Artificial Neural Networks

Computational And Cognitive Neuroscience

Practical 2

18/12/2020

Rosamelia Carioni Porras ***(Student ID: 6247275)***
Bianca Caissotti di Chiusano **(Student ID: i*6245461*)**

Table Of Contents

1.0 Introduction

Consider complex interconnections of neurons that allow the transmission of signals throughout a person's body. This biological neural network can be representent and simulated computationally through the use of Artificial neural networks (ANNs). One of the advantages of ANNs is that, even though it makes use of very simple computational operations, it can be applied to a wide range of complex problems (Kwon, 2011). They possess the ability to learn from an experimental set of data, without having any knowledge of the system. This type of learning is known as "Supervised Learning" (Section 1.2). Moreover, according to Seoyun Kwon: artificial neural networks have become useful tools for modelling and can be combined with other "artificial intelligence tools". An example of this could be, for example, its combination with Genetic Algorithms (GA) (Kwon, 2011).

This report will explain and outline the implementation of a simple "perceptron" (further explained in **Section 1.1**) whilst analysing and answering the report's research question and subquestions, described in **Section 1.3**.

1.1 The Perceptron

One of the first neural computation models is called "The Perceptron", and dates back to the late 1950s. This model can be compared to a biological neuron, but in more scientific terms, it is a single-layer, feed-forward neural network. By feed-forward network, it is meant that it only has connections in one direction, and by single-layer, that its inputs are connected directly to its outputs.

The construction of this neural network unit consists of: assembling an experimental set of data, then creating and finally training the network object. Throughout the training, the outputs of several nodes interconnect with the adjustable weighted inputs of the receiving node. These, in turn, will generate an output as a function with respect to the weighted inputs. Each node adopts two functions in order to achieve their corresponding output. These functions are known as the "Input" and "Activation" functions (further discussed in section 2.0 of the report).

What is important to note as of now, however, is that a network is made of connected nodes, each of which intake inputs and generate output. These connections all have an adjustable weight which will change while the network is learning. To achieve a better picture of this system, consider a biological neural network. The connections between nodes reassemble synapses between neurons, and the weights of these connections represent the synaptic strength.

1.2 Perceptron Learning Rules: Supervised Learning

The perceptron learning rule is a process in which the weights of a network are modified in order for the network to perform the wanted task. The learning rule that is going to be considered throughout this report is “Supervised Learning”, however it is important to note that there are several other types of neural network learning rules.

In supervised learning, a training set for the considered network behaviour is provided. Consider the following example (Hagan, M, et. al):

$$\{p_1,\ t_1\}, \{p_2,\ t_2\}, \ldots., \{p_q,\ t_q\}$$

In this example p_q would be the input of the network and t_q is what we are trying to reach, in other words, our “target”. The input and the outputs will be compared, the supervised learning rule will try to close the gap difference between these two, by adjusting the weights of the network. Further detail on this rule will be outlined in section 2.4.

1.3 Sub-Questions

The following sub questions will be considered throughout the outlined “Experiments and Results” in Section 4.0 :

1. Show the output from your program reproducing the AND learning in the final lecture slides, where the weights are outputted at every application of the weight-change rule.
2. Adjust your program so that it calculates y = 2x+1 and show that it works.
3. What is the effect when you add more target values (if there is any effect)? Try to show data/results of running your program that supports your answer.

2.0 Method

2.1 Initialize Weights

The first step taken in this program is to give training data to the algorithm, this data consists of input links and output links. The weights of the network represent the strength of the connection between units and are used to transform input data. These influence largely on the output because they represent the relationships between units. Their values are calculated during the training of the model, however to start the program they must be initialized. The program initializes randomly the weights of all input links. Considering the same probability for positive and negative values and a range of (-1,1). Throughout the program these will change until the correct output is reached.

2.2 Input Function

The implementation of the *Input Function* consists of the sum of each weight multiplied by it's input link, which are the outputs of the previous node. The following formula outlines it:

$$\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

i is the value of the input link and W is the value of its weight.

2.3 Activation Function

The purpose of the Activation Function is to introduce nonlinearity into the output of the unit. The activation function used in this assignment is based on the *Step Function*, which takes in the result of the *Input Function* as a parameter and, based on this, assigns the value 0 or 1 to the output.
If the input value is less than the threshold then it returns 0, else returns 1. The program outlined in this report will consider a threshold of 0.

2.4 Perceptron learning rule

The perceptron learning rule states that the algorithm must learn the optimal weight coefficients in order to classify data efficiently. This process is executed with supervised learning, which implies that the correct outputs are already known.

The generated outputs of the *Activation Function* are compared with the correct outputs. If these are not the same, the error is propagated back and the weights are adjusted; the new weight is equal to original weight, plus the result of the learning rule. The new outputs are calculated and this process is repeated until the sum of the error, which is calculated as target value, minus perceptron output, is less than a certain value. In this case the margin of error considered is 0.01.

Another important component of the learning rule is the *learning rate*. This reveals how fast the weight change rule is going to evolve, (In this report the chosen learning rate is 0.1). Lastly, the network considers an additional node, known as the *bias*, with an input link of 1 that adds another dimension to the input space and remains unchanged (with the exception of its weight). This allows the "line" to fit the prediction in a better way.

$$\Delta w_i = \underset{\substack{\uparrow \\ \text{learning} \\ \text{rate}}}{\eta}(\underset{\substack{\uparrow \\ \text{target} \\ \text{value}}}{t} - \underset{\substack{\uparrow \\ \text{perceptron} \\ \text{output}}}{o})\underset{\substack{\uparrow \\ \text{input} \\ \text{value}}}{x_i}$$

