MSIS 5223: Tutorial 15 – Neural Networks in Python

1. Biological Neural Networks

Various biological systems send signals through the organism in order to create some kind of response, such as body movement, vision, auditory inputs, speaking, or touch. These signals are electrical, similar in concept to the electrical wires that you find in your home, mobile device, computers, televisions, and cars. The wires in modern electronics typically are made of copper and conduct an electrical current where electrons move from one side to the other.

Other modern conductors of electrical current include various electrolytes. For example, salt water can act as an electrolyte to carry charges. Electrolytes serve a similar purpose in the human body. Athletes, when participating in a sport, drink electrolyte-based solutions to "rehydrate" their bodies. Examples include Gatorade and Powerade. These drinks contain salt, which is essential for conducting electrical currents within liquid.

Within the human body, these electrical currents are performed within neurons by way of the electrolytes. For example, when a basketball player runs across a court, the body uses massive amounts of electrolytes to send signals communicating the legs should move swiftly. When the human body undergoes stress such as this, these electrolytes are "used up" and the body becomes fatigued when no more are available. Sports drinks replenish the available electrolytes within the body.

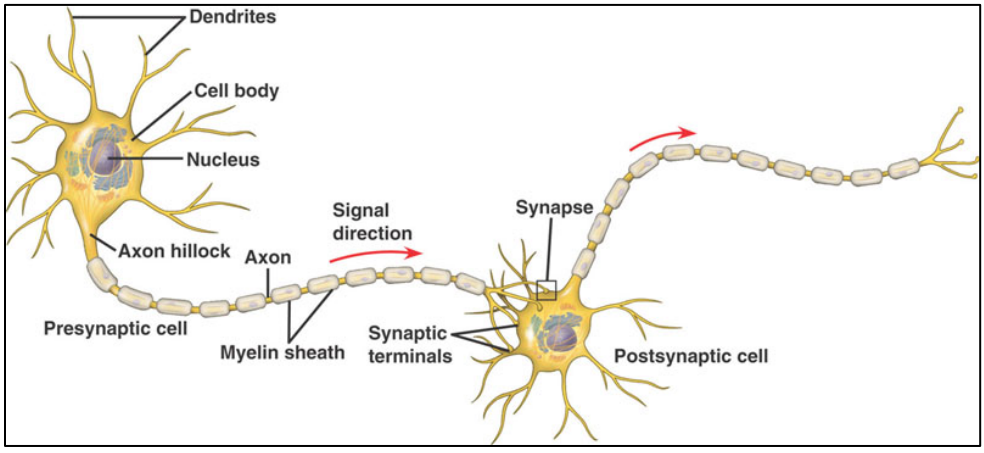

Figure 1 Neural Connections and Their Parts

Figure 1 shows two neurons connected with each other within a human body. A neuron is comprised of three main parts: 1) dendrites, 2) cell body, and 3) the axon. The dendrites act as the input into the cell body. These branches accept incoming electrical signals into the cell. The axon of another neuron attaches to the dendrites of another neuron. These connections are where many memories are stored.

The cell axon is the long "tail" that extends from a neuron. This is the output from a cell body. This output sends the electrical signal from one neuron to another. The axon has a conductive barrier around it comprised of many cells; these cells are myelin sheaths. Think of the myelin sheath as the rubber coating on a wire in your computer. Without it, interference is possible and loss of signal can occur.

The cell body acts as the computer for the electrical signals coming in through the dendrites. The cell body uses a summation function to calculate the signal to pass on. This is done in several ways, such as summation based on the number of inputs, summation based on time, and summation based on a combination of the two.

Why is it important to know all this information about biological neurons when this course is not in biology? This helps you understand how and why an artificial neural network operates. While many benefits exist for using artificial neural networks, knowing how they work helps you identify weaknesses; thus, you develop an understanding of the appropriateness of their usage.

2. Artificial Neural Networks

Artificial neural networks, or ANN for short, are one of many machine learning algorithms. What is machine learning? It is a concept where explicit, defined programs or implementations are not feasible. Many tasks have no explicit scope or boundary. A simple example of this is email filter for spam. Since spam can take multiple forms the computation for pattern matching must be complex and intelligent enough to sift through thousands of emails that can potentially evolve over time.

ANN is a replication of the brain's ability to connect multiple nodes to conduct unstructured machine learning or pattern recognition. ANN has seen many application areas such as finance, marketing, manufacturing, operations, information systems, and

supply chain management. The most common form of ANN is multilayer perceptron (MLP).

Many unstructured problems are not linear, so traditional methods are not capable of solving them. The biological neural network, due to its infinite connections and processing power, is the best system for solving unstructured problems. What better system to mimic than the human brain for such problems? Similar to a biological neural network, an artificial neural network contains the three main parts found in a neuron:

- Input – layer in which data enters the network; this mimics the dendrites of a biological system.
- Hidden – artificial neurons that weight each input variable, performs a Combination Function, and passes the results on to the output layer; this acts as the cell body of a neuron.
- Output – the Transfer Function occurs here and outputs the result based on the nature of the target variable (i.e. categorical, binary, continuous); like an axon, this sends the processed signal.

The *Combination Function* assesses and merges (usually summation) inputs based on weights into a single value. These weights are similar to the coefficients in regression; the larger the weight the more important a variable is. Unlike regression, this is usually non-linear. The *Transfer Function* outputs the merged value using a specified technique similar to regression, logistic regression, classification, etc. Both of these functions combined together are referred to as the *Activation Function*, or the activity of taking inputs in the Combination Function and creating outputs in the Transfer Function.

