Testing and Implementation of Perceptron

Tautvydas Lisas

Dt0021a/4 DSP and Machine Learning

David Doran

Table of Contents

Table of Figures

1 Introduction

In this report the design, implementation and testing of a simple neural network called the perceptron. The Iris dataset will be used to validate and find the model limitations, for both - the implemented design and a built-in design in Matlab. The outline of the procedure taken, including some theoretical part will be discussed. This will allow the reader to obtain a better understanding of how a bare minimum neural network operates. These methods could furthermore be used to build a better understanding of neural networks.

The validation process was used to determine the limits of the model so datasets could be chosen according. Also, by testing ideas for further improvement could be acknowledged.

2 Background

The perceptron was the first machine learning algorithm. Originally proposed by Frank Rosenblatt in the late 50's, and later refined and carefully analysed by Minsky and Papert in 1969 [1]. The perceptron is not the sigmoid neuron that is used today in artificial neural networks, but a general computational model that takes and input, aggregates it and returns a 1 or a 0. The returned value depends on the calculated value, if the value is higher than the threshold it returns a logical 1, but if it is lower than the threshold it returns a logical 0.

The single layer perceptron is typically only used for binary classification problems that have a linear decision boundary.

The perceptron consists of 4 parts:

1. Input Layer – Contains the initial data being inputted into the network.
2. Weights – Weights control the strength of the connection between neurons, they decide how much influence the input will have on the output.
3. Bias – Are a constant which are an addition to the input which will usually have the value of 1 or -1. They have an outgoing connection to their own weights, and they guarantee that even when all the inputs are zeros there will still be an activation in the neuron.
4. Net sum – The sum of weights multiplied by input plus the bias.
5. Activation Function (Sigmoid) – Decides if the neuron should fire or not. It rounds the net sum value to either 0 or 1. A sigmoid function is used compress the output of the learning equation closest to either 0 or 1. The delta rule is then used to subtract calculated values from the targets, if there is no difference between them then the weights converge and are not updated anymore.

$$Sigmoid\ function\ (x) = \frac{1}{1 + e^{-x}}$$

The initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small a random number.
2. Multiply weights by input and sum them up.
3. Compare result against the threshold to compute the output (1 or 0).
4. Update the weights.
5. Repeat.

3 Implementation

A simple single layer neural network known as a perceptron is implemented and limitations tested on iris-flower dataset. The perceptron has the ability to differentiate between linearly separated data. It does this by calculating the weights which will enable a single straight line to separate the two datasets.

Preparation of Data

The simple perceptron was first of all programmed to predict the output of an AND gate. After, a more difficult data set was used to confirm if the perceptron was operating

correctly. The second dataset was created visually to assure a linear boundary was present.

```
% AND gate data set
input = [0 0; 0 1; 1 0; 1 1];
desired_out = [0;1;1;1];

% manually generated dataset
dataset_train = [2.7810836 2.550537003 0;
            1.465489372 2.362125076 0;
            3.396561688 4.400293529 0;
            1.38807019 1.850220317 0;
            3.06407232 3.005305973 0;
            7.627531214 2.759262235 1;
            5.332441248 2.088626775 1;
            6.922596716 1.77106367 1;
            8.675418651 -0.242068655 1;
            7.673756466 3.508563011 1];

dataset_test = [8.673756466 4.508563011 1;
               2.465489372 3.362125076 0;
            4.396561688 5.400293529 0;
            2.38807019 2.850220317 0;
            4.06407232 4.005305973 0;
            8.627531214 3.759262235 1;
            6.332441248 3.088626775 1;
            7.922596716 2.77106367 1;
            9.675418651 -1.242068655 1;
            3.7810836 3.550537003 0];
```

