_loss: 104859.9187
Epoch 2/1000
298/298 - 0s - loss: 53735.2417 - val_loss: 10033.3467
Epoch 3/1000
298/298 - 0s - loss: 3456.0443 - val_loss: 2832.0205
Epoch 4/1000
298/298 - 0s - loss: 4912.1159 - val_loss: 5504.1926
Epoch 5/1000
298/298 - 0s - loss: 4154.7669 - val_loss: 2042.1780
Epoch 6/1000
298/298 - 0s - loss: 1411.5907 - val_loss: 1259.3724
Epoch 7/1000
298/298 - 0s - loss: 1189.8836 - val_loss: 1435.5145
Epoch 8/1000
298/298 - 0s - loss: 1207.4120 - val_loss: 1259.7002
Epoch 9/1000
298/298 - 0s - loss: 1069.7891 - val_loss: 1189.8975
Epoch 10/1000
298/298 - 0s - loss: 1068.2267 - val_loss: 1188.1633
Epoch 11/1000
298/298 - 0s - loss: 1068.9461 - val_loss: 1175.8650
Epoch 12/1000
298/298 - 0s - loss: 1044.6897 - val_loss: 1185.7492
Epoch 13/1000
298/298 - 0s - loss: 1056.0984 - val_loss: 1178.5605
Epoch 14/1000
298/298 - 0s - loss: 1041.7714 - val_loss: 1157.2365
Epoch 15/1000
298/298 - 0s - loss: 1031.7727 - val_loss: 1146.1638
Epoch 16/1000
298/298 - 0s - loss: 1026.6840 - val_loss: 1140.5295
Epoch 17/1000
298/298 - 0s - loss: 1019.7115 - val_loss: 1131.8495
Epoch 18/1000
298/298 - 0s - loss: 1010.8711 - val_loss: 1122.4224
Epoch 19/1000
298/298 - 0s - loss: 1013.6087 - val_loss: 1111.3609
Epoch 20/1000
298/298 - 0s - loss: 995.9503 - val_loss: 1105.5188
Epoch 21/1000
298/298 - 0s - loss: 987.5903 - val_loss: 1094.2863
Epoch 22/1000
298/298 - 0s - loss: 990.0723 - val_loss: 1089.7853
Epoch 23/1000
298/298 - 0s - loss: 968.7077 - val_loss: 1074.3502
Epoch 24/1000
298/298 - 0s - loss: 968.9280 - val_loss: 1065.4332
Epoch 25/1000
298/298 - 0s - loss: 955.3398 - val_loss: 1055.7287
Epoch 26/1000
298/298 - 0s - loss: 955.5287 - val_loss: 1052.8219
Epoch 27/1000
298/298 - 0s - loss: 935.7177 - val_loss: 1035.1746
Epoch 28/1000
```

```
298/298 - 0s - loss: 938.9435 - val_loss: 1026.7096
Epoch 29/1000
298/298 - 0s - loss: 921.2798 - val_loss: 1021.8623
Epoch 30/1000
298/298 - 0s - loss: 918.7541 - val_loss: 1021.8645
Epoch 31/1000
298/298 - 0s - loss: 903.5642 - val_loss: 994.2775
Epoch 32/1000
298/298 - 0s - loss: 896.2183 - val_loss: 984.4263
Epoch 33/1000
298/298 - 0s - loss: 886.1336 - val_loss: 978.4129
Epoch 34/1000
298/298 - 0s - loss: 877.7422 - val_loss: 964.1715
Epoch 35/1000
298/298 - 0s - loss: 871.3048 - val_loss: 956.3459
Epoch 36/1000
298/298 - 0s - loss: 861.6707 - val_loss: 948.6097
Epoch 37/1000
298/298 - 0s - loss: 850.0068 - val_loss: 932.7441
Epoch 38/1000
298/298 - 0s - loss: 846.9615 - val_loss: 921.6213
Epoch 39/1000
298/298 - 0s - loss: 830.3624 - val_loss: 913.5166
Epoch 40/1000
298/298 - 0s - loss: 831.6781 - val_loss: 907.8736
Epoch 41/1000
298/298 - 0s - loss: 814.4517 - val_loss: 889.8433
Epoch 42/1000
298/298 - 0s - loss: 804.2001 - val_loss: 879.6267
Epoch 43/1000
298/298 - 0s - loss: 793.5329 - val_loss: 869.0650
Epoch 44/1000
298/298 - 0s - loss: 786.6698 - val_loss: 857.7609
Epoch 45/1000
298/298 - 0s - loss: 775.7591 - val_loss: 847.1539
Epoch 46/1000
298/298 - 0s - loss: 767.7103 - val_loss: 836.7088
Epoch 47/1000
298/298 - 0s - loss: 756.9816 - val_loss: 825.8035
Epoch 48/1000
298/298 - 0s - loss: 747.9103 - val_loss: 819.3103
Epoch 49/1000
298/298 - 0s - loss: 739.1126 - val_loss: 805.0508
Epoch 50/1000
298/298 - 0s - loss: 734.6592 - val_loss: 795.2228
Epoch 51/1000
298/298 - 0s - loss: 724.3488 - val_loss: 783.2872
Epoch 52/1000
298/298 - 0s - loss: 710.7389 - val_loss: 779.2385
Epoch 53/1000
298/298 - 0s - loss: 702.9931 - val_loss: 762.7323
Epoch 54/1000
298/298 - 0s - loss: 694.2653 - val_loss: 751.5614
Epoch 55/1000
298/298 - 0s - loss: 682.2225 - val_loss: 744.4663
Epoch 56/1000
```