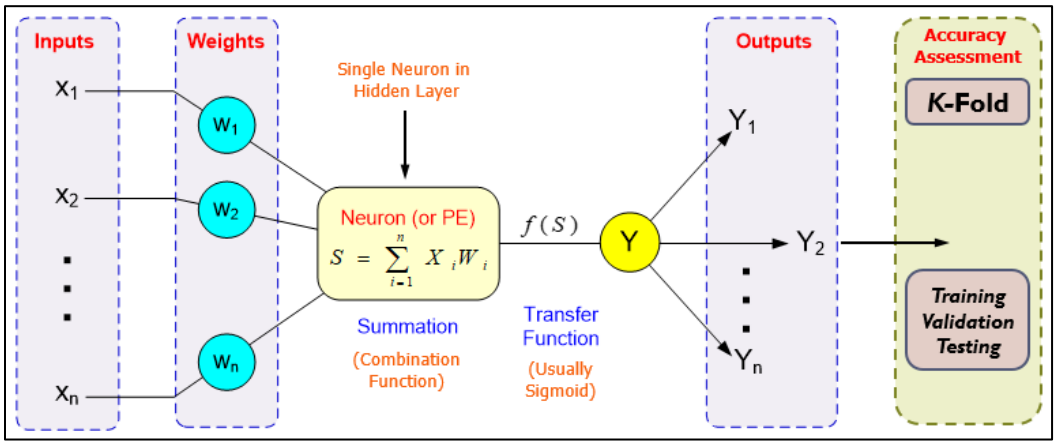

Figure 2 Artificial Neural Network

Within an ANN, the neuron is referred to as the *Processing Element*, or where the Combination Function takes place. Figure 2 provides the overall structure of an ANN. The Input Layer includes the Inputs or variables x_1, x_2, x_3,…x_n where n is the number of variables you have chosen. These variables are the columns you have selected you're your data. Each of these variables, or inputs, is then given a weight by the processing element, or artificial neuron. The weights and the neuron is the Hidden Layer. This name is apt because ANNs are considered black boxes. This will be discussed in a later section. It is possible to have multiple processing elements.

Once the processing elements are complete, the Transfer Function passes the outputs from the Hidden Layer to the Output Layer. Notice, in Figure 2, the Output Layer contains multiple outputs y_1, y_2, y_3,…y_n. The number of outputs depends on the type of function you are using. For example, if you are using a logistic function, then the output will consist of two items; if the function is classification, then the output will be based on the number of categories; if time series, then the number of outputs is based on the number of periods forecasted.

After the Output Layer, if chosen, an accuracy assessment is performed to determine the fit of the model. A word of caution is given here. Like many other techniques, because this is unstructured, the solution provided may not be a global optimum, but a local optimum dependent on the data and the random numbers generated.

Due to the power inherent in artificial neural networks, they have many application areas that have already been covered in this course and some that have not. The following are various applications of ANNs:

- Classification: Feedforward networks (e.g. MLP), radial basis function, and probabilistic NN
- Regression: Feedforward networks (e.g. MLP), radial basis function
- Clustering: Adaptive Resonance Theory (ART) and Self-Organizing Maps
- Association Analysis: Hopfield networks

3. Input Layer

Like many other statistical techniques, the variables should undergo preprocessing prior to usage in the ANN. For example, variables should be restricted to small ranges.

This prevents larger-range variables from appearing more important than smaller-range variables. If you recall, an example used in a previous tutorial compared the variable salary to job satisfaction. Salary can potentially have a range from $1 to $2.5 million while job satisfaction is a 5-point Likert scale ranging from 1 to 5. A value of 5 in both variables is not equal, but the system does not know this. To a certain extent, after multiple iterations, the ANN will decrease the weight of salary to compensate for its large range. This is not always enough and never perfect.

One possible solution to this is standardizing values first. Standardizing continuous variables may use the min-max normalization procedure:

$$x^* = {x - \min(x)}\Big/{range(x)} = {x - \min(x)}\Big/{\max(x) - \min(x)}$$

Another important note about the Input Layer is that ANNs do not handle missing values, unlike decision trees. While many statistical packages give the appearance of this, they either remove records with missing values or automatically impute values. Prior to using an ANN, ensure that your data has no missing values.

Categorical variables should be converted over to numerical values. For variables with fewer categories, use indicator variables. For categorical data with lots of categories, try binning the data. Be careful when working with categorical variables in neural networks when mapping the variables to numbers. Mapping introduces an ordering which the neural network takes into account, such as ordering a categorical variable of marital status.

4. Hidden Layer

At the heart of the Hidden Layer is the Summation Function. This is a specific mathematical formula involving the inputs and their weights. The Summation Function can involve one processing element or multiple elements. In fact, the Summation Function can involve an infinite number of neurons in a single Hidden Layer. The benefit of increasing the number of neurons is the ability to detect patterns within the data more precisely. This benefit comes with a detriment: overfitting the model to the data. This can happen because the ANN memorizes the pattern on the dataset. That is, the model can predict the sample with precision; however, because you want to generalize to the

population, not the sample, the model needs to predict only the shared variance, not the total variance.

The following figures illustrate how a Hidden Layer can be comprised of multiple neurons. The first figure, to the right, is a simple MLP ANN. This ANN has two inputs, x_1 and x_2 with their respective weights. Both inputs and weights route through the Combination Function by way of summation. This model contains a single output, y. The equation used in the Combination Function is provided within the figure. The equation is given as follows:

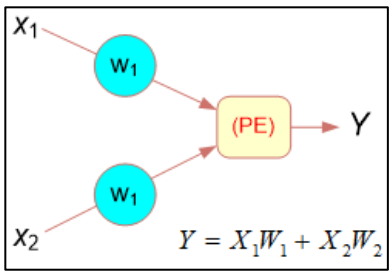

Figure 3 ANN with Single PE

$$y = x_1 w_1 + x_2 w_2$$

This equation looks very similar to regression. As mentioned earlier, the weights are akin to the coefficients in regression. Unfortunately, these weights do not operate like they do in regression. That is, the interpretation is not similar. This is because the Transfer Function plays a role in generating the value of y.