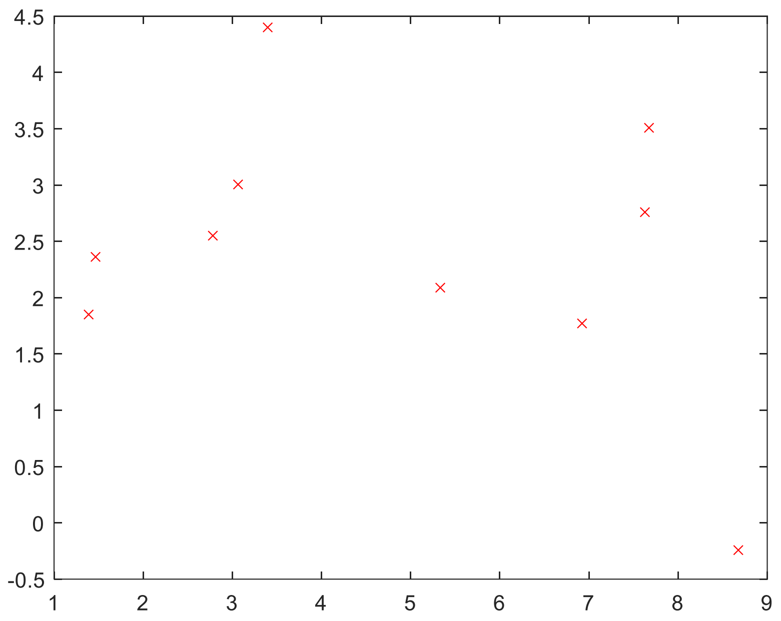


Figure 1: Manually generated dataset plot.

3.1 Implementation of Perceptron

To create the perceptron first steps were to select the bias, learning rate and to create a matrix for the weights. The bias is used to adjust the output along with the weighted sum of the inputs to the neuron. The bias acts like a constant which helps the model to fit the given data by moving the line up and down thus fitting the prediction with the data better. The bias was selected as -1, and the learning rate as 0.7. Different learning rates can be chosen to give a faster or slower learning rate, but the accuracy will be affected if the learning rate is too high. The weights were generated randomly in a 3 by 1 matrix.

The perceptron equation is then used to calculate the output y. The perceptron equation uses the bias multiplied by the first weight, added to the first input column which is multiplied by the second weight, added to the second input column which is multiplied by the third weight. The input columns are iterated by each row. The Sigmoid function is then used to adjust the output to either a 0 or a 1.

$$y = bias * (weights) + input(j,1) * weights(2,1) + input(j,2) * weights(3,1)$$

$$out(j) = \frac{1}{1 + \exp{(-y)}} = 0 \; or \; 1$$

The delta function is used to determine the difference between the output just calculated and the desired output. For example, if an AND gate is used, and the two inputs are [0 0] you would expect the output to be 0. Further, if the two input were [0 1], [1 0] or [1 1] you would expect the output to be 1. If the output is not what is expected, then delta is a non-zero value.

The weights are then adjusted dependent on the previous weight value, added to the sum of the coeff, bias, input and delta. Each weight is calculated respectfully to the input and stored. If delta is 0, or the expected output is as the output from the sigmoid function, then the weights are not updated anymore, and the last value is stored.

$$w_i = w_i + x_i(delta) \times bias$$

```
% activate bias and coefficient
bias = -1;
coeff = 0.7;

weights = -1*2.*rand(3,1);
iterations = 1000;
for i = 1:iterations
      out = zeros(10,1);
      for j = 1:length(input)
            y = bias*weights(1,1)+...
            input(j,1)*weights(2,1)+input(j,2)*weights(3,1);
            out(j) = 1/(1+exp(-y));
            delta = desired_out(j)-out(j);
            weights(1,1) = weights(1,1)+coeff*bias*delta;
            weights(2,1) = weights(2,1)+coeff*input(j,1)*delta;
            weights(3,1) = weights(3,1)+coeff*input(j,2)*delta;
      end
end
```

4 Testing/Validation

The iris data set was used with 3 different flowers – Setosa, Verginica and Versicolor. Measurements of sepal length and petal length were extracted from the dataset. A function was also created for non-seen data to validate the training function. Accuracy of the perceptron was 100% for linearly divided data. Limitations were also explored by using non-linearly divided data.

4.1 Preparation of Data

4.1.1 Linearly Separated Data

Setosa and Versicolor datasets were selected as they are linearly separated. The training data contained 60 datapoints in total, 30 of Setosa and 30 of Versicolor. The testing data contained a total of 40 datapoints, 20 of Setosa and 20 of Versicolor. Some of the points overlap so when you count the plot points, it is not noticeable.