3.0 Implementation

3.1 Input Function

3.1.1 JAVA

```
public int inputFunction(int[] inputs){
     double result = 0;
     for (int i=0; i<weights.length; i++){
          Result += (weights[i] * inputs[i]);
     }
     //call activationFunction
     return activationFunction(result);
}
```

3.2 Activation Function

3.2.1 JAVA

```
public int activationFunction (double result){
     //threshold =0
     if (result < -EPSILON){
        return 0;
     } else {
        return 1;
     }
}
```

3.3 Perceptron learning rule

3.3.1 JAVA

```
    public int run(){
         double someValue=0.01;     //considering a marginal error of 0.01
         sumError=1;
         int activation;
         int error;
         int counter = 0;
         while (sumError > someValue){
             sumError=0;
             for (int i=0; i< INPUT_NODES.length; i++){
               activation = inputFunction(INPUT_NODES[i]);
               error = INPUT_NODES[i][3] - activation;
               sumError+= Math.abs(error);
               if (sumError!=0) {
                    calculateDeltaWeight(error,INPUT_NODES[i]);
                    break;
               }
             }
               counter++;
         }
         return counter;
    }
```

```
    public void calculateDeltaWeight (int error, int [] row) {
         double  deltaWeight;
         for (int i=0; i<weights.length; i++) {
               deltaWeight= LEARNING_CONSTANT * error *row[i];
               weights[i]+= deltaWeight;
         }
    }
```

4.0 Results

4.1 Sub-Question 1

"Show the output from your program reproducing the AND learning in the final lecture slides, where the weights are outputted at every application of the weight-change rule."

4.1.1 The AND learning: Experiment 1

The AND function follows the AND logic operator defined below:

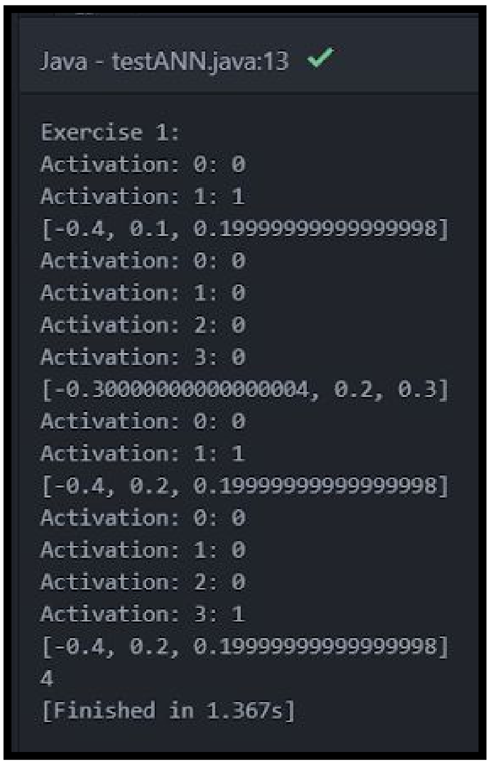
x_0	x_1	x_2	Output
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

In the outlined program (previously shown in Section 3.0) as:

```
AND_INPUT_NODES = {{1,0,0,0},{1,0,1,0},{1,1,0,0},{1,1,1,1}};
```

- The **initial weights** are: [-0.3],[0.1],[0.3]
- The **bias** is the first column of the matrix.
- The **inputs** x_1 **amd** x_2 : columns 2 and 3 (respectively)
- The **target outputs** (correct outputs): column 4

The following is an example test run, taking a total of 4 iterations to the program to calculate the weights.


```
Java - testANN.java:13 ✓

Exercise 1:
Activation: 0: 0
Activation: 1: 1
[-0.4, 0.1, 0.19999999999999998]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
[-0.30000000000000004, 0.2, 0.3]
Activation: 0: 0
Activation: 1: 1
[-0.4, 0.2, 0.19999999999999998]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 1
[-0.4, 0.2, 0.19999999999999998]
4
[Finished in 1.367s]
```

Note that our program was designed to generate and give random weights, however for this exercise we provided the weights to be the same as the example shown during our lecture.

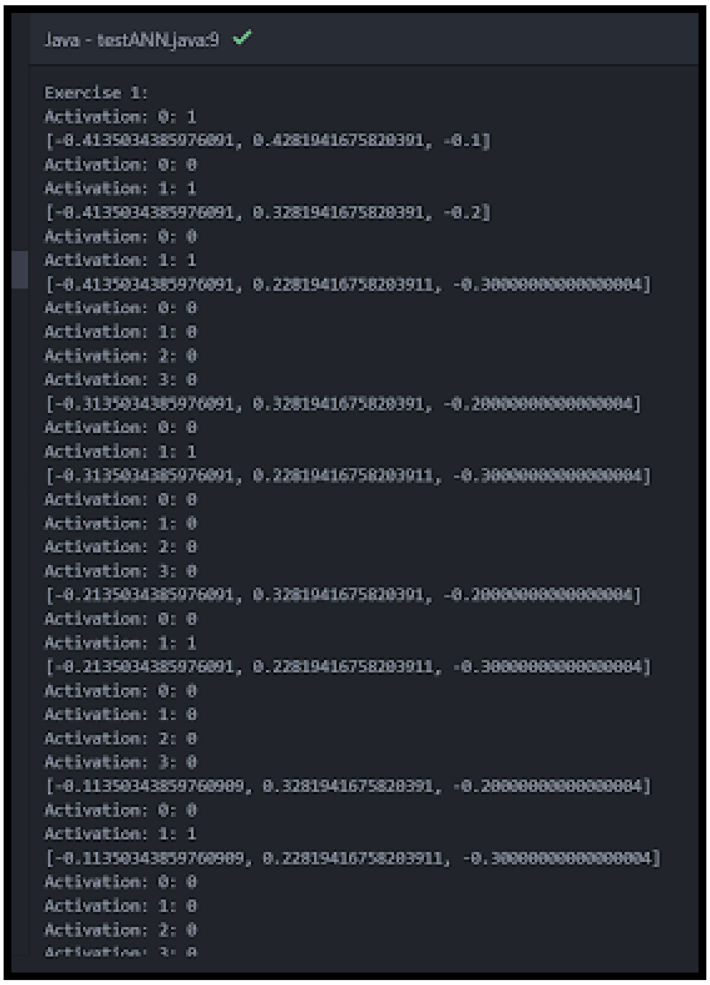
4.1.2 The AND learning: Experiment 2

Experimenting with randomly generated initial weights:

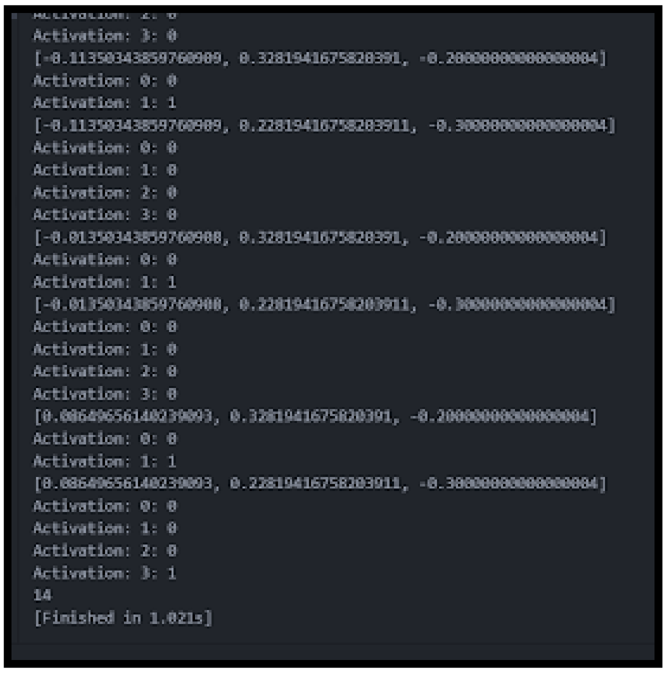
- The **bias** is the third column of the matrix.
- The **inputs** x_1 **amd** x_2 : columns 1 and 2 (respectively)
- The **target outputs** (correct outputs): column 4

```
AND_INPUT_NODES = {{0,0,1,0},{0,1,1,0},{1,0,1,0},{1,1,1,1}};
```

The following is an example test run. It took 14 runs of our algorithm to calculate the weights.



```
Exercise 1:
Activation: 0: 1
[-0.4135034385976091, 0.4281941675820391, -0.1]
Activation: 0: 0
Activation: 1: 1
[-0.4135034385976091, 0.3281941675820391, -0.2]
Activation: 0: 0
Activation: 1: 1
[-0.4135034385976091, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
[-0.3135034385976091, 0.3281941675820391, -0.20000000000000004]
Activation: 0: 0
Activation: 1: 1
[-0.3135034385976091, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
[-0.2135034385976091, 0.3281941675820391, -0.20000000000000004]
Activation: 0: 0
Activation: 1: 1
[-0.2135034385976091, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
[-0.11350343859760909, 0.3281941675820391, -0.20000000000000004]
Activation: 0: 0
Activation: 1: 1
[-0.11350343859760909, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
```



```
Activation: 2: 0
Activation: 3: 0
[-0.11350343859760909, 0.3281941675820391, -0.20000000000000004]
Activation: 0: 0
Activation: 1: 1
[-0.11350343859760909, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
[-0.01350343859760908, 0.3281941675820391, -0.20000000000000004]
Activation: 0: 0
Activation: 1: 1
[-0.01350343859760908, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 0
[0.08649656140239093, 0.3281941675820391, -0.20000000000000004]
Activation: 0: 0
Activation: 1: 1
[0.08649656140239093, 0.22819416758203911, -0.30000000000000004]
Activation: 0: 0
Activation: 1: 0
Activation: 2: 0
Activation: 3: 1
14
[Finished in 1.021s]
```

4.2 Sub-Question 2

“Adjust your program so that it calculates y = 2x+1 and show that it works.

a) What is it in the perceptron that represents the line (see figure)?

b) What would the binary (yes/no; 0/1) output of the perceptron encode (and what would be the inputs)?

c) How many target values (= points = ’truth-table’ lines) would you need minimally?

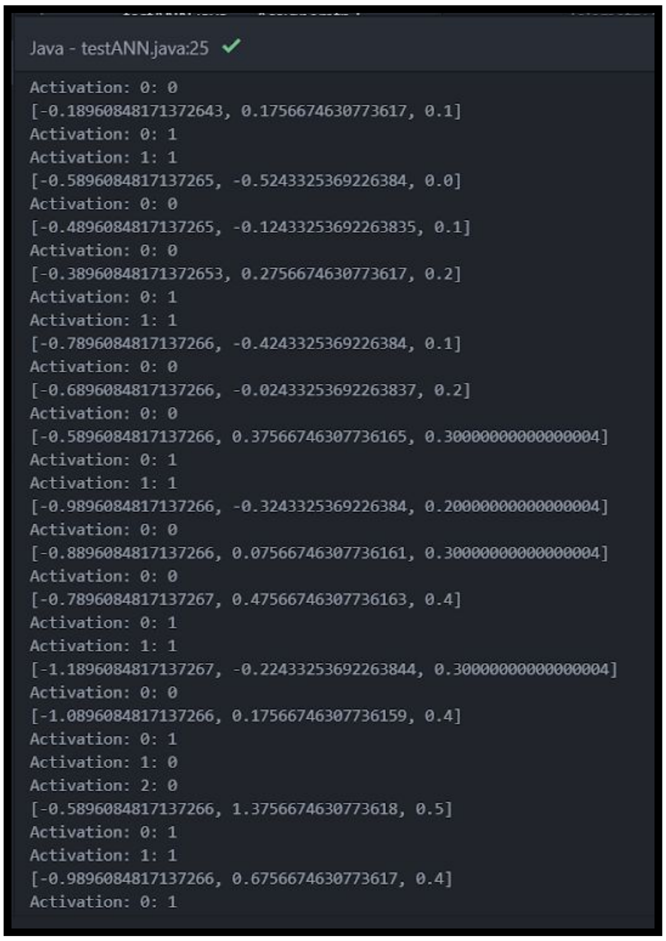
d) How would you choose your target values?”