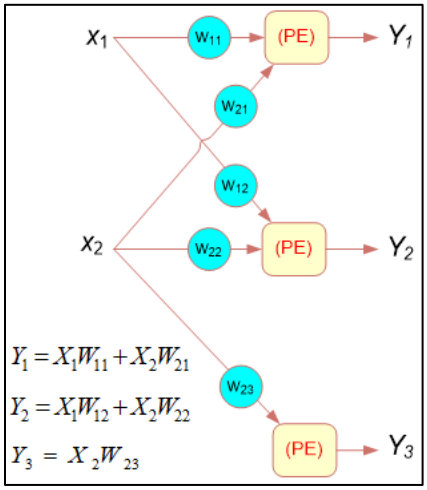

Figure 4 ANN with Three PEs

Figure 4 shows an ANN containing three processing elements. In addition to three processing elements, this ANN also contains three outputs (it is possible to have an ANN with three inputs and a single output). Look at how each input is connected to each output. Do you notice anything odd?

Only the first two outputs have a weight associated with each input. The third output, y_3, only is related to x_2, not x_1. This is a glimpse into the potential complexity neural networks. Also notice the weights associated with each input-output relationship is different than the others. For example, the weight between x_1 and y_1 is w_{11} while the weight between x_1 and y_2 is w_{12}. The equations for the three outputs is shown in the figure and reproduced below:

$$y_1 = x_1 w_{11} + x_2 w_{21}$$

$$y_2 = x_1 w_{12} + x_2 w_{22}$$

$$y_3 = x_2 w_{23}$$

A neural network can have multiple hidden layers, as shown in Figure 5 below. This ANN contains two Hidden Layers. The first layer has three neurons; the second layer has two neurons. The first layer has weights between the processing elements and the inputs. The second layer has weights between the processing elements of the first Hidden Layer and the second Hidden Layer. Like the relationship between an input and processing element, each processing element between two Hidden Layers do not need to have a relationship. Hidden Layers do not need to possess the same number of processing elements either. An ANN can have an infinite number of Hidden Layers. The more layers are added, the longer the computational time on a computer.

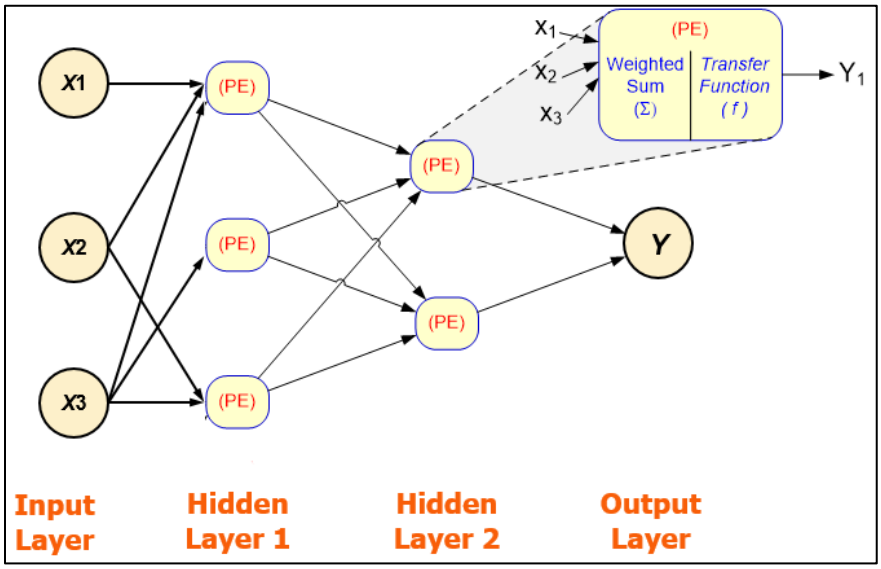

Figure 5 Example of Multiple Hidden Layers

As mentioned earlier, the ANNs are black boxes. No single method exists to determine the best weights (unlike regression and its coefficients). Potentially, an iterative process of trying different neural network models on training, testing, and validation data may reveal how weights operate. In fact, you may have to create two, three, five, etc. numbers of different ANN models. Each ANN model will have different weights; the version with the least error for the testing data is the optimal solution. Once you have a final version for each model, compare their performance accuracy against each other.

5. Output Layer

The last layer in the neural network contains the Transfer Function. Typically, this takes the form of a sigmoid function that is bounded within a specified range. This sigmoid function is S-shaped and usually is a logistic function or a tangential function. Why a sigmoid? A sigmoid contains linear, curvilinear, and constant behavior all in a single function. Looking at Figure 6, this is apparent. This allows a neural network flexibility to tackle a wide variety of data. Regression is limited to linear relationships, and even when augmented by polynomial terms is restricted to specific forms of data. By utilizing complex functions like a sigmoid, ANNs are robust.

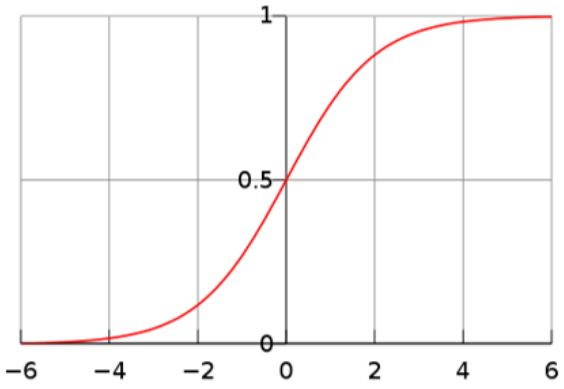

Figure 6 Sigmoid Function

As mentioned, the sigmoid as Transfer Function can be one of two forms. The first is the logistic function given as the following equation:

$$\log(y) = {}^{1}/_{1 + e^{-y}}$$

where *y* is the neuron's value derived from the Combination Function.

The other form is the tangential function, given as follows:

$$\tanh(y) = {}^{e^{2y} - 1}/_{e^{2y} + 1}$$

Neural networks produce continuous values for outputs regardless of the output type (i.e. categorical). For example, if you have a categorical variable with binary values 1 and -1, you may end up with values such as 0.83, 0.67, 0.23, 0.999 or even -0.452. Output values closer to 1 tend toward that value; values closer to -1 tend toward that value. These continuous values are a result of the Transfer Function.