```
298/298 - 0s - loss: 683.8359 - val_loss: 738.8125
Epoch 57/1000
298/298 - 0s - loss: 673.9678 - val_loss: 723.7866
Epoch 58/1000
298/298 - 0s - loss: 655.8523 - val_loss: 715.6897
Epoch 59/1000
298/298 - 0s - loss: 649.6330 - val_loss: 704.0192
Epoch 60/1000
298/298 - 0s - loss: 643.9476 - val_loss: 691.1572
Epoch 61/1000
298/298 - 0s - loss: 627.9205 - val_loss: 685.6211
Epoch 62/1000
298/298 - 0s - loss: 630.9766 - val_loss: 675.3809
Epoch 63/1000
298/298 - 0s - loss: 620.4021 - val_loss: 664.9146
Epoch 64/1000
298/298 - 0s - loss: 601.4826 - val_loss: 655.5067
Epoch 65/1000
298/298 - 0s - loss: 602.5151 - val_loss: 644.4906
Epoch 66/1000
298/298 - 0s - loss: 584.9831 - val_loss: 631.7765
Epoch 67/1000
298/298 - 0s - loss: 582.3892 - val_loss: 620.7529
Epoch 68/1000
298/298 - 0s - loss: 580.3517 - val_loss: 617.2255
Epoch 69/1000
298/298 - 0s - loss: 575.3606 - val_loss: 603.6507
Epoch 70/1000
298/298 - 0s - loss: 551.4546 - val_loss: 598.6873
Epoch 71/1000
298/298 - 0s - loss: 552.0443 - val_loss: 583.6519
Epoch 72/1000
298/298 - 0s - loss: 536.8391 - val_loss: 576.5555
Epoch 73/1000
298/298 - 0s - loss: 529.9672 - val_loss: 564.6031
Epoch 74/1000
298/298 - 0s - loss: 522.4439 - val_loss: 556.2015
Epoch 75/1000
298/298 - 0s - loss: 513.4194 - val_loss: 548.1135
Epoch 76/1000
298/298 - 0s - loss: 505.7009 - val_loss: 537.8890
Epoch 77/1000
298/298 - 0s - loss: 496.8726 - val_loss: 530.3638
Epoch 78/1000
298/298 - 0s - loss: 488.8692 - val_loss: 520.3936
Epoch 79/1000
298/298 - 0s - loss: 481.2276 - val_loss: 512.5432
Epoch 80/1000
298/298 - 0s - loss: 477.8306 - val_loss: 503.1329
Epoch 81/1000
298/298 - 0s - loss: 473.3998 - val_loss: 494.9358
Epoch 82/1000
298/298 - 0s - loss: 465.8867 - val_loss: 490.6273
Epoch 83/1000
298/298 - 0s - loss: 453.1066 - val_loss: 479.8850
Epoch 84/1000
```