The following is an example test run, showing that the algorithm implemented and outlined in this report is able to calculate the weights for 10 target values:

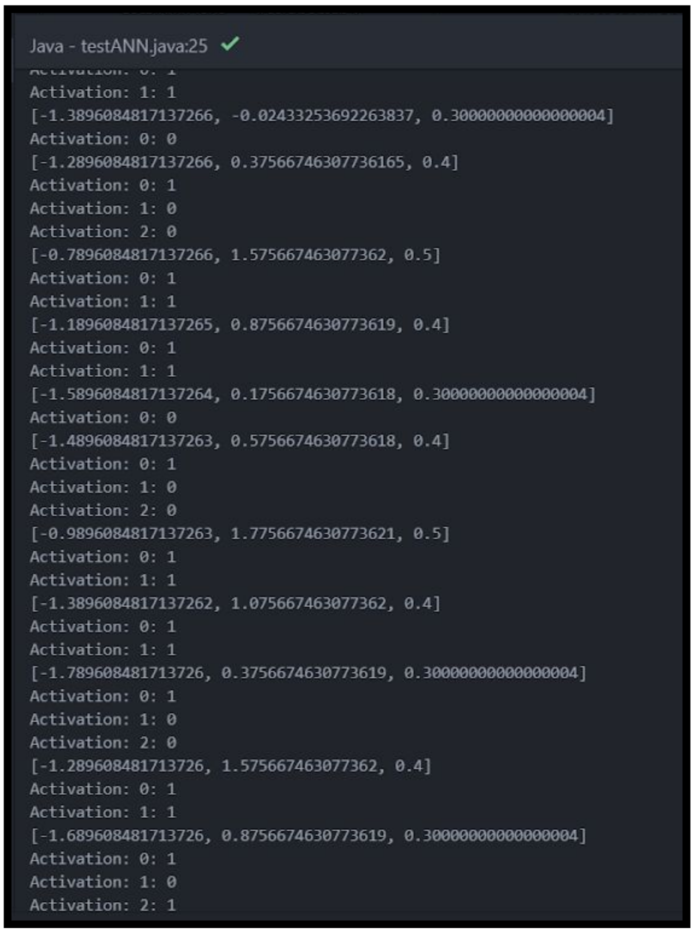
- The target values used were:

```
[[1, 4, 1, 1], [4, 7, 1, 0], [5, 12, 1, 1], [3, 8, 1, 1], [5, 13, 1, 1],
[10, 20, 1, 0], [3, 6, 1, 0], [6, 11, 1, 0], [4, 6, 1, 0], [5, 13, 1, 1]].
```

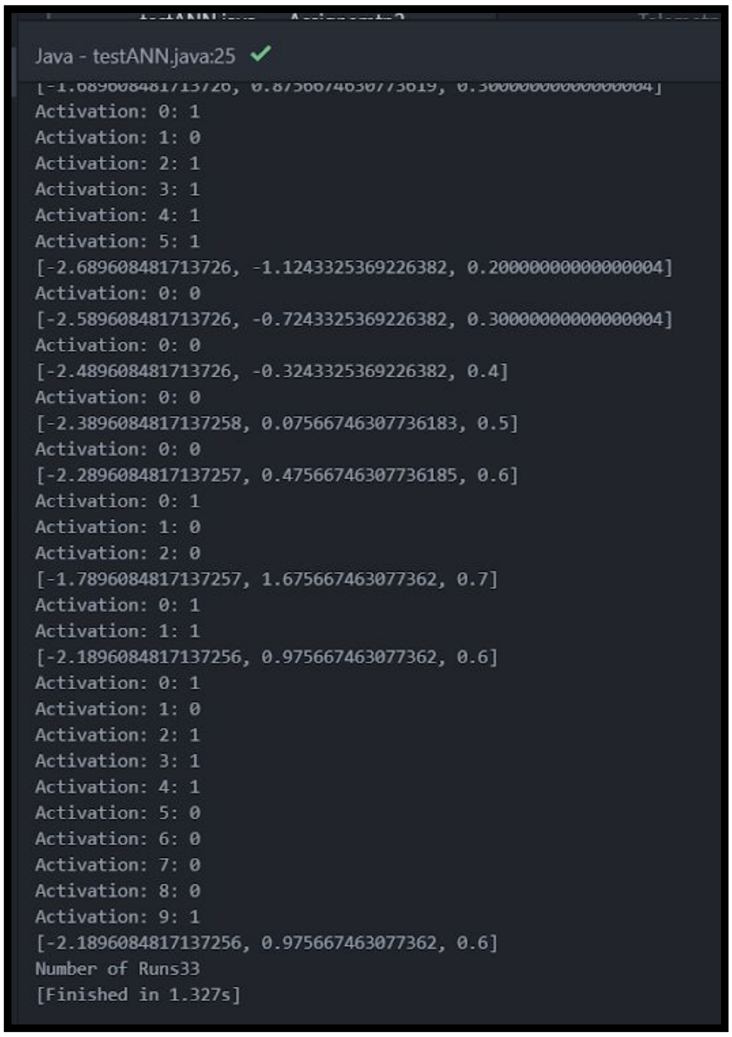
- The **bias** is the third column of the matrix.
- The **target outputs** (correct outputs): column 4