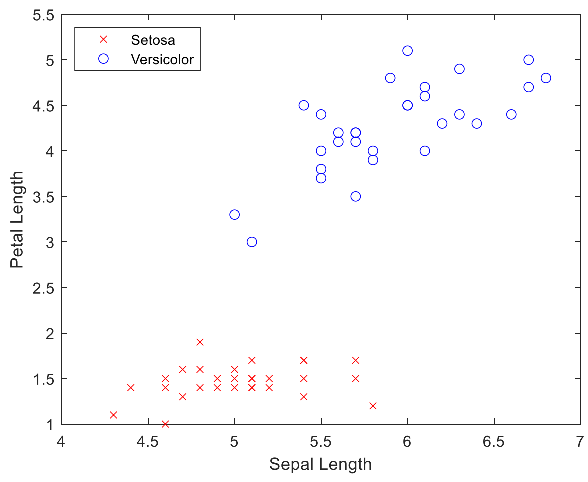


Figure 2:Linearly separated training data

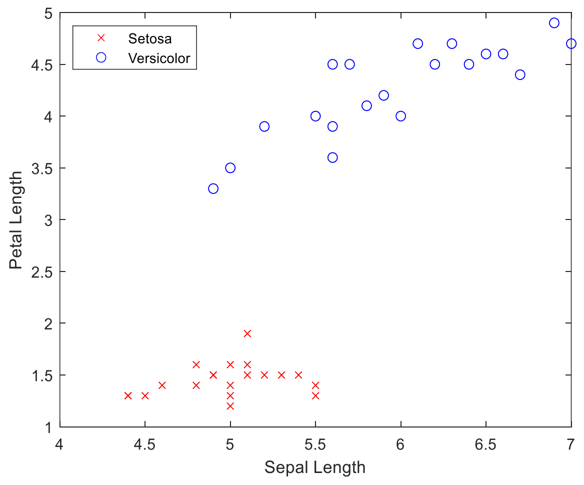


Figure 3: Linearly separated test data plotted.

.

4.1.2 Non-Linearly Separated Data

Versicolor and Virginica flower datasets are not linearly separated. Overall, the training data contains 60 points, 30 points of Versicolor and 30 of Virginica. Testing data contains 20 points of each, with a total of 40 data points.

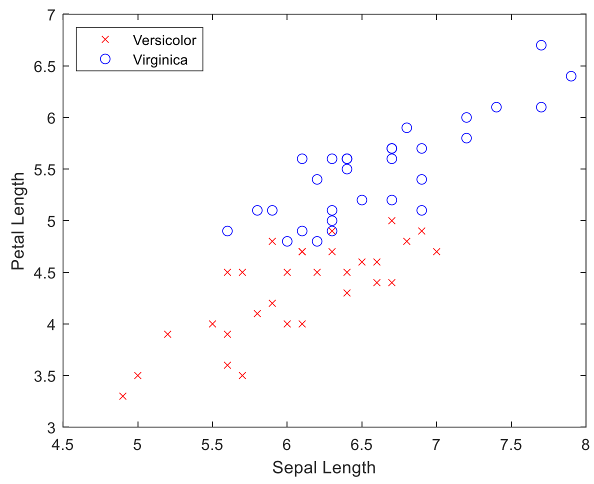


Figure 4: Non-linearly separated training data.