```
298/298 - 0s - loss: 445.6094 - val_loss: 471.7849
Epoch 85/1000
298/298 - 0s - loss: 444.9835 - val_loss: 462.8412
Epoch 86/1000
298/298 - 0s - loss: 443.5763 - val_loss: 456.5965
Epoch 87/1000
298/298 - 0s - loss: 436.6940 - val_loss: 453.5159
Epoch 88/1000
298/298 - 0s - loss: 414.3947 - val_loss: 447.0089
Epoch 89/1000
298/298 - 0s - loss: 416.7841 - val_loss: 433.9080
Epoch 90/1000
298/298 - 0s - loss: 403.5432 - val_loss: 423.4334
Epoch 91/1000
298/298 - 0s - loss: 403.1473 - val_loss: 415.1188
Epoch 92/1000
298/298 - 0s - loss: 390.5989 - val_loss: 408.5711
Epoch 93/1000
298/298 - 0s - loss: 385.0042 - val_loss: 400.7886
Epoch 94/1000
298/298 - 0s - loss: 380.2837 - val_loss: 394.4561
Epoch 95/1000
298/298 - 0s - loss: 382.1260 - val_loss: 388.9179
Epoch 96/1000
298/298 - 0s - loss: 371.1698 - val_loss: 380.5425
Epoch 97/1000
298/298 - 0s - loss: 359.9534 - val_loss: 373.7457
Epoch 98/1000
298/298 - 0s - loss: 358.0036 - val_loss: 366.5114
Epoch 99/1000
298/298 - 0s - loss: 348.6594 - val_loss: 359.0750
Epoch 100/1000
298/298 - 0s - loss: 344.6860 - val_loss: 352.5845
Epoch 101/1000
298/298 - 0s - loss: 338.0005 - val_loss: 345.9701
Epoch 102/1000
298/298 - 0s - loss: 331.2779 - val_loss: 340.2206
Epoch 103/1000
298/298 - 0s - loss: 325.3663 - val_loss: 334.1550
Epoch 104/1000
298/298 - 0s - loss: 319.3072 - val_loss: 327.7170
Epoch 105/1000
298/298 - 0s - loss: 313.7492 - val_loss: 322.3784
Epoch 106/1000
298/298 - 0s - loss: 313.7471 - val_loss: 315.4883
Epoch 107/1000
298/298 - 0s - loss: 304.8789 - val_loss: 309.6435
Epoch 108/1000
298/298 - 0s - loss: 301.6150 - val_loss: 304.9265
Epoch 109/1000
298/298 - 0s - loss: 300.2148 - val_loss: 299.9399
Epoch 110/1000
298/298 - 0s - loss: 289.3050 - val_loss: 292.6603
Epoch 111/1000
298/298 - 0s - loss: 282.8135 - val_loss: 286.8729
Epoch 112/1000
```

```
298/298 - 0s - loss: 283.6183 - val_loss: 281.0534
Epoch 113/1000
298/298 - 0s - loss: 274.6550 - val_loss: 275.6063
Epoch 114/1000
298/298 - 0s - loss: 269.9542 - val_loss: 271.9059
Epoch 115/1000
298/298 - 0s - loss: 265.6656 - val_loss: 265.1887
Epoch 116/1000
298/298 - 0s - loss: 262.1005 - val_loss: 260.1739
Epoch 117/1000
298/298 - 0s - loss: 256.3500 - val_loss: 255.2909
Epoch 118/1000
298/298 - 0s - loss: 251.3900 - val_loss: 252.0265
Epoch 119/1000
298/298 - 0s - loss: 247.2246 - val_loss: 245.5129
Epoch 120/1000
298/298 - 0s - loss: 241.7555 - val_loss: 240.8349
Epoch 121/1000
298/298 - 0s - loss: 237.9977 - val_loss: 236.3335
Epoch 122/1000
298/298 - 0s - loss: 233.5239 - val_loss: 231.7200
Epoch 123/1000
298/298 - 0s - loss: 229.3251 - val_loss: 227.2675
Epoch 124/1000
298/298 - 0s - loss: 225.5864 - val_loss: 222.6441
Epoch 125/1000
298/298 - 0s - loss: 221.2191 - val_loss: 218.1110
Epoch 126/1000
298/298 - 0s - loss: 217.8098 - val_loss: 213.9518
Epoch 127/1000
298/298 - 0s - loss: 214.3937 - val_loss: 210.5598
Epoch 128/1000
298/298 - 0s - loss: 210.2760 - val_loss: 205.6227
Epoch 129/1000
298/298 - 0s - loss: 206.5413 - val_loss: 202.4728
Epoch 130/1000
298/298 - 0s - loss: 202.3109 - val_loss: 197.9401
Epoch 131/1000
298/298 - 0s - loss: 199.8272 - val_loss: 196.1144
Epoch 132/1000
298/298 - 0s - loss: 197.1229 - val_loss: 190.0905
Epoch 133/1000
298/298 - 0s - loss: 192.5514 - val_loss: 186.7910
Epoch 134/1000
298/298 - 0s - loss: 189.2665 - val_loss: 184.1961
Epoch 135/1000
298/298 - 0s - loss: 185.1848 - val_loss: 179.9203
Epoch 136/1000
298/298 - 0s - loss: 186.1516 - val_loss: 176.2954
Epoch 137/1000
298/298 - 0s - loss: 182.4030 - val_loss: 173.4539
Epoch 138/1000
298/298 - 0s - loss: 177.4716 - val_loss: 169.6453
Epoch 139/1000
298/298 - 0s - loss: 173.9908 - val_loss: 166.0001
Epoch 140/1000
```

