Project 3:
Multilayer Neural Network

Ian McAtee and Stein Wiederholt

EE5650: Object and Pattern Recognition
Professor Wright
University of Wyoming

December 10, 2021

Abstract

This project provides an introduction to multilayer nerual networks. A brief background of neural networks is provided and a neural network is constructed in MATLAB using artificially produced training and test data. A robust presentation of the neural network design and testing is given. The implemented neural network achieved an accuracy of 93.75% upon classification of the test data. The neural network results are then compared to those of the k_n-nearest neighbor and Bayesian classifiers from McAtee and Wiederholt [4]. Finally, a general discussion of strengths and weaknesses of neural networks is presented.

1 Introduction and Background

In recent years, neural networks have become a ubiquitous solution to many object and pattern classification problems. Modern computational power has allowed deep learning neural networks to solve many complex classification problems that were previously intractable via other methods. With the advent of simple open-source platforms and APIs for their development, neural networks have become a popular and nearly default choice for classification problems. Unfortunately, these simple tools have empowered developers to quickly design and deploy neural networks without having an intimate knowledge of how these networks work. Consequently, one can find many examples of erroneous applications of neural networks or the implementation of a neural network towards problems in which other classification techniques would have been more simple and appropriate. Further, even though research into neural networks is ongoing, they are are widely considered "black-box" techniques. That is, it is difficult to parse how exactly these networks arrive at specific classifications. This lack of mathematical validation may make neural networks inadvisable in critical applications.

The goal of this project is to provide a rudimentary background on neural network theory and design. The framework developed in this discussion is used to develop a simple multilayer neural network in MATLAB using artificial training and test data. A robust discussion of the network design process and the achieved results is then presented. Lastly, a brief comparison of neural networks to other classification methods is given.

1.1 Neural Networks

The concept of neural networks arises from generalized linear discriminant functions and the idea that modifiable weight vectors can describe decision boundaries. While the linear hyperplanar discriminants from such classifiers can achieve good results, they often do not generalize to demanding classification problems. That is, there are many problems in which generalized linear discriminant functions simply cannot obtain minimum classification error. This is largely due to the linear constraint of such classifiers. If one could define optimal non-linear functions, such networks may achieve minimum error. The crux of traditional neural networks is to implement linear discriminants in a space to which the inputs have been non-linearly mapped.

1.1.1 Architecture and Feedforward Operation

Neural networks accomplish the mapping of the inputs to new spaces by feeding the input data through weighted consecutive layers. Adjacent layers are connected by a series of weights that scale the outputs of the previous layer. Each layer consists of nodes that apply activation functions to the weighted sum of its input and outputs the result to the next layer. To better visualize this architecture, figure 1 depicts a simple three layer neural network:

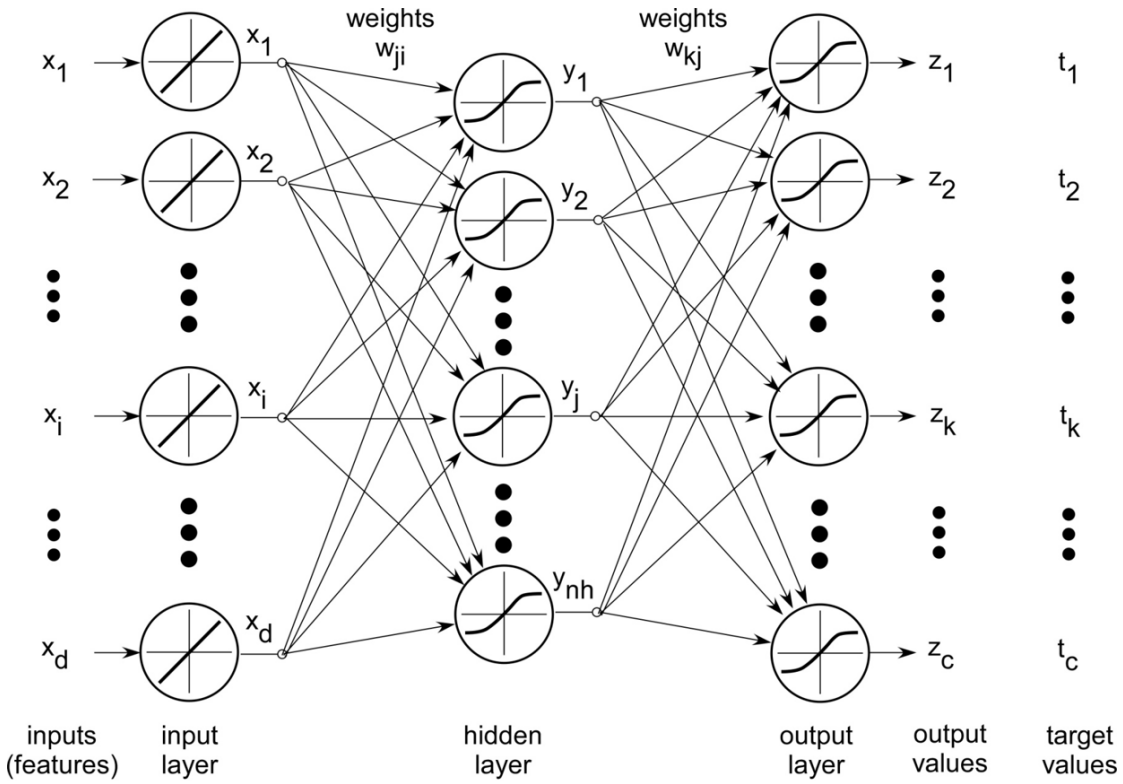

Figure 1: A three-layer fully connected neural network (bias not shown) [1]

The neural network represented in figure 1 exhibits a fully connected architecture. That is, the output of each node is connected to every node of the subsequent layer. This network consists of the input layer (layer i), a single hidden layer (layer j), and the output layer (layer k). It is standard to include a single bias unit that forms weighted connections to nodes across all layers other than the input layer (this is not shown in figure 1). As expected, the input layer accepts the input, with one node for each dimension of the data. Note, traditionally a one is appended to the beginning of each input vector to account for the bias. The input nodes usually do not apply an activation to the input and instead simply output the corresponding component of the input vector. Each dimension i of the input vector is fed to all j hidden nodes scaled by the weight w_{ji}. Each hidden node performs the weighted sum of its inputs to form its *net activation* or just *net* such that:

$$net_j = \sum_{i=0}^{d} x_i w_{ji} = \boldsymbol{w}_j^t \boldsymbol{x} \tag{1}$$

It is at this point that the non-linear mapping begins to be performed. Each node of the hidden layer produces an output y_j that is a nonlinear function of its net_j:

$$net_j = f(net_j) \tag{2}$$

This function $f()$ is known as the *activation function* of the node. There are many valid choices of activation functions for neural networks. The activation functions of the nodes is

a user defined hyperparameter of the network and should be chosen based on a knowledge of the specific classification problem. A common choice is the the sigmoid function, shown in the bipolar form in figure 2 below:

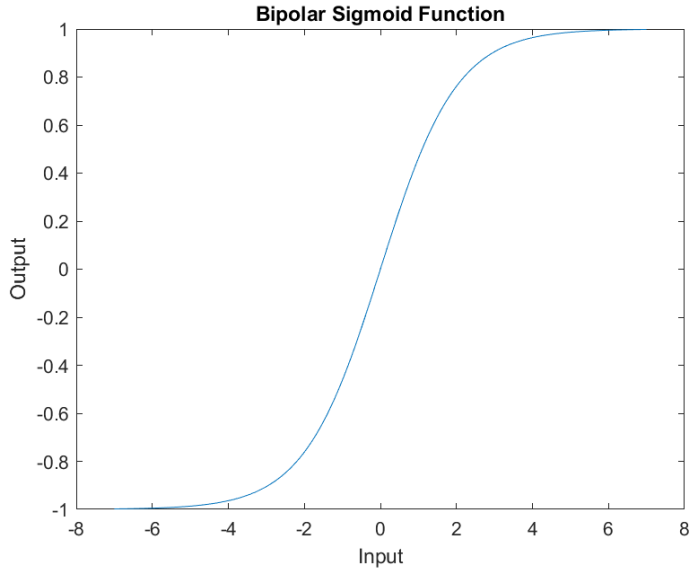

Figure 2: Bipolar Sigmoid Function with Slope Parameter $\alpha = 1$

The bipolar sigmoid function is given by:

$$sigmoid(net) = \frac{2}{1 + exp(-\alpha \cdot net)} - 1 \tag{3}$$

Here, α is the slope parameter and controls the steepness of the function. The sigmoid function can be generalized as:

$$sigmoid(net) = a \cdot tanh(\beta \cdot net) = a\frac{1 - exp(-2\beta \cdot net)}{1 + exp(-2\beta \cdot net)}, \tag{4}$$

such that it is defined on the interval between the saturation value $-a$ and a with a slope parameter of β. An examination of the sigmoid function in figure 2 can provide a cursory notion of why this function is well-suited for node activation functions. One can see that near the origin, the sigmoid exhibits a pseudo-linear region while the non-linearity increases as the function approches the positive and negative saturation values. Therefore, based on weights, the net_j of a node can be mapped linearly or non-linearly to various degrees.

In figure 1, the activation function output y_j of each of the j hidden nodes is fed to each of the k output layer nodes weighted by w_{kj}. Similarly, each output node computes its net activation net_k as:

$$net_k = \sum_{j=0}^{n_H} y_j w_{kj} = \boldsymbol{w}_k^t \boldsymbol{y}, \tag{5}$$

where n_H is the number of hidden layer nodes. Each output layer then applies its activation function to its net_k to produce:

$$z_k = f(net_k) \tag{6}$$

Each output z_k represents the different discriminant functions. With that, it should be apparent that the number of output nodes should match the number of classes in the classification problem. Such as with normal discriminant functions, the class is chosen that corresponds to the maximal z_k.

1.1.2 Learning and the Backpropagation Algorithm

However, the difficulty in this method lies with determining the weights such that the inputs are non-linearly mapped as to minimize the error. The most general method to accomplish this is known as the *backpropagation algorithm*. At the core of the backpropagation algorithm is the gradient descent of an error surface. In a supervised learning paradigm, this involves presenting training data to the network along with a known target label and adjusting the weights such that the network output is more similar to the target values.

Specifically, the network output $\boldsymbol{z}_k$ is compared against the target vector $\boldsymbol{t}_k$ as a measure of the error corresponding to that particular training pattern. Typically the target vector $\boldsymbol{t}_k$ is set such that the index relating to the class of the input pattern is set to 1 and the rest are set to -1. This leads to an optimization problem in one wishes to minimize the following criterion function:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}(\boldsymbol{t} - \boldsymbol{z})^2 \tag{7}$$

Here, $\boldsymbol{w}$ is all the weights of the network, and $\boldsymbol{t}$ and $\boldsymbol{z}$ are vectors of length equal to the number of classes c, representing the target values and network outputs, respectively. This minimization can be accomplished based on well-established gradient decent techniques. The weights are updated such that the direction reduces the error:

$$\Delta\boldsymbol{w} = -\eta\frac{\partial J}{\partial \boldsymbol{w}} \tag{8}$$

The amount that the weights change along the negative gradient of the error surface is determined by the learning rate η. This process is applied in an iterative manner as training samples are presented to the network with the weight update at iteration m for the mth sample taking the form:

$$\boldsymbol{w}(m+1) = \boldsymbol{w}(m) + \Delta\boldsymbol{w} \tag{9}$$

Learning the weights of the network involves the chain-rule differentiation of equation (8). The basis is that each individual weight update in a neural network can be evaluated via application of the chain rule. Consider the network presented in 1. For a particular hidden layer to output layer weight w_{jk}, one can use the chain rule to write:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = \delta_k\frac{\partial net_k}{\partial w_{kj}} \tag{10}$$

δ_k is known as the sensitivity of node k and represents how the overall error is affected by the activation of node k. The sensitivity is defined by:

$$\delta_k = -\frac{\partial J}{\partial net_k} \tag{11}$$

Performing the differentiation of the sensitivity yields:

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} \tag{12}$$

By examining equation (7) and provided the derivative of the activation has a solution, one can simplify equation (12) to:

$$\delta_k = (t_k - z_k) f'(net_k) \tag{13}$$

The last derivative from equation (10) can be evaluated by examining equation (5) as simply y_j. Putting this together, equation (10) can be expressed as:

$$\frac{\partial J}{\partial w_{kj}} = (t_k - z_k) f'(net_k) y_j \tag{14}$$

Thus, the backpropagation weight update for output layer to hidden layer weight w_{kj} is:

$$\boxed{\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j} \tag{15}$$

One can solve for the input to hidden layer weights in much the same way with an intelligent application of the chain rule. Once again, from equation (8), one can write:

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \tag{16}$$

First examine the $\partial J / \partial y_j$ term:

$$\begin{aligned}
\frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2\right] \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k) f'(net_k) w_{jk}
\end{aligned} \tag{17}$$

Note how using equation (17), one can define the sensitivity of the jth hidden node:

$$\delta_j = f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k \tag{18}$$

That is, the sensitivity of hidden node j is a summation of the sensitivity of the output units as weighted by the w_{jk} weights. Using this, the weight update rule for the input to hidden layer weights is:

$$\boxed{\Delta w_{ji} = \eta x_i \delta_j = \eta x_i f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k} \tag{19}$$

1.1.3 Training Protocols and Validation

The backpropagation algorithm provides a method to learn the weight updates given a training data sample and its associated target value. However, one must still determine how these samples are presented to the network. This is known as the training protocol used by the network. Training is usually described by epochs, or the number of times that the full training set has been presented to the network. The three most common training protocols are stochastic, batch, and on-line training. In stochastic training, during each epoch, the training patterns are presented to the network in a random order. Here, weight updates occur at the point each sample is presented to the network. Conversely, in batch training, all training data samples are presented to the network before weight updates are applied. Lastly, on-line training consists of presenting each training sample to the network only once.