The following example helps illustrate the relationship between the Combination Function and the Transfer Function. The Input Layer contains three variables x_1, x_2, and x_3. Assume this data has multiple rows of data, each a customer of a giant retail chain. The variables are defined as follows:

- x_1 is a 5-point Likert scale indicating length of stay in the store
- x_2 is a 5-point Likert scale for satisfaction of the store appearance

- x_3 is a 5-point Likert scale for service satisfaction

A single row is selected from the data with the following values: x_1 is 3, x_2 is 1, and x_3 is 2. The target variable is a binary response indicating whether the customer purchased something or not. A -1 indicates purchase and a 1 indicates no purchase. Figure 7 illustrates the neural network's operation for this single customer. The processing element determines the weights for x_1, x_2, and x_3 to be 0.2 for w_1, 0.4 for w_2, and 0.1 for w_3 respectively.

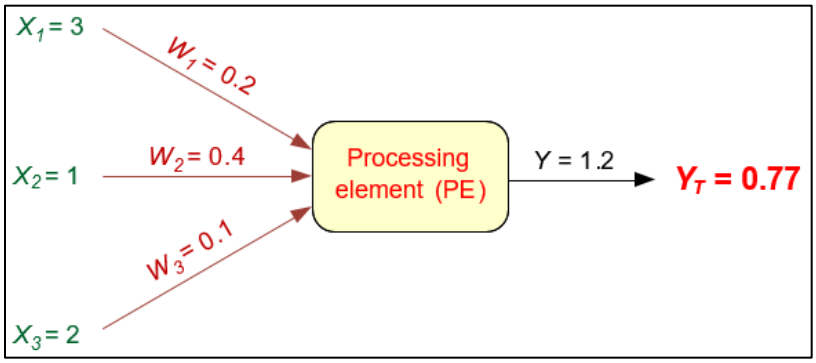

Figure 7 Example Neural Network

The Combination Function uses a simple summation function within the processing element. Each weight is multiplied by its respective input and then derived values are summed together. The Transfer Function uses a logistic function based on the value output from the processing element. The following equations reveal the functions:

Summation Function:

$$y_1 = x_1 w_1 + x_2 w_2 + x_3 w_3$$

$$y = 3(0.2) + 1(0.4) + 2(0.1) = 1.2$$

Transfer Function:

$$y_t = 1/(1 + e^{-y})$$

$$y_t = 1/(1 + e^{-1.2}) = 0.77$$

Assuming the actual value is a binary response between 1 and -1, the predicted value of this row is closer to 1, indicating the customer will not make a purchase. The interpretation of the output depends on the chosen transformations utilized on the data prior to and during the neural network analysis. For example, if a log transformation was used to normalize skewed data, then the log transformation needs to be reversed like in regression.

One common criticism of neural networks is they operate like black boxes. This is because of the lack of explicability; that is, the weights are opaque and the model does not explain itself. A response to this is a sensitivity analysis. The following steps provide an overview of how to perform this type of analysis.

1. Find the average value for each input
2. Measure the output(s) when all inputs are set to the average
3. Measure the output(s) as each input is individually adjusted up and down through its range

Results illustrate the relative importance of input variables compared to each other. Be careful, because this only looks at inputs as individuals—an ANN is a combination of all inputs.

6. ANN Machine Learning

Earlier, we discussed why an artificial neural network is classified as machine learning. A neural network is capable of working on unstructured problems. What was not discussed is how this occurs. You know the basic process of inputs moving through the Combination Function and the Transfer Function, but how does the neural network know what values to assign each weight and whether those values are too high or too low?

The process a Multi-Layer Perceptron Artificial Neural Network uses to identify weight values, and therefore learn, is back propagation. *Back propagation* is a process defined by three steps:

1. Iterate through each record in the data and calculate an output or predicted value.
2. Calculate the error by comparing the predicted value to the actual value
3. The error is sent back to the Input Layer to adjust the weights by determining where the greatest error occurred; weights are adjusted up or down

The adjustment performed by the back propagation method is based on the momentum and the learning rate. The *momentum* is the tendency of a weight to increase or decrease and continue in that direction. The *learning rate* is the extent to which weights change; that is, the size of the jump for weights. This jump is initially very great

and then gradually decreases over time. The GMAT operates similarly to this concept in determining the difficulty of questions to present to you.

A word of caution is given at this point. When an ANN converges on a set of weights, these weights may be a local optimum, not a global optimum. A local optimum is a good fitting model for your sample, but not necessarily for the population. If you apply your local optimum solution to a different sample, you may find great errors in prediction.

Another word of warning is not to include a large number of variables. Just like in regression, more variables leads to the possibility of overfitting your data. Again, this can lead to a local optimum instead of a global optimum.

7. Example of an ANN

An example will help further your understanding of all the previous concepts. This example will extend the previous example of customer purchasing at a giant retailer. This time, an additional processing element will be added. The Input Layer contains three variables as defined:

- x_1 is a 5-point Likert scale indicating length of stay in the store
- x_2 is a 5-point Likert scale for satisfaction of the store appearance
- x_3 is a 5-point Likert scale for service satisfaction

A single row is selected from the data with the following values: x_1 is 3, x_2 is 1, and x_3 is 2. The target variable is a binary response indicating whether the customer purchased something or not. A -1 indicates purchase and a 1 indicates no purchase. The Hidden Layer contains two nodes, Node A, or N_a, and Node B, or N_b. Each input has a relationship with each node. The following weights are randomly selected during the initial phase:

- w_{1a}: Weight between x_1 and N_a with value 0.6
- w_{2a}: Weight between x_2 and N_a with value 0.8
- w_{3a}: Weight between x_3 and N_a with value 0.5
- w_{1b}: Weight between x_1 and N_b with value 0.9
- w_{2b}: Weight between x_2 and N_b with value 0.8
- w_{3b}: Weight between x_3 and N_b with value 0.4

In addition to these weights, because two nodes exist in the Hidden Layer, they have a respective weight with the Output Layer. This model only has a single output node, Y_1. The weights are also equal because the system assumes equality between these two nodes. These weights are given as follows:

- w_{aY1}: Weight between N_a and Y_1 with value 0.85
- w_{bY1}: Weight between N_b and Y_1 with value 0.85

Each layer contains an error value. The ANN uses this error to capture any noise inherent in the data. This is an important aspect of neural networks, because all samples include error or noise, regardless of the statistical technique used.