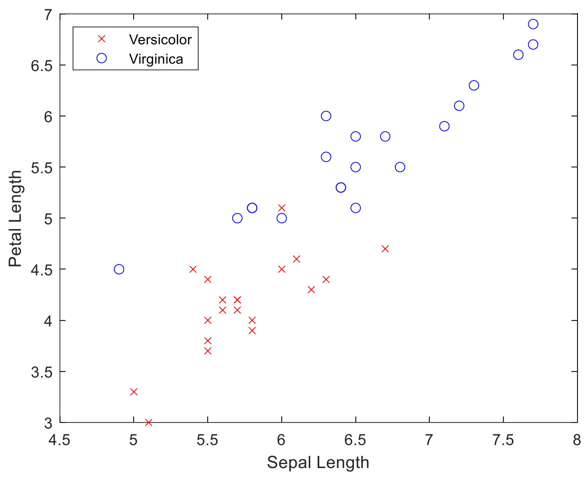


Figure 5: Non-linearly separated testing data.

```
% Non-linearly seperated data
data_train_fail = [data(1,51:80),data(1,121:150);data(3,51:80),data(3,121:150)]';
targets_fail = [T(2,51:80),T(2,121:150)]';

data_test_fail = [data(1,81:120);data(3,81:120)]';
test_targets_fail = T(2,81:120)';

plot(data_train_fail(1:30,1), data_train_fail(1:30,2), 'rx')
hold on
plot(data_train_fail(31:60,1),data_train_fail(31:60,2), 'bo')
xlabel('Sepal Length')
ylabel('Petal Length')
legend('Versicolor','Virginica','Location','northwest')
```

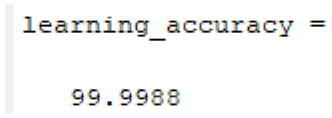
4.1.3 Testing of Neural Network

First of all, the perceptron was trained using the training dataset previously selected. The weights are then stored and used in the validation function to validate the perceptron using the testing dataset. The accuracy is compared by subtracting the targets and output result of the training function. The linearly separated data has 99.9988% accuracy as seen below.

To validate the perceptron, validation function is executed using the testing data. The accuracy is then calculated using the output. The validation of the perceptron, while using linearly separated data is 100%.

```
% train perceptron function and calculate the weights.
[weights,out] = perceptron_train(data_train, targets_train, bias, coeff);

% test accuracy of the perceptron learning
learning_accuracy = 100 - abs(((sum(abs(out - targets_train))/...
    length(out))*100))
```



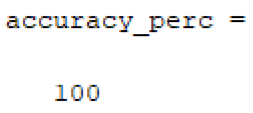
```
learning_accuracy =

   99.9988
```

Figure 6: Output of learning accuracy.

```
% validate perceptron accuracy
out_validate = p_validate(data_test, weights, bias);

accuracy_perc = 100 - abs(((sum(abs(out_validate - test_targets))/...
    length(out_validate))*100))
```



```
accuracy_perc =

   100
```

Figure 7: Accuracy of test data.

To test perceptron limitations, it is trained and validated using the non-linearly separated dataset. Surprisingly, the perceptron training accuracy was 96.6% and validation accuracy was 95%. This is exceptionally high, if you look at figure 7, by eye, you can estimate that there is at least 2 or 3 points that are overlapping. However, the testing data accuracy validates figure 8, as you can see visually there is only 1 point that overlaps. This training method uses 10,000 epochs. If it is reduced to 1,000, the testing accuracy decreases to 85%.

To test the validation furthermore 3 points were changed on the test data. It was found that the accuracy decreased, and the points that were changed were incorrectly predicted. The accuracy dropped to 87% at 10,000 epochs. If the epochs were reduced to 1,000, the validation accuracy drops to 80%.

Also, by using manipulated data from figure 14 to train the perceptron we can identify limitations in this design. If you train the node using that data, it will output learning accuracy of 96.57%. Even though, the training accuracy is very high, if you input testing data into the validation function, the accuracy drops to 50%. This indicates that the perceptron is not operating correctly.