```
298/298 - 0s - loss: 173.2805 - val_loss: 162.8689
Epoch 141/1000
298/298 - 0s - loss: 172.0611 - val_loss: 159.8967
Epoch 142/1000
298/298 - 0s - loss: 165.5859 - val_loss: 157.3186
Epoch 143/1000
298/298 - 0s - loss: 162.3572 - val_loss: 153.6901
Epoch 144/1000
298/298 - 0s - loss: 161.0297 - val_loss: 151.6905
Epoch 145/1000
298/298 - 0s - loss: 164.1645 - val_loss: 148.8484
Epoch 146/1000
298/298 - 0s - loss: 156.3238 - val_loss: 145.0771
Epoch 147/1000
298/298 - 0s - loss: 152.3051 - val_loss: 142.4923
Epoch 148/1000
298/298 - 0s - loss: 149.8716 - val_loss: 140.4517
Epoch 149/1000
298/298 - 0s - loss: 147.7921 - val_loss: 137.0884
Epoch 150/1000
298/298 - 0s - loss: 144.5433 - val_loss: 134.4882
Epoch 151/1000
298/298 - 0s - loss: 144.0840 - val_loss: 134.6734
Epoch 152/1000
298/298 - 0s - loss: 142.7512 - val_loss: 129.2658
Epoch 153/1000
298/298 - 0s - loss: 138.6744 - val_loss: 126.8691
Epoch 154/1000
298/298 - 0s - loss: 136.3120 - val_loss: 125.7347
Epoch 155/1000
298/298 - 0s - loss: 134.8607 - val_loss: 122.1199
Epoch 156/1000
298/298 - 0s - loss: 132.3261 - val_loss: 120.8815
Epoch 157/1000
298/298 - 0s - loss: 130.2538 - val_loss: 117.9441
Epoch 158/1000
298/298 - 0s - loss: 127.5774 - val_loss: 116.9117
Epoch 159/1000
298/298 - 0s - loss: 128.5830 - val_loss: 114.8769
Epoch 160/1000
298/298 - 0s - loss: 123.8368 - val_loss: 112.2658
Epoch 161/1000
298/298 - 0s - loss: 121.8774 - val_loss: 110.3176
Epoch 162/1000
298/298 - 0s - loss: 121.1990 - val_loss: 108.8108
Epoch 163/1000
298/298 - 0s - loss: 119.1470 - val_loss: 106.4554
Epoch 164/1000
298/298 - 0s - loss: 117.1019 - val_loss: 104.7673
Epoch 165/1000
298/298 - 0s - loss: 114.4462 - val_loss: 102.9108
Epoch 166/1000
298/298 - 0s - loss: 113.8899 - val_loss: 100.6241
Epoch 167/1000
298/298 - 0s - loss: 113.7473 - val_loss: 99.1480
Epoch 168/1000
```

```
298/298 - 0s - loss: 109.9129 - val_loss: 98.5171
Epoch 169/1000
298/298 - 0s - loss: 111.6148 - val_loss: 95.8686
Epoch 170/1000
298/298 - 0s - loss: 109.5533 - val_loss: 97.8955
Epoch 171/1000
298/298 - 0s - loss: 110.5111 - val_loss: 92.7941
Epoch 172/1000
298/298 - 0s - loss: 110.6292 - val_loss: 96.9406
Epoch 173/1000
298/298 - 0s - loss: 108.2353 - val_loss: 90.7488
Epoch 174/1000
298/298 - 0s - loss: 103.7881 - val_loss: 88.2208
Epoch 175/1000
298/298 - 0s - loss: 100.4373 - val_loss: 89.0537
Epoch 176/1000
298/298 - 0s - loss: 100.0941 - val_loss: 85.4782
Epoch 177/1000
298/298 - 0s - loss: 97.8368 - val_loss: 85.8181
Epoch 178/1000
298/298 - 0s - loss: 95.8849 - val_loss: 83.0792
Epoch 179/1000
298/298 - 0s - loss: 94.7138 - val_loss: 84.0111
Epoch 180/1000
298/298 - 0s - loss: 93.9980 - val_loss: 80.8398
Epoch 181/1000
298/298 - 0s - loss: 92.4562 - val_loss: 81.9521
Epoch 182/1000
298/298 - 0s - loss: 91.9720 - val_loss: 81.2425
Epoch 183/1000
298/298 - 0s - loss: 93.9076 - val_loss: 77.1700
Epoch 184/1000
298/298 - 0s - loss: 92.0447 - val_loss: 76.0691
Epoch 185/1000
298/298 - 0s - loss: 92.4003 - val_loss: 77.9899
Epoch 186/1000
298/298 - 0s - loss: 87.6844 - val_loss: 73.9357
Epoch 187/1000
298/298 - 0s - loss: 86.4119 - val_loss: 74.5456
Epoch 188/1000
298/298 - 0s - loss: 85.1260 - val_loss: 73.0177
Epoch 189/1000
298/298 - 0s - loss: 85.2527 - val_loss: 71.2634
Epoch 190/1000
298/298 - 0s - loss: 84.7504 - val_loss: 73.4859
Epoch 191/1000
298/298 - 0s - loss: 83.9971 - val_loss: 70.3122
Epoch 192/1000
298/298 - 0s - loss: 82.2615 - val_loss: 68.4355
Epoch 193/1000
298/298 - 0s - loss: 86.2356 - val_loss: 76.1497
Epoch 194/1000
298/298 - 0s - loss: 82.0077 - val_loss: 70.1432
Epoch 195/1000
298/298 - 0s - loss: 84.0382 - val_loss: 74.0556
Epoch 196/1000
```