Java - testANN.java:25

```
Activation: 0: 0
[-0.18960848171372643, 0.1756674630773617, 0.1]
Activation: 0: 1
Activation: 1: 1
[-0.5896084817137265, -0.5243325369226384, 0.0]
Activation: 0: 0
[-0.4896084817137265, -0.12433253692263835, 0.1]
Activation: 0: 0
[-0.38960848171372653, 0.2756674630773617, 0.2]
Activation: 0: 1
Activation: 1: 1
[-0.7896084817137266, -0.4243325369226384, 0.1]
Activation: 0: 0
[-0.6896084817137266, -0.02433253692263837, 0.2]
Activation: 0: 0
[-0.5896084817137266, 0.37566746307736165, 0.30000000000000004]
Activation: 0: 1
Activation: 1: 1
[-0.9896084817137266, -0.3243325369226384, 0.20000000000000004]
Activation: 0: 0
[-0.8896084817137266, 0.07566746307736161, 0.30000000000000004]
Activation: 0: 0
[-0.7896084817137267, 0.47566746307736163, 0.4]
Activation: 0: 1
Activation: 1: 1
[-1.1896084817137267, -0.22433253692263844, 0.30000000000000004]
Activation: 0: 0
[-1.0896084817137266, 0.17566746307736159, 0.4]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 0
[-0.5896084817137266, 1.3756674630773618, 0.5]
Activation: 0: 1
Activation: 1: 1
[-0.9896084817137266, 0.6756674630773617, 0.4]
Activation: 0: 1
```


Java - testANN.java:25

```
Activation: 1: 1
[-1.3896084817137266, -0.02433253692263837, 0.30000000000000004]
Activation: 0: 0
[-1.2896084817137266, 0.37566746307736165, 0.4]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 0
[-0.7896084817137266, 1.575667463077362, 0.5]
Activation: 0: 1
Activation: 1: 1
[-1.1896084817137265, 0.8756674630773619, 0.4]
Activation: 0: 1
Activation: 1: 1
[-1.5896084817137264, 0.1756674630773618, 0.30000000000000004]
Activation: 0: 0
[-1.4896084817137263, 0.5756674630773618, 0.4]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 0
[-0.9896084817137263, 1.7756674630773621, 0.5]
Activation: 0: 1
Activation: 1: 1
[-1.3896084817137262, 1.075667463077362, 0.4]
Activation: 0: 1
Activation: 1: 1
[-1.789608481713726, 0.3756674630773619, 0.30000000000000004]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 0
[-1.289608481713726, 1.575667463077362, 0.4]
Activation: 0: 1
Activation: 1: 1
[-1.689608481713726, 0.8756674630773619, 0.30000000000000004]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 1
```



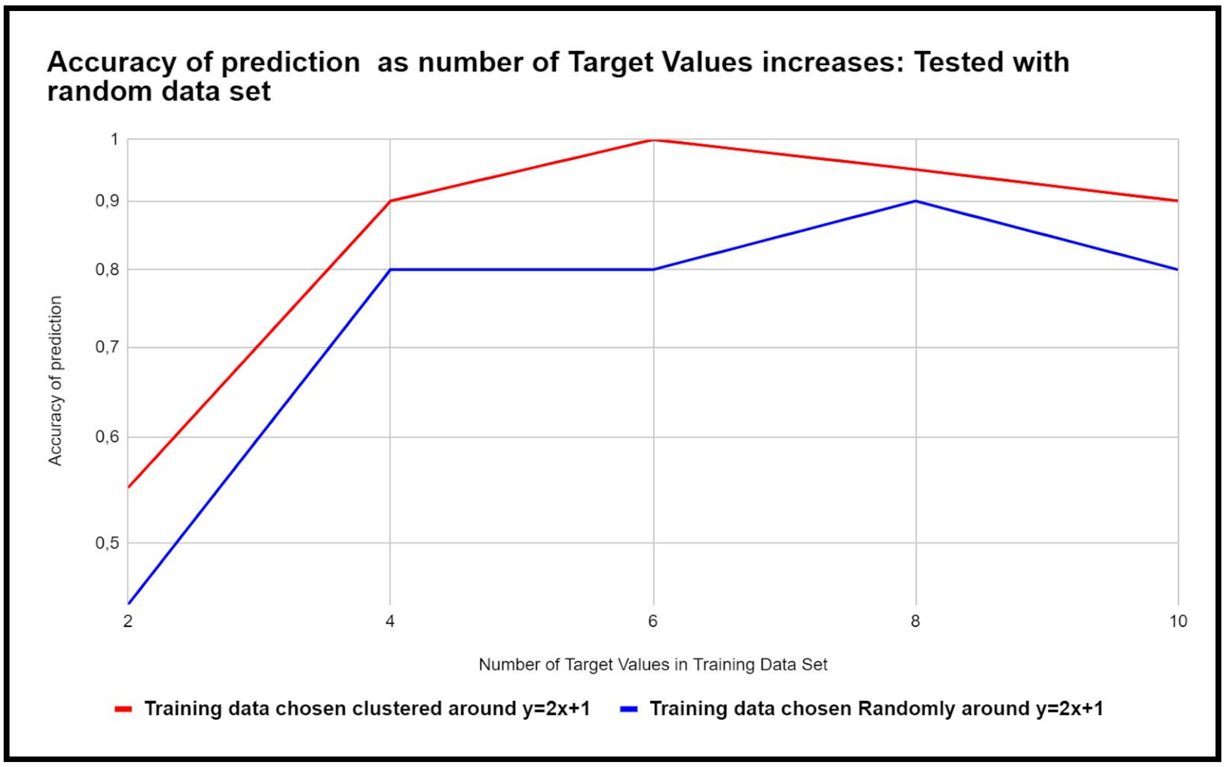
```
Java - testANN.java:25
[-1.689608481713726, 0.8756674630773619, 0.30000000000000004]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 1
Activation: 3: 1
Activation: 4: 1
Activation: 5: 1
[-2.689608481713726, -1.1243325369226382, 0.20000000000000004]
Activation: 0: 0
[-2.589608481713726, -0.7243325369226382, 0.30000000000000004]
Activation: 0: 0
[-2.489608481713726, -0.3243325369226382, 0.4]
Activation: 0: 0
[-2.3896084817137258, 0.07566746307736183, 0.5]
Activation: 0: 0
[-2.2896084817137257, 0.47566746307736185, 0.6]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 0
[-1.7896084817137257, 1.675667463077362, 0.7]
Activation: 0: 1
Activation: 1: 1
[-2.1896084817137256, 0.975667463077362, 0.6]
Activation: 0: 1
Activation: 1: 0
Activation: 2: 1
Activation: 3: 1
Activation: 4: 1
Activation: 5: 0
Activation: 6: 0
Activation: 7: 0
Activation: 8: 0
Activation: 9: 1
[-2.1896084817137256, 0.975667463077362, 0.6]
Number of Runs33
[Finished in 1.327s]
```