Recall that learning is performs a gradient decent across an error surface as defined by the criterion function based on the training data. Because this is based on the training data, performing learning over many epochs can result in what is known as overtraining. This occurs when the network has learned the training data to such an extent that it fails to generalize to test data. Unfortunately, it is difficult to quantitatively determine the exact point at which the network has not overtrained yet has learned sufficiently. One method to attempt to estimate this point is through the use of validation data. Validation data is a batch of samples, separate from the training data, that is used to evaluate the performance of the network. That is, at certain epoch intervals, the network is used to classify both the training and validation data. As the epochs progress, both the training and validation error should diminish. However, if overtraining begins to occur, one will see the validation error increase while the training error continues to decrease. Therefore, it is common to end the training at the lowest point of validation data error to ensure a network that will still generalize and perform well on test data.

2 Methods and Results

A three layer neural network was developed in MATLAB using artificially produced two-featured training data for three classes. The neural network was implemented in such a way that the number of hidden nodes, learning rate, and validation training termination hyperparameters could be easily changed as to analyze their effect on the network performance. All developed MATLAB code is given in its entirety in the Code Listing section of the Appendix to this report. This section presents a detailed explanation of the design of the neural network as well as a presentation of the achieved results.

2.1 Data Set Development

For the purposes of training and testing the designed neural network, artificial training and test data was acquired. The data was acquired from the data generated in McAtee and Wiederholt [4]. Specifically, this data represented three two-featured classes and was drawn from the following Gaussian distributions:

$$\mathcal{N}_1(\mu_1, \Sigma_1), \text{ where } \mu_1 = \begin{bmatrix} -4 \\ 10 \end{bmatrix}, \Sigma_1 = \begin{bmatrix} 8 & 4 \\ 4 & 15 \end{bmatrix} \tag{20}$$

$$\mathcal{N}_2(\mu_2, \Sigma_2), \text{ where } \mu_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 10 & -2 \\ -2 & 4 \end{bmatrix} \tag{21}$$

$$\mathcal{N}_3(\mu_3, \Sigma_3), \text{ where } \mu_3 = \begin{bmatrix} 5 \\ 10 \end{bmatrix}, \Sigma_3 = \begin{bmatrix} 5 & 0 \\ 0 & 10 \end{bmatrix} \tag{22}$$

The training and test data were organized as Nx2 matrices such that each row constituted a data sample with the elements in the two columns representing two features:

$$\boldsymbol{D_c} = \begin{bmatrix} d_{1,1} & d_{1,2} \\ \vdots & \vdots \\ d_{N,1} & d_{N,2} \end{bmatrix} \tag{23}$$

Here, $\boldsymbol{D_c}$ is the data matrix for class c, and $d_{1...N,1}$ and $d_{1...N,2}$ are vectors containing the two features. The data samples were drawn from the acquired data such that the number of samples was $N_1 = 1000$, $N_2 = 600$, and $N_3 = 400$ samples of training, validation, and test data for classes one, two, and three, respectively.

Since the training and test data were taken from a subset of the data used in [4], the data should exhibit approximately the same statistics. Further, because of the number of samples drawn for each class is a simple scaling of the number of samples produced in [4], the prior probabilities remain unchanged in this project. However, since this project is using a smaller sample size, the mean vectors and covariance matrices can be further from the desired values. In order to assess if this is occurring, the mean vector and covariance matrix was found for each class, as seen in figure 3.

	Class 1		Class 2		Class 3	
	Training	Test	Training	Test	Training	Test
Sample Mean $\left(\hat{\mu} = \begin{bmatrix} \hat{\mu}_1 \\ \hat{\mu}_2 \end{bmatrix}\right)$	$\begin{bmatrix} -4.089 \\ 10.016 \end{bmatrix}$	$\begin{bmatrix} -4.102 \\ 9.941 \end{bmatrix}$	$\begin{bmatrix} 1.980 \\ 1.966 \end{bmatrix}$	$\begin{bmatrix} 1.925 \\ 2.003 \end{bmatrix}$	$\begin{bmatrix} 4.901 \\ 10.325 \end{bmatrix}$	$\begin{bmatrix} 4.989 \\ 10.07 \end{bmatrix}$
Unbiased Covariance Matrix $(\hat{\Sigma})$	$\begin{bmatrix} 7.661 & 3.017 \\ 3.017 & 13.23 \end{bmatrix}$	$\begin{bmatrix} 8.177 & 4.151 \\ 4.151 & 15.62 \end{bmatrix}$	$\begin{bmatrix} 11.19 & -2.425 \\ -2.425 & 3.932 \end{bmatrix}$	$\begin{bmatrix} 9.367 & -1.834 \\ -1.834 & 3.708 \end{bmatrix}$	$\begin{bmatrix} 4.874 & 0.063 \\ 0.063 & 8.910 \end{bmatrix}$	$\begin{bmatrix} 4.152 & -0.471 \\ -0.471 & 10.03 \end{bmatrix}$
Prior Probability $(P(\omega_i))$	0.50	0.50	0.30	0.30	0.20	0.20

Figure 3: Training and Test Data Statistics by Class

The mean vectors are roughly equivalent, but the covariance matrices have a small but noticeable variance. This may change the performance of the neural network when compared to the Bayesian and k_n nearest neighbor classifiers found in Project 2 [4]. For reference, the acquired training, validation, and test data are plotted in figures 4, 5, and 6.

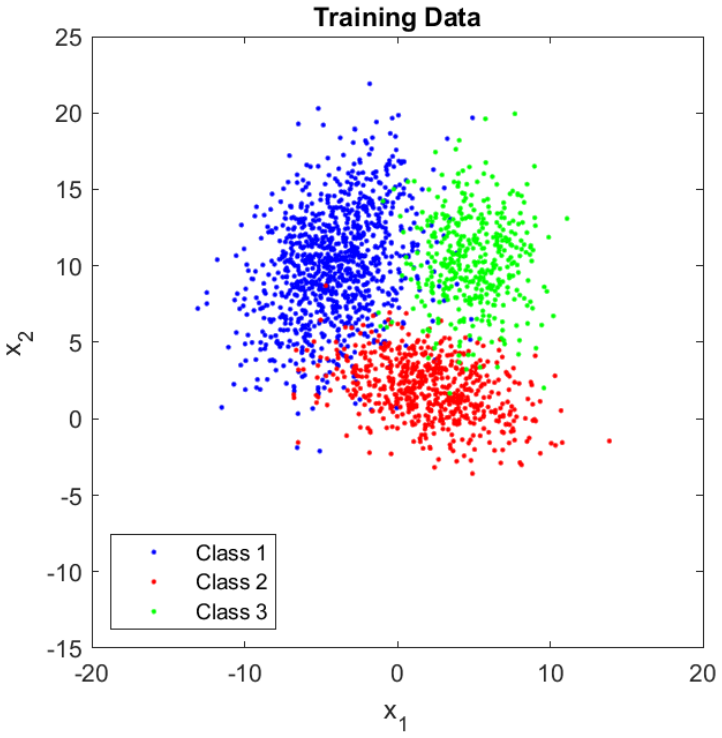

Figure 4: Scatter Plot of Training Data

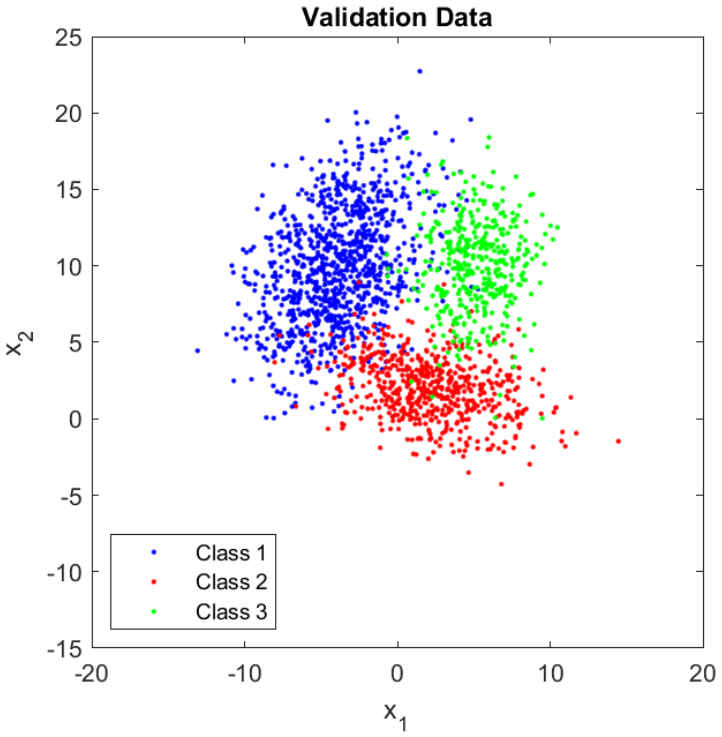

Figure 5: Scatter Plot of Validation Data

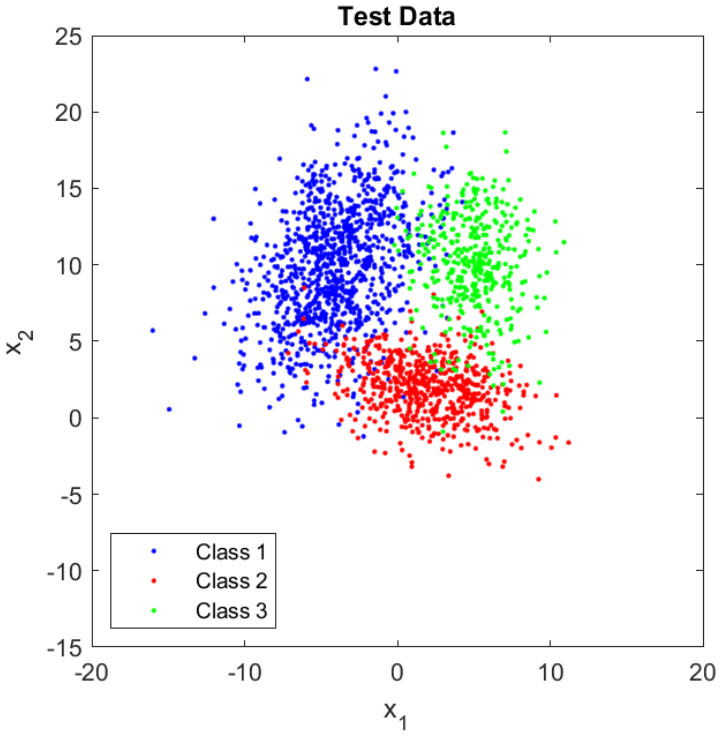

Figure 6: Scatter Plot of Test Data

2.2 Neural Network Development

This section details the development of the neural network as well as a discussion and justification of the design choices.