- x_0: Input Layer error with value 1.0
- w_{0a}: Hidden Layer error weight between x_0 and N_a with value 0.5
- w_{0b}: Hidden Layer error weight between x_0 and N_b with value 0.6
- w_{0Y}: Output Layer error weight with value 0.5

Combination Function

The function used in this example will be a simple linear combination, or a summation function. Each node has a calculated value based on the combination of weights and values. The following table provides a breakdown of all these values.

Input Layer	**Hidden Layer**	
	N_a	N_b
x_0 1.0	w_{0a} 0.5 1.0 * 0.5 = 0.5	w_{0b} 0.6 1.0 * 0.6 = 0.6
x_1 3.0	w_{1a} 0.6 3.0 * 0.6 = 1.8	w_{1b} 0.9 3.0 * 0.9 = 2.7
x_2 1.0	w_{2a} 0.8 1.0 * 0.8 = 0.8	w_{2b} 0.8 1.0 * 0.8 = 0.8
x_3 2.0	w_{3a} 0.5 2.0 * 0.5 = 1.0	w_{3b} 0.4 2.0 * 0.4 = 0.8

Table 1 Combination Function and Values

The first column, Input Layer, contains the values of each input including the error. The second column contains the weights of each input with each node. The last column is part of the combination function. This column is a multiplication of the input with its respective weight. The equations that follow are the combination function results for the first node, N_a.

$$N_a = 0.5x_0 + 0.6x_1 + 0.8x_2 + 0.5x_3$$

$$N_a = 0.5(1.0) + 0.6(3.0) + 0.8(1.0) + 0.5(2.0)$$
$$N_a = 0.5 + 1.8 + 0.8 + 1.0$$
$$N_a = 4.1$$

The second node, N_b also is calculated using the summation function. The following equations provide the linear combination.

$$N_b = 0.6x_0 + 0.9x_1 + 0.8x_2 + 0.4x_3$$
$$N_b = 0.6(1.0) + 0.9(3.0) + 0.8(1.0) + 0.4(2.0)$$
$$N_b = 0.6 + 2.7 + 0.8 + 0.8$$
$$N_b = 4.9$$

Transfer Function

Once the Combination Function has completed calculations for each node and layer, the Transfer Function consolidates the values and pushes the values forward. In this example, we will use a logistic function, as is shown below.

$$\log(y) = {}^{1}/_{1 + e^{-y}}$$

Each node will push through the logistic function, and then the results will be combined in a linear combination (i.e. summation function), and then pushed through a logistic function one final time. The first node is given as follows:

$$f(N_a) = {}^{1}/_{1 + e^{-y}}$$
$$f(N_a) = {}^{1}/_{1 + e^{-4.1}}$$
$$f(N_a) = {}^{1}/_{1.017}$$
$$f(N_a) = 0.98$$

The result of the logistic function is 0.984. The value from the second node, N_b, is pushed through the logistic function as shown below.

$$f(N_b) = {}^{1}/_{1 + e^{-y}}$$
$$f(N_b) = {}^{1}/_{1 + e^{-4.9}}$$
$$f(N_b) = {}^{1}/_{1.0075}$$
$$f(N_b) = 0.99$$

The result of the logistic function is a value of 0.993. Both of the values returned from the logistic function are similar, which is a good indication that the neural network is operating well.

Once the logistic functions compute their respective values, they need to be run through a summation function. At this point, the error for the Output Layer is taken into consideration. The summation function is given below.

$$net_{ab} = 0.5N_0 + 0.85N_a + 0.85N_b$$
$$net_{ab} = 0.50(1.0) + 0.85(0.98) + 0.85(0.993)$$
$$net_{ab} = 0.50 + 0.84 + 0.84$$
$$net_{ab} = 2.18$$

With the summation function complete, the value is run through a logistic function, similar to the original node values.

$$f(net_{ab}) = y_t = {}^{1}/_{1 + e^{-2.18}}$$
$$y_t = {}^{1}/_{1 + e^{-2.18}}$$
$$y_t = 0.90$$

The result of this process is the predicted value of 0.90. Recall, target variable is binary with possible values of 1 and -1. The predicted value of 0.90 is neither 1 or -1. This is because of the logistic function applied during the Transfer Function. Since it is a sigmoid function, the predicted value is a close approximation of the actual value. In this case, because 0.90 is closer to the actual value of 1, then for all intents and purposes the predicted value can be considered “1”.

Back Propagation

The last step is to perform the back propagation. In this step the error is calculated. The error is based on the difference between the predicted value and the actual value. After the error score is calculated, the score is returned to the beginning of the ANN and the weights are adjusted for another pass.

With a typical dataset, the error score is calculated for each row of data, not just a single row like this example. This is important because a single error score may indicate drastic changes to the weights while the rest indicate little change. In other words, a distribution of error scores is created for an entire dataset and potentially some outlying

error scores are created. If you rely on a single error score to adjust the weights, and that error score is an outlying value, then the weight adjustments will falsely change.

Returning to the example, the predicted value is 0.90 and the actual score in the data is 1.0. The error is calculated using the following equation where y_t is the predicted value and y_i is the actual value.

$$error = y_t(1 - y_t)(y_i - y_t)$$
$$error = 0.90(1 - 0.90)(1.0 - 0.90)$$
$$error = 0.90(0.10)(0.10)$$
$$error = 0.009$$

Once the error is calculated, the error score is then used to calculate the change in weight. This is done using the selected learning rate. For this example, the learning rate is 0.17, and is represented as *lr* in the equation below. The calculation is given as follows:

$$weight\ change = lr(error)(1)$$
$$weight\ change = 0.17(0.009)(1)$$
$$weight\ change = 0.0015$$

This weight change results in a positive value. A positive value is an indication that the predicted value is too low. If the result had been negative, then that would indicate the predicted value was too high. To adjust the weights, the weight change is added to the Output Layer error weight w_{0Y}. The value of the error weight is 0.5. The new error weight is calculated as

$$w_{0Y} = 0.5 + 0.0015$$
$$w_{0Y} = 0.5015$$

Notice the adjustment in the weight appears to be miniscule. This is because the learning rate is low, a value of 0.17. These miniscule adjustments will require a multitude of iterations in order to converge on a good solution. To increase the speed by which the adjustment happens, the learning rate can be increased to a higher value. Be aware, that as the learning rate increases, the likelihood of a global optimum solution decreases.