The line $y = 2x + 1$ is represented in the perceptron after it has been trained and it then has the ability to classify data as either above or below it. The classification is achieved through the combination of the weights after they have been trained, the input function and activation function.
The training data set consists of the inputs (together with their corresponding outputs) which are first given to the perceptron. After this, the new inputs represent the data that needs to be classified and the binary outputs, their position. In other words, this defines to which group they belong to and which label should be given to them (1 or 0).

When training the perceptron we faced the dilemma of either choosing the training data set randomly (while still giving them the correct output) or by clustering points around the actual graph. After a few experiments we came to the conclusion that the best approach to take is to choose the training data set, using points close to the graph because this implies that the line generated is more accurate and it makes better predictions. (Results represented in Graphs 1 and 2). Nevertheless we decided to investigate what the test data should be: randomly selected or clustered around the line. We expected that randomly selected data would give a wider testing spectrum, closer to a real life situation and therefore being the best option. However, after obtaining elevated accuracy of prediction, we realised that testing the algorithm with clustered data was best because we can actually choose to test it with points near to the line.

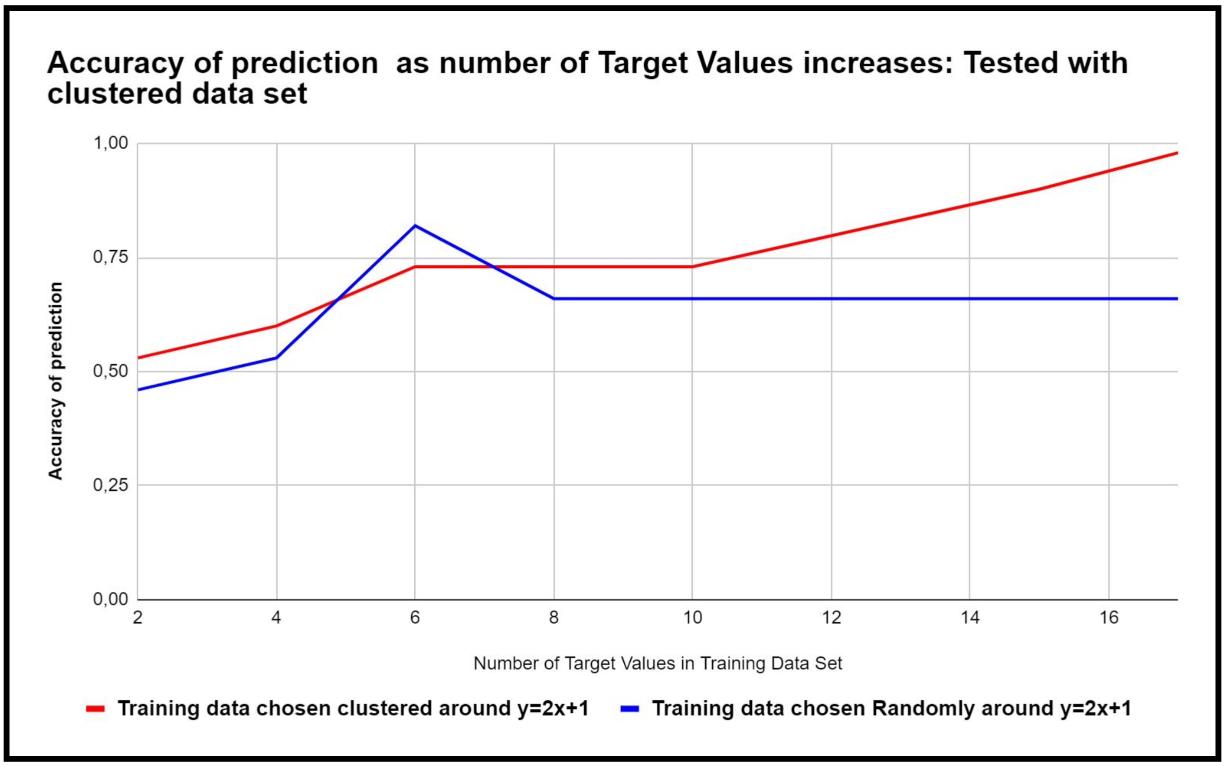
Finally, Sub-Question 2.d asked about the quantity of target values that should be used to train the perceptron and to answer this we ran two types of experiments. **Graph 1** shows that when the Data Set and the Testing Data are both randomly selected, the minimum number of target values to obtain an accuracy of 90% is 8. On the other hand, **Graph 2** shows that if the Training Data and the Testing Data are clustered around the line then the minimum number of target values to obtain an accuracy of at least 95% is 16.