```
298/298 - 0s - loss: 79.0808 - val_loss: 65.3704
Epoch 197/1000
298/298 - 0s - loss: 77.5371 - val_loss: 65.6799
Epoch 198/1000
298/298 - 0s - loss: 76.7543 - val_loss: 63.9797
Epoch 199/1000
298/298 - 0s - loss: 75.4548 - val_loss: 65.2337
Epoch 200/1000
298/298 - 0s - loss: 75.2814 - val_loss: 62.7816
Epoch 201/1000
298/298 - 0s - loss: 78.7884 - val_loss: 66.5500
Epoch 202/1000
298/298 - 0s - loss: 74.7617 - val_loss: 62.7047
Epoch 203/1000
298/298 - 0s - loss: 73.3059 - val_loss: 63.5815
Epoch 204/1000
298/298 - 0s - loss: 73.3379 - val_loss: 60.1637
Epoch 205/1000
298/298 - 0s - loss: 72.8527 - val_loss: 59.5174
Epoch 206/1000
298/298 - 0s - loss: 71.3816 - val_loss: 58.9280
Epoch 207/1000
298/298 - 0s - loss: 71.0684 - val_loss: 58.5003
Epoch 208/1000
298/298 - 0s - loss: 69.5999 - val_loss: 58.6842
Epoch 209/1000
298/298 - 0s - loss: 69.6249 - val_loss: 63.3405
Epoch 210/1000
298/298 - 0s - loss: 70.2080 - val_loss: 56.3041
Epoch 211/1000
298/298 - 0s - loss: 68.7671 - val_loss: 56.1137
Epoch 212/1000
298/298 - 0s - loss: 67.4164 - val_loss: 56.0807
Epoch 213/1000
298/298 - 0s - loss: 67.4641 - val_loss: 60.8322
Epoch 214/1000
298/298 - 0s - loss: 70.2280 - val_loss: 55.4504
Epoch 215/1000
298/298 - 0s - loss: 70.7004 - val_loss: 56.4594
Epoch 216/1000
298/298 - 0s - loss: 69.3142 - val_loss: 66.7034
Epoch 217/1000
298/298 - 0s - loss: 70.9057 - val_loss: 52.7473
Epoch 218/1000
298/298 - 0s - loss: 63.8462 - val_loss: 53.7675
Epoch 219/1000
298/298 - 0s - loss: 65.2959 - val_loss: 56.9050
Epoch 220/1000
298/298 - 0s - loss: 63.8828 - val_loss: 52.9221
Epoch 221/1000
298/298 - 0s - loss: 66.2621 - val_loss: 61.4800
Epoch 222/1000
298/298 - 0s - loss: 66.0702 - val_loss: 51.9835
Epoch 223/1000
298/298 - 0s - loss: 62.1414 - val_loss: 50.4767
Epoch 224/1000
```

