1 Introduction and objectives

The content of this report will focus on Artificial Neural Networks (ANN) with several optimization techniques which have been inspired by nature such as Particle Swarm Optimisation (PSO), Simulated annealing (SA) and Genetic Algorithms (GA). The feed-forward neural network (FFNN) will be implemented in this assignment where the forward and backward propagation's will be discussed as a separate topology.

The dataset used for this experiment will be the problen1 dataset, which has nine different example types to train, test, and validate the algorithm. Due to the number of datasets, the input vector, weights, and bias all need to be hyper-parameters that will be initialized at the start of the implementations with random values. The two optimizer typologies have additional requirements, one of which is a stopping criteria that is used to end the algorithm's training operations due to meeting the desired epochs or having met the ideal accuracy value. Additionally, a comparison will be made between the naive Bayesian classifier (NB) and the FFNN trained on the MNIST dataset.

2 Implementation of basic feedforward network with backprop

2.1 Discussion and derivation of theory

2.1.1 Feed-forward Network

$$y(\mathrm{x}, \mathrm{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathrm{x})\right) \tag{1}$$

The neural network is a connection of inputs, hidden layers and outputs. Each hidden layer is a node that performs some kind of activation on the input. With equation 1 representing the output of a nonlinear activation function performed on a neuron which is the weighted sum of inputs and bias [1].

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2}$$

There are two parts being summed at every junction used by the activation given by equation 2 and represented by a_j. The notation used in the equation for the weights is given by $w_{ji}^{(1)}$ and the biases noted by $w_{j0}^{(1)}$. The pre-activation given by each node at each layer is the weighted sum of the inputs from the previous layer plus bias value.

$$z_j = h\left(a_j\right) \tag{3}$$

The activation function is passed into a nonlinear function denoted by $h()$ and are generally a type of sigmoidal function such as a sigmoid or tanh function. This equation represents the output for a single layer NN and can be expanded upon by a second hidden layer by the equation to follow.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{4}$$

The second hidden layer activation function is described by equation 4, with K an integer value ranging from K= 1 to K = K, with K the total number of output nodes.

$$y_k = \sigma\left(a_k\right) \tag{5}$$

The last step is to derive the output of the unit activation which is done by transforming the weighted sum of the bias and weights using an appropriate activation function with σ used to denote the final activation function and can be expressed as follows,

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \tag{6}$$

The activation function shown in equation 6 is of the logistic sigmoid form.

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{7}$$

This network model is known as a feed-forward network because of the manner in which information is transferred through the system. The information only travels forward through the neural network from input through to the hidden layers and lastly out of the output node. Equation 7 shows the entire feed-forward propagation of information through the NN. Which comes to a conclusions that the neural network is a nonlinear function from a set of input variables to output variables which is controlled by adjusting the weight vector.

2.1.2 Back Propagation

This segment of the report will cover methods used to determine the error function of a given input vector training set x_n, where n is a integer value such that n = 1, 2, ... N and a known target value denoted by t_n. These two vectors are used to compute the error function which is described by,

$$E(\mathrm{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}\left(\mathbf{x}_n, \mathbf{w}\right) - \mathbf{t}_n\|^2 . \tag{8}$$

The Error function is the sum-of-squares of the difference between the expected input value and target value summed across all data points N. It is regraded that the neural network is the minimization of the error function which is the same as the maximization of the likelihood which is shown in equation 9.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \{y\left(\mathbf{x}_n, \mathbf{w}\right) - t_n\}^2 \tag{9}$$

Although the measure of the value of the error function for a neural network of this caliber is strictly on finding a value for the wight vector such that E(w) is a minimum. The process of achieving this results is done so theoretically by stepping through weight space in incremental proportions denoted by w+δw which is a proportional shift of δw. The error function is a smooth continuous function which is generally the results of using a soft activation function. The procedure used to local the global minimum of the error function is by find the stationary point of E(w) such that,

$$\nabla E(\mathbf{w}) = 0 \tag{10}$$

However, the probability of finding the global minimum of the error function is unlikely and is something that will generally go unknown. It is alright to use the local minima with the smallest value found in the weight space. A common practice is to pick a random initial position for the weight vector and incrementally step through it in weight space by,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}, \tag{11}$$

With the steps denoted by tau. There have been many proposed methods of achieving this result, with many of the functions requiring the values to perform the algorithm. It has thus become suitable to update the weight according to the error function after every iteration. For this experiment the gradient descent optimization method was performed which is regarded as the simplest approach when using gradient information and has been implemented by,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E\left(\mathbf{w}^{(\tau)}\right). \tag{12}$$

With the learning rate of the algorithm denoted by η where $\eta > 0$. The process is done by evaluating the prediction and updating the weight vector accordingly. The evaluation of the

weights is only possible once the set has been predicted completely with this method known as a **batch** method. With each iteration of the error function moving the gradient to the point with the most decrease in value which is marked as gradient decent method.