In Graph 1 and Graph 2 you can see the results of our experiments, it is worth mentioning that one of the lines shows an algorithm trained with a random set (red) and the other with a selected clustered set (blue).



Graph 1: Testing accuracy with a randomly selected data set (APPENDIX A)

One thing to remark about this graph is the drop in accuracy of prediction when the number of target values increased to more than 6 for the algorithm trained with clustered points (red line).



Graph 2: Testing Accuracy with a clustered data set. (APPENDIX B)

4.3 Sub-Question 3

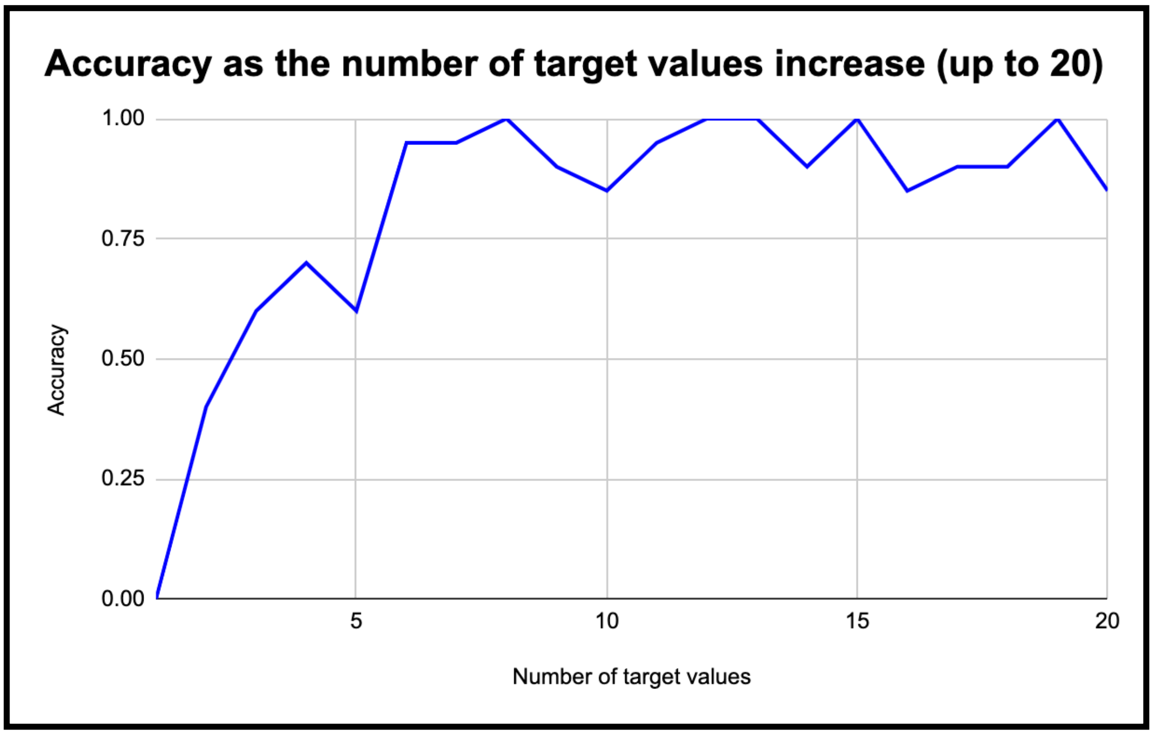
"What is the effect when you add more target values (if there is any effect…)? Try to show data/results of running your program that support your answer."

To answer this question we ran two experiments. In the first, shown in **Graph 3**, we trained the perceptron on a number of different target values, in a range from 1 to 20. We saw that as the number of target values increased from 1 to 8 there was a noticeable improvement in its accuracy. However, after this point, it more or less reached the maximum accuracy. It consistently scored between 0.85 and 1.

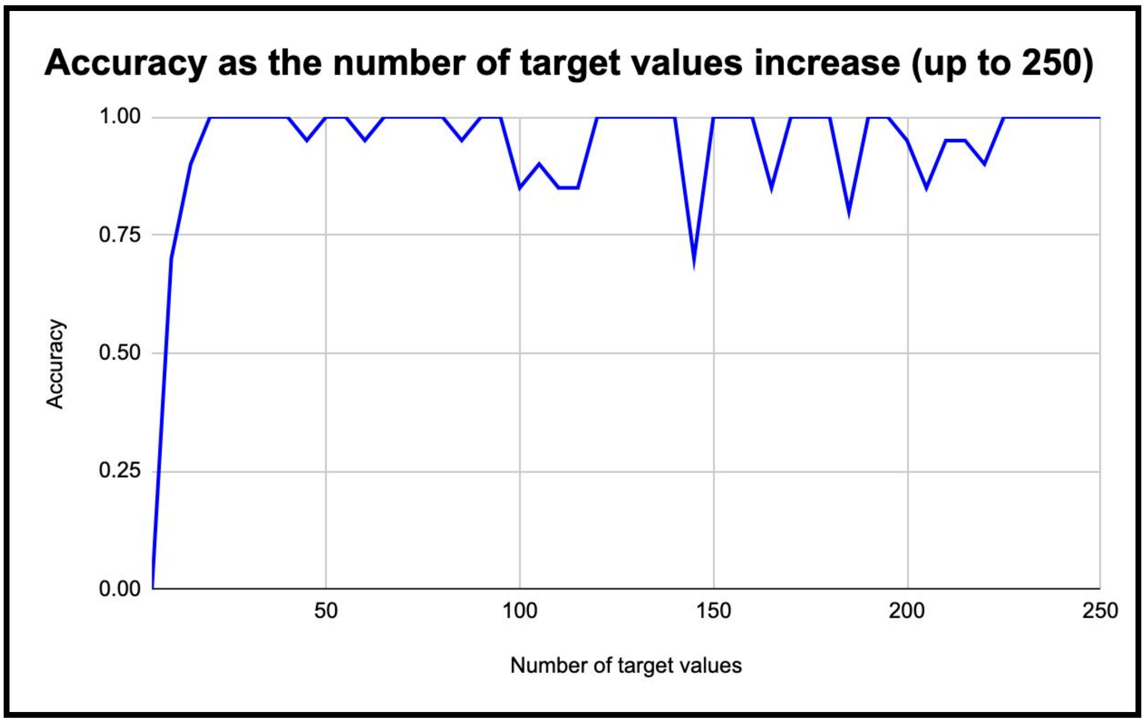
We decided to run a second experiment to try and find the maximum number of target values. We expected to see the phenomenon of *overfitting*. By this we mean that if the input data set is too big, the perceptron will still be able to "memorize" all of the examples, but it will not be able to generalize the inputs that have not been seen before, or better, it will but only if the weights are kept small (Russel, 2010). This should cause the accuracy to decrease because it will fit too well to the training data set and would not be able to predict correctly inputs never seen before.