```
298/298 - 0s - loss: 60.9776 - val_loss: 51.0747
Epoch 225/1000
298/298 - 0s - loss: 61.1262 - val_loss: 49.3356
Epoch 226/1000
298/298 - 0s - loss: 59.9358 - val_loss: 56.2200
Epoch 227/1000
298/298 - 0s - loss: 61.8749 - val_loss: 48.5184
Epoch 228/1000
298/298 - 0s - loss: 59.3500 - val_loss: 49.2315
Epoch 229/1000
298/298 - 0s - loss: 58.7732 - val_loss: 49.6212
Epoch 230/1000
298/298 - 0s - loss: 59.0191 - val_loss: 47.4893
Epoch 231/1000
298/298 - 0s - loss: 58.5962 - val_loss: 52.0647
Epoch 232/1000
298/298 - 0s - loss: 57.5451 - val_loss: 47.0744
Epoch 233/1000
298/298 - 0s - loss: 57.4292 - val_loss: 48.5805
Epoch 234/1000
298/298 - 0s - loss: 57.2974 - val_loss: 46.4830
Epoch 235/1000
298/298 - 0s - loss: 59.5053 - val_loss: 48.0127
Epoch 236/1000
298/298 - 0s - loss: 57.6045 - val_loss: 48.8987
Epoch 237/1000
298/298 - 0s - loss: 55.6797 - val_loss: 45.2071
Epoch 238/1000
298/298 - 0s - loss: 54.9872 - val_loss: 46.9131
Epoch 239/1000
298/298 - 0s - loss: 55.2195 - val_loss: 44.9971
Epoch 240/1000
298/298 - 0s - loss: 54.4574 - val_loss: 47.3636
Epoch 241/1000
298/298 - 0s - loss: 55.7777 - val_loss: 43.9272
Epoch 242/1000
298/298 - 0s - loss: 56.6080 - val_loss: 43.6550
Epoch 243/1000
298/298 - 0s - loss: 53.3914 - val_loss: 44.0960
Epoch 244/1000
298/298 - 0s - loss: 53.3937 - val_loss: 46.4250
Epoch 245/1000
298/298 - 0s - loss: 52.5582 - val_loss: 43.4441
Epoch 246/1000
298/298 - 0s - loss: 52.2242 - val_loss: 42.5886
Epoch 247/1000
298/298 - 0s - loss: 53.1087 - val_loss: 45.4969
Epoch 248/1000
298/298 - 0s - loss: 51.2835 - val_loss: 42.2982
Epoch 249/1000
298/298 - 0s - loss: 51.9679 - val_loss: 42.0797
Epoch 250/1000
298/298 - 0s - loss: 50.6096 - val_loss: 41.9481
Epoch 251/1000
298/298 - 0s - loss: 52.3675 - val_loss: 42.1443
Epoch 252/1000
```

```
298/298 - 0s - loss: 52.0081 - val_loss: 41.5254
Epoch 253/1000
298/298 - 0s - loss: 52.4647 - val_loss: 46.1836
Epoch 254/1000
298/298 - 0s - loss: 49.0224 - val_loss: 40.2575
Epoch 255/1000
298/298 - 0s - loss: 50.8724 - val_loss: 40.5554
Epoch 256/1000
298/298 - 0s - loss: 48.6178 - val_loss: 40.2881
Epoch 257/1000
298/298 - 0s - loss: 48.1621 - val_loss: 40.1415
Epoch 258/1000
298/298 - 0s - loss: 47.9184 - val_loss: 39.6353
Epoch 259/1000
298/298 - 0s - loss: 47.7817 - val_loss: 44.1131
Epoch 260/1000
298/298 - 0s - loss: 48.0547 - val_loss: 38.6934
Epoch 261/1000
298/298 - 0s - loss: 49.1476 - val_loss: 38.5595
Epoch 262/1000
298/298 - 0s - loss: 48.3410 - val_loss: 38.4703
Epoch 263/1000
298/298 - 0s - loss: 47.1575 - val_loss: 43.8495
Epoch 264/1000
298/298 - 0s - loss: 47.5766 - val_loss: 37.7489
Epoch 265/1000
298/298 - 0s - loss: 45.9611 - val_loss: 37.8400
Epoch 266/1000
298/298 - 0s - loss: 45.3411 - val_loss: 37.4187
Epoch 267/1000
298/298 - 0s - loss: 44.8844 - val_loss: 40.0926
Epoch 268/1000
298/298 - 0s - loss: 45.0760 - val_loss: 36.9468
Epoch 269/1000
298/298 - 0s - loss: 45.1810 - val_loss: 40.3046
Epoch 270/1000
298/298 - 0s - loss: 44.8097 - val_loss: 37.5340
Epoch 271/1000
298/298 - 0s - loss: 44.2911 - val_loss: 38.6985
Epoch 272/1000
298/298 - 0s - loss: 43.8413 - val_loss: 37.3905
Epoch 273/1000
298/298 - 0s - loss: 43.3722 - val_loss: 36.7338
Epoch 274/1000
298/298 - 0s - loss: 43.0023 - val_loss: 35.9522
Epoch 275/1000
298/298 - 0s - loss: 43.3070 - val_loss: 42.2387
Epoch 276/1000
298/298 - 0s - loss: 43.3620 - val_loss: 35.6415
Epoch 277/1000
298/298 - 0s - loss: 44.2254 - val_loss: 35.0081
Epoch 278/1000
298/298 - 0s - loss: 43.6141 - val_loss: 35.4647
Epoch 279/1000
298/298 - 0s - loss: 42.5499 - val_loss: 37.3217
Epoch 280/1000
```