The evaluation of the performance of the system is however, incomplete and will require that the same process be re-run with different random starting points. The alternative approach is to update the weight vectors after every iteration and is express by the following equation,

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{13}$$

This method is better used for large datasets and is known as a one-point algorithm. With the final calculation of the error function defined by,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n \left(\mathbf{w}^{(\tau)} \right) \tag{14}$$

Back propagation is going from the output through the hidden layers back to the input and adjusting the weights along the way. By taking the derivative with respect to the weights, with E(w) being a scalar value which is shown by,

$$\frac{\partial E}{\partial w} = \sum_{j} \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w}. \tag{15}$$

By using the chain rule the input derived towards the error function is computed to back propagate through the algorithm to adjust the weights incrementally.

2.2 Discussion and motivation neural network and backprop

2.2.1 Activation function(s)

The relevance of using activation functions is to scale the data to be within a range, as the data back and forward propagates, it updates the values of the parameters and with each update the values could tend towards infinity which is undesirable. It is not possible to fit a linear function to a non linear function, all that can be done is approximate the value of the non-linear function.

The activation function is applied after each neuron in the hidden layers and output layers, however the output layer has a different activation function applied to it than the hidden

layers which will be discussed later. For the application of classifying several non-linear functions, the use of a Sigmoid activation function was considered to be used in the hidden layer and is achieved by using the following equations,

$$y = \frac{1}{1 + e^{-x}} \tag{16}$$

where y is the output from the activation function. The activation function on a linear space has been plotted to show the outcome from such a function.

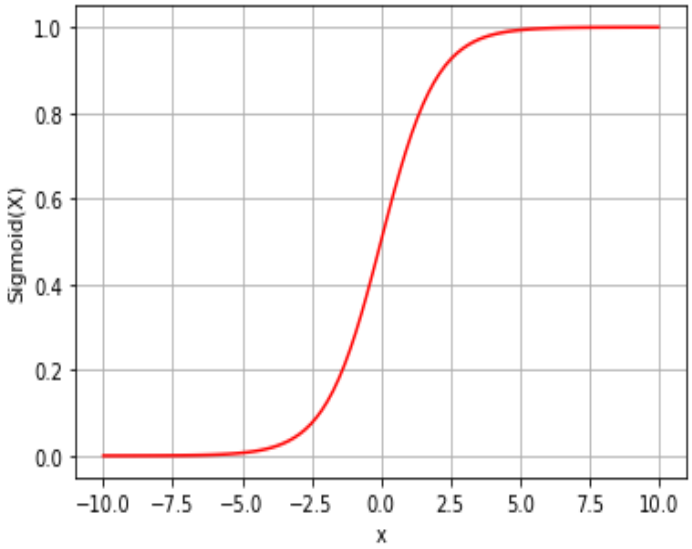


Figure 1. Sigmoid activation function

With the response of this activation function the value will be either zero when negative and increase with a steep gradient until its a positive value where it will saturate to one as it approaches infinity. The choice of using alternative activation come when considering the choice of optimizer. Given that the Sigmoid activation function resembles that of a step function it outputs a value of either one or negative one. These marginal values makes it highly difficult to distinguish for the optimizer how off the actual values is. This propelled the reason to use a ReLu Activation function and is expressed in the following equation,

$$f(x) = \max(0, x) = \begin{cases} x_i \text{ if } x_i > 0 \\ 0 \text{ if } x_i < 0 \end{cases} \tag{17}$$

With the function outputting a zero if the input value is less than zero and scaled according to y = x if the input value is greater than one.