We ran the model with a range of 5 to 250 target values. In this experiment, we saw no general decrease of accuracy. In fact, the accuracy of the model stayed in the range of 0.8 to 1, apart from a couple of runs (as shown in **Graph 4**).

We thus concluded that either we had not reached the point of overfitting or what we think is more likely that the data we use does not produce the phenomenon of overfitting. This could be due to the fact that the line we are trying to graph is straight, so there is no reason for the algorithm to generalize. And, that the data is clean and always accurately labelled. If the data was more complex, it is likely that overfitting will appear.



Graph 3: Testing Accuracy in a range of 1-20 target values. (APPENDIX C)



Graph 4: Testing Accuracy in a range of 20-250 target values. (APPENDIX D)

5.0 Conclusion and Discussion

Throughout this report we discussed and outlined the implementation of a feed-forward single layer perceptron. We investigated and implemented the perceptron learning rule on the AND logical operator and discussed our findings if the perceptron had to represent the function $y = 2x + 1$.

The perceptron consists of three main components: the input function, the activation function and the perceptron learning rule. By combining these elements we were able to produce a system which could accurately model the AND operator and classify data as either above or below the given linear equation.

Having built the perceptron, we carried out a number of experiments where we altered its parameters to try and optimise our solution. We interrogated the results by way of asking three questions. For the first question, we aimed to give information about what was happening inside the perceptron as the system was training for the AND operator. In the second question, we detailed how the perceptron could classify data in relation to a linear equation. We demonstrated how the selection methods for choosing the training and testing data sets affected results. In the third and final question, we investigated the effect of increasing the number of target values and proposed that a significant increase could introduce overfitting to the algorithm.

Finally, it is also important to note the following: Throughout this report we are only considering models in which a solution actually exists, meaning that the perceptron learning rule is expected and has to reach a solution after a certain number of steps. According to Martin T. Hagan (et. al), this occurs if the perceptron can divide the input space into two different regions. This was the case for both experiments conducted in order to answer sub-questions 1 and 2. In other words, the input vectors were always linearly separable. This brings us to the real limitation of the perceptron learning rule, which is its inability to solve problems which are not linearly separable, for example the XOR gate. The consequence of this limitation led to the investigation of more complex networks such as multilayer perceptrons (Hagan et. al., *Neural Network Design*).

6.0 References

- Kwon, S. J. (2011). *Artificial Neural Networks*. Nova Science Publishers, Inc.
- Hagan, M et al. Neural Network Design. PWS Publishing Company. https://dke.maastrichtuniversity.nl/westra/Education/ANO/NNFL.pdf
- Russel, S. Norvig, P. (2010). *Artificial Intelligence, A Modern Approach*. Pearson Education.

APPENDIX

APPENDIX A

Raw Data For Graph 1: Accuracy of prediction as the number of target values increase - using a random data set

Number of Target Values	2	4	6	8	10
Accuracy of prediction	0,55	0,9	1	0,95	0,9
Randomly generated	0,45	0,8	0,8	0,9	0,8

APPENDIX B

Raw Data For Graph 2: Accuracy of prediction as the number of target values increase - using clustered data

Number of Target Values	2	4	6	8	10	15	17
Accuracy of prediction	0,53	0,6	0,73	0,73	0,73	0,9	0,98
Randomly generated	0,46	0,53	0,82	0,66	0,66	0,66	0,66

APPENDIX C

Raw Data for Graph 3: Testing Accuracy in a range of 1-20 target values

Number of target values	**Accuracy**		**Number of target values**	**Accuracy**
1	0		11	0,95
2	0,4		12	1
3	0,6		13	1
4	0,7		14	0,9
5	0,6		15	1
6	0,95		16	0,85
7	0,95		17	0,9
8	1		18	0,9
9	0,9		19	1
10	0,85		20	0,85

APPENDIX D

Raw Data for Graph 4: Accuracy as the number of target values up to 250

Number of target values	Accuracy		60	0,95
5	0		65	1
10	0,7		70	1
15	0,9		75	1
20	1		80	1
25	1		85	0,95
30	1		90	1
35	1		95	1
40	1		100	0,85
45	0,95		105	0,9
50	1		110	0,85
55	1		115	0,85

Number of target values	Accuracy		Number of target values	Accuracy
120	1		190	1
125	1		195	1
130	1		200	0,95
135	1		205	0,85
140	1		210	0,95
145	0,7		215	0,95
150	1		220	0,9
155	1		225	1
160	1		230	1
165	0,85		235	1
170	1		240	1
175	1		245	1
180	1		250	1
185	0,8			