2.2.1 Data Preprocessing

For most neural networks to achieve good performance, the input data must first be preprocessed prior to its presentation to the network. A common approach is to transform the data such that as a collective of all the classes, it has zero mean and unit variance. This is done in an attempt to prevent the weights from biasing towards one feature during training. Preprocessing the data in this way does not affect class discriminability and therefore, does not affect classification performance.

For this neural network, an approximately zero mean for the training data was obtained by simply subtracting off the weighted mean of the class data, such that:

$$\boldsymbol{D}^* = \boldsymbol{D} - \left[\sum_{j=1}^{c} P(\omega_j)\boldsymbol{\mu}_j\right]^t \tag{24}$$

Here, $\boldsymbol{D}^*$ is the data exhibiting approximately zero mean, and the summation term is the average mean found by the weighting the class means $\boldsymbol{\mu}_j$ by the priors $P(\omega_j)$.

To achieve a unit variance, a whitening transform was applied to training data via eigendecomposition. A whitening transformation works by decorrelating the data components and then scaling the components to unit variance, thus producing an identity covariance matrix. First, eigendecomposition was applied to the estimated covariance matrix of the training data such that:

$$\boldsymbol{\Sigma} = \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^{-1} \tag{25}$$

$\boldsymbol{\Phi}$ is the matrix of eigenvectors with the associated diagonal matrix of $\boldsymbol{\Lambda}$ eigenvalues. One can decorrelate the data by forming a new random variable $\boldsymbol{Y}$ that is drawn from an uncorrelated distribution with the diagonal covariance $\boldsymbol{\Lambda}$:

$$\boldsymbol{Y} = \boldsymbol{\Phi}^{-1}\boldsymbol{D} = \boldsymbol{\Phi}^t\boldsymbol{D} \tag{26}$$

This forms a new random variable $\boldsymbol{Y}$ with variances of λ_i on the diagonal of its covariance matrix. Lastly the new samples are scaled to unit length such that the variances are all unity:

$$\boldsymbol{W} = \boldsymbol{\Lambda}^{1/2}\boldsymbol{Y} \tag{27}$$

Which yields the preprocessed whitened data $\boldsymbol{W}$ that exhibits unit variance.

The preprocessing parameters (the weighted mean, $\boldsymbol{\Phi}$, and $\boldsymbol{\Lambda}$) were then saved to use in the preprocessing of the validation and training data. It is important that all data to be presented to the neural network for evaluation be preprocessed in an identical manner. Failure to preprocess test data in the same manner as the training data will result in poor

classification performance because the training data is no longer representative of the test data.

The mean and covariance of the preprocessed training data was calculated to verify the validity of the preprocessing techniques. The mean was found to be:

$$\boldsymbol{\mu}_{training} = \begin{bmatrix} 0.2971 \\ -0.2003 \end{bmatrix} \cdot 10^{-15} \approx \mathbf{0} \tag{28}$$

The covariance was found to be equal to the identity matrix. This demonstrates that the designed preprocessing achieved the desired results. The preprocessed training, validation, and test data are plotted for reference in figures 7, 8, and 9. Tables with the preprocessed data statistics and overall data before and after comparison are also provided in figures 10 and 11.

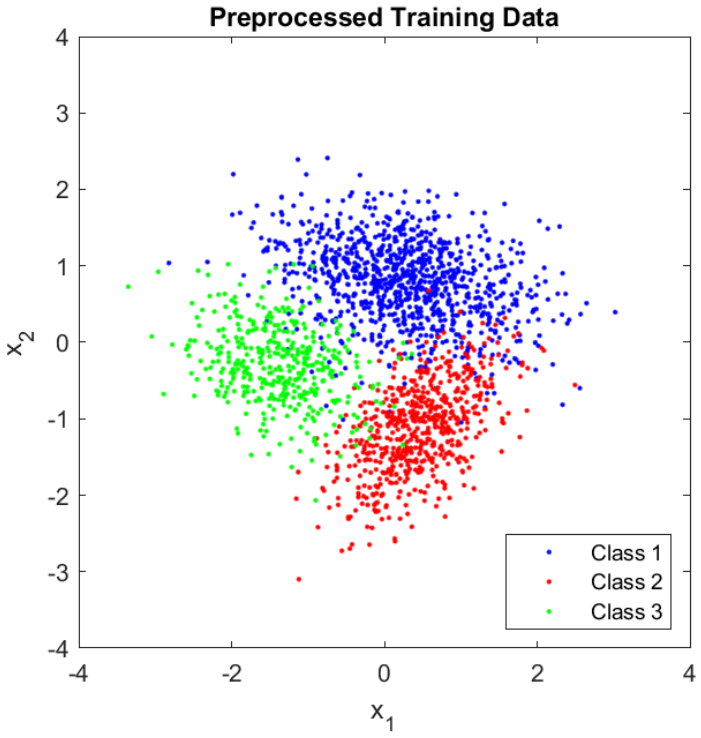

Figure 7: Scatter Plot of Preprocessed Training Data

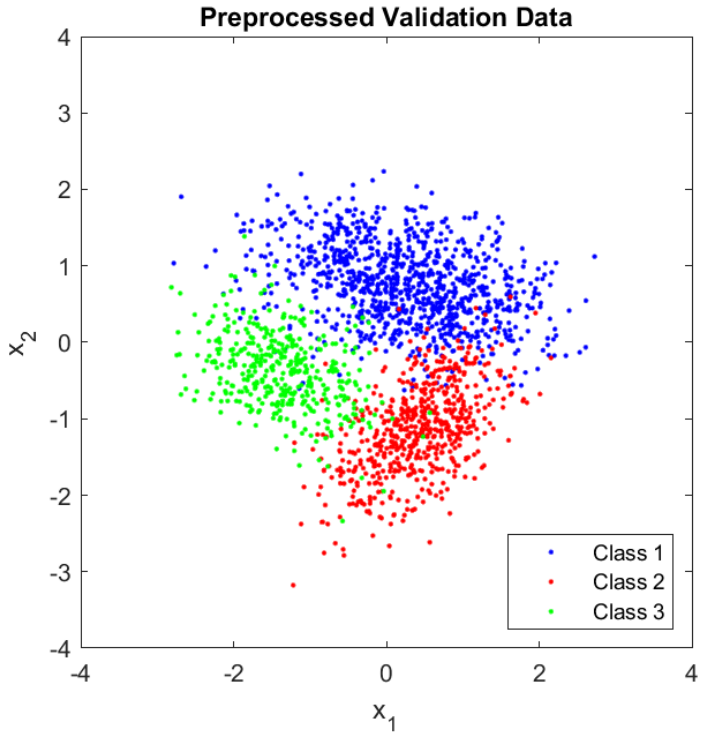

Figure 8: Scatter Plot of Preprocessed Validation Data

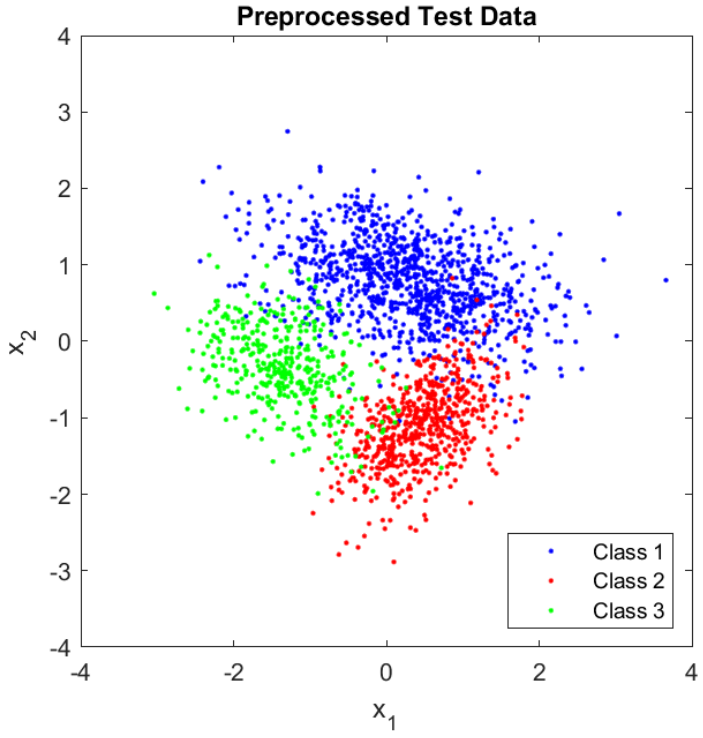

Figure 9: Scatter Plot of Preprocessed Test Data

	Class 1		Class 2		Class 3	
	Training	**Test**	**Training**	**Test**	**Training**	**Test**
Sample Mean $\left(\hat{\boldsymbol{\mu}} = \begin{bmatrix}\hat{\boldsymbol{\mu}}_1 \\ \hat{\boldsymbol{\mu}}_2\end{bmatrix}\right)$	$\begin{bmatrix}0.274 \\ 0.789\end{bmatrix}$	$\begin{bmatrix}0.288 \\ 0.780\end{bmatrix}$	$\begin{bmatrix}0.452 \\ -1.120\end{bmatrix}$	$\begin{bmatrix}0.457 \\ -1.108\end{bmatrix}$	$\begin{bmatrix}-1.363 \\ -0.294\end{bmatrix}$	$\begin{bmatrix}-1.338 \\ -0.342\end{bmatrix}$
Unbiased Covariance Matrix $\left(\hat{\boldsymbol{\Sigma}}\right)$	$\begin{bmatrix}0.725 & -0.142 \\ -0.142 & 0.282\end{bmatrix}$	$\begin{bmatrix}0.860 & -0.189 \\ -0.189 & 0.298\end{bmatrix}$	$\begin{bmatrix}0.311 & 0.175 \\ 0.175 & 0.343\end{bmatrix}$	$\begin{bmatrix}0.281 & 0.136 \\ 0.136 & 0.288\end{bmatrix}$	$\begin{bmatrix}0.371 & -0.090 \\ -0.090 & 0.257\end{bmatrix}$	$\begin{bmatrix}0.346 & -0.128 \\ -0.128 & 0.286\end{bmatrix}$
Prior Probability $(P(\omega_i))$	0.50	0.50	0.30	0.30	0.20	0.20

Figure 10: Preprocessed Training and Test Data Statistics by Class

	Original		Pre-Processed	
	Training	Test	Training	Test
Sample Mean $\left(\hat{\boldsymbol{\mu}} = \begin{bmatrix}\hat{\boldsymbol{\mu}}_1 \\ \hat{\boldsymbol{\mu}}_2\end{bmatrix}\right)$	$\begin{bmatrix}-0.470 \\ 7.663\end{bmatrix}$	$\begin{bmatrix}-0.476 \\ 7.585\end{bmatrix}$	$\begin{bmatrix}0.000 \\ 0.000\end{bmatrix}$	$\begin{bmatrix}0.013 \\ -0.010\end{bmatrix}$
Unbiased Covariance Matrix $\left(\hat{\boldsymbol{\Sigma}}\right)$	$\begin{bmatrix}22.28 & -4.794 \\ -4.794 & 23.52\end{bmatrix}$	$\begin{bmatrix}22.01 & -4.151 \\ -4.151 & 24.28\end{bmatrix}$	$\begin{bmatrix}1.000 & 0.000 \\ 0.000 & 1.000\end{bmatrix}$	$\begin{bmatrix}1.045 & -0.027 \\ -0.027 & 0.988\end{bmatrix}$

Figure 11: Original vs. Preprocessed Overall Data Statistics

2.2.2 Feedforward and Backpropagation Development

The equations developed in section 1.1.1 were implemented to construct a MATLAB function to perform the feedforward operation of the neural network. This function (*evaluateNN.mat*) is input with the data to be classified, the weights, and the preprocessing parameters. There is no need to specify the neural network architecture, as the function automatically extrapolates the right architecture from the dimensionality of the weight matrices. Using the preprocessing parameters, the function preprocesses the input data and appends a one as a bias to form augmented training data prior to presenting it to the network. The activation function of the input nodes are set to the identity function. The hidden nodes and output nodes use the bipolar sigmoid as the activation function with a saturation value of $a = 1.7159$ and a slope parameter of $\beta = 2/3$.