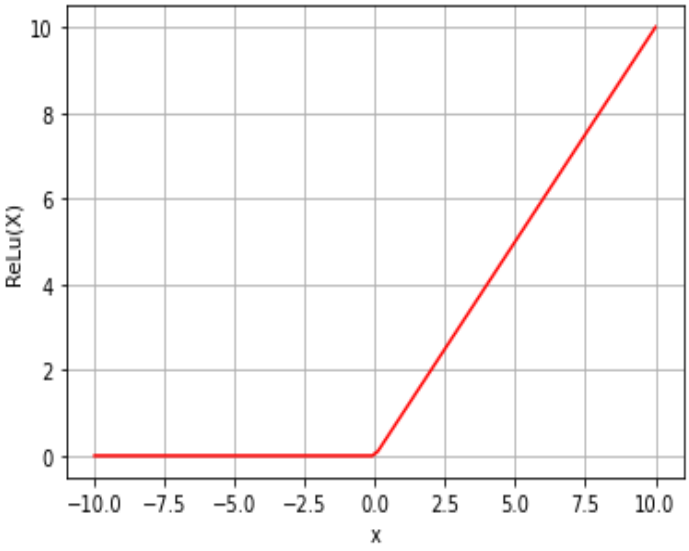


Figure 2. ReLU activation function

Figure 2 is a plot of the ReLu activation function using a linear space to show the output which is expressed by equation 17. The two primary reasons of using a ReLu activation function is because its granular (scale-able) and fast since it is not as computationally intensive as other activation functions such as the Sigmoid activation function.

The **Softmax activation function** applied to the final output of the hidden layers is used to normalized the data across the algorithm with a more general uniform distribution used for decision making. The process of applying Softmax activation to the output is by applying an exponential function with the input vector as the variable and then normalized across the hidden layer which will give a probability distribution ranging from zero to one and summing up to one showing the most probable answer for the FFNN. With the equation used to implement the activation function expressed by,

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{18}$$

2.2.2 Initialisation

For the initialization there are two parameters to initialize, being the weights and the bias. Its important to consider the input weights to be initialized in a range greater than or less than zero. Zero has a special property where anything multiplied by zero will be zero and will propagates through the entire training will meaningless results. That's why its important to

initialize the weights with non-zero values. Generally speaking the weight parameters are a range bound random function of the input vector. It carries the shape of the input vector by the number of neurons in the hidden layer. Thus the bias and weights will be initialized as **bias[x] = random** and **weight[x] = random** with the value of random ranging from zero to one. Additional parameters such as cost and lost will also be declared however, will be initialized to zero.

2.2.3 Learning rate

The learning rate is a highly important hyper-parameter used to decide the best possible result when training your algorithm. This is due to the fact of having too small of a training size, lets say one-point. will result in the algorithm attempting to update the weights on every iteration and will notice that the computational intensive process could achieve better results if it were to take in larger sample sizes. By increasing the batch size the algorithm is able to make better assumptions and improvements on the parameters as to not over fit or under fit. If all the samples were used to adjust the parameters of the weights, you will find that the algorithm over-fitted and will likely damage your parameters.

The learning rate controls the frequency at which the model changes in response to the error estimations made. By using a smaller learning rate the algorithm will require more epochs to steadily saturate into its maximum accuracy. For this assignment the learning rate will be a minimum of **learning rate = 0.01** and a maximum of **learning rate = 0.1**. Due to the FFNN updating after every cycle, the amount of epochs required to make a meaningful update is quite large in comparison to batching. Thus the maximum value is preferred for this experiment, if not the amount of epochs required could be as much as 10 times more.

2.2.4 Stopping criteria and data

Depending on the algorithm implemented, the stop criteria will change however the standard consideration to the stopping criteria is once the accuracy of the algorithm on the training dataset starts to saturate and flatten out, the training algorithm has begun to over fit the data and thus can be stopped.