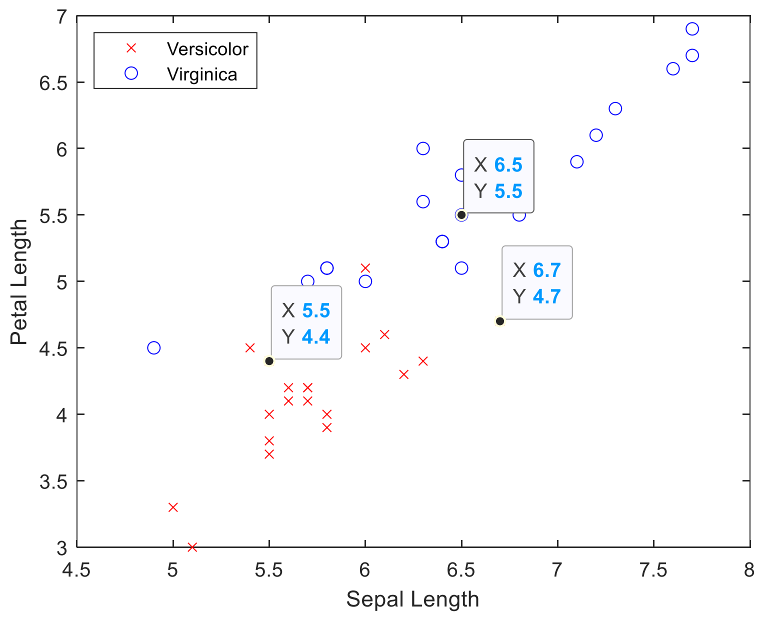


Figure 8: Target data changed

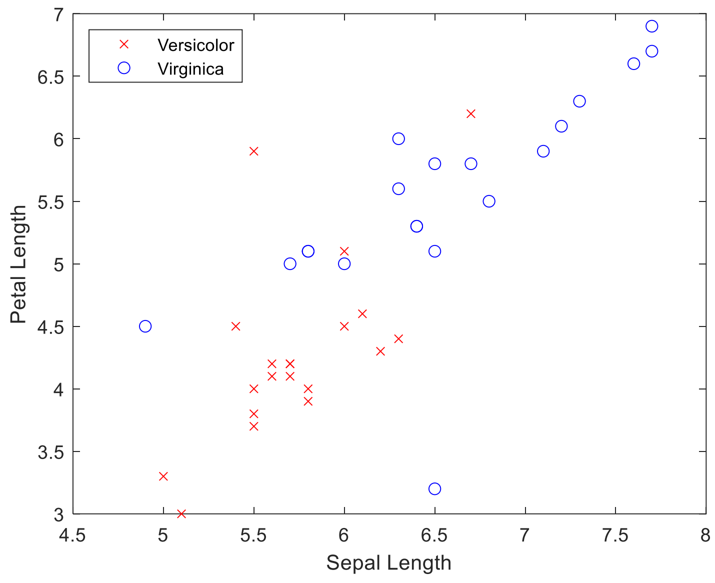
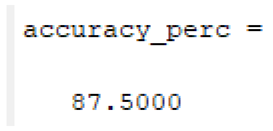


Figure 9: New test data.



```
accuracy_perc =

    87.5000
```

Figure 10: Accuracy of changed data.

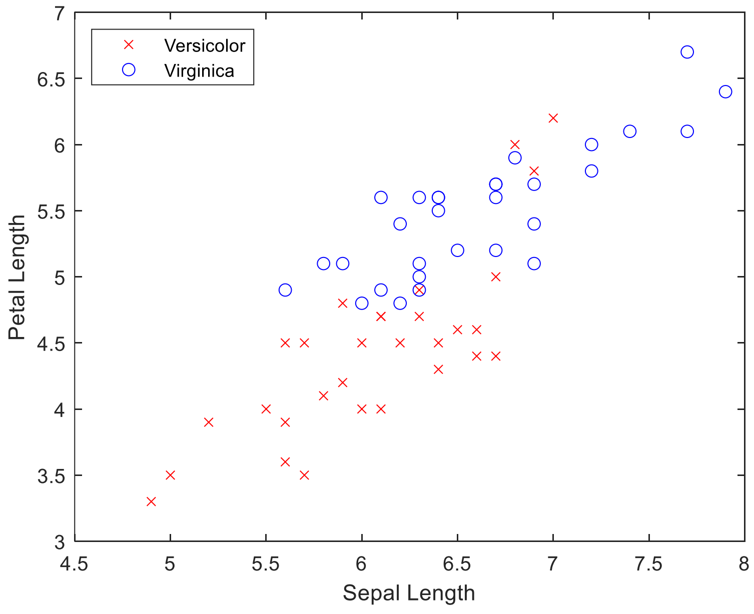


Figure 11: Matlab perceptron limitations.