8. Coding in Python: Regression

This tutorial introduces three new modules from SciKit Learn to aid in data mining. First, SciKit Learn contains a neural network package that we will use for both regression and classification. Second, it contains a simple library that splits data into training and testing subsamples. The last library is the preprocessing package, which provides the ability to scale data; remember, scaling provides quicker processing and a less-biased neural network model. The following screenshot reveals the libraries imported for this tutorial.

```
#Neural Network
from sklearn.neural_network import MLPRegressor
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import train_test_split
from sklearn import preprocessing
```

Figure 8 Libraries for Neural Network

The regression neural network will use the ozone data that was used previously in the regression tutorial. Recall the data contains four variables *radiation*, *temperature*, *wind*, and *ozone* where *ozone* is the target variable. All four columns of data are numeric.

Prior to performing the neural network analysis, the data must be prepared by splitting the data and then scaling the two samples. The function train_test_split() is used to split the data into training and testing samples. The output from the function is an array of four elements in order: training data predictor variables, testing data predictor variables, training data target variable, and testing data target variable. The next figure contains the code for splitting the ozone data.

```
In [9]: ozone_data_train, ozone_data_test, y_train, y_test = train_test_split(ozone_data[['rad','wind','temp']],
   ...:                                                                       ozone_data.ozone,
   ...:                                                                       test_size=0.4)
   ...:
```

Figure 9 Splitting Data into Training and Testing

While the function contains many arguments, the three most important are used in the example above. The first argument passes the predictor variables into the function; in this case the variables *radiation*, *wind*, and *temperature*. The second argument is the target variable, *ozone*. The third argument is the size of the testing dataset based on the percentage of the total data. In this example, the testing dataset will contain 40% of the data from the ozone data, leaving 60% for training.

Once the splitting is complete, the next step is to scale the data so all inputs for the neural network have the same "influence." Figure 10 provides the code and results after completion. After creating a scaler object, the scaling is based on the training data. This scaling will then be forced onto the training and testing data.

```
In [14]: scaler = preprocessing.StandardScaler()

In [15]: scaler.fit(ozone_data_train)
Out[15]: StandardScaler(copy=True, with_mean=True, with_std=True)

In [16]: ozone_data_train = scaler.transform(ozone_data_train)

In [17]: ozone_data_test = scaler.transform(ozone_data_test)
```

Figure 10 Scaling Training and Testing Data

This is an odd process because the scaling for the training data is applied to itself. The reason this happens is the scaling is a normalization process. Look at Figure 11 below. The output on the left is the first five rows of unscaled training data. The output on the right is the first five rows of the scaled data. Recall, the purpose of normalization is to force the mean of the data to zero.

```
Out[23]:
    rad  wind  temp
60   24  13.8    81
30   37   9.2    65
41  175   7.4    89
25  291  13.8    90
68   71  10.3    77
```

```
In [26]: ozone_data_train[0:5,:]
Out[26]:
array([[-1.66223566,  1.03158404,  0.28211157],
       [-1.51827703, -0.25355718, -1.40099474],
       [ 0.00989925, -0.75643852,  1.12366472],
       [ 1.29445323,  1.03158404,  1.22885887],
       [-1.14176983,  0.0537592 , -0.13866501]])
```

Figure 11 Original Data vs. Scaled

The data can be run through the neural network function. The code bellows shows the process and output. This is very similar to the process of regression, classification, and clustering in Python. The activation function uses a logistic function and stochastic gradient descent as the solver. Additionally, this neural network has two hidden layers, each with 20 processing elements.

```
nnreg1 = MLPRegressor(activation='logistic', solver='sgd',
                      hidden_layer_sizes=(20,20),
                      early_stopping=True)
nnreg1.fit(ozone_data_train, y_train)

nnpred1 = nnreg1.predict(ozone_data_test)
```

Figure 12 Neural Network Regressor

How good is this neural network model? Using the metrics library, error scores can be obtained including the R-Square. Specifically, the mean absolute error and the

mean squared error are given (see Figure 13 below). These error values are extremely high. Compare these results with those from the multiple regression model used in the earlier tutorial (see Figure 14). Regressing performs better overall.

```
In [47]: metrics.mean_absolute_error(y_test, nnpred1)
Out[47]: 26.16939070417445

In [48]: metrics.mean_squared_error(y_test, nnpred1)
Out[48]: 874.15133702654839

In [49]: metrics.r2_score(y_test, nnpred1)
Out[49]: -0.096733545481151317
```

Figure 13 ANN Model Performance Metrics

```
In [50]: metrics.mean_absolute_error(y_test, linpred1)
Out[50]: 14.602328389592788

In [51]: metrics.mean_squared_error(y_test, linpred1)
Out[51]: 333.43674585149421

In [52]: metrics.r2_score(y_test, linpred1)
Out[52]: 0.58166137946397656
```

Figure 14 Regression Model Performance Metrics

These results do not indicate that neural networks cannot perform better, but the model we specified is not very good. As this data is linear, using a linear function may prove better. The neural network for SciKit Learn includes the rectified linear unit function or RELU. Using this for the activation function results in major improvements to the error as shown in Figure 15.

```
In [73]: metrics.mean_absolute_error(y_test, nnpred2)
Out[73]: 16.118731182601032

In [74]: metrics.mean_squared_error(y_test, nnpred2)
Out[74]: 422.84174367728008

In [75]: metrics.r2_score(y_test, nnpred2)
Out[75]: 0.46949148839827104
```

Figure 15 ANN Model Using RELU

While not as good as multiple regression, this is still very good. In addition to using RELU as the activation function, the number of hidden layer nodes was removed allowing python to dictate the number. How many nodes were used by Python? The code below reveals 100 nodes were used in a single layer.

```
In [76]: nnreg2.hidden_layer_sizes
Out[76]: (100,)
```

Figure 16 Nodes and Layers

9. Coding in Python: Classification

The data for the classification example uses the taxonomy data containing eight columns: *Taxon*, *Petals*, *Internode*, *Sepal*, *Bract*, *Petiole*, *Leaf*, *Fruit*. The target variable, *Taxon*, contains the four values I, II, III, and IV indicating the island the plants are from. The other variables are all numeric. Like the ozone data, the taxonomy data is split into training and testing, this time placing 30% of the data into testing. Scaling is based on the training subset. The figure below presents the code for the classifier neural network. Again, keep in mind this is a multilayer perceptron neural network.

```
nnclass1 = MLPClassifier(activation='logistic', solver='sgd',
                         hidden_layer_sizes=(100,100))
nnclass1.fit(taxon_data_train, taxon_train)

nnclass1_pred = nnclass1.predict(taxon_data_test)
```

Figure 17 Classifier Neural Network

A confusion matrix in text form and a gradient-color-based form are presented below. For the color-based confusion matrix, remember that the more red the color is, the more accurate the prediction. The accuracy of this model is atrocious, to say the least. This is very apparent from Figure 20 which provides the precision of the prediction.

```
In [109]: cm = metrics.confusion_matrix(taxon_test, nnclass1_pred)
     .....: print(cm)
[[ 0  9  0  0]
 [ 0  6  0  0]
 [ 0  8  0  0]
 [ 0 13  0  0]]
```

Figure 18 Confusion Matrix

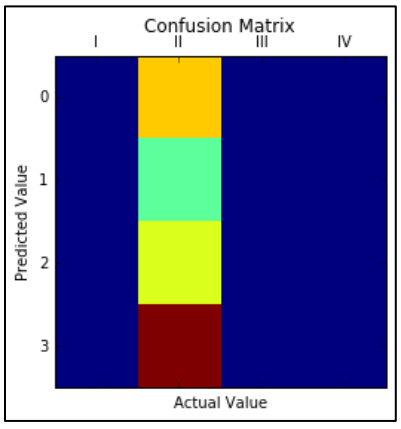

Figure 19 Color-Based Confusion Matrix

```
In [111]: print(metrics.classification_report(taxon_test, nnclass1_pred))
             precision    recall  f1-score   support

          I       0.00      0.00      0.00         9
         II       0.17      1.00      0.29         6
        III       0.00      0.00      0.00         8
         IV       0.00      0.00      0.00        13

avg / total       0.03      0.17      0.05        36
```

Figure 20 Classification Report of ANN

The model is tried again, this time using RELU instead of logistic. The confusion matrix and the classification report are given below in Figure 21 and Figure 22 respectively. A definite improvement is seen over the previous model. This model still has far to go and may possibly need more layers or more nodes per layer.

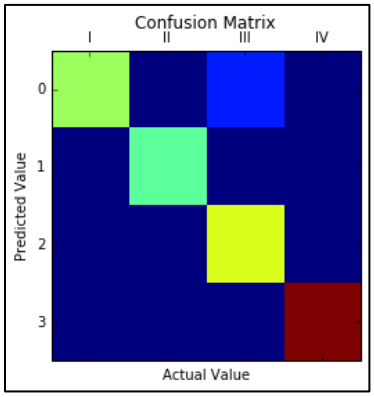

Figure 21 Confusion Matrix of RELU ANN

```
             precision    recall  f1-score   support

          I       1.00      0.78      0.88         9
         II       1.00      1.00      1.00         6
        III       0.80      1.00      0.89         8
         IV       1.00      1.00      1.00        13

avg / total       0.96      0.94      0.94        36
```

Figure 22 Improvement in Accuracy

10. Time Series Neural Networks

Due to the complexity and robustness of neural networks—due in part to the Combination Function and Transfer Function—they are capable of performing various classes of analyses. As discussed in the previous tutorial, some of these include classification, regression, clustering, and association analysis. In addition to these, neural networks can be used for time series analysis.

Recall that an artificial neural network (ANN) can have multiple inputs. A time series can also have multiple inputs depending on the number of autoregressive and moving average components. An ANN can mimic a time series, such as an ARMA(3, 0) by representing each autoregressive component with a neural network input. Figure 1 illustrates this idea.

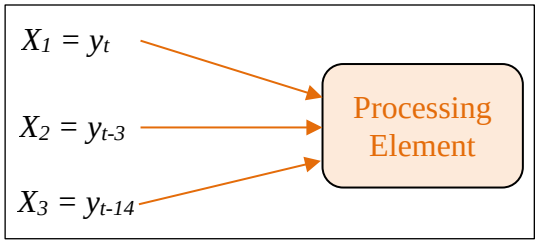

Figure 23 Time Series ANN

Assume that the two autoregressive parameters are lags of 3 and 14. Figure 23 shows a neural network with three inputs where x_1 is the current time period, x_2 is Lag 3, and x_3 is Lag 14. In this way, each lagged autoregressive component is represented within the neural network. Each lag effect is an input in the system. If more lag effects are added, then more inputs are connected to the processing element.

The example will use the unemployment data from Maine using the file maine_unemployment.txt. This dataset contains three columns: *unemployment*, *month* and *year*. In the tutorial on time series, this data exhibited an ARMA(2, 1) model with a second-order and a sixth-order lag effect. The data does not contain these autoregressive lag effects nor will any ANN libraries in Python automatically add these columns for you. As a data scientist, you need to create and add these columns into the data.