Next, a function was developed to perform the backpropagation-based learning of the network weights. This function (*trainNN.mat*) uses code from the feedforward function to perform the feedforward evaluation of the network and then applies the backpropagation method presented in section 1.1.2 to update the weights. The target values used to evaluate the backpropagation criterion function were bipolar one-hot labels. That is, the targets t for the classes are:

$$t_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, t_3 = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} \tag{29}$$

As should be obvious from an examination of the backpropagation equations, the weights of a network cannot be initialized to zero. Many authors have proposed weight initialization schemes to minimize the time needed to train a network. However, the neural network designer should still carefully consider weight initialization choices as it can affect possible convergence of the network. Often times, prior information about a classification problem can help serve as a guide to initialize weights. In this project, the weights were initialized following a uniform random distribution between the values of -1 and 1, as suggested in [5]. Other weight initializations were tested, such as normal distributions of the weights and varying a uniform distribution over different ranges. However, experimentation showed that an initialization with a normal distribution between -1 to 1 caused networks to converge most rapidly.

The learning rate η also had to be considered. Recall that the learning rate affects the step size of the gradient decent during backpropagation. A learning rate that is too large can cause the minimum error to be overshot, while too small a step can lengthen the time to convergence. It is often recommended that a learning rate of $\eta = 0.1$ be used as a starting point. Through experimentation, it was found that in this project, a learning rate of $\eta = 0.01$ offered decent performance while not elongating the training duration to a substantial amount.

The implemented backpropagation follows a stochastic training protocol. That is, the weight updates are performed each time a training data sample is presented to the network and the training samples are presented in a random sequence. A sequential training protocol was also attempted. In this paradigm, the training samples are presented to the neural network in a sequential order. This protocol was found to be far inferior to stochastic training. This is likely due to the ordering of the training samples. In this project, the samples were collected into a large training matrix where each class is grouped together. Consequently, the weights get biased towards the last occurring class, as it only encounters samples of that class prior to the end of learning. For example, this project contained 2000 training samples, of which, the last 400 samples are class three. Therefore, in each training epoch, the last 400 weight updates will bias the entire classifier towards class three. Thus, upon classification, the classifier will misclassify many patterns of class one and two as class three. The stochastic training method alleviates this issue as the random nature of the training sample presentation will prevent bias from accumulating in the network.

To verify that the designed backpropagation learning was working correctly, the average cost per epoch was calculated as:

$$Cost_{epoch} = \frac{\sum_1^n \frac{1}{2} \sum_{k=1}^c (t_k - z_k)^2}{n} \tag{30}$$

Here, n is the number of training samples. The average cost should be expected to decrease throughout the epochs, given the neural net is performing the backpropagation correctly. Figure 12 shows an example of the obtained average cost per epoch using an instance of a network with 18 hidden units and a learning rate of $\eta = 0.01$ as trained over 50 epochs:

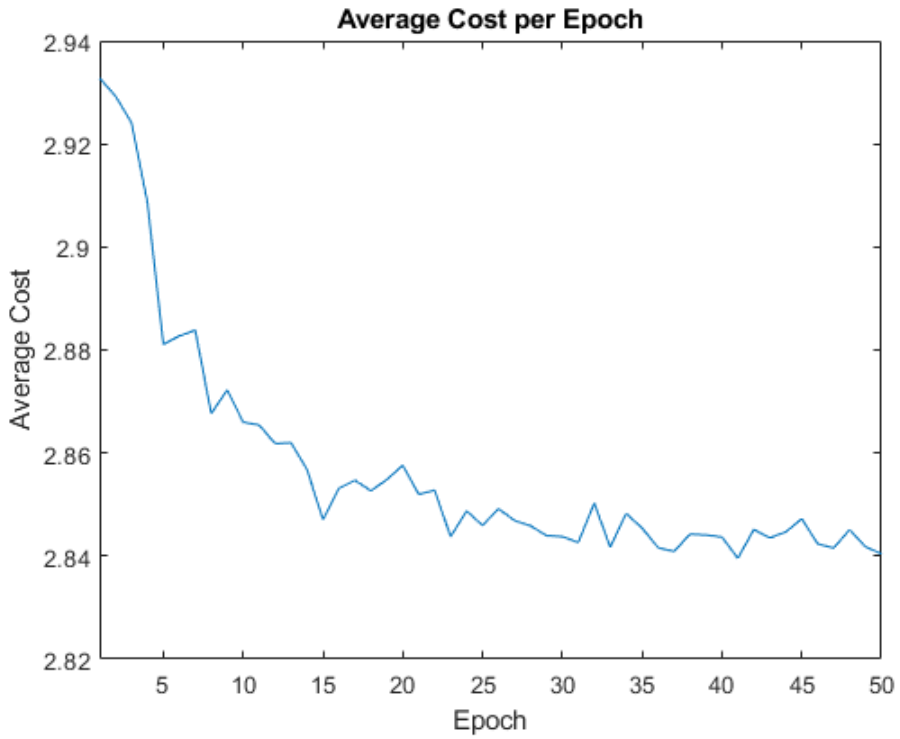

Figure 12: Average Cost per Epoch for a Network with 18 Hidden Nodes

An analysis of 12 shows that the average cost decreases throughout the epochs of training. Thus showing effective backpropagation learning. However, it should be noted that the average cost after the first epoch is already quite low and only decreases by approximately 0.1 over the course of training. This is not an issue as it an indication that the first epoch was quite effective in training the network and that subsequent epochs only offer marginal improvements.

Figure 12 also hints that perhaps such a high number of training epochs is unnecessary for this classification problem. To determine an appropriate stopping point for the neural network training, validation was incorporated. Generally, validation data is used as a metric to determine if the network is becoming over-trained. That is, at certain epoch intervals, the network is used to classify both the training and validation data. If the training classification errors continue to fall and validation errors begin to rise, this is an indication that over-training may be occurring and training should be stopped. However, it was discovered that to observe this phenomenon with this data, the learning rate must be set quite small and training had to be run over a extraordinarily large number of epochs. Figure 13 shows an example of the training and validation classification errors obtained for a network with 18 hidden nodes and a learning rate of 0.01.

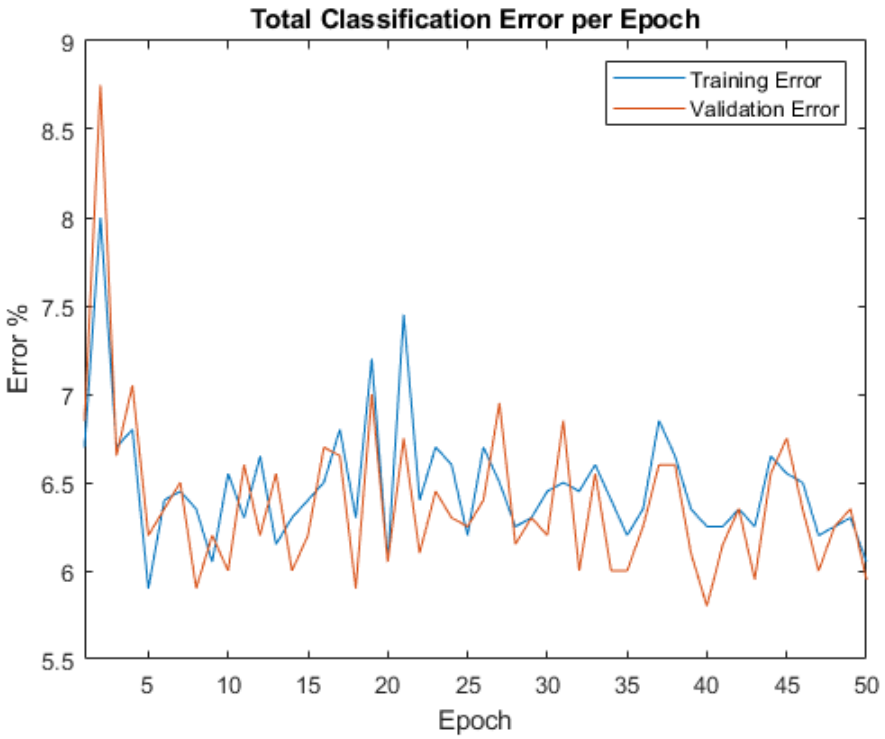

Figure 13: Training and Validation Classification Error per Epoch for a Network with 18 Hidden Nodes

Therefore, an alternative criterion to ceasing training was developed. This project utilized an error threshold as a metric for stopping training. Both the training and validation classification errors are checked each epoch of training and if both error percentages are below an error threshold, training is stopped. Through experimentation, it was found that good classification performance could be achieved if the error threshold was set at 6%.

2.2.3 Determining the Optimal Architecture

The number of input and output nodes of a neural network are dictated by the dimensionality of the input data and the number of classes, respectively. However, the number of hidden nodes to incorporate into the hidden layer is a user-defined hyper parameter. Unfortunately, a concise solution to the optimal number of hidden nodes in a network does not exist. There are many authors who suggest guidelines for an initial estimation of how many hidden nodes may work. However, a neural network designed should still adjust this initial estimation to maximize the performance of a network to the individual classification problem.

To determine the optimal architecture for this data, several networks were trained with a varying number of hidden nodes and their performance was evaluated. Specifically, the number of hidden nodes were varied from $n_H = 6$ to $n_H = 32$ in two node increments. At each value of n_H, ten networks were trained and evaluated on the test data, and the best performing network was selected as the optimal network for that particular n_H architecture. Figure 14 below shows the error obtained by the optimal networks for $n_H = 6$ to $n_H = 32$.

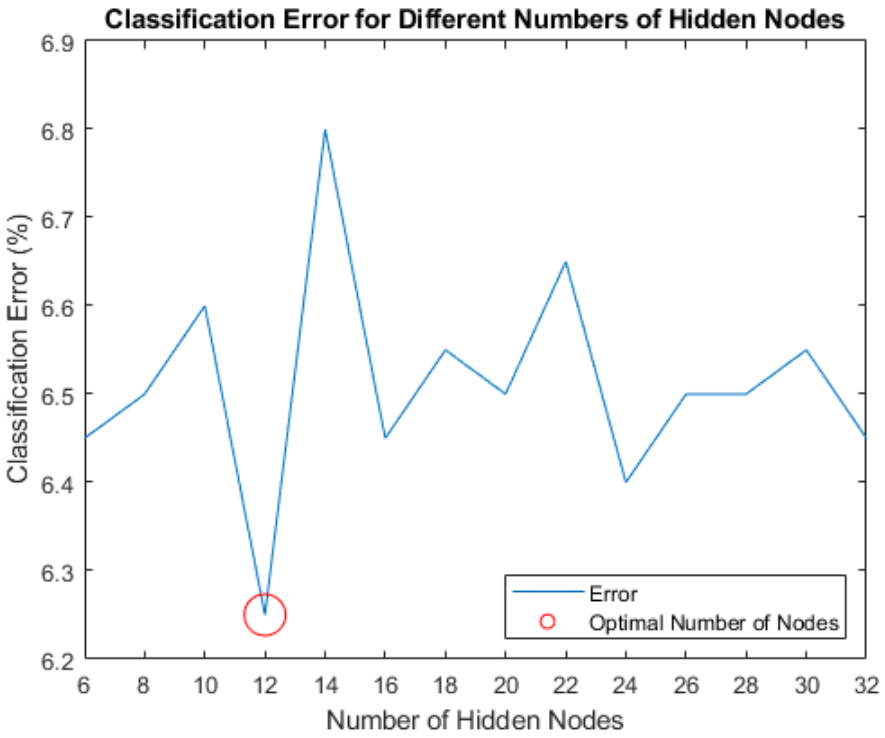

Figure 14: Percent Classification Error of Test Data by Optimal Networks with n_H Architectures Ranging from $n_H = 6$ to $n_H = 32$

From an examination of figure 14, one can see that the lowest classification error arose from a network consisting of twelve hidden nodes. Therefore, this architecture was selected to be the final architecture of the developed neural network. It should be noted that in this case, access to test data was readily available. However, in real-world applications, one rarely will have access to test data. Therefore, it is best practice to instead use the validation data classification accuracies or errors as a metric to determine the number of hidden nodes in the network. The decision to use the test data to set the architecture of the network was made purely to ensure that a network that exhibited maximal classification performance was presented. The final neural network architecture is depicted in figure 15.