4.2 Comparison to Matlab Perceptron

The training and test data were also used to evaluate the built-in perceptron in Matlab, the results could then be used to compare the two designs. As predicted the linearly divided data had training and testing data results of 100%. However, when using the non-linearly split data for Versicolor and Virginica flower, the performance of the perceptron drops significantly. If you check figure 15, you can see that 10 data points of Virginica are classified as Versicolor, which coincides with the confusion matrix where we get 10 wrong classifications. Testing the unseen data for non-linearly separated dataset gave an accuracy of 87.5%. By using the modified testing data from figure 12, accuracy drops further down to 82.5%. Furthermore, if data in figure 14 is used to train the perceptron, the training accuracy drops to 50%, meaning that the perceptron is no longer differentiating between the two data types and classifying all the points as one type of data. The testing was done at 1,000 epochs.

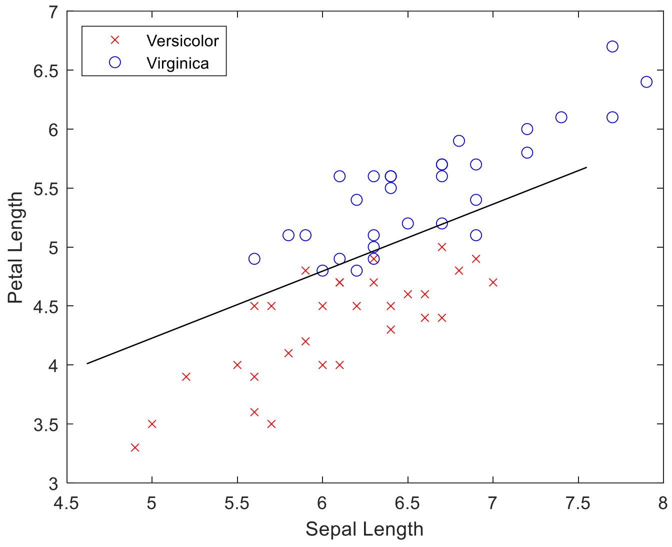


Figure 12: Training data set.

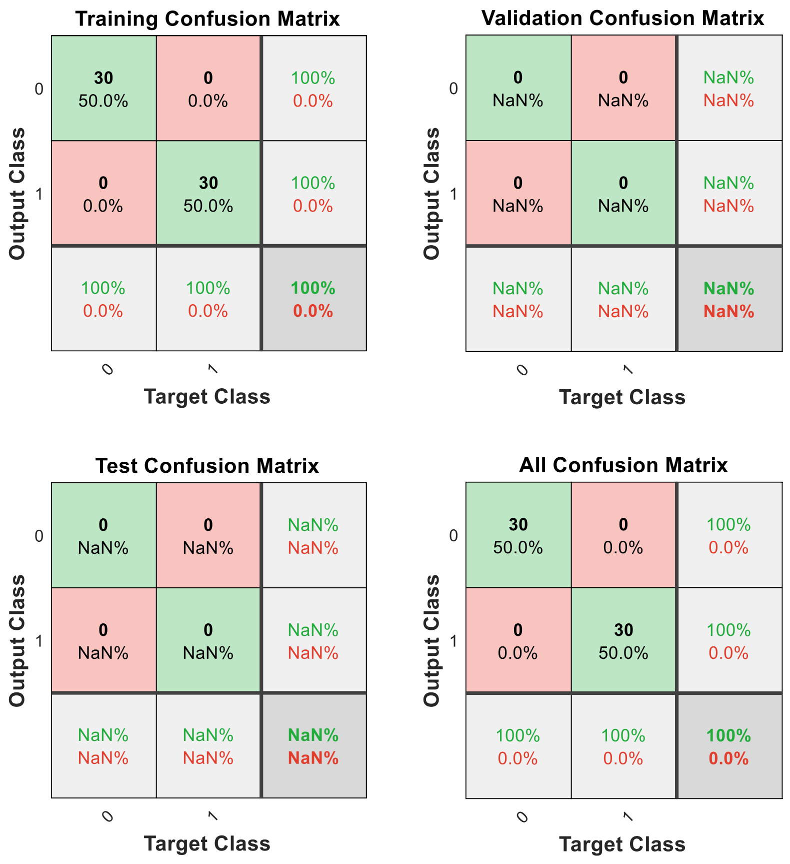


Figure 13: Training confusion Matrix (linearly divided data).

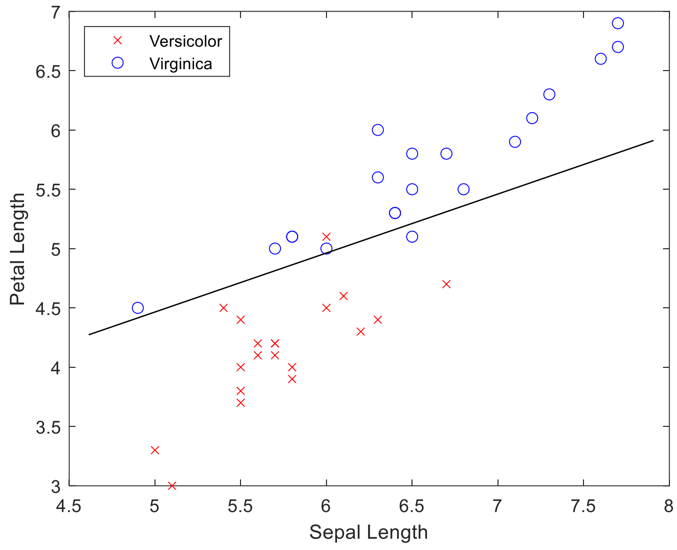


Figure 14: Testing dataset.

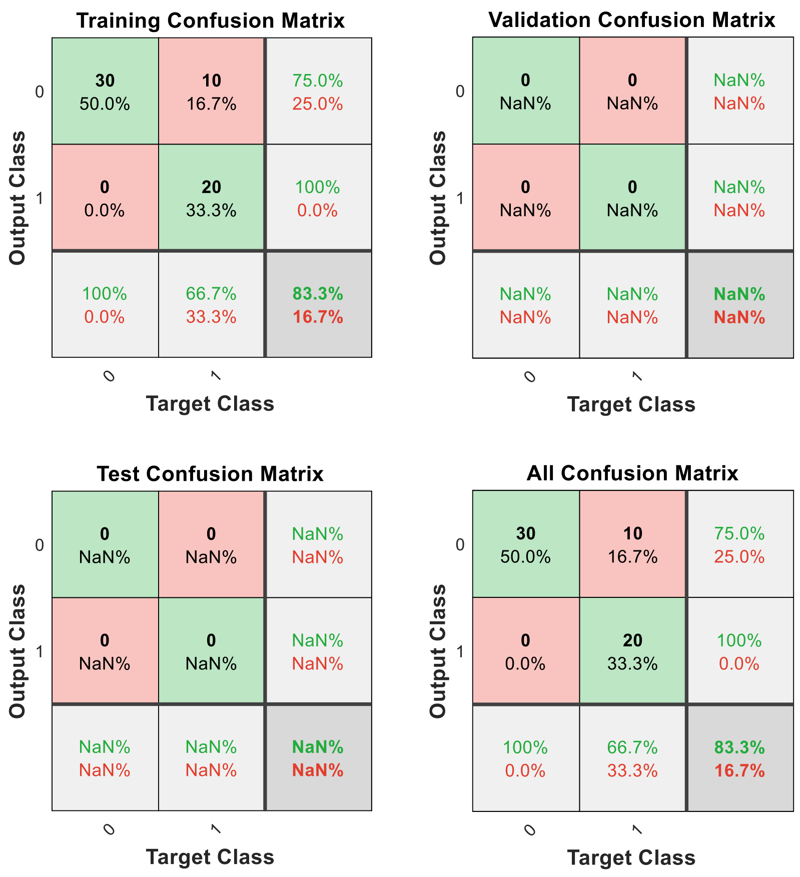


Figure 15: Training non-linearly separated data.

4.3 Conclusions & Future Work

In this report the design of a perceptron was designed, implemented, and tested using the Iris flower dataset. First of all, theory of how the perceptron works was discussed to familiarise with the model. A preliminary dataset was then created for design testing purposes. After the perceptron was implemented, a more complex dataset was used to test and validate the perceptron design. The model was then compared using a built-in perceptron in Matlab. These two models were tested with linearly and non-linearly separated data. The testing and training data were manipulated furthermore to find model limitations.