```
298/298 - 0s - loss: 42.4206 - val_loss: 36.6365
Epoch 281/1000
298/298 - 0s - loss: 41.8326 - val_loss: 33.9366
Epoch 282/1000
298/298 - 0s - loss: 40.7090 - val_loss: 35.2874
Epoch 283/1000
298/298 - 0s - loss: 41.1847 - val_loss: 35.7322
Epoch 284/1000
298/298 - 0s - loss: 40.2632 - val_loss: 33.2830
Epoch 285/1000
298/298 - 0s - loss: 40.4647 - val_loss: 33.4544
Epoch 286/1000
298/298 - 0s - loss: 41.8345 - val_loss: 33.3342
Epoch 287/1000
298/298 - 0s - loss: 40.1833 - val_loss: 33.5219
Epoch 288/1000
298/298 - 0s - loss: 42.5633 - val_loss: 45.5246
Epoch 289/1000
298/298 - 0s - loss: 43.4740 - val_loss: 32.2915
Epoch 290/1000
298/298 - 0s - loss: 40.7724 - val_loss: 35.0065
Epoch 291/1000
298/298 - 0s - loss: 40.1270 - val_loss: 41.1526
Epoch 292/1000
298/298 - 0s - loss: 41.5003 - val_loss: 35.0315
Epoch 293/1000
298/298 - 0s - loss: 39.4004 - val_loss: 33.8747
Epoch 294/1000
298/298 - 0s - loss: 41.5784 - val_loss: 31.3118
Epoch 295/1000
298/298 - 0s - loss: 38.1686 - val_loss: 31.1514
Epoch 296/1000
298/298 - 0s - loss: 38.6330 - val_loss: 37.5739
Epoch 297/1000
298/298 - 0s - loss: 38.8436 - val_loss: 30.6906
Epoch 298/1000
298/298 - 0s - loss: 37.6227 - val_loss: 32.6170
Epoch 299/1000
298/298 - 0s - loss: 36.6737 - val_loss: 30.3784
Epoch 300/1000
298/298 - 0s - loss: 36.7113 - val_loss: 30.9689
Epoch 301/1000
298/298 - 0s - loss: 36.3901 - val_loss: 31.8580
Epoch 302/1000
298/298 - 0s - loss: 36.4300 - val_loss: 29.8985
Epoch 303/1000
298/298 - 0s - loss: 36.6609 - val_loss: 31.8773
Epoch 304/1000
298/298 - 0s - loss: 39.6073 - val_loss: 29.4928
Epoch 305/1000
298/298 - 0s - loss: 37.2211 - val_loss: 29.9193
Epoch 306/1000
298/298 - 0s - loss: 38.2181 - val_loss: 42.4494
Epoch 307/1000
298/298 - 0s - loss: 41.9627 - val_loss: 36.3420
Epoch 308/1000
```

```
298/298 - 0s - loss: 35.2754 - val_loss: 29.0452
Epoch 309/1000
298/298 - 0s - loss: 34.5570 - val_loss: 28.5060
Epoch 310/1000
298/298 - 0s - loss: 35.0860 - val_loss: 28.4189
Epoch 311/1000
298/298 - 0s - loss: 34.4839 - val_loss: 29.8177
Epoch 312/1000
298/298 - 0s - loss: 37.1565 - val_loss: 27.9970
Epoch 313/1000
298/298 - 0s - loss: 38.1949 - val_loss: 34.7456
Epoch 314/1000
298/298 - 0s - loss: 35.7598 - val_loss: 29.5360
Epoch 315/1000
298/298 - 0s - loss: 34.7382 - val_loss: 30.6052
Epoch 316/1000
298/298 - 0s - loss: 34.0591 - val_loss: 29.3044
Epoch 317/1000
Restoring model weights from the end of the best epoch.
298/298 - 0s - loss: 32.9764 - val_loss: 29.1071
Epoch 00317: early stopping
```

Out[4]:

```
<tensorflow.python.keras.callbacks.History at 0x22a9acc8608>
```

Finally, we evaluate the error.

In [5]:

