1 Introduction

This report describes the designing, building and fine-tuning of a multi-layer neural network which will be used in a robot that does groceries. The design of this Artificial Neural Network (ANN) is a feed-forward neural network with back-propagation algorithm. The robot would be able to recognize 7 different types of products, such as fruit, meat and candy. The robot can base these conclusions on 10 features, such as roundness, colour, weight, etc.

2 Method

2.1. Input and output

There were a couple of architectural decisions to be made before coding the neural network could start. Based on the data that was given, it was immediately clear this neural network would consist of 10 inputs. However, for the number of output neurons this was less evident. Since a total of 7 states was necessary for the output, one could argue that working with binary output would make 3 outputs sufficient. This was first considered to be the most effective method, because there were fewer output states thus increasing the chance of getting it right when calculating. Ironically, however, this method proved inadequate due to a high error rate, and so the number of output neurons was increased to 7. In this architecture, there should be only one active node after each calculation, where each node represents one of the 7 states.

2.2. Perceptron

Each node in the neuron is nothing more than a perceptron, which multiplies all inputs with weights, and then sums the results. Mathematically:

$$net_j = \sum_i w_i x_i$$

This output is fed into the activation function, which in our case is a sigmoid function (see Chapter 2.3. Hidden layers):

$$o_j(p) = sigmoid[\sum_i w_i x_i - \theta_j]$$

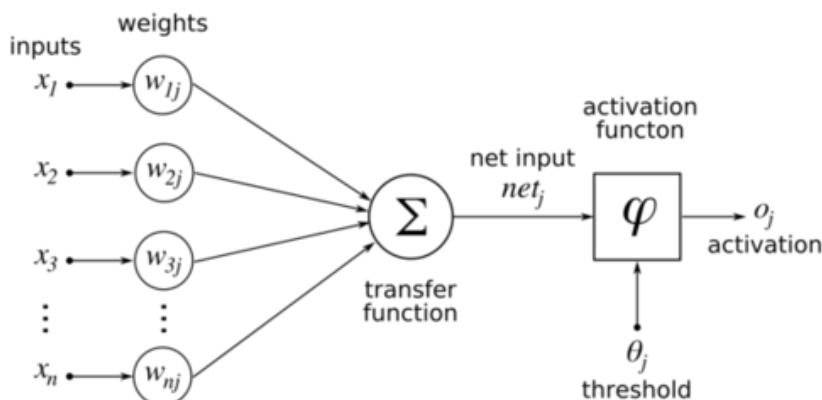The principle of the perceptron can be seen graphically in Figure 1.

Figure 1: Perceptron schematic

2.3. Hidden layers

It was soon thought that making code that was flexible to the number of neurons in a hidden layer would take some extra time to code, but was also the most convenient. Due to time constraints, the number of hidden layers was initially fixed at two layers. Later, however, two extra scripts were made to facilitate one or three hidden layers as well. To update the weights of the output layer, a backpropagation algorithm was implemented:

$$w_{jk}(p + 1) = w_{jk}(p) + \Delta w_{jk}(p)$$

Where:

$$\Delta w_{jk}(p) = \beta * \Delta w_{jk}(p - 1) + \alpha * y_j(p) * \delta_k(p)$$

Where:

$$\delta_k(p) = y'_k(p) * e_k(p)$$

In these functions, w_{jk} means the weight between node 'j' from the previous layer, and node 'k' from the layer for which the weights are calculated. α is the learning rate, which increases the changes in updates at larger values, but also makes it more likely to surpass an optimum. β is called the momentum constant, which has a stabilizing effect on training by making the changes less susceptible to spikes in the learning surface. e_k is the error, which is nothing more than the desired output minus the actual output.

The sigmoid function was used as an activation function for the perceptron (see Chapter 2.2. Perceptron) because it is relatively easy to differentiate compared to a step function. The gradient $\delta_k(p)$ can now be written as:

$$\delta_k(p) = y_k(p) * [1 - y_k(p)] * e_k(p)$$

Because the error of a hidden layer is impossible to compute (one does not know the desired output), the following function is used for the gradient of hidden layers:

$$\delta_j(p) = y_k(p) * [1 - y_k(p)] * \sum_k w_{jk}(p)\delta_k(p)$$

The summation is nothing more than a sum of all weights on the output of the hidden neuron multiplied by the gradient of the node they are connected to.

2.4. Other features

A couple of features were implemented to decrease the amount of epochs required to reach an optimum. This includes the momentum constant β (described in Chapter 2.3. Hidden layers) and an adaptive learning rate. The adaptive learning increases the learning rate by 10% every time the mean square error (MSE) decreases between two epochs. It decreases by 50% every time the MSE increases between two epochs. The learning rate is capped at a certain value alpha_max.

The code also calculates an error rate, which is a percentage of the data that is incorrectly categorized. If the error rate changes less than a certain constant, the calculation stops, because the changes are too insignificant to continue calculating.

2.5. Overview

An overview of the network can be found in Figure 2. The number of hidden layers in can vary (in the figure there are two hidden layers), as well as the number of nodes per hidden layer. Note that the amount of nodes in a layer can only go as high as the number of nodes in the previous layer (except for the output layer). The input layer is fixed at 10 inputs and the output at 7 outputs during the tests (although the code is flexible to changes).
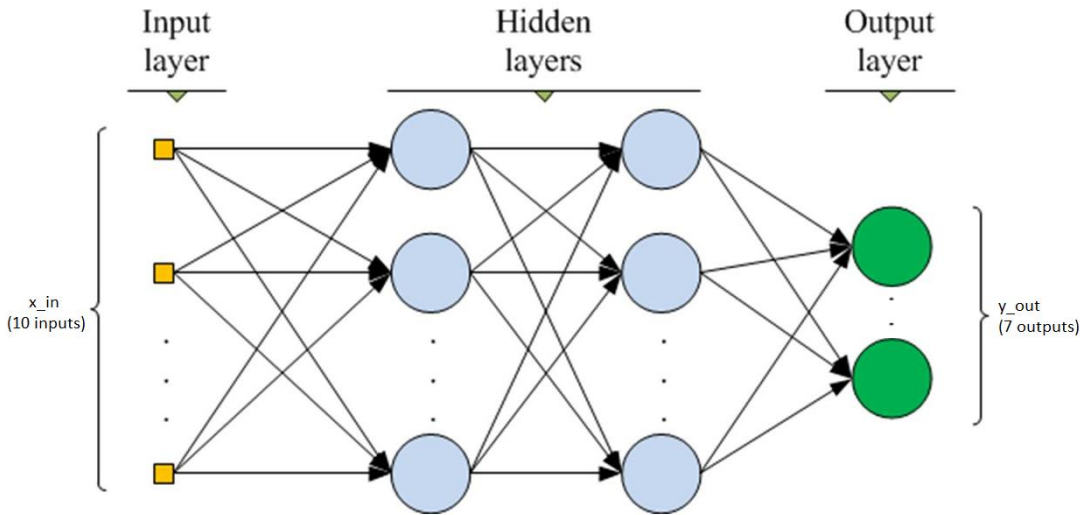
Figure 2: Schematic of the ANN

3 Results

3.1. Training

The data used has 7854 training sets consisting of 10 inputs and a corresponding category. It was decided to use three quarters of the sets for training. The remaining quarter is distributed over a validation set and a test set. The training set is used for the calibration of the weights of ANNs with different variables. Each ANN is then tested with the validation set. The best ANN is then picked to run the test set, which makes sure the chosen ANN is really working properly. This method ensures the ANN works on data that it did not train for. For example, if the ANN is overfitted, it will work perfectly on the training but it will not when working with new data. A separate script was made to calculate the validation and test sets, as well as unknown sets.

Training results are evaluated by calculating the error rate (see Chapter 2.4. Other features) and is stopped when the given number of epochs is reached or when the change in the error rate is too insignificant.

A couple of training sessions were done with the same constant values to see how initialization affects performance. The results after 30 epochs can be found in Table 1. It is clear that (apart from test 9) the validation test results in a higher error rate. This is expected, because this is data that the network has not been trained on. Also notice that the initialization of the weights can have a significant impact on performance, with up to 3.2% more failed calculations.

Table 1: results of tests with different initializations

		Test No.									
Error rate (%)		1	2	3	4	5	6	7	8	9	10
	Training	7.67	8.78	8.03	9.76	8.28	7.55	10.10	7.74	9.78	8.18
	Validation	9.16	8.96	10.18	10.59	9.37	10.79	10.90	9.88	9.06	9.16

There was also a plot made of the error rate and alpha at each epoch of test 10, this can be seen in Figure 3. One can clearly see the increase of alpha at the start, and the decrease after roughly 10 epochs. The increase after every decrease of alpha is most likely because the decrease rate is chosen too high. The error rate plot shows us that it diverges quite fast after roughly 10 epochs to about 10%, while fine tuning (to about 8% error rate) takes about 20 more epochs.
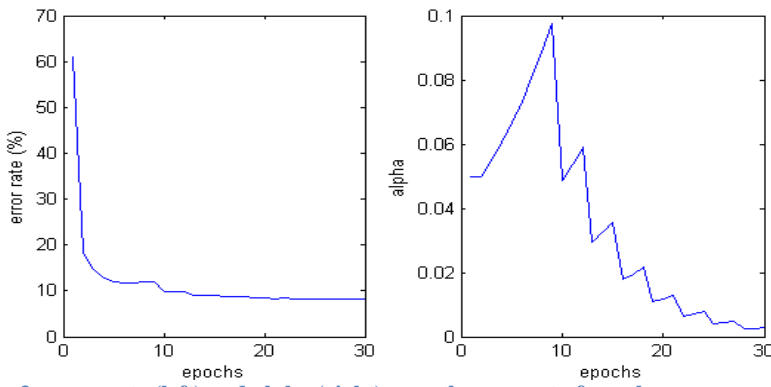
Figure 3: error rate (left) and alpha (right) over the amount of epochs

3.2. Optimization

After training, a couple of calculations were done with different number of neurons (3 tests for each number of neurons). A plot of these calculations can be found in Figure 4. In the figure, the first three tests (tests with 8, 12 and 18 neurons) were computed in an ANN with 2 hidden layers. The neurons were evenly spread, so half of the number of neurons in each layer. The second two were computed in an ANN with 3 hidden layers, because the 2-layer ANN was limited to 18 neurons (a layer cannot exceed the 10 inputs). Here, too, the numbers were evenly spread over all layers.
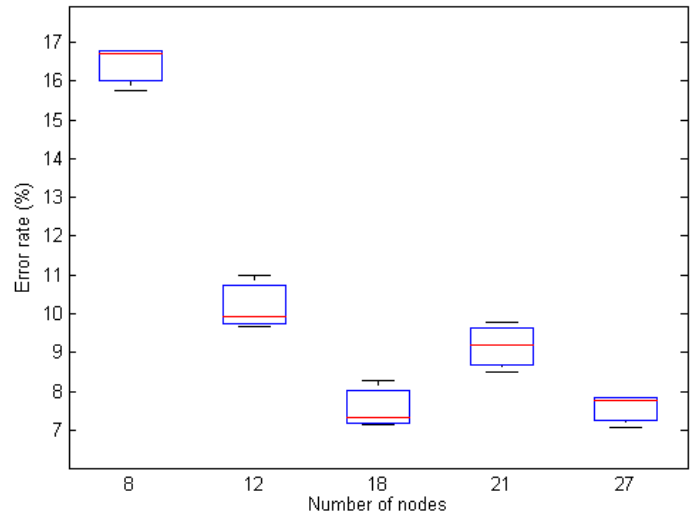
One can see that there is hardly any difference between 2 hidden layers and 3 hidden layers. This is probably due to the fact that the added complexity does not take into account any more information compared to an ANN with a fewer neurons.

Figure 4: Boxplot with error rates of different number of neurons

To decide which architecture to use, a validation test was run on all the data. The results can be found in Table 2. Neuron configurations are noted from the first hidden layer to the last, where each number represents the number of neurons in that layer (so 8,6 means 8 neurons in the first hidden layer, and 6 in the second).The numbers again show that the third layer has hardly any added value, and even seems to have a negative effect on some configurations. On average, the 9,9 configuration seems to score the best, but there is one 9,9,9 configuration which scored the best with 8.55%. This is therefore the network that will be used.

Table 2: Validation results of optimalization

	Neuron configuration	9,9,9	9,9,9	9,9,9	7,7,7	7,7,7	7,7,7			
Error rate (%)	Training set	7.84	7.77	7.08	8.50	9.79	9.22			
	Validation set	9.78	9.47	8.55	10.08	10.29	10.34			
	Neuron configuration	9,9	9,9	9,9	6,6	6,6	6,6	4,4	4,4	4,4
Error rate (%)	Training set	7.33	7.15	8.27	10.98	9.93	10.98	16.69	15.74	16.79
	Validation set	8.76	8.86	10.08	10.29	9.98	11.10	17.11	16.80	17.72

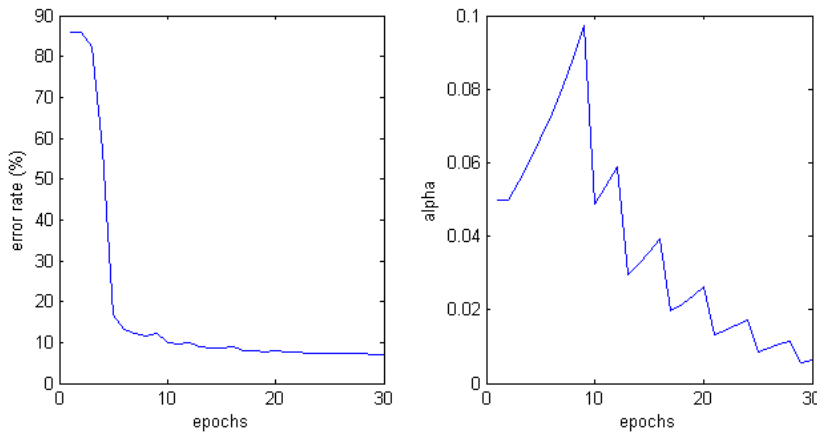A plot of the error rate and the alpha can be found in Figure 5.

Figure 5: plot of error rate (left) and alpha (right) over the epoch number of 9,9,9

3.3. Evaluation

Running the network that was chosen on the test set resulted in an error rate of 8.35%, this is marginally better than the validation set. MATLAB has a built-in neural network program which was run to compare with the results of the designed network. Results of a network with 27 neurons can be found in Figure 6. The results of MATLAB are better, with 0.4% to 1.6% fewer mistakes compared to the designed network. This was expected, because the MATLAB function has gone through a lot more optimisation and the engineers have a lot more knowledge at their hand. What was more of a surprise, though, is the speed at which the programs calculate. While the designed code takes roughly 2 seconds per epoch, the MATLAB function takes less than a second to run through 21 epochs. The MATLAB function is also much more convenient to use because of the GUI.

Results	Samples	CE	%E
Training:	5498	2.38273e-0	6.69334e-0
Validation:	1178	6.65427e-0	5.94227e-0
Testing:	1178	6.65252e-0	7.55517e-0

Figure 6: results of a 27-neuron network generated by the MATLAB NN function

4 Conclusion

It was noticed during training (see Chapter 3.1. Training) that initialization of weights can have a significant impact on the performance, with up to 2.5% more failed calculations. It was also clear that the validation set usually resulted in higher error rates, ranging up to about 2% more errors compared to the training set.

Training also revealed that the error rate usually approached a value after roughly 10 epochs, and using another 20 epochs to refine by another 2% (see Figure 3). The adaptive alpha seems to be working as designed, resulting in a higher alpha in the early stages, and a lower alpha in the last 10 epochs or so (see Figure 3).

The final ANN is adequate in what it does, the error rate remains reasonably stable over every test, recording 7.08% in the training set, 8.55% in the validation set, and 8.35% in the test set. This is promising for future data used in this ANN, where the output is unknown. Comparing the built ANN with the built-in MATLAB function was not very surprising, resulting in 6% to 40% fewer mistakes compared to the designed network. It was evident, though, that the built-in MATLAB function was much faster, taking less than a second to train over 21 epochs. The built ANN takes roughly 2 seconds for one epoch.