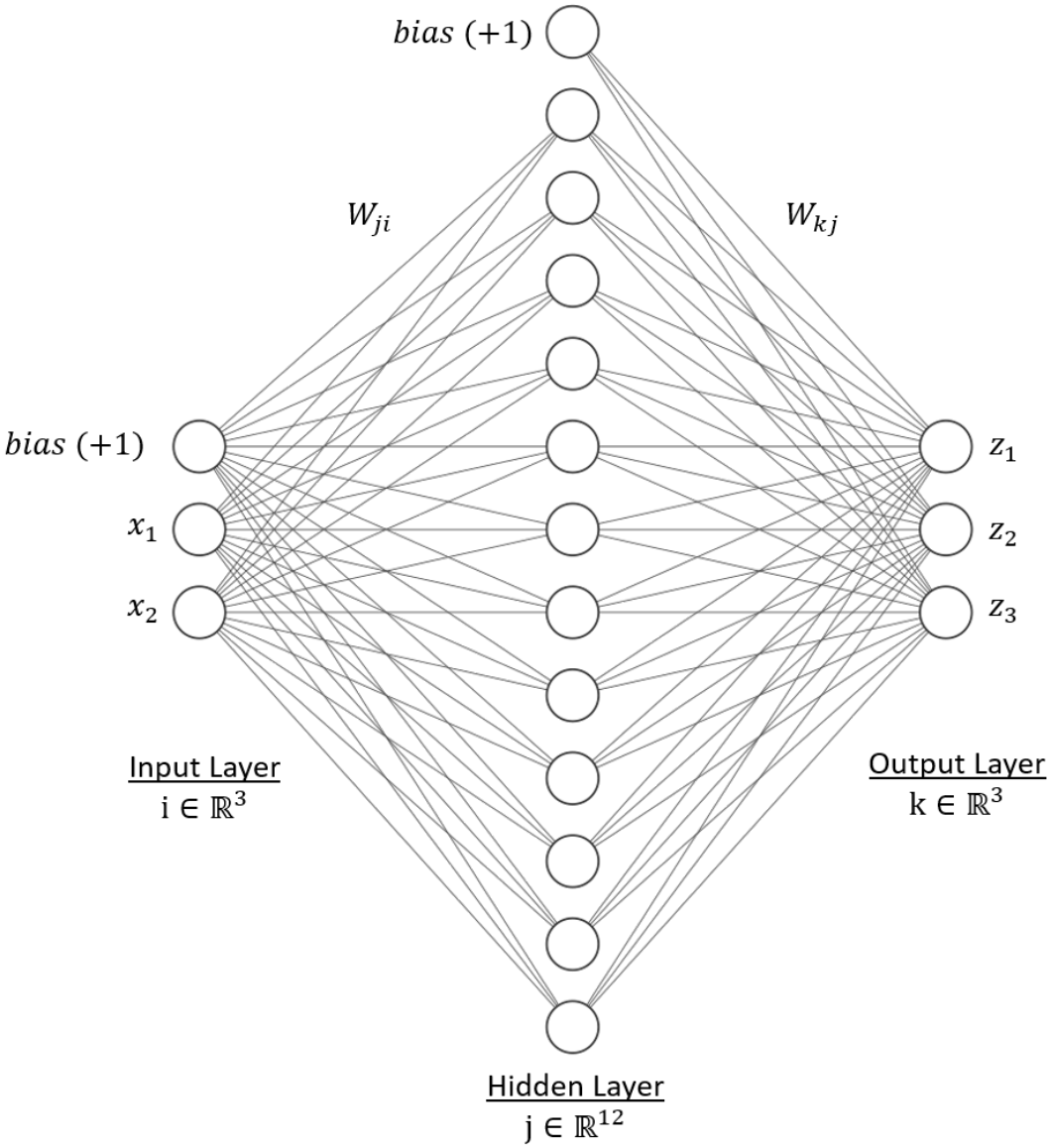

Figure 15: Final Neural Network Architecture

The final network depicted in figure 15 exhibited the input layer to hidden layer weights as described in figure 16 and the hidden layer to output layer weights as given in figure 17.

	$w_{j,bias}$	$w_{j,1}$	$w_{j,2}$
$w_{1,i}$	-0.69	0.69	-0.71
$w_{2,i}$	-0.60	-1.40	0.02
$w_{3,i}$	1.91	-0.12	2.19
$w_{4,i}$	-0.44	-0.10	0.93
$w_{5,i}$	-0.99	1.03	1.09
$w_{6,i}$	1.26	-1.01	1.43
$w_{7,i}$	-0.69	0.82	-1.00
$w_{8,i}$	0.39	-2.40	-0.88
$w_{9,i}$	1.48	-1.56	2.20
$w_{10,i}$	0.07	-1.03	-2.54
$w_{11,i}$	0.30	-0.70	-0.16
$w_{12,i}$	1.55	-0.18	-1.05

Figure 16: Input Layer to Hidden Layer Weights

	$w_{1,j}$	$w_{2,j}$	$w_{3,j}$
$w_{k,bias}$	-0.53	-0.64	-0.64
$w_{k,1}$	-0.95	-1.32	0.10
$w_{k,2}$	-0.43	-0.59	0.69
$w_{k,3}$	-0.04	0.00	0.17
$w_{k,4}$	-0.05	0.42	0.19
$w_{k,5}$	0.10	-0.42	0.74
$w_{k,6}$	-0.43	-0.16	-0.08
$w_{k,7}$	0.96	0.82	0.30
$w_{k,8}$	0.26	-0.12	-0.09
$w_{k,9}$	0.70	-0.58	0.52
$w_{k,10}$	-0.73	0.21	0.68
$w_{k,11}$	0.19	0.35	-0.06
$w_{k,12}$	0.14	-0.19	0.70

Figure 17: Hidden Layer to Output Layer Weights

2.3 Neural Network Classification

The final developed neural network defined by figures 15, 16, 17, was used to classify the test data. Upon evaluation, the overall classification accuracy obtained was 93.75%. Figure 18 shows a breakdown of the overall classification metrics and the class-by-class metrics.

	Class 1	Class 2	Class 3	Overall
Accuracy (%)	94.30	94.67	91.00	93.75
Error (%)	5.70	5.33	9.00	6.25%

Figure 18: Class-by-Class and Overall Neural Network Classification Results

To gain a better sense of these classification results, the decision regions learned from the neural network are plotted with the preprocessed training and test data superimposed in figures 19 and 20 below:

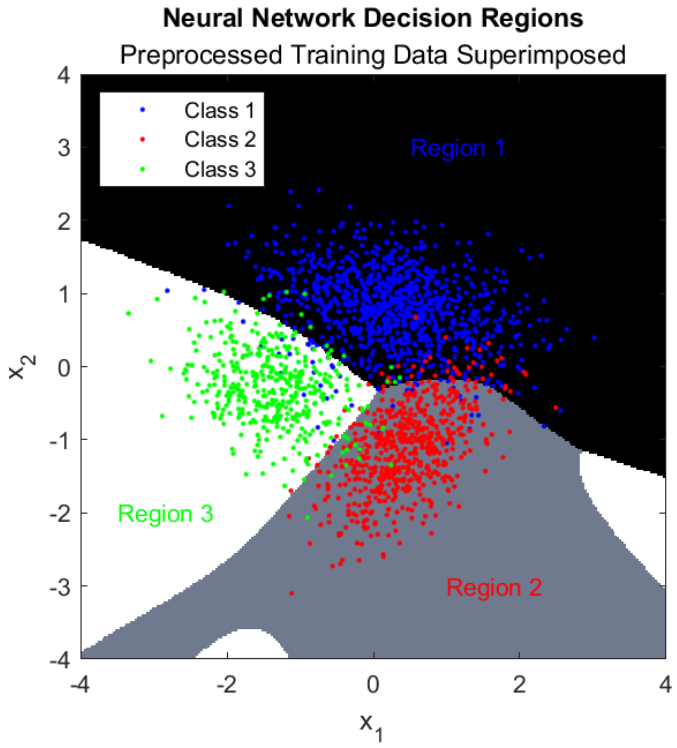

Figure 19: Neural Network Decision Regions with Training Data Superimposed

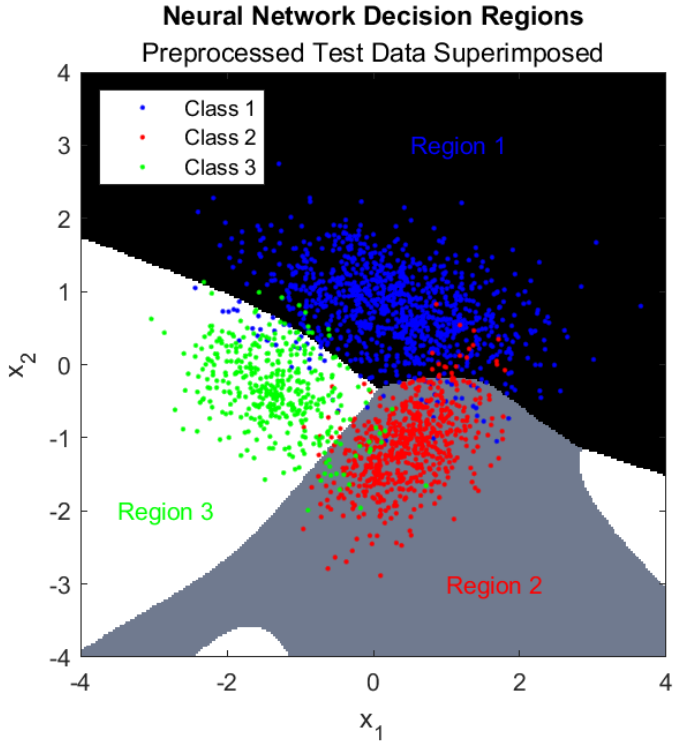

Figure 20: Neural Network Decision Regions with Test Data Superimposed

An analysis of figures 19 and 20 shows that this neural network is able to form decision regions that seem to be mostly optimal for classification of this data. There are, however, some discontinuities in the decision regions. This may be due to the fact that the training for this neural network only lasted 12 epochs before training was terminated. With more training, it was found that the decision regions become simply connected within the axis of observation. However, this may not mean that the regions are simply connected across the entire possible feature space. Unfortunately, it is difficult to extrapolate exactly why these neural networks exhibit particular decision regions.

Lastly, to get an understanding of the misclassifications made by the network, figure 21 depicts the misclassified test data points superimposed over the neural network decision regions.

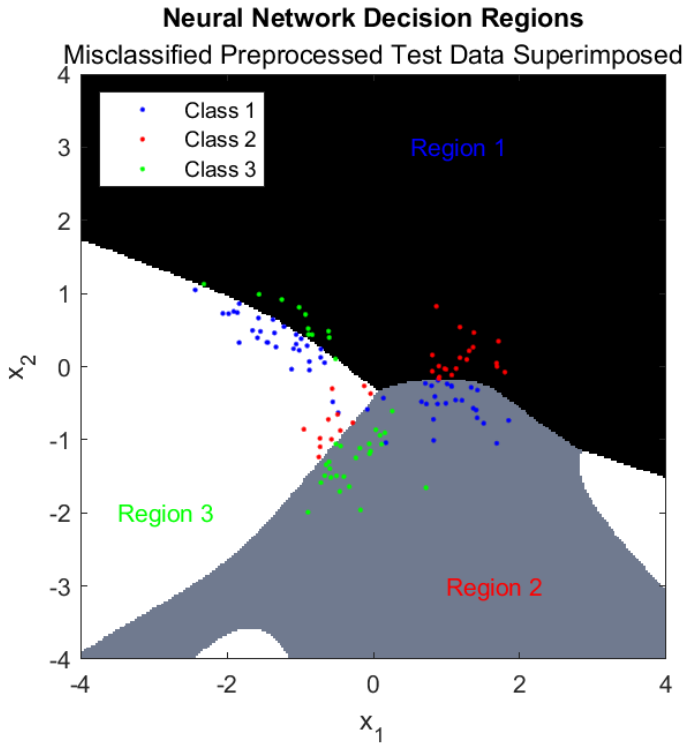

Figure 21: Neural Network Decision Regions with Misclassified Test Data Superimposed

Figure 21 shows that there are relatively few misclassifications made by the neural network. Further, the points that were misclassified are such points that even an optimal Bayesian classifier would most likely have misclassified. Such a comparison will be explored in more depth in section 3.

3 Discussion and Conclusions

It has been shown here that the designed neural network achieves a high classification accuracy. However, a neural network approach is by no means the only method of classifying the given test data. Further, using a neural network here is by far not an optimal approach. In

a previous project, k-nearest neighbors classifiers and a Bayesian classifier were tested using an identically distributed set of data as used in this project. In fact, as mentioned, the data used here was a subset of that data. Both the Bayesian and k-nearest neighbor approaches also achieved high levels of performance. For reference, figure 22 tabulates the performance metrics for each classifier.

	Class 1	Class 2	Class 3	Overall
Bayesian Error (%)	4.88	6.37	10.47	6.44
Bayesian Accuracy (%)	95.12	93.63	89.53	93.56
KNN				
$k_n = 1$ **Error (%)**	7.42	10.70	13.08	9.54
$k_n = 1$ **Accuracy (%)**	92.58	89.30	86.92	90.47
$k_n = 3$ **Error (%)**	8.55	7.97	8.22	8.31
$k_n = 3$ **Accuracy (%)**	91.45	92.03	91.78	91.69
$k_n = 5$ **Error (%)**	9.00	7.38	7.27	8.17
$k_n = 5$ **Accuracy (%)**	91.00	92.62	92.72	91.83
Neural Network				
NN Error (%)	5.70	5.33	9.00	6.25
NN Accuracy (%)	94.30	94.67	91.00	93.75

Figure 22: Classifier Performance Breakdown

An examination of figure 22 shows that all three classifiers achieved over a 90% classification accuracy. A cursory review of the results show that the neural network performed best. This may lead one to falsely assert that a neural network approach to this problem is better than the others, including a Bayesian approach. While it is true that the neural network performed better than the Bayesian, one must examine the results in context. The Bayesian classifier was tested on a data set that was ten times the length of the data set used here. Therefore, such direct comparisons of the results are not valid. However, there are some important insights to be gleaned from a comparison of the different classifier implementations that are worth discussing.

First, it is quite incorrect to posit that a neural network implementation can achieve better classification accuracies than a Bayesian approach. A Bayesian classifier can define the optimal decision boundaries by taking into account the statistics of the data. Thus, it can provide for classification at or near the Bayes error limit. A neural network may be able to emulate such decision regions with the right architecture and training paradigm. However, this is not a guarantee. For example, figure 23 shows the decision regions produced under a variety of network hyperparameters.

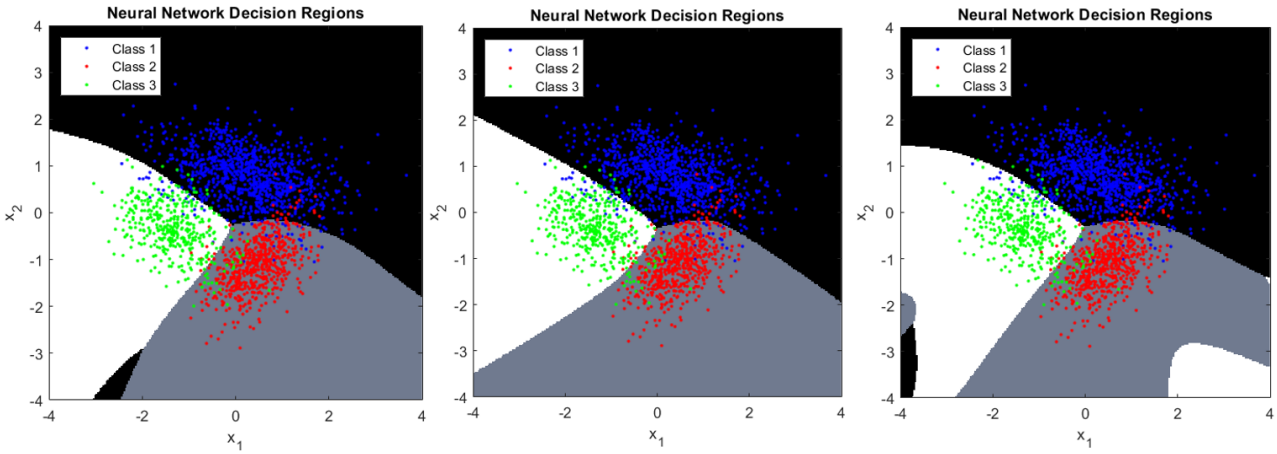

Figure 23: Different Neural Network Classification Regions

As one can see from figure 23, different neural network architectures and hyperparameters choices will produce different decision regions. This demonstrates that utilizing a neural network is not a substitute to the Bayesian approach that will produce the optimal decision regions given accurate knowledge of the data distribution and parameters.

However, this leads to one of the greatest strengths of a neural network approach. When implementing a neural network, one does not need knowledge of the data statistics to be able to form a network that will provide good classification results. Often times, real-world data is complex, and it is difficult to impossible to determine its underlying distribution. This is where neural networks excel. In particular, deep neural networks, that is, neural networks with many hidden layers, can easily classify complex real-world data that other statistical based methods cannot. However, designers and users must use such approaches with caution. Because there is not an underlying mathematically proven statistical basis to neural network decisions, one is unable to completely prove that the classifier will always exhibit that performance. This "black-box" nature of neural networks needs to be taken into account, especially if the classifier is to be used for mission-critical applications.

The popularity of neural networks has been on the rise in recent years and one can see them widely used in the literature and in industry for classification tasks. An argument could be made that neural networks are slowly becoming the default choice for classification problems, even those where they are not warranted. As mentioned, there are real-world complex problems in which the use of a neural network is indicated. However, when approaching a pattern classification problem, one needs to consider if simpler approaches may be more beneficial. This project shows that the construction and optimization of a neural network is a complicated process. However, for this project, this added complexity only yielded approximately an additional 2-3% accuracy over a simple k-nearest approach. This is a moderate disparity in performance given that a k-nearest neighbor approach is substantially more simple to implement. This shows that for simple classification problems, one needs to consider whether a moderate performance increase gained with neural networks is worth the added training complexity. Further, if one is truly concerned with maximal performance, and one knows the underlying data distribution and parameters, or can accurately estimate them, implementing a Bayesian classifier is a more straightforward process as compared to a neural

network approach and can provide for optimal classification.

Neural networks are undoubtedly a powerful tool for classification problems. But their rise in popularity has eclipsed other well-founded techniques and has resulted in neural network utilization for many tasks where their suitability or necessity is questionable. Moreover, this prevalence has coincided with the development of tools that make the training and deployment of neural networks simple enough that a user with little knowledge of their underlying theory can successfully implement them. This ease, however, means that many simply implement neural network based approaches without first considering other options. This project has shown that while neural networks are well-performing classifiers, there are many situations in which other classification techniques may be superior or far simpler to implement. This is not to say that neural networks should not be considered as a potential solution. But it is paramount that the designer have a solid theoretical understanding of their operation and can justify their use over other classification approaches.

4 References

[1] I. McAtee and S. Wiederholt, "Project 0: Bayesian Classifier", University of Wyoming, 2021

[2] I. McAtee and S. Wiederholt, "Project 1: A Comparison of Bayesian and Linear Discriminant Analysis Classifiers", University of Wyoming, 2021

[3] C.G. Wright, "Object and Pattern Recognition EE5650 Chapter 4, part 3" University of Wyoming, 2021

[4] I. McAtee and S. Wiederholt, "Project 2: A Comparison of K-Nearest Neighbors and Bayesian Classification Techniques", University of Wyoming, 2021

[5] C.G. Wright, "Object and Pattern Recognition EE5650 Chapter 6, part 4" University of Wyoming, 2021

5 Appendix

Listing 1: Project 3: Main Function

```
%Project 03: Neural Network
%Author: Ian McAtee and Stein Wiederholt
%Date: 11/23/2021
%DESCRIPTION: Main function for EE5650 Project 3. Function can
   [+]find a
    %well-performing neural network for the training and test data
       [+], or
```

```
    %users can specify their own network configuration, or one can
        [+] simply
    %use the weights provided in FinalWeights12.mat to examine the
    %predefined weights found by the authors with a 12 hidden node
        [+] network

close all
clear all

%% DATA SETUP
%Load in data
load('training2.mat')
load('test2.mat')

%Load in weights found previously by the author
load('FinalWeights12.mat')

%Set number of training samples for each class
N = [1000,600,400];
totalN = sum(N);

%Form the training data
train1 = class1_train(1:N(1),:);
train2 = class2_train(1:N(2),:);
train3 = class3_train(1:N(3),:);
train = [train1;train2;train3];

%Form the validation data
val1 = class1_train(N(1)+1:N(1)*2,:);
val2 = class2_train(N(2)+1:N(2)*2,:);
val3 = class3_train(N(3)+1:N(3)*2,:);
val = [val1;val2;val3];

%Form the test data
test1 = class1_test(1:N(1),:);
test2 = class2_test(1:N(2),:);
test3 = class3_test(1:N(3),:);
test = [test1;test2;test3];

%Form labels
labels = [ones(N(1),1);2*ones(N(2),1);3*ones(N(3),1)];

%Preprocess training data and get preprocessing parameters
fprintf('PREPROCESSING DATA\n')
[trainProcess,preProcessParam] = preprocessTrain(train,[N]);
```

```
49
%Setup a structure for the neural network architecture and
   [+]parameters
51 %These hyperparameters can be changed by the user
initialNNParam.hiddenNodes = 18;
53 initialNNParam.learnRate = 0.01;
initialNNParam.maxEpochs = 1000;
55
%% NEURAL NETWORK TRAINING
57 %Leave only one ****section**** uncommented

59 %Default to setting the trained network to the optimal found
   [+]previously
%
   [+]*********************************************************************
   [+]
61 trainedNNParam.wj = wj;
trainedNNParam.wk = wk;
63 %
   [+]*********************************************************************
   [+]

65 %Train several networks and select the best
%
   [+]*********************************************************************
   [+]
67 % %Set the number of networks tested for each number of hid units
   [+]instance
% %Depending on how you set these values, this can take a long
   [+]time
69 % numberNetworks = 10;
% maxHidUnits = 32;
71 % minHidUnits = 6;
% count = 0;
73 % %Loop through networks with differing number of hidden units
% for i = minHidUnits:2:maxHidUnits
75 %     fprintf('TRAINING NEURAL NETWORK\n')
%     count = count+1;
77 %     errors = zeros(numberNetworks,1);
%     initialNNParam.hiddenNodes = i; %Set number of hidden units
79 %     %Train networks and evaluate performace, tabulate error
%     for j = 1:numberNetworks
81 %         trainedNN(j).Param = trainNN(trainProcess,val,labels,
   [+]initialNNParam,preProcessParam);
```

```
%          class = evaluateNN(test,trainedNN(j).Param,
    [+]preProcessParam);
%          errors(j) = length(find(labels-class))/totalN * 100;
%      end
%      %Find network with lowest error
%      optimal = find(errors == min(errors));
%      opError(count) = errors(optimal(1));
%      trainedNNParam.wj = trainedNN(optimal(1)).Param.wj;
%      trainedNNParam.wk = trainedNN(optimal(1)).Param.wk;
%      %Save networks as optimal for that num of hid units
%      save(int2str(i),'-struct','trainedNNParam');
% end
% %Set the NN to the best performing network overall
% overallOp = find(opError == min(opError));
% overallOpIndex = minHidUnits + 2*(overallOp(1)-1);
% load(strcat(int2str(overallOpIndex),'.mat'));
% trainedNNParam.wj = wj;
% trainedNNParam.wk = wk;
% save(strcat('OptimalNetwork',int2str(overallOpIndex)),'-struct
    [+]','trainedNNParam');
%
    [+]*****************************************************************
    [+]

%Train a single new network
%
    [+]*****************************************************************
    [+]
%fprintf('TRAINING NEURAL NETWORK\n')
%trainedNNParam = trainNN(trainProcess,val,labels,initialNNParam,
    [+]preProcessParam);
%
    [+]*****************************************************************
    [+]


%% NEURAL NETWORK EVALUATION
fprintf('CLASSIFYING TEST DATA\n')

%Perform the classification of the test data with the network
class = evaluateNN(test,trainedNNParam,preProcessParam);

%Find the percent error and accuracy of classification
mis = find(labels-class);
mis1 = find(labels(1:N(1))-class(1:N(1)));
```

```
mis2 = find(labels(N(1)+1:N(1)+N(2))-class(N(1)+1:N(1)+N(2)));
mis3 = find(labels(N(1)+N(2)+1:totalN)-class(N(1)+N(2)+1:totalN));
error = length(mis)/totalN * 100;
error1 = length(mis1)/N(1) * 100;
error2 = length(mis2)/N(2) * 100;
error3 = length(mis3)/N(3) * 100;

%Display classification metrics
fprintf('Results:\n')
fprintf('\tCLASS 1:\n')
fprintf('\t\tError:      %.2f%%\n',error1)
fprintf('\t\tAccuracy:   %.2f%%\n',100-error1)
fprintf('\tCLASS 2:\n')
fprintf('\t\tError:      %.2f%%\n',error2)
fprintf('\t\tAccuracy:   %.2f%%\n',100-error2)
fprintf('\tCLASS 3:\n')
fprintf('\t\tError:      %.2f%%\n',error3)
fprintf('\t\tAccuracy:   %.2f%%\n',100-error3)
fprintf('\tOVERALL:\n')
fprintf('\t\tError:      %.2f%%\n',error)
fprintf('\t\tAccuracy:   %.2f%%\n',100-error)

%% GENERATE PLOTS

%Run the function to generate the plots for this project
genProj3Plots(train,test,val,mis,N,trainedNNParam,preProcessParam)
   [+];
```

Listing 2: Training Data Preprocessing Function

```
%FUNCTION: preProcessTrain.m
%AUTHOR: Ian McAtee
%DATE: 11/20/2021
%DESCRIPTION: Function to perform the preprocessing of training
   [+]data prior
    %to its input to a neural network. Transform the data such
       [+]that the
    % mean is approximately zero and the covariance is the
       [+]identity matrix
%INPUT:
    %train: A nxD matrix of training data samples
    %N: A 1xc row vector containing the number of samples in each
       [+]class
    %preProcessParameters: A structure containing the weighted
       [+]mean of the
        %original training data, as well as the phi and lamda used
```

```
            [+] in the
        %preprocessing whitening transfrom, used to preprocess the
13      %validation data
%OUTPUT:
15      %processedTrain: A nxD matrix of preprocessed training data
           [+]samples
        %preProcessParameters: A structure containing the weighted
           [+]mean of the
17         %original training data, as well as the phi and lamda used
              [+] in the
           %preprocessing whitening transfrom, used to preprocess the
19         %validation data

21  function [processedTrain,preProcessParam] = preprocessTrain(train,
       [+]N)

23  %Get necessary variables
    numFeat = size(train,2); %Number of feature dimensions
25  numClasses = length(N); %Number of classes
    totalN = sum(N); %Total number of samples
27
    %Preallocate a numFeatures x numClasses mean matrix
29  means = zeros(numFeat,numClasses);
    weightMean = zeros(numFeat,1);
31
    %Calculate the weighted mean of the training data
33  startIndex = 1;
    endIndex = 0;
35  for i = 1:numClasses
        endIndex = endIndex + N(i);
37      means(:,i) = mean(train(startIndex:endIndex,:))'; %Mean
        weightMean = weightMean + (N(i)/totalN)*means(:,i); %Weighted
           [+]mean
39      startIndex = startIndex + N(i);
    end
41
    %Make the training data approximately zero mean
43  train = train-weightMean';

45  %Use a whitening transformation make the training covariance = I
    covTrain = cov(train);
47  [phi,lam] = eig(covTrain);
    Y = phi*train';
49  processedTrain = (sqrt(lam)\ Y)';
```

```
%Set outputs preprocessing parameters
preProcessParam.weightMean = weightMean;
preProcessParam.lam = lam;
preProcessParam.phi = phi;

%Find mean and covariance of processed training data
m = mean(processedTrain);
c = cov(processedTrain);

%Error threshold for determining successful processing
thresh = 1e-10;

%Preprocessing Successful if mean and cov within thresh of 0 and I
if (sum(m)<thresh)&&((sum(eye(numFeat)-c,'all'))<thresh)
    fprintf('Training Data Preprocessing Successful\n')
%Preporcessing failed
else
    error('Training Data Preprocessing FAILED')
end

%If dimensionality 2, display the preprocessed train data mean and
   [+] cov
if (numFeat==2)
    fprintf('\tPreprocessed Training Data Mean:\n')
    fprintf('\t%.2f\n',m);
    fprintf('\tPreprocessed Training Data Covariance:\n')
    fprintf('\t%.2f \t%.2f\n',c(1,1),c(1,2));
    fprintf('\t%.2f \t%.2f\n',c(2,1),c(2,2));
end
```

Listing 3: Neural Network Training Function

```
%FUNCTION: trainNN.m
%AUTHOR: Ian McAtee
%DATE: 11/29/2021
%DESCRIPTION: Function to perform the neural network training via
   [+]the
    %backpropagation algorithm and validation stoppage
%INPUT:
    %train: A nxD matrix of preprocessed training data samples
    %val: A nValxD matrix of validation data samples
    %labels: a nX1 vector of class labels of the training data
    %initialNNParam: A structure containing the number of desired
       [+]hidden
        %nodes, the learning rate, and max number of training
           [+]epochs
```

```
12      %preProcessParameters: A structure containing the weighted
            [+]mean of the
            %original training data, as well as the phi and lamda used
                [+] in the
14          %preprocessing whitening transfrom, used to preprocess the
            %validation data
16  %OUTPUT:
        %trainedNNParam: A structure containing wj (the learned input
            [+]to hidden
18          %layer weights and wk (the learned hidden to output layer
                [+]weights)

20  function trainedNNParam = trainNN(train,val,labels,initialNNParam,
        [+]preProcessParam)

22  %Find length of training and validation data
    lenTrain = length(train);
24  lenVal = length(val);

26  %Find the number of classes from the labels
    numClasses = length(unique(labels));
28
    %Find the number of samples in each class
30  N = zeros(1,numClasses);
    for i = 1:numClasses
32      N(i) = length(labels(labels==i));
    end
34
    %Form target values
36  t = [];
    for i = 1:numClasses
38      target = -1*ones(1,numClasses);
        target(i) = 1;
40      t = [t;repmat(target,N(i),1)];
    end
42
    %Extract the preprocessing parameters
44  weightMean = preProcessParam.weightMean;
    lam = preProcessParam.lam;
46  phi = preProcessParam.phi;

48  %Preprocess val data
    %Make the entire val set zero mean
50  val = val-weightMean';
```

```
%Whiten the varience of the validation data via a whitening
   [+]transform
Y = phi*val';
val = (sqrt(lam)\ Y)';

%Form augmented training and validation data
augTrain = [ones(lenTrain,1),train];
augVal = [ones(lenVal,1),val];

%Find number of features
numFeat = size(train,2);

%Setup Neural Network Architecture
numInNodes = numFeat + 1;
numHidNodes = initialNNParam.hiddenNodes;
numOutNodes = numClasses;

%Preallocate and Initialize weight matrices
%Uncomment below for gaussian distributed weights
%wj = randn(numInNodes,numHidNodes);
%wk = randn(numHidNodes+1,numOutNodes);

%Initialize weights as uniform dist. between -1 and 1
a = -1; b = 1;
wj = (b-(a)).*rand(numInNodes,numHidNodes) + a;
wk = (b-(a)).*rand(numHidNodes+1,numOutNodes) + a;

%Extract the hyperparameters
numEpochs = initialNNParam.maxEpochs;
eta = initialNNParam.learnRate;

%Preallocate variables to hold training metrics
classTrain = zeros(lenTrain,1);
classVal = zeros(lenVal,1);
errorTrain = zeros(1,numEpochs);
errorVal = zeros(1,numEpochs);
cost = zeros(lenTrain,1);
costPerEpoch = zeros(numEpochs,1);

%Set the training stoppage error threshold
errorThresh = 6.0; % error of 6%

%Neural Network Training
%Loop through epochs
fprintf('Training: Epoch ');
```

```
96  for epoch = 1:numEpochs

98      %Format the console output to keep track of epochs
        numBackSpace = length(num2str(epoch));
100     backSpace = repmat('\b',1,numBackSpace);
        fprintf('%d',epoch);
102
        %Loop stochastically through training samples
104     for n = randperm(lenTrain) %Random Sample

106         %%%%%%%% FORWARD PROPAGATION %%%%%%%%
            netj = wj'*augTrain(n,:)'; %Net of hidden nodes
108         yj = sigmoidF(netj); %Output of hidden nodes
            y = [1;yj]; %Add bias
110
            netk = wk'*y; %Net of output nodes
112         zk = sigmoidF(netk); %Output of output nodes

114         %Sace the cost
            cost(n) = 0.5*sum((t(n)'-zk).^2);
116
            %%%%%%%% BACKPROPAGATION %%%%%%%%
118         %Find Wkj Update Deltas
            for k = 1:numOutNodes
120             deltaWk(:,k) = (eta*(t(n,k)-zk(k))*sigDeriv(netk(k)).*
                    [+]y)';
            end
122
            %Find Wji Update Deltas
124         for j = 1:numHidNodes
                delJ = sigDeriv(netj(j))*sum(wk(j+1,:)'.*(t(n,:)'-zk)
                    [+].*sigDeriv(netk));
126             deltaWj(:,j) = eta*delJ.*augTrain(n,:)';
            end
128
            %Update weights
130         wj = wj + deltaWj;
            wk = wk + deltaWk;
132
        end
134
        %Find the average cost per epoch
136     costPerEpoch(epoch) = sum(cost)/lenTrain;

138     %%%%%%%% VALIDATION %%%%%%%%
```

```
    %Use the network at current epoch to classify training data
    for n = 1:lenTrain
            netj = wj'*augTrain(n,:)'; %Net of hidden nodes
            yj = sigmoidF(netj); %Output of hidden nodes
            y = [1;yj]; %Add bias

            netk = wk'*y; %Net of output nodes
            zk = sigmoidF(netk); %Output of output nodes

            %Save classifications
            [~,classTrain(n)] = max(zk);
    end

    %Use the network at current epoch to classify validation data
    for n = 1:lenVal
            netj = wj'*augVal(n,:)'; %Net of hidden nodes
            yj = sigmoidF(netj); %Output of hidden nodes
            y = [1;yj]; %Add bias

            netk = wk'*y; %Net of output nodes
            zk = sigmoidF(netk); %Output of output nodes

            %Save classifications
            [~,classVal(n)] = max(zk);
    end

    %Find the training and validation classification errors
    errorTrain(epoch) = length(find(labels~=classTrain))/lenTrain *
       [+] 100;
    errorVal(epoch) = length(find(labels~=classVal))/lenVal * 100;

    %End training if both train and val errors are below error
       [+]threshold
    if((errorVal(epoch)<errorThresh)&&(errorTrain(epoch)<
       [+]errorThresh))
        epochStop = epoch;
        fprintf('\nTraining Successful \n')
        break
    end

    %Save the epoch that one stops at
    epochStop = numEpochs;
    fprintf(backSpace);
end

```

```
%Return the learned weights
trainedNNParam.wj = wj;
trainedNNParam.wk = wk;

%Plots

%Plot Cost per Epoch
figure()
plot([1:epochStop],costPerEpoch(1:epochStop)')
title('Average Cost per Epoch')
xlim([1,epochStop])
xlabel('Epoch')
ylabel('Average Cost')

%Plot Classification Error per Epoch
figure()
plot([1:epochStop],errorTrain(1:epochStop))
hold on
plot([1:epochStop],errorVal(1:epochStop))
hold off
title('Total Classification Error per Epoch')
xlim([1,epochStop])
xlabel('Epoch')
ylabel('Error %')
legend('Training Error','Validation Error','location','northeast')

end
```

Listing 4: Neural Network Evaluation (Classification) Function

```
%FUNCTION: evaluateNN.m
%AUTHOR: Ian McAtee
%DATE: 11/29/2021
%DESCRIPTION: Function to perform the neural network evaluation
    % (classification) of test data
%INPUT:
    %test: A nxD matirx of test data samples
    %trainedNNParam: A structure containing wj (the learned input
       [+]to hidden
        %layer weights and wk (the learned hidden to output layer
           [+]weights)
    %preProcessParameters: A structure containing the weighted
       [+]mean of the
        %original training data, as well as the phi and lamda used
           [+] in the
        %preprocessing whitening transfrom, used to preprocess the
```

```
13          %validation data
%OUTPUT:
15      %class: A nx1 vector of classifications determined by the
            [+]network for
            %the test data
17
function class = evaluateNN(test,trainedNNParam,preProcessParam)
19
%Find length of test data
21 lenTest = length(test);

23 %Extract the trained weights
wj = trainedNNParam.wj;
25 wk = trainedNNParam.wk;

27 %Extract the preprocessing parameters
weightMean = preProcessParam.weightMean;
29 lam = preProcessParam.lam;
phi = preProcessParam.phi;
31
%Preprocess test data
33 %Make the entire test set zero mean
test = test-weightMean';
35
%Whiten the varience of the test data via a whitening transform
37 Y = phi*test';
test = (sqrt(lam)\ Y)';
39 %Uncomment below to see that mean ~= 0 and cov ~= I
% mean(test)
41 % cov(test)

43 %Form the augmented test data
augTest = [ones(lenTest,1),test];
45
%Preallocate classification vector
47 class = zeros(lenTest,1);

49 %Forward propagate the test data through trained network
for n = 1:lenTest
51          %FORWARD PROPAGATION
            netj = wj'*augTest(n,:)';
53          yj = sigmoidF(netj);
            y = [1;yj];
55
            netk = wk'*y;
```

```
        zk = sigmoidF(netk);

        %Perform classifcation by taking max of output
        [~,class(n)] = max(zk);
end

fprintf('Classification Successful\n')
end
```

Listing 5: Plot Generation Function

```
%FUNCTION: genProj3Plots.m
%AUTHOR: Ian McAtee
%DATE: 11/31/2021
%DESCRIPTION: Function to plot various plots corresponding to
   [+]EE5650
    %project 3
%INPUT:
    %train: A nxD matrix of  training data samples
    %train: A nTestxD matrix of  test data samples
    %val: A nValxD matrix of validation data samples
    %mis: A mx1 vector containing the indices of the misclassified
       [+] points
    %N: A 1xC row vector of the number of samples in each class
    %trainedNNParam: A structure containing wj (the learned input
       [+]to hidden
        %layer weights and wk (the learned hidden to output layer
           [+]weights)
    %preProcessParameters: A structure containing the weighted
       [+]mean of the
        %original training data, as well as the phi and lamda used
           [+] in the
        %preprocessing whitening transfrom, used to preprocess the
        %validation data
%OUTPUT:
    %Scatter plots of the training, test, and validation data
    %Scatter plots of the preprocessed training, test, and val
       [+]data
    %Scatter plots of the neural network decision regions
       [+]superimposed with
        %the preprocessed training, test, validation, and
           [+]misclassified
        %points

function genProj3Plots(train,test,val,mis,N,trainedNNParam,
   [+]preProcessParam)
```

```
26
   %Find indices for easier plotting
28 N1s = 1;
   N1e = N(1);
30 N2s = N(1)+1;
   N2e = N(1)+N(2);
32 N3s = N(1)+N(2)+1;
   N3e = sum(N);
34 totalN = sum(N);

36 %Extract the trained weights
   wj = trainedNNParam.wj;
38 wk = trainedNNParam.wk;

40 %Extract preprocessing parameters
   weightMean = preProcessParam.weightMean;
42 lam = preProcessParam.lam;
   phi = preProcessParam.phi;
44
   %Preprocess train, test, and validation data
46 %Make the entire test set zero mean
   trainP = train-weightMean';
48 testP = test-weightMean';
   valP = val-weightMean';
50
   %Whiten the varience of the test data via a whitening transform
52 Ytrain = phi*trainP';
   Ytest = phi*testP';
54 Yval = phi*valP';
   trainP = (sqrt(lam)\ Ytrain)';
56 testP = (sqrt(lam)\ Ytest)';
   valP = (sqrt(lam)\ Yval)';
58
   %Set some axis options
60 axisOpt = [-20,20,-15,25];
   axisOpt2 = [-4,4,-4,4];
62
   %% Plot Data
64
   %Training Data
66 figure()
   scatter(train(N1s:N1e,1),train(N1s:N1e,2),'.b')
68 hold on
   scatter(train(N2s:N2e,1),train(N2s:N2e,2),'.r')
70 hold on
```

```
scatter(train(N3s:N3e,1),train(N3s:N3e,2),'.g')
axis(axisOpt)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Training Data')
legend('Class 1','Class 2','Class 3','Location','southwest')

%Validation Data
figure()
scatter(val(N1s:N1e,1),val(N1s:N1e,2),'.b')
hold on
scatter(val(N2s:N2e,1),val(N2s:N2e,2),'.r')
hold on
scatter(val(N3s:N3e,1),val(N3s:N3e,2),'.g')
axis(axisOpt)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Validation Data')
legend('Class 1','Class 2','Class 3','Location','southwest')

%Test Data
figure()
scatter(test(N1s:N1e,1),test(N1s:N1e,2),'.b')
hold on
scatter(test(N2s:N2e,1),test(N2s:N2e,2),'.r')
hold on
scatter(test(N3s:N3e,1),test(N3s:N3e,2),'.g')
axis(axisOpt)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Test Data')
legend('Class 1','Class 2','Class 3','Location','southwest')

%% Plot Preprocessed Data

%Prepocessed Training Data
figure()
scatter(trainP(N1s:N1e,1),trainP(N1s:N1e,2),'.b')
hold on
```

```
scatter(trainP(N2s:N2e,1),trainP(N2s:N2e,2),'.r')
hold on
scatter(trainP(N3s:N3e,1),trainP(N3s:N3e,2),'.g')
axis(axisOpt2)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Preprocessed Training Data')
legend('Class 1','Class 2','Class 3','Location','southeast')

%Prepocessed Validation Data
figure()
scatter(valP(N1s:N1e,1),valP(N1s:N1e,2),'.b')
hold on
scatter(valP(N2s:N2e,1),valP(N2s:N2e,2),'.r')
hold on
scatter(valP(N3s:N3e,1),valP(N3s:N3e,2),'.g')
axis(axisOpt2)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Preprocessed Validation Data')
legend('Class 1','Class 2','Class 3','Location','southeast')

%Prepocessed test Data
figure()
scatter(testP(N1s:N1e,1),testP(N1s:N1e,2),'.b')
hold on
scatter(testP(N2s:N2e,1),testP(N2s:N2e,2),'.r')
hold on
scatter(testP(N3s:N3e,1),testP(N3s:N3e,2),'.g')
axis(axisOpt2)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Preprocessed Test Data')
legend('Class 1','Class 2','Class 3','Location','southeast')

%% Plot Decision Regions

%Create temp x data to evaluate over
x = (-4:0.03:4); %Reduce the step size to for higher res. plots
```

```
L = length(x); %Get number of temp samples

temp = [];
%Form temp data
for i=1:L  % loop columns
    for j=1:L  % loop rows
        temp = [temp;x(i),x(j)]; %Form vector
    end
end

%Find length of temp data
lenTemp = length(temp);

%Form Augmented Temp data
augTemp = [ones(lenTemp,1),temp];

%Preallocate temp classification vector
classTemp = zeros(lenTemp,1);

%Forward propagate the temp data through trained network
for n = 1:lenTemp
        %FORWARD PROPAGATION
        netj = wj'*augTemp(n,:)';
        yj = sigmoidF(netj);
        y = [1;yj];

        netk = wk'*y;
        zk = sigmoidF(netk);

        %Perform classifcation by taking max of output
        [~,classTemp(n)] = max(zk);
end

%Form a matrix of classification regions
classRegions = zeros(L,L);
index1 = find(classTemp == 1);
index2 = find(classTemp == 2);
index3 = find(classTemp == 3);
classRegions(index1) = 1;
classRegions(index2) = 2;
classRegions(index3) = 3;

%Plot the decison boundaries with training data
figure()
imagesc(x,x,classRegions,'AlphaData',1.0)
```

```
206 axis ('xy')
    colormap('bone')
208 hold on
    scatter(trainP(N1s:N1e,1),trainP(N1s:N1e,2),'.b')
210 hold on
    scatter(trainP(N2s:N2e,1),trainP(N2s:N2e,2),'.r')
212 hold on
    scatter(trainP(N3s:N3e,1),trainP(N3s:N3e,2),'.g')
214 hold off
    axis(axisOpt2)
216 axis ('square')
    box on
218 xlabel('x_1')
    ylabel('x_2')
220 title('Neural Network Decision Regions')
    subtitle('Preprocessed Training Data Superimposed')
222 legend('Class 1','Class 2','Class 3','Location','northwest')
    text(0.5,3,'Region 1','Color','b')
224 text(-3.5,-2,'Region 3','Color','g')
    text(1,-3,'Region 2','Color','r')
226
    %Plot the decison boundaries with test data
228 figure()
    imagesc(x,x,classRegions,'AlphaData',1.0)
230 axis ('xy')
    colormap('bone')
232 hold on
    scatter(testP(N1s:N1e,1),testP(N1s:N1e,2),'.b')
234 hold on
    scatter(testP(N2s:N2e,1),testP(N2s:N2e,2),'.r')
236 hold on
    scatter(testP(N3s:N3e,1),testP(N3s:N3e,2),'.g')
238 hold off
    axis(axisOpt2)
240 axis ('square')
    box on
242 xlabel('x_1')
    ylabel('x_2')
244 title('Neural Network Decision Regions')
    subtitle('Preprocessed Test Data Superimposed')
246 legend('Class 1','Class 2','Class 3','Location','northwest')
    text(0.5,3,'Region 1','Color','b')
248 text(-3.5,-2,'Region 3','Color','g')
    text(1,-3,'Region 2','Color','r')
250
```

```
mis1 = mis(mis<=N1e);
mis2 = mis(mis>N1e & mis<=N2e);
mis3 = mis(mis>N2e & mis<=N3e);

%Plot the decison boundaries with misclassified
figure()
imagesc(x,x,classRegions,'AlphaData',1.0)
axis ('xy')
colormap('bone')
hold on
scatter(testP(mis1,1),testP(mis1,2),'.b')
hold on
scatter(testP(mis2,1),testP(mis2,2),'.r')
hold on
scatter(testP(mis3,1),testP(mis3,2),'.g')
hold off
axis(axisOpt2)
axis ('square')
box on
xlabel('x_1')
ylabel('x_2')
title('Neural Network Decision Regions')
subtitle('Misclassified Preprocessed Test Data Superimposed')
legend('Class 1','Class 2','Class 3','Location','northwest')
text(0.5,3,'Region 1','Color','b')
text(-3.5,-2,'Region 3','Color','g')
text(1,-3,'Region 2','Color','r')


end
```

Listing 6: Sigmoid Function

```
%FUNCTION: sigmoidF.m
%AUTHOR: Ian McAtee
%DATE: 11/21/2021
%DESCRIPTION: Function to compute the sigmoid funcion
%INPUT:
    %net: A dx1 vector that can represent the weighted input to a
       [+]neural
          %network node
%OUTPUT:
    %sigNet: A dx1 vector containing the sigmoid of the net input

function sigNet = sigmoidF(net)
    a = 1.7159;
```

```
    beta = 2/3;
    netLen = length(net);
    sigNet = zeros(netLen,1);
    for i = 1:netLen
        sigNet(i) = a*tanh(beta*net(i));
    end
end
```

Listing 7: Sigmoid Derivative Function

```
%FUNCTION: sigD.m
%AUTHOR: Ian McAtee
%DATE: 11/21/2021
%DESCRIPTION: Function to compute the derivative of the sigmoid
   [+]funcion
%INPUT:
    %net: A dx1 vector that can represent the weighted input to a
       [+]neural
          %network node
%OUTPUT:
    %sigD: A dx1 vector containing the sigmoid derivative of the
       [+]net input

function sigD = sigDeriv(net)
    a = 1.7159;
    beta = 2/3;
    netLen = length(net);
    sigD = zeros(netLen,1);
    for d = 1:netLen
        sigD(d) = (beta/a)*(a+sigmoidF(net(d)))*(a-sigmoidF(net(d)
           [+]));
    end
end
```