```python
# Measure RMSE error.  RMSE is common for regression.
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")
```

```
Final score (RMSE): 5.291219300799398
```

In [ ]:

```

```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [Video] [Notebook]
- Part 3.2: Introduction to Tensorflow and Keras [Video] [Notebook]
- Part 3.3: Saving and Loading a Keras Neural Network [Video] [Notebook]
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [Video] [Notebook]
- **Part 3.5: Extracting Weights and Manual Calculation** [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Part 3.5: 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.

In [2]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import numpy as np

# Create a dataset for the XOR function
```

```
x = np.array([
    [0,0],
    [1,0],
    [0,1],
    [1,1]
])

y = np.array([
    0,
    1,
    1,
    0
])

# Build the network
# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

done = False
cycle = 1

while not done:
    print("Cycle #{}".format(cycle))
    cycle+=1
    model = Sequential()
    model.add(Dense(2, input_dim=2, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(x,y,verbose=0,epochs=10000)

    # Predict
    pred = model.predict(x)

    # Check if successful.  It takes several runs with this
    # small of a network
    done = pred[0]<0.01 and pred[3]<0.01 and pred[1] > 0.9 \
        and pred[2] > 0.9
    print(pred)
```

```
Cycle #1
[[0.49999997]
 [0.49999997]
 [0.49999997]
 [0.49999997]]
Cycle #2
[[0.33333334]
 [1.        ]
 [0.33333334]
 [0.33333334]]
Cycle #3
[[0.33333334]
 [1.        ]
 [0.33333334]
 [0.33333334]]
Cycle #4
[[0.]
 [1.]
 [1.]
 [0.]]
```

In [3]:
```
pred[3]
```

Out[3]:
```
array([0.], dtype=float32)
```

The output above should have two numbers near 0.0 for the first and fourth spots (input [0,0] and [1,1]). The middle two numbers should be near 1.0 (input [1,0] and [0,1]). These numbers are in scientific notation. Due to random starting weights, it is sometimes necessary to run the above through several cycles to get a good result.

Now that we've trained the neural network, we can dump the weights.

In [4]:
```
# Dump weights
for layerNum, layer in enumerate(model.layers):
    weights = layer.get_weights()[0]
    biases = layer.get_weights()[1]

    for toNeuronNum, bias in enumerate(biases):
        print(f'{layerNum}B -> L{layerNum+1}N{toNeuronNum}: {bias}')

    for fromNeuronNum, wgt in enumerate(weights):
        for toNeuronNum, wgt2 in enumerate(wgt):
            print(f'L{layerNum}N{fromNeuronNum} \
                  -> L{layerNum+1}N{toNeuronNum} = {wgt2}')
```

```
0B -> L1N0: 1.3025760914331386e-08
0B -> L1N1: -1.4192625741316078e-08
L0N0                    -> L1N0 = 0.659289538860321
L0N0                    -> L1N1 = -0.9533336758613586
L0N1                    -> L1N0 = -0.659289538860321
L0N1                    -> L1N1 = 0.9533336758613586
1B -> L2N0: -1.9757269598130733e-08
L1N0                    -> L2N0 = 1.5167843103408813
L1N1                    -> L2N0 = 1.0489506721496582
```

If you rerun this, you probably get different weights. There are many ways to solve the XOR function.

In the next section, we copy/paste the weights from above and recreate the calculations done by the neural network. Because weights can change with each training, the weights used for the below code came from this:

```
0B -> L1N0: -1.2913415431976318
0B -> L1N1: -3.021530048386012e-08
L0N0 -> L1N0 = 1.2913416624069214
L0N0 -> L1N1 = 1.1912699937820435
L0N1 -> L1N0 = 1.2913411855697632
L0N1 -> L1N1 = 1.1912697553634644
1B -> L2N0: 7.626241297587034e-36
L1N0 -> L2N0 = -1.548777461051941
L1N1 -> L2N0 = 0.8394404649734497
```

In [5]:

```
input0 = 0
input1 = 1

hidden0Sum = (input0*1.3)+(input1*1.3)+(-1.3)
hidden1Sum = (input0*1.2)+(input1*1.2)+(0)

print(hidden0Sum) # 0
print(hidden1Sum) # 1.2

hidden0 = max(0,hidden0Sum)
hidden1 = max(0,hidden1Sum)

print(hidden0) # 0
print(hidden1) # 1.2

outputSum = (hidden0*-1.6)+(hidden1*0.8)+(0)
print(outputSum) # 0.96

output = max(0,outputSum)

print(output) # 0.96
```

```
0.0
1.2
0
1.2
0.96
0.96
```

In [ ]: