A NEURAL APPLICATION OF WINE CLASSIFICATION

Hemant Heer

Contents

Introduction

To apply the subject matter from ISE 419 (Applied Soft Computing), neural networks were deemed interesting for solving classification problems. Neural networks take in "inputs" and return "outputs" from previously gathered data to tune the parameters (weights) within the nodal architecture. The following paper dives into the application of supervised learning with categorical classification. In laymen terms, supervised learning uses a "teacher" to classify the samples into classes appropriate within the context of the data provided [1]. To relate classes and categories, the definition of *categorical* in this context is a *label* for the training sample that was fed through the network. This label assumes that humans can understand these classes and that there exists some relationship among all the "features" of a sample from a data set. A datum typically contains attributes, or "features", of interest that are quantitatively measured from the environment.

For example, data taken from breast cancer patients can include patient age, gender, cholesterol level, blood sugar, red blood cell count, white blood cell count, and others. These variables are considered the principal "features" of a single sample. However, a single data point cannot predict the behavior of say, a certain disease, primarily because the model lacks information from the environment. Although the data set being analyzed in this paper does not involve breast cancer data from patients, the principle remains the same. A significant number of samples should be collected so a neural model can approximate classifications. This paper will begin with a general background of the data set used, describe the approach used, provide

a detailed explanation of the implementation, and converge to the results from which conclusions will be generated.

Background on the Application

In order to perform classifications, an artificial neural network was chosen to perform classifications on a wine data set. This data set was donated in 1991 by Stefan Aeberhard to the University of California, Irvine's Machine Learning Repository and contains data on wine types from chemical analyses [2]. Very briefly, wines are produced via the fermentation of grapes and there may be different "strains" of grapes that result in a particular type of wine. The data set in UCI's machine learning repository has a collection of 178 samples that are neatly defined into three different kinds of "cultivars." A cultivar is a grouping of plants that have characteristics specific to that set of plants. In relation to the wine data set, these samples reside in one of three types of cultivars. The first attribute of a sample taken from the data set contains a class identifier. This class identifier is either 1, 2, or 3 and is used to classify that sample into a certain cultivar. The other attributes of the sample include features such as:

1. Alcohol
2. Malic Acid
3. Ash
4. Alcalinity of ash
5. Magnesium
6. Total phenols
7. Flavanoids
8. Nonflavanoid phenols

9. Proanthocyanins
10. Color Intensity
11. Hue
12. OD280/OD315 of diluted wines
13. Proline

These attributes were measured and recorded but units were not provided in the data set's description. Each sample in the data set has the attributes and the class identifier recorded. The site makes clear that there were other attributes recorded but they weren't available, suggesting that the 13 provided attributes were not the only ones measured [2].

This data set provides a set of defined inputs (the attributes for each sample) and outputs (class identifiers) and can be used in the application of an artificial neural network. A neural network can be generated to predict the cultivar that a sample belongs to. Artificial neural networks were seen as a project worthy of consideration primarily because the data site's repository makes it clear that the classes are "separable" [2]. In supervised learning, a separable system allows a neural network to clearly separate samples into different classes [1]. The wine data set contains 13 attributes for each sample, 3 classes that a sample can belong to, and a small data set that would allow time to test modifications to the neural network's architecture. In addition, the class identifier can be used to generate the number of output nodes in the output layer of a neural network. Since the class identifier can range from 1 to 3, appropriate binary values can be used as the "output" for a particular sample. In clearer terms, if the sample belonged to "cultivar 1", the final output can be understood as [1 0 0]. The positions in the row correspond to the nodes in the output layer of the neural network. A value

of 1 in the first output means that the sample belonged to cultivar 1 and zeroes in the 2nd and 3rd output nodes signify that the sample does not belong to those cultivars. As a final example, if the sample belonged to "cultivar 2", the true output would be a [0 1 0]. Since the class identifiers could be represented in a vector containing binary numbers, a neural architecture could be applied appropriately by creating 3 nodes in the output layer and 13 nodes in the input layer and a varying number of hidden neurons. A neural network could be used appropriately in this context.

Background on the Approach

A neural network functions in the same way a brain functions. On a very fundamental level, neurons (analogous to the weights on links within a neural network) are being strengthened over time to classify information every day. Although this is not the technical explanation, neurons fire in response to inputs and this is modeled within a neural network. The most basic neural networks contains an input layer, a hidden layer, an output layer, and a plethora of links connecting every node. Inputs are fed into the input layer and coupled with the weights connecting the input layer to the hidden layer. The weights on these links essentially control the effect certain nodes have on the output in the hidden layer. A singular output for each node in the hidden layer is created and passed through a sigmoidal activation function. This activation function mimics the behavior of neurons firing in response to inputs from the environment. The same process is repeated as the outputs from the hidden layer must reach the output layer.

As in linear regression, the goal of a neural network is to minimize an error function to construct a model that closely fits the data but allows the system to generalize. The most

intuitive explanation would be the creation of a best-fit line in Excel for a simple set of inputs and outputs. In linear regression, inputs are linearly related to the output. However, a neural network is preferable when the attributes are related to each other in a non-linear manner. For the wine classification data set, the color intensity and hue may not have a linear relationship because the other attributes' values affect the cultivar that a sample belongs to. In this case, the 13 inputs that are passed into the neural architecture may not have a simple function that relates them. Therefore, a neural network can act as a "black box" that uses certain parameters, like weights, to behave as a multidimensional non-linear function. In a neural network, weights are manipulated based on the attributes that are passed into the network. As samples are passed into a network, the network produces its best estimate of the output given random weights. This output differs from the gold standard output (the outputs provided by the wine classification data set). The error function is the difference between the true output and the calculated output. Minimizing this and backpropagating the error through the network will tune the parameters so better predictions can be made. Backpropagation occurs via gradient descent and uses partial derivatives to determine the optimal direction to minimize an error function. There are two ways of performing this type of training. Weights can be adjusted as each training sample is fed through the network or partial derivative accumulators can be used to adjust the weights after all the training samples have passed through the network. The latter will be used and discussed in the details of implementation.

Details of Implementation

Preprocessing

The preprocessing stage involves normalization of the entire data set. Since the neural network contains a sigmoidal activation function, it's best to normalize all attributes of the data set between 0 and 1. Feeding the non-normalized data set through the neural network may be a possible solution but it's best to standardize all the attributes because there's a significant amount of variation in the magnitudes of each attribute. If there is a significant difference in the magnitudes of the attributes, then they may alter the predicted output from the neural network. It's best to normalize so training is more efficient and less prone to these significant alterations to the final output [3]. A "linear transformation" is performed on the data set and mapped to values between 0 and 1 [4]. The formula for Max-min normalization is:

$$X_{i,new} = \left(\frac{X_i - min_{attribute}}{max_{attribute} - min_{attribute}}\right) * (new_{max} - new_{min}) + new_{min}$$

This neural network was implemented in Matlab and the wine classification data set was imported as a 178 x 13 matrix. There are 178 training samples and 13 columns for the attributes of each sample. In normalization, the maximum and minimum of that column (the attribute) are taken across all 178 samples. Then, for each particular attribute, the minimum is subtracted from every training sample's value for that attribute and this is divided by the attribute's range. The objective of normalization is to scale between 0 and 1, so the new maximum value is 1 and the new minimum value is 0. This range is multiplied by the quantity in parentheses and the new minimum value (0) is added to the result. Therefore, all the values in attribute 1 are mapped between 0 and 1 and this process is repeated for the other 12 attributes.

Splitting data set into training, validation, and testing sets

After the data set was normalized, it was randomly partitioned into a training set, a validation set, and a testing set. There are 134 samples in the training set, 39 samples in the validation set, and 5 samples in the testing set. In basic pseudocode, the entire application is performed like this:

For each epoch

Import training set, calculate outputs, calculate sum squared error, and adjust weights via backpropagation

Import validation set and calculate sum squared error

end

Import testing set and calculate outputs from network

In addition, within all of these sets of data for the network, the class identifier attribute (otherwise known as the gold standard column) is separated from the inputs (the remaining 13 columns) so the calculations for error in the output layer are easier to program. So the training set has a set of inputs and the corresponding outputs provided from the wine classification data. The same goes for the validation and testing set. The pseudocode above is a high level organization of the implementation used in this project. Backpropagation is only performed when the training set is being used in an epoch primarily because the validation set and testing set should not affect the network's ability to generalize. The validation set and testing sets contain samples that the network haven't seen as to allow the network to provide reasonable results in the output layer.

Initializing network parameters

This is a 13-(2-3)-3 artificial neural network. The input layer contains 13 input neurons (the training sample contains 13 attributes). The hidden layer contains 2-3 hidden neurons so the effect of modifying the number of nodes in the hidden layer could be examined in the results section. Finally, the output layer contains 3 nodes because each sample from the wine classification data set could belong in cultivar 1, cultivar 2, or cultivar 3. Binary values (either a 0 or 1) will be used to signify the cultivar that the sample belongs to. Sample outputs are shown below:

If the sample belongs to cultivar 1, the true output should be

[1 0 0]

If the sample belongs to cultivar 2, the true output should be

[0 1 0]

If the sample belongs to cultivar 3, the true output should be

[0 0 1]

As you can see, the positions in the output vector correspond to the positions of the nodes in the neural architecture.

Weights for the connections from the input to the hidden layer and the connections from the hidden to the output layer are randomly generated. The weights are randomly generated from a standard uniform distribution between 0 and 1. In Matlab, 2 arrays are created for the weights and a snippet of code is shown below for the exact implementation:

```
% m is the number of input nodes

% l is the number of hidden nodes
l = 3;

% n is the number of output nodes
```

```
n = 3;

w1 = rand(l , m + 1) % Weights from input to hidden layer
                     % m+ 1 to account for 13 fan-in units and the bias
                     % neuron
w2 = rand(n , l + 1) % Weights from hidden to output layer, + 1 for the bias neuron
```

L stores the number of hidden neurons and m stores the number of input neurons. The rows are the hidden neurons and the columns for each row are essentially the “weighting” of all the links from each input neuron. Since the number of input neurons is 13, batch training includes a bias neuron when feeding the input layer to the hidden layer. This bias neuron has a weighting for all the connections it makes to the hidden layer.

Moving to the w2 matrix, it is a (n x l+1) matrix.n stores the number of output neurons and l stores the number of hidden neurons. When forward propagating, a bias neuron is included. Each row of the resultant w2 matrix corresponds to a neuron in the output layer and the columns are the “weighting” of the connections from each neuron in the hidden layer to that output node. Explanations for the weights are given because a visualization would look highly complicated for a neural architecture with 13 neurons in the input layer.

The learning factor is also initialized early in the implementation because that value controls the degree to which the weights are manipulated at the end of an epoch. Since weights are floating point numbers, the learning factors adjust those weights with arithmetic. A typical value for the learning rate is 0.1 but values as high as 1 can be used with caution. Larger learning factors may affect the ability of the network to converge to a global minimum.

Finally, the number of epochs are initialized in the implementation because it controls the number of epochs for adjusting the weights from the training set.

```
% Learning factor
rho = 0.1;

% Number of epochs to train for
epochs = 30;
```

Feeding the inputs through the neural network

Initialization of the partial derivative matrices, forward propagation all samples, backpropagation of the error, and weight adjustment are the main processes that occur cyclically every epoch. For example, in Epoch 1, c1 and c2 (the partial derivative accumulators for the weights going to the hidden layer and output layer, respectively) are matrices containing zeroes with the same dimensions as the weight matrices.

```
    % Zero out the partial accumulators every epoch
    c1 = zeros(l,m+1);
    c2 = zeros(n,l+1);
```

After the partial accumulator matrices have been initialized, they will only be used when the back propagation occurs. A detailed examination of the code implementation follows. Assume that we are in the first epoch and only going through the first training sample.

```
Epoch 1

    % Zero out the partial accumulators every epoch
    c1 = zeros(l,m+1);
    c2 = zeros(n,l+1);
```

Since forward propagation and backpropagation are occurring every epoch and with many training samples, it would be redundant to talk about the same processes for every single training sample and epoch.

Forward propagation

A training sample is loaded in as an input vector (1 x 13 vector) and the bias neuron is incorporated as the '0^{th} ' element in this input vector. The bias neuron, a value of 1, is

incorporated according to the batch training paradigm. This vector now contains the bias neuron in addition to the 13 attributes of the sample that were read into the program. This vector is now forward propagated to the hidden layer using the linear basis function. Since there are three nodes in the hidden layer and the input vector has connections to all three nodes in the hidden layer, a dot product is calculated between the input vector and the weight vector. The weight matrix is a 3 x 14 matrix in which the rows correspond to the number of hidden neurons and the columns correspond to the actual weighting of all the connections from each input. An entire row of that matrix is the weighting for all the connections from the input layer to a particular hidden neuron. This dot product is performed for each row of the weight matrix. The final product is 1 x 3 vector containing the values produced from the linear basis function and each element of that vector is passed through a sigmoidal activation function, $\frac{1}{1+e^{-s}}$. Sample code of this implementation is shown below:

```
bias_a0 = 1; % Bias node for input layer
bias_a1 = 1; % Bias node for hidden layer
read_in_input = norm_wine_features_training(r,:);
input_to_hid = [bias_a0,read_in_input];

% Forward propagate
for k = 1:l

 s = sum(input_to_hid.*w1(k,:)); %Linear combination of weights and biases
 a1(1,k) = 1./(1+exp(-s));

end
```

To go from the hidden layer to the output layer requires the addition of a bias neuron in the input vector. The code below shows the bias neuron added as the 0^{th} neuron in this new input vector. The input vector's dimensions are 1 x 4. There are a new set of weights going from the hidden layer to the output layer. This new weight matrix, w_2 , is a 3 x 4 matrix in which each

row corresponds to the weighting of the connections that link the neurons in the hidden layer to a particular neuron in the output layer. A dot product is calculated between the 1 x 4 input vector and a row of the w_2 matrix. The result from this dot product is passed through the sigmoidal activation function. After all rows of the weight matrix have been completed, the final output, a_2, is a 1 x 3 vector that attempts to resemble the gold standard output. Sample code of this implementation is shown below.

```
        hid_to_output = [bias_a1,a1];

        for j = 1:n
            s = sum(hid_to_output.*w2(j,:)); % Linear combination of weights and
biases
            a2(1,j) = 1./(1+exp(-s));
        end
```

Back Propagation

After the output nodes have been calculated, it's now time to backpropagate the error through the network to adjust the weights accordingly. Each training sample contains a gold standard 1 x 3 vector. It's important to calculate the error difference between each element in the output vector and the gold standard vector. This value is called epsilon. For each output neuron, this epsilon value is calculated, multiplied with the corresponding output, and multiplied with the expression (1 – the corresponding output). That is a 1 x 3 vector containing part of the partial derivative (d_2). Since each output neuron has an epsilon error, a matrix containing the errors for each training sample is generated. This is an r x n matrix and the program refers to this as the "epsilon tracker" matrix.

```
        % Backward propagate

        for j = 1:n
            epsilon = a2(1,j) - raw_wine_outputs_training(1,j); % Difference between
calculated output and true output
```

```
            epsilon_tracker(r,j) = epsilon;
            d2(1,j) = epsilon.*a2(1,j).*(1 – a2(1,j));
        end
```

The partial derivative matrix, c_2 , from the hidden to the output layer is being populated. D_2 contains part of the partial derivative matrix and must be multiplied by the output, a_1, in the hidden layer.

```
        for j = 1:n
            for k = 1:l
                c2(j,k) = c2(j,k) + d2(1,j).*a1(1,k); % Partial derivative matrix
from hidden to output layer
            end
        end
```

The partial derivative matrix from the input to the hidden layer is being created here using the same principles. D_1 partially contains the partial derivative so it must be multiplied by the actual inputs that existed in the input layer (input_to_hid). The c1 matrix is a 3 x 14 matrix and so is the c2 matrix.

```
        for k = 1:l

            stuff = 0;
            for j = 1:n
                stuff = stuff + w2(j,k).*d2(1,j);
            end

            d1(1,k) = a1(1,k).*(1-a1(1,k)).*stuff;
        end

        for k = 1:l
            for i = 1:m+1
                c1(k,i) = c1(k,i) + d1(1,k).*input_to_hid(1,i);
            end
        end
```

After all the training samples have passed through, the weights are updated using the partial derivative matrices that were created. The partial derivative matrices, c1 and c2, are multiplied by the learning factor, ρ, and a subtracted from the weight matrices, w1 and w2. The

code below uses for loops in a complicated and non-intuitive manner but it boils down to basic matrix subtraction. Both w1 and c1 are 3 x 14 matrices while w2 and c2 3 x 4 matrices. In epoch 1, the randomly generated weights are adjusted according to the error difference between the gold standard output and neural network's output.

```
%%% Updating neural network weights %%%%%%%%%%%

for k = 1:l
    for i = 1:m+1
        w1(k,i) = w1(k,i)-rho.*c1(k,i);
    end
end

for j = 1:n
    for k = 1:l+1
        w2(j,k) = w2(j,k) - rho.*c2(j,k);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

For the validation and testing sets, only forward propagation is performed but the "epsilon tracker" matrix is generated for these as well. The sum squared error for each epoch is created using this formula [5].

$$\hat{E} = \sum_{r=1}^{p} E_r = \frac{1}{2} \sum_r \sum_j \varepsilon_r^2[j]$$

Once again, the r x j matrix contains the epsilon value between the true output and the calculated output for each node and for each training sample. Square all the errors in the matrix, sum the rows, and sum down the final columns, and divide the final answer by 2 to obtain the sum squared error for that particular epoch. This error can be plotted against epoch to track the trend of the error over time.

Description of Results

In order to test the efficacy of the network, the network was trained for 800 epochs in order to determine the epoch that begins to increase the validation error. The figure below plots the training and validation errors over 800 epochs. The neural network contains 3 neurons in the hidden layer and used a learning factor of 0.8.

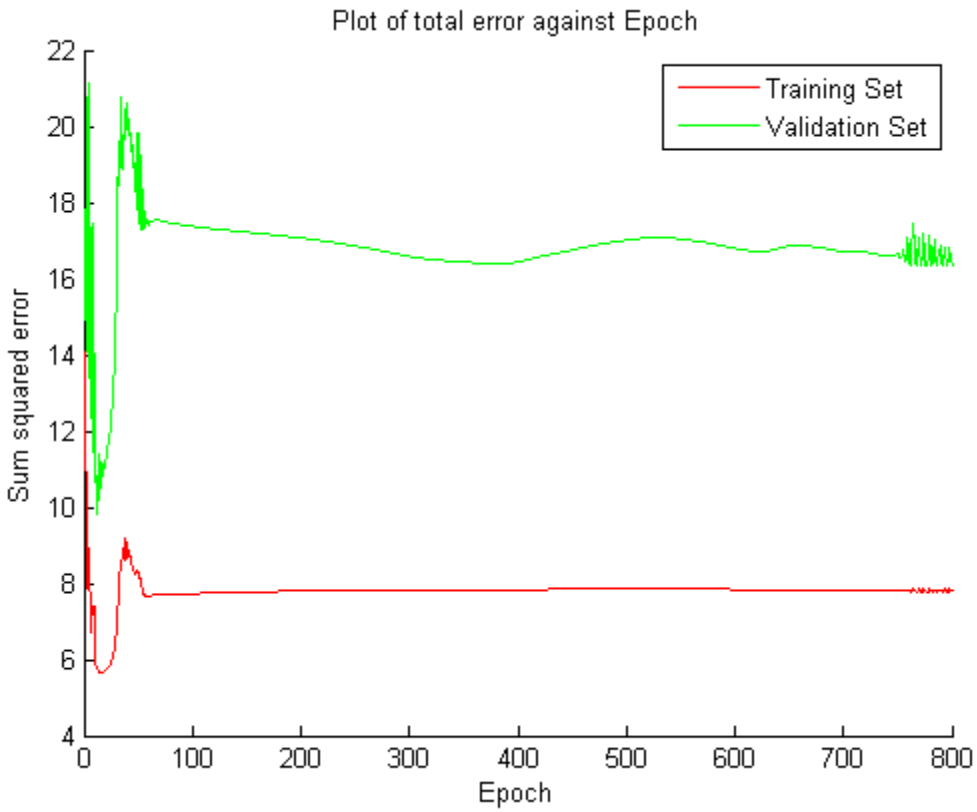

Figure 1: Neural Network trained for 800 epochs and SSE plotted for Training and Validation Sets. 3 hidden neurons and a learning factor of 0.8

The training and validation errors follow the same trends. It's also notable that the validation error is shifted up relative to the training error. This behavior is expected because the network didn't adjust its weights from the validation set. Naturally, the neural network would have a harder time producing reasonable outputs for the validation set, thus resulting in a higher error. However, a behavior that is not typical of neural networks is that the network error did not converge to 0.01. As you

can see in the figure above, training error converges to 6 which is many orders of magnitudes higher than 0.01. Attempts were made to reduce the error to 0.01 but the algorithm provided in the course took an unreasonable number of epochs, ranging in the millions, and still did not converge to 0.01. It's hypothesized that the algorithm's pseudocode was faulty or there weren't enough training samples to sufficiently reduce the sum squared error. In addition, it was decided that 50 epochs was the appropriate cut-off for training done in the future because the validation error began to increase after that period. Although it is not made clear in Figure 1, ideally the network should continue minimizing the training set's error while the validation set's error should increase after a certain epoch because the network's weights are being tuned specifically for the training set. Modifications to the neural network's architecture were studied after setting the maximum number of epochs for training at 50.

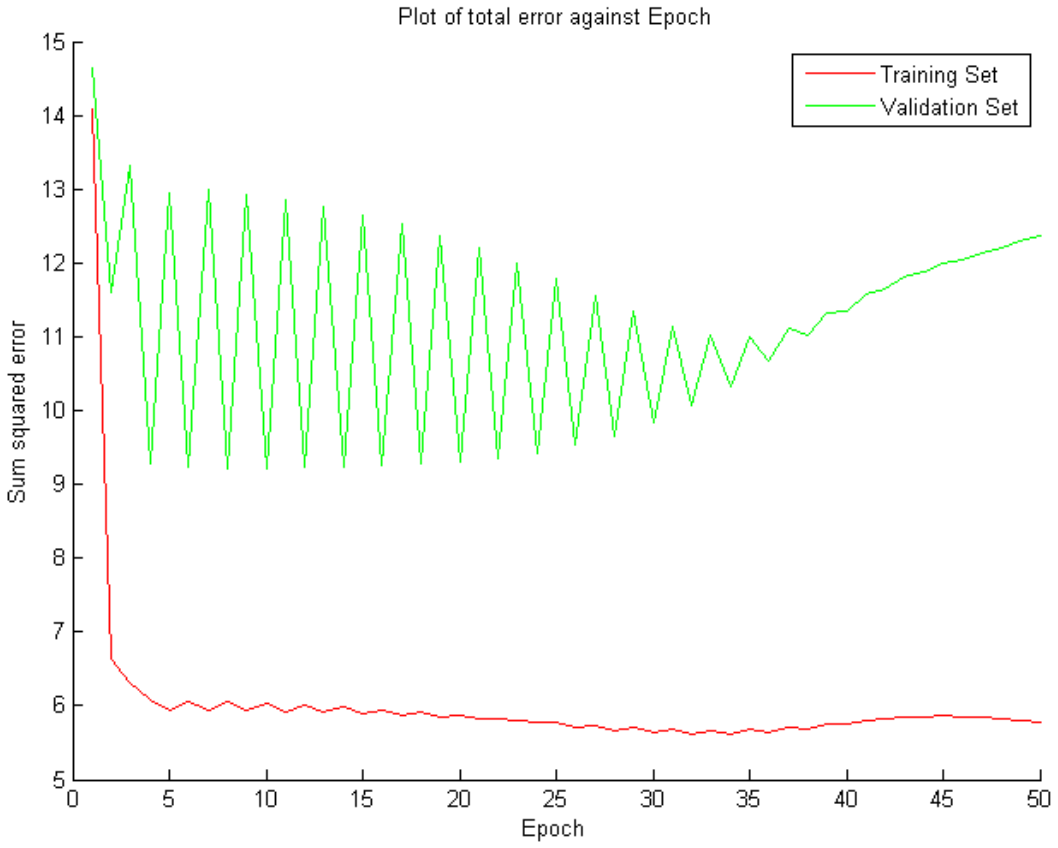

Figure 2: Neural Network trained for 50 epochs and SSE plotted for Training and Validation Sets. 3 hidden neurons and a learning factor of 0.5

Figure 2 shows the variability in the validation set's error with a higher learning rate, although this learning rate is lower than 0.8, Figure 2's graph was trained for a smaller number of epochs but it's clear that the training set's error decreased over time but there are significant fluctuations in the validation set's error. The high learning rate resulted in the validation error to make significant jumps in the error values. In addition, as the training error decreased with the number of epochs, the validation error began to increase after 30 epochs, suggesting that there may be a relationship between a higher learning rate and the error's divergence. This issue is fixed with a lower learning rate and results in a validation error that smoothly converges to a minimum. A lower learning rate stabilizes the error and results in convergence with minimal variability in the sum squared error. This behavior is noted in Figure 3 below.

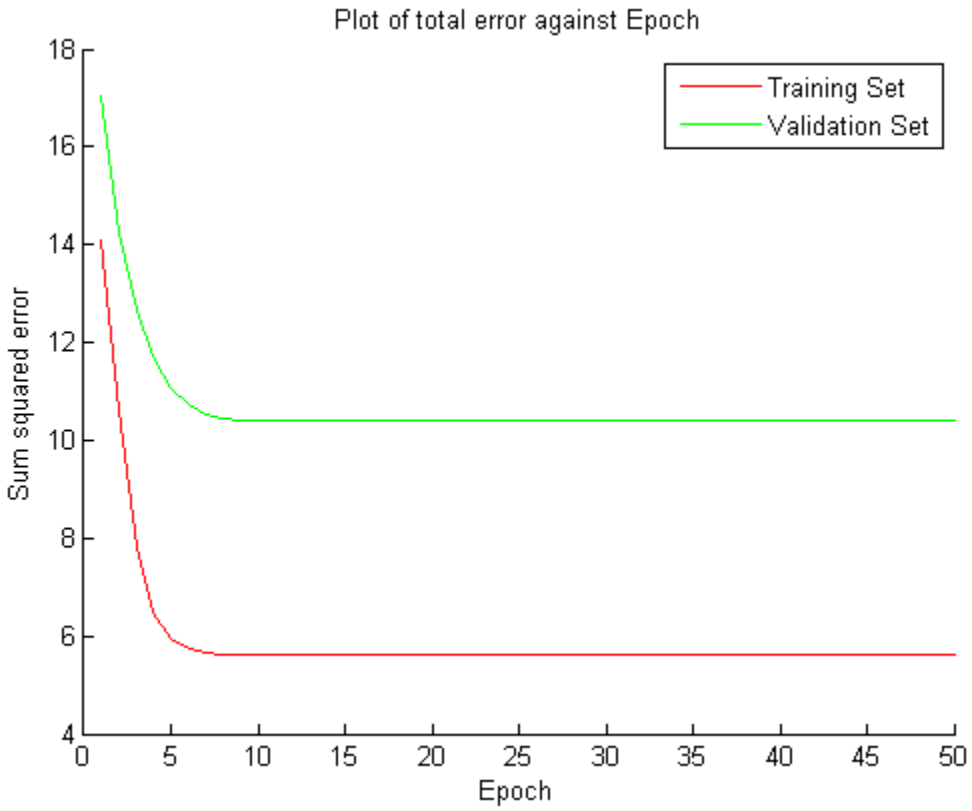

Figure 3: Neural Network trained for 50 epochs and SSE plotted for Training and Validation Sets. 3 hidden neurons and a learning factor of 0.1

After studying 3 hidden neurons in the hidden layer, 2 hidden layers were tested with three learning rates.

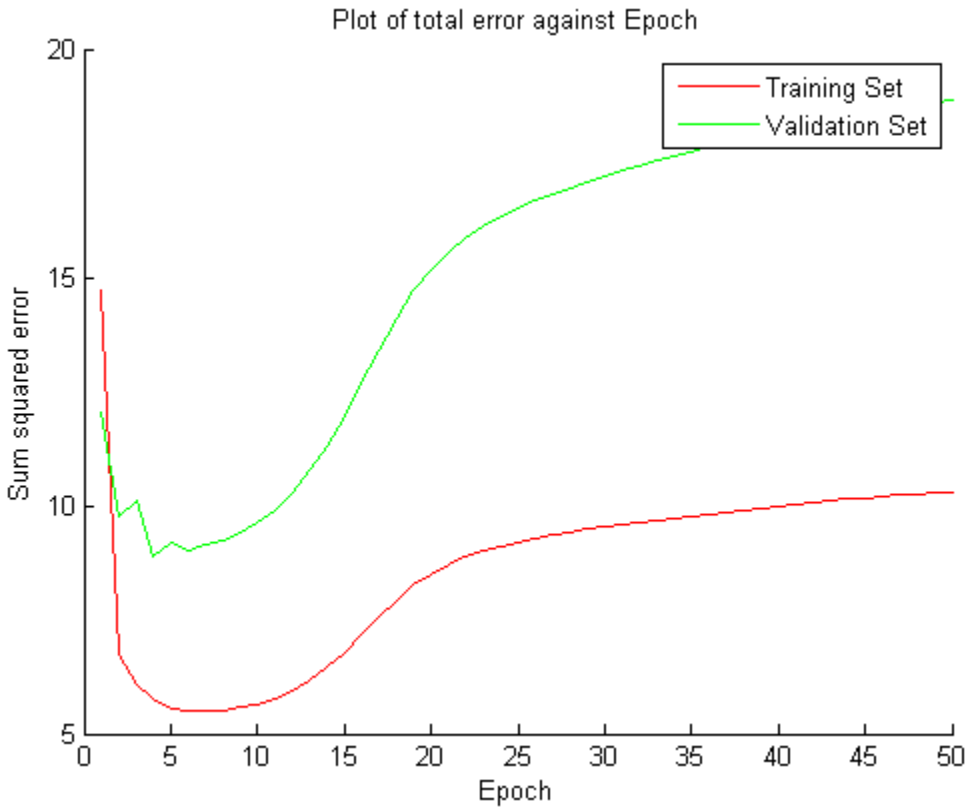

Figure 4: Neural Network trained for 50 epochs and SSE plotted for Training and Validation Sets. 2 hidden neurons and a learning factor of 0.8

Compared to a neural network with 3 hidden neurons, the issue of divergence in Figure 4 with large learning rates is preserved, however there is a sharp and earlier divergence in the error for both the training and validation sets. The errors are minimizing until 5 epochs, suggesting that the learning rate is making large adjustments to the weights and resulting in outputs that are not matching the gold standard outputs. It's interesting to note that divergence likely occurs earlier in the epochs with less neurons in the hidden layer. This is better understood when compared to Figure 2 which has 3 hidden neurons and a higher learning rate. Increasing the number of hidden neurons likely plays a role in error divergence over time.

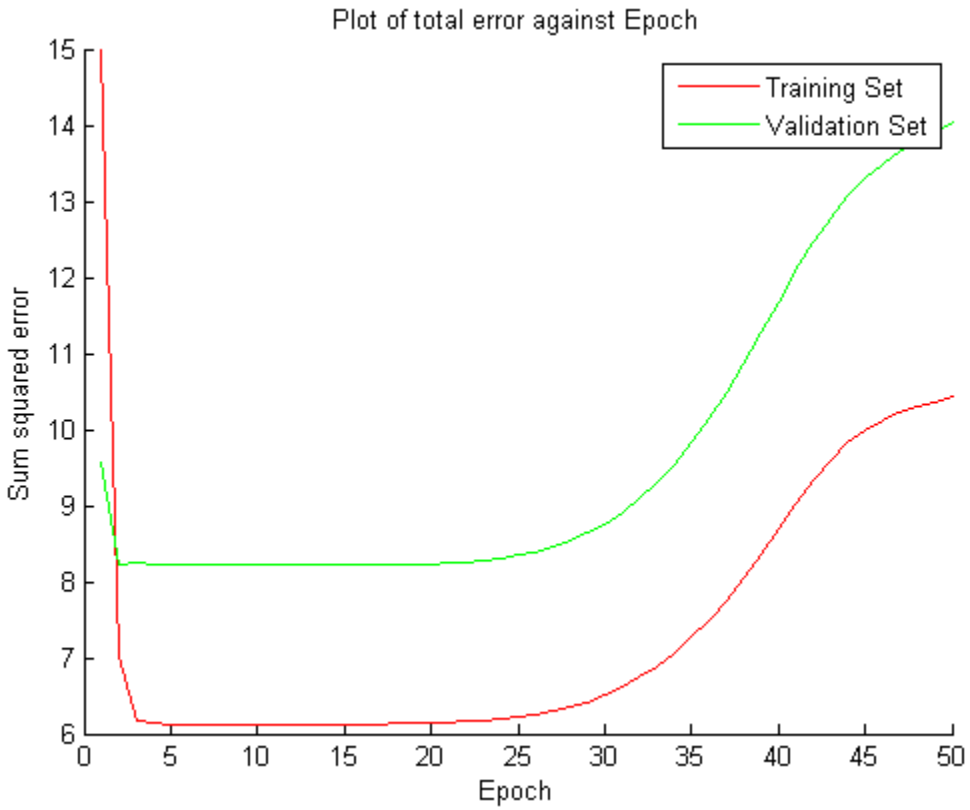

Figure 5: Neural Network trained for 50 epochs and SSE plotted for Training and Validation Sets. 2 hidden neurons and a learning factor of 0.1

Figure 5 shows two interesting results that concern the number of hidden neurons and the effect of learning rate. Decreasing the learning rate helps create smoother curves but divergence in the validation error still occurs. However, divergence occurs at 30 epochs compared to 5 epochs in Figure 4 (which has a learning rate of 0.8).

All the figures above illustrate the relationship between the sum squared error and training and validation sets. It's also important to consider the outputs from a testing set that the network has never seen. Figure 6 below shows the output of a neural network trained for 50 epochs with a learning rate of 0.8 and 2 hidden neurons.

```
raw_wine_outputs_testing =

    0.1000    0.9000    0.1000
    0.1000    0.9000    0.1000
    0.1000    0.9000    0.1000
    0.1000    0.1000    0.9000
    0.9000    0.1000    0.1000
    0.1000    0.9000    0.1000


testing_outputs =

    0.1662    0.8381    0.1855
    0.1631    0.8612    0.1901
    0.2060    0.8069    0.2124
    0.0159    0.9900    0.0552
    0.0084    0.9968    0.0439
    0.0420    0.9711    0.0932
```

Figure 6: True outputs from wine data set compared to a trained Neural Network's outputs

6 samples were set aside for the testing set in the beginning. Previously it was mentioned that the true outputs within the wine data set would use binary values of 0 and 1. Figure 6 uses values like 0.1 and 0.9 because these are the values a sigmoidal activation would ideally give after the linear basis function in the output layer. Exact binary values are ideal outputs but values closer to 0.1 and 0.9 should not significantly alter the calculations for backpropagation. As seen in Figure 6, the first three outputs belong to cultivar 2; the fourth output belongs to cultivar 3; the fifth output belongs to cultivar 1; and the last sample belongs to cultivar 2. The outputs from the neural network, after weights were adjusted through training, were either close to the gold standard or significantly different from the true outputs. In the outputs from the neural network, the first three outputs closely match the first three gold standard outputs, suggesting that the network recognized that those samples belonged to cultivar 2. This is also the case for the last sample. However, differences arise in the fourth and fifth samples. The

fourth sample belongs in cultivar 3 but the network classified it as cultivar 2. The fifth sample belongs in cultivar 1 but the network classified it as cultivar 2. It seems the network may have been trained on a significant number of samples which belonged to cultivar 2. The network is not a very accurate classifier according to these results.

Conclusions

While the project may not have been a complete success, a few pieces of knowledge were obtained from the implementation in this project. A neural network trained with gradient descent may get stuck at a local minimum instead of a global minimum. The figures in the results section showed that the training error converged at 6, which was many orders of magnitude higher than 0.01. In this case, the global minimum would be 0.01 so there may have been problems with the network continuously converging to a local minimum or there was an error in the implementation of the algorithm. Sum squared error summed the error across all nodes and training samples, so the error for each epoch may have been significantly larger than it should have been. In addition, the validation and training errors did not behave like textbook examples of error plots. Ideally, the training error would continuously minimize and approach the error tolerance of 0.01 while the validation error would increase after a certain time because the network was tuning its weights too closely with the training set. The training and validation curves in the figures essentially mirrored each other and were shifted relative to each other. In addition, modifying the learning rate significantly altered the variability in the validation error of all the graphs. Higher learning rates resulted in fluctuating validation errors while lower learning rates stabilized the curves so they could converge to a minimum. Hidden

neurons affected the divergence of the training error and validation error. Increasing the number of hidden neurons allows the error to converge as time goes on.

However, given these results, a few modifications to the project could be made to ensure that the results were accurate. The algorithm may have contained a minor error in the implementation within Matlab. Aside from the algorithm, the data set did not contain enough samples to appropriately train the network, validate the network, and test the network with a testing set. 178 training samples was too limited for the scope of this project and the project would have benefitted if there were more samples in the data set from the machine learning repository. In addition, the data set was missing attributes that may have been vital in the training process. Aside from sample size, gradient descent was seen as inefficient and may have been computationally limiting the results. Matlab's neural network tool contains more complicated and advanced techniques for gradient descent that are more efficient and less cumbersome than the algorithm used from the lecture course. The other problem with gradient descent lies with the likelihood of converging to a local minimum. Gradient descent uses partial derivatives to optimize a function and the error may hit a local minimum. So, it may be interesting to incorporate features like "momentum" to prevent the network from reaching a local minimum instead of a global minimum. In addition, it may be interesting to use backpropagation with biases that are meant to be subtracted. In the current implementation, the bias is always a value of 1 and incorporated as the 0^{th} input. It may be interesting to simply subtract the bias values from the output of the linear basis summation function. These may result in differences that significantly improve the results from the neural network.

In total, this project provided an opportunity to learn how to implement an artificial neural network with real world data. Learning how to create a neural architecture that implemented backpropagation was an experience that explored a basic, yet fundamental machine learning paradigm. In addition, the ability to create an architecture from information provided from the wine classification data set was an educational experience. Using real world data and a neural network allowed for a clearer understanding of model creation. As a tool in soft computing, it was made far clearer that neural networks were a way to create a model that related various attributes in a non-linear manner. In the case of wine classification, the attributes outlined in the background were related in a manner that could not have been captivated via a function that could be modeled using techniques like linear regression. Using a neural architecture and tuning weights among all the connections using standard gradient descent techniques truly allowed a deeper appreciation of model creation.

Bibliography

[1] Land, Walker. *Adaptation and Learning in Complex Systems*. Deer Park: Linus Publications, 2011. Print.

[2]http://archive.ics.uci.edu/ml/datasets/Wine

[3]http://visualstudiomagazine.com/articles/2014/01/01/how-to-standardize-data-for-neural-networks.aspx

[4]https://learnit.itu.dk/pluginfile.php/113449/mod_resource/content/1/3point5point2_normalization.pdf

[5] A Gentle (but mathematical) Introduction to Training Neural Nets—Part 2