The process is fairly straight forward: 1) Copy the original column of data, 2) add in empty data to "push" the data into the past, and 3) add the new data back into the original dataframe. The code for the entire

```
#==============================
# Step 1: Make a copy of data
#==============================
s1 = maine_data.unemploy

#================================
# Step 2: Create Lag Effects by
# adding in empty data, or zeroes
#================================
#Lag 2 Effect
lag2col = pd.Series([0,0])
lag2col = lag2col.append(s1, ignore_index=True)
lag2col = lag2col.ix[0:127,]

#Lag 6 Effect
lag6col = pd.Series([0,0,0,0,0,0])
lag6col = lag6col.append(s1, ignore_index=True)
lag6col = lag6col.ix[0:127,]

#====================================
# Step 3: Add data back into dataframe
#====================================
newcols1 = pd.DataFrame({'lag2': lag2col})
maine_data2 = pd.concat([maine_data, newcols1], axis=1)

newcols2 = pd.DataFrame({'lag6': lag6col})
maine_data3 = pd.concat([maine_data2, newcols2], axis=1)
```

Figure 24 Creating Lag Effects in Python

process is shown in Figure 24. Step one is simple and could be skipped if you want to directly reference the *unemploy* column, but makes it easier to work with. Step two is more complicated. This requires you to visualize what the lag effect should look like. A second-order lag effect means the unemployment of the current time period y_t is influenced by the unemployment y_{t-2}, or two time periods ago (that is, two months ago). The table below illustrates this idea. The number of new data points to add is equal to the order of the lag. For a lag effect on the order of two, add in two rows of empty data; for a lag of six, add in six rows.

Time	Data	Lag of 2	Lag of 6
1996 Jan	6.7		
1996 Feb	6.7		
1996 Mar	6.4	6.7	
1996 Apr	5.9	6.7	
1996 May	5.2	6.4	
1996 Jun	4.8	5.9	
1996 Jul	4.8	5.2	6.7
1996 Aug	4.0	4.8	6.7
1996 Sep	4.2	4.8	6.4
1996 Oct	4.4	4.0	5.9

Table 2 Lag Effects in Unemployment Data

Return to Figure 24 and look at Step 2. The first line of code creates "empty space" for two time periods. Instead of using "null" or "N/A" for empty space a "0" is used in time series data. The same process is followed for adding in a lag effect of six, where six zeroes are created and added to beginning of the data. After running the code for creating the lag effect of 2, the new column will contain two zeroes in the first two rows, as shown in the figure to the right.

```
In [13]: lag2col
Out[13]:
0       0.0
1       0.0
2       6.7
3       6.7
4       6.4
5       5.9
6       5.2
7       4.8
8       4.8
```

Figure 25 Lag 2 Column

The last step, Step 3, takes the newly created columns and adds them to the dataset. Your dataframe should look like the following figure. The lag effects appear correct with the appropriate lagging in the past.

	unemploy	lag2	lag6	month	year
0	6.7	0.0	0.0	1	1996.000000
1	6.7	0.0	0.0	2	1996.000000
2	6.4	6.7	0.0	3	1996.000000
3	5.9	6.7	0.0	4	1996.000000
4	5.2	6.4	0.0	5	1996.000000
5	4.8	5.9	0.0	6	1996.000000
6	4.8	5.2	6.7	7	1996.000000
7	4.0	4.8	6.7	8	1996.000000
8	4.2	4.8	6.4	9	1996.000000
9	4.4	4.0	5.9	10	1996.000000
10	5.0	4.2	5.2	11	1996.000000
11	5.0	4.4	4.8	12	1996.000000
12	6.4	5.0	4.8	1	1997.000000
13	6.5	5.0	4.0	2	1997.000000
14	6.3	6.4	4.2	3	1997.000000

Figure 26 Unemployment Data with Lag Effects

Now that the lag effects are added to the dataframe one last step must be taken to format the data. The time variable is not usable in its current state. Remember, unemployment is regressed onto time. The time variable needs to represent an increase over time, which currently neither month nor year do that: month repeats every 12 data points and year is not an incremental variable.

To create this new variable, simply use the index to create an incremental, single digit variable. The figure below provides the code for doing this. Take the time to understand the code presented here as well as that given for creating lag effects. None of this code should be new to you as most of it was presented in Tutorial 2 and other tutorials.

```
#======================
# Create Time Variable
#======================
timelen = len(maine_data3.index) + 1
newcols3 = pd.DataFrame({'time': list(range(1,timelen))})
maine_data4 = pd.concat([maine_data3, newcols3], axis=1)

#Finalized data with 2 lag effects
maine_data5 = maine_data4[['unemploy','time','lag2','lag6']]
```

Figure 27 Creating an Incremental Time Variable

Now that the data is polished and prepared it is time to split the data into training and testing subsets. A warning needs to be given here. Time series data cannot be randomly drawn and placed into subsets. The data is dependent on itself and random selection violates and removes the dependency. The data must be selected in chunks. For

this example, the training data will include 70% of the data; the other 30% is for the testing sample.

```
splitnum = np.round((len(maine_data5.index) * 0.7), 0).astype(int)
splitnum
#### Result: 70% of data includes 90 records

maine_train = maine_data5.ix[0:90,]
maine_test = maine_data5.ix[91:127,]
```

Figure 28 Splitting Data for Time Series

You may be thinking that this form of splitting has inherent weaknesses. For example, the time period the training data covers is not the same as the testing data. This means that model built may not have enough variance to model reality. This is true. Other techniques exist to create a balanced training and testing set of data that allow both to cover the same time period. Unfortunately, these techniques are advanced and beyond the scope of this course. For now, use the method presented above.

With training and testing data at hand, it is possible to use a neural network. Just like the other neural networks, use the training data to create the model and then use the test data to create prediction scores. The MAE and MSE are calculated resulting in 5.5 and 30.6 respectively. Not bad at all for the first run. Like the other ANN examples, this model can be tweaked and adjusted until an optimal solution is found.

```
nnts1 = MLPRegressor(activation='relu', solver='sgd')
nnts1.fit(maine_train[['time','lag2','lag6']], maine_train.unemploy)

nnts1_pred = nnts1.predict(maine_test[['time','lag2','lag6']])

metrics.mean_absolute_error(maine_test.unemploy, nnts1_pred)

metrics.mean_squared_error(maine_test.unemploy, nnts1_pred)
```

Figure 29 Neural Network for Unemployment Data