During testing it was found that both of the models operated with training and validation accuracy of 100% for the linearly separated data. When using the non-linearly separated data, the perceptron accuracy started to decrease. For the non-manipulated testing and training data, both designs operated at around 87.5% accuracy at 1,000 epochs. For the manipulated testing data, the accuracy decreased to around 80%-82.5% for 1,000 epochs. When the manipulated training data was used the training accuracy dropped to 50%, which suggest the perceptron cannot differentiate between data points and the model limitations have been reached. Furthermore, it was found that increasing epochs to 10,000 increased the training and validation accuracy by about 10%.

This report validated the perceptron limitations and confirmed that it is only useful with data that is linearly separated or if it is not linearly separated, the points that are not linearly separated must not be too far into the cluster. If a different class datapoint is in the opposite cluster, the perceptron cannot distinguish between the data sets and only classifies one data set.

Further testing could be done by acquiring different datasets, but it is not necessary as the limitations were tested and validated using two models. Other improvements could be made by increasing the number of nodes. By introducing another node into the model, it would allow a more flexible class separation.

5 References

[1] ROSENBLATT, F. (1958). The perceptron. A probabilistic model for information storage and organisation in the brain. Psychol. Rev. 65, 386.

[2] Du, K.-L & Swamy, M.N.s. (2014). Perceptrons. 10.1007/978-1-4471-5571-3_3.

[3] E. Alpaydin, *Introduction to machine learning*. Cambridge, Mass.: MIT Press, 2010.

6 Appendix

```
% Linearly separated data
% Training data
[data,T] = iris_dataset;

% select Sepal and Petal length for Setosa and Versicolor dataset to train
data_train = [data(1,1:30),data(1,71:100);data(3,1:30),data(3,71:100)]';
targets = [T(1,1:30),T(1,71:100)]';

% Testing data
data_test = [data(1,31:70);data(3,31:70)]';
test_targets = T(1,31:70)';

plot(data_train(1:30,1),data_train(1:30,2),'rx')
hold on
plot(data_train(31:60,1),data_train(31:60,2),'bo')
xlabel('Sepal Length')
ylabel('Petal Length')
legend('Setosa','Versicolor','Location','northwest')

plot(data_test(1:20,1),data_test(1:20,2),'rx')
hold on
plot(data_test(21:40,1),data_test(21:40,2),'bo')
xlabel('Sepal Length')
ylabel('Petal Length')
legend('Setosa','Versicolor','Location','northwest')
```

Figure 16: Dataset code.

```
% train perceptron function and calculate the weights.
[weights,out] = perceptron_train(data_train_fail_2, targets_fail,
bias, coeff);

% test accuracy of the perceptron learning
learning_accuracy = 100 - abs(((sum(abs(out - targets_fail))/...
    length(out))*100))

% validate perceptron accuracy
out_validate = p_validate(data_test_fail, weights, bias);

accuracy_perc = 100 - abs(((sum(abs(out_validate -
test_targets_fail))/...
    length(out_validate))*100))

function [weights,out] = perceptron_train(input, desired_out, bias,
coeff)
    weights = -1*2.*rand(3,1);
    iterations = 100000;
    for i = 1:iterations
         out = zeros(10,1);
         for j = 1:length(input)
              y = bias*weights(1,1)+...
                   input(j,1)*weights(2,1)+input(j,2)*weights(3,1);
              out(j) = 1/(1+exp(-y));
              delta = desired_out(j)-out(j);
              weights(1,1) = weights(1,1)+coeff*bias*delta;
              weights(2,1) = weights(2,1)+coeff*input(j,1)*delta;
              weights(3,1) = weights(3,1)+coeff*input(j,2)*delta;
         end
    end
end


%% Validate perceptron
function out_round = p_validate(input_test, weights, bias)
    iterations = 100;
    for i = 1:iterations
        out_validate = zeros(20,1);
         for j = 1:length(input_test)
              y = bias*weights(1,1)+...

input_test(j,1)*weights(2,1)+input_test(j,2)*weights(3,1);
               out_validate(j) = 1/(1+exp(-y));
         end
        out_round = round(out_validate);
    end
end
```

Figure 17: Final code - without dataset.