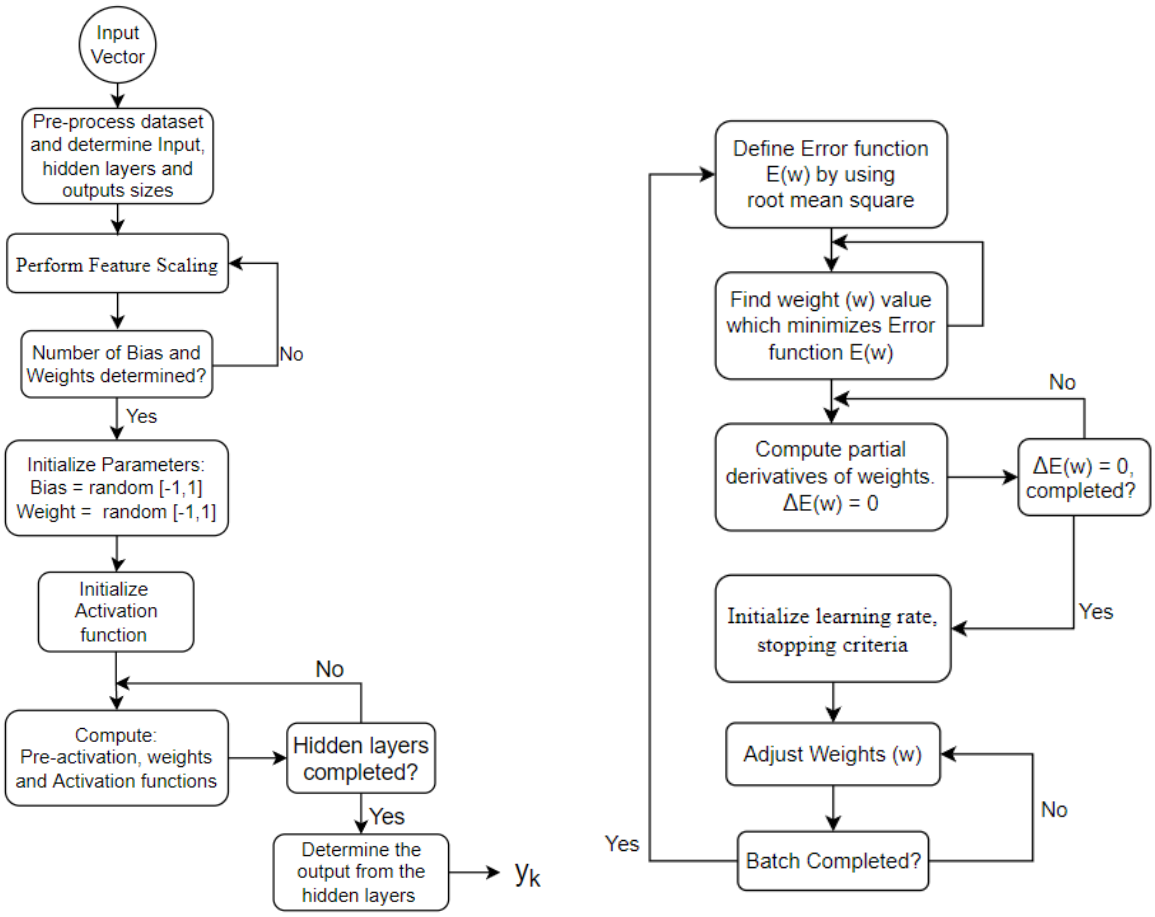
2.2.5 Feature scaling

The data that is fed into the FFNN is different with regards to the input length and shape. However, does not vary in the type of data being presented as it will either be in the form of a one hot encoded value or floating point. Many techniques have been used to prepare the different data sets in a dynamic manner by normalizing, and standardization. The idea behind normalizing is to scale the input vectors values between zero and one.

2.3 Algorithmic description and implementation

The process of training an algorithm on the feed-forward neural network is divided into two steps, The first is the feed-forward algorithm which starts with pre-processing the raw input vector by transforming its shape and size to match the input of the FFNN. Once the correct shape and size has been determined, it is important to separate the input data from the output data which has been stored in the same text document descried in Section 3. The input vectors must pass through some sort of feature scaling given that the system is dynamic in the sense that it is able to be trained on different datasets. Its not possible for the FFNN to be hard-coded for this experiment and thus the system must determine the correct number of weights and biases before any initialization.

The initialization for the FFNN only considers the input vector shape and size, output vector size, number of hidden layers, bias and weight vectors. The initial parameters for the bias and weight vectors are normally scaled according to the input vector however, for this experiment they were assigned random values between the values of negative one and one. The hidden layers are then determined by summing the values of the input times the weight plus the bias at each neuron in each hidden layer that is then passed to a activation function which for this assignment has been chosen as the ReLU which is discussed in Section 2.2.1. Once all the hidden layers have been determined a second, Softmax activation function can then be applied to the neuron which will then produce an output vector denoted by y_k which is expressed in equation 7. This concludes the forward propagation of the FFNN and can be see by Figure 3a



(a) FFNN Algorithmic description **(b)** Back propagation Algorithmic description

Figure 3. Classifier algorithmic development

The gradient decent based back-propagation algorithm has been used for training. This approached used to determine the error function E(w) is expressed in equation 9 in Section 2.1.2 which is known and the root mean square error function. The goal is to minimize the loss function and ultimately improve the accuracy of the algorithms predictions. This is done is many ways however, for this experiment the derivative of the error function is determine and equated to zero. another important parameter to consider when making the adjustments through the system are the learning rate which determines the step sizes of the adjustments made to the weights and is discussed in Section 2.2.3. Once the algorithm has completed its back propagation, it then takes in another input vector and repeats the entire process.

2.3.1 Implementation

The process used to code the FFNN is described in this section and is divided into several algorithms to have a better overview of the topic.

Algorithm 1 Pre-Processing

Require: $f = open('xxx1.dt', 'r')$ ▷ Open training set
Require: $in\_list = f.read()$
Require: $f.close()$
for x in range(0, len(in_list), n+2): **do** ▷ Generate input into list
    partition = in_list[x: n+x]
    if $len(partition) \leq n$: **then**
        $partition = partition +$
        for None y in range(n-len(partition)): **do**
        end for
    end if
    Yield partition ▷ Yield input list
end for
for x in range(0, len(in_list), n+2): **do** ▷ Generate Output into list
    partition = in_list[x+8: n+x+2] ▷ Structure output size
    if $len(partition) \leq 2$: **then**
        $partition = partition +$
        for None y in range(n-len(partition)): **do**
        end for
    end if
    Yield partition ▷ Yield output list

The initial step is to extract the raw data from a text file and store it in a variable list. This list contains both the input vector and target output results. However the data types may vary from dataset to dataset. Algorithm 1 shows the steps used to perform this process of extracting the raw data and separating the input variables from the output.

Algorithm 2 Neural Network Initialization

while $N \neq 0$ **do**
    weight[N] = 0.01 * np.random.randn(n_inputs, Hidden_layer) ▷ weights
    bias[N] = np.zeros((1, Hidden_layer)) ▷ bias
    output = np.dot(inputs[N], weights[N]) + bias[N] ▷ dot operation

Once the data as been extrapolated and pre-processed, the second phase can start by initializing all the necessary parameters. The length of the hidden layers, size of the input, shape

of the output are pre-determined by the raw dataset and are omitted from the algorithm however calculated. The Bias and initial weights are assigned non-zero values by using the numpy library function. The next step is to find the output from each neuron and is simply done by computing the dot product between the input vector, the weight and the addition of the bias value. This process is shown in the algorithm above, Algorithm 2.

Algorithm 3 ReLu activation function

```
def ReLu(inputs):                                   ▷ ReLu Function
 return output = np.maximum(0, inputs)
```

The ReLu activation function algorithm is shown above in Algorithm 3. Where the function simply returns the maximum value of the input.

Algorithm 4 Softmax activation function

```
                                    ▷ Subtract the matrix by the largest value in the row
def forward(inputs):
 numerator = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))     ▷ Row operation
 softmax = numerator / np.sum(numerator, axis=1, keepdims=True)          ▷ Transpose
 return softmax
```

The Softmax activation function was used on the output of the hidden layers and was done so by implementing the algorithm shown above by Algorithm 4. The process was to determine the numerator which was done by taking the exponential value of the input subtracted by the highest value in the row which allowed the range of the exponential value to be bounded between negative infinity and zero. It was highly important to keep the shape of the output, thus the signal was therefore transposed. The denominator was simply the summation of the the input values. The final process was to divide the numerator by the denominator and return the output.

The process of back propagating through the entire algorithm is broken done into several smaller steps. TO successfully update the hyperparameters, the error derived towards the weight given by $\frac{\partial E}{\partial W}$, the error derived towards the bias given by $\frac{\partial E}{\partial B}$ and the derivative derived towards the input given by $\frac{\partial E}{\partial X}$ need to be calculated. This is done by making use of the chain rule, layer by layer.

Algorithm 5 Back propagation and Gradient decent

Require: $\frac{\partial E}{\partial w}$

Require: *Dictionary* ▷ To store derivative values

Determine length of Hidden layers

dz = In.T - Y.T ▷ Output minus target

$dw = dZ.dot(Dictionary[\text{"}A\text{"} + str(self.neuron_length)].T)$

$db = np.sum(dz, axis = 1, keepdims = True)$

for x in range(0, $len(neuron\_length)$): **do**

    Determine dz ▷ Compute Hyperparameters

    Determine dw ▷ Store values in Dictionary

    Determine db

end for ▷ End =0

The pseudo code used to implemented the back propagation topology for the FFNN is shown in Algorithm 5. With the idea of calculating the difference between the target value and output form the FFNN used as the error function. The algorithm will iterate through all the hidden layers and determine the new values of the output, weights and bias. Once this iteration is done then dictionary values will be returned containing