Neural Networks - Assignment 1

Michelle Willebrands & Eva van Weenen

July 18, 2019

1 Introduction

The human brain is a fascinating network that is able to *learn*, where the act of learning is defined as "gaining or acquiring knowledge of or skill in (something) by study, experience, or being taught" [4]. Without realizing it, we can easily recognize faces, drive cars, read handwritten texts, translate languages and recognize recurring patterns in images, which is quite difficult to do for computes. If we can let computers 'learn', we can find solutions to difficult algorithmic problems: problems that for example are too expensive for computers to simply compute or problems that involve huge amounts of data that we as humans cannot process. These solutions from learning are often less expensive and much more accurate.

An example of learning that proves to be extremely difficult for a computer by simply programming, is pattern recognition. The human brain is able to distinct many digits and characters, but for computers this turns out to be rather difficult. Introducing a neural network solves these problems. A neural network comprises a number artificial neurons (nodes) in different layers, with connections between them. There is an input- and output layer and possibly hidden layers in between. Signals can be transmitted along the connections, where the weights on the signals are adjusted during the learning process. The network minimizes the difference between its output and the desired output on a training data set (these are the errors) and is actually able to learn in this fashion.

In this report, we will research the different methods of learning for computers and investigate their effectiveness. We will classify handwritten digits of the MNIST data set by using a simple distance feature (section 2.2), by a Bayesian classifier (section 2.3) and by using a neural network with a perceptron algorithm (section 2.4). Furthermore, we will predict the outcomes of the XOR-function using a neural network with the gradient descent algorithm (section 2.5).

2 Methods and results

2.1 MNIST data set

The Modified National Institute of Standards and Technology (MNIST) database is a database of handwritten digits commonly used in machine learning [2]. We will work with a simplified version of this data set, consisting of 2707 16×16 pixel images of handwritten digits (input) and the digit they represent (desired output or label). Each pixel has a value between -1 and 1. This set is split into a training set consisting of 1707 images and a test set of 1000 images. The output of the test set is usually unknown (even though in our case it is known) and we use these data to test the accuracy of our algorithms.

2.2 Simplest classifier

2.2.1 Center

The first method to classify handwritten digits is by implementing a "Shortest distance classifier". Each image consists of 16×16 pixels and can be interpreted as a 256 dimensional vector. In 256 dimensional space, all handwritten digits of the training set will cluster together into clouds C_d at a different location for the different digits $d = 0, 1, ..., 9$. For each cloud, we can calculate its center c_i, defined by the mean of all images in a cloud.

The calculated centers are depicted in figure 1. We can interpret these images as the mean image of all images in a cloud. We see that some images show more 'vague' features than others, meaning that the images in a cloud are spread over a larger area. The center of cloud C_1 is a very distinct image, meaning that all points in the cloud are very close together and thus that the handwritten digits 1 are all written in almost the same way. The center of cloud C_5 is more vague, thus meaning that points in cloud C_5 cover a larger area and that the handwritten digits 5 vary more.

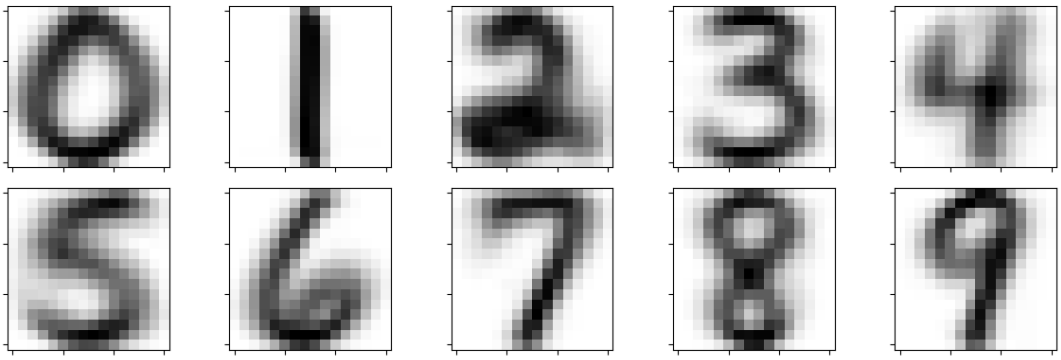

Figure 1: Centers of all digits calculated by taking the mean of all images in a cloud C_i.

2.2.2 Radius

A way of indicating the dispersion of a cloud is by calculating the radius. The radius r_i is defined by the maximum distance between the center of a cloud and all points in this cloud. Furthermore, we can calculate n_i, the number of points belonging to every cloud C_i.

Cloud C_i	0	1	2	3	4	5	6	7	8	9
Number of images n_i	319	252	202	131	122	88	151	166	144	132
Radius r_i	10.0	3.3	9.8	9.3	7.5	11.9	8.3	6.1	7.3	8.1

Table 1: The radius r_i and number of images n_i for every digit cloud of the training set.

The calculated radii of the training set can be observed in table 1. We see that the r_1 is the smallest radius and that r_5 is the largest radius. This coincides with our previous observations where we said C_1 seemed to have a low dispersion and C_5 a high dispersion. We can therefore hypothesize that digit 1 might be much easier to classify than digit 5. However, we must note that our conclusions from the radius are not too reliable, as the radius defines the maximum position from the center of a cloud, and is therefore slightly dependent on the number of points in a cloud n_i. One could for example imagine that the chances of having an outlier (which defines our radius in this case) in a data set is higher when the number of points in the data set is higher. To actually have an accurate indication of the dispersion of a cloud we would have to calculate the 68.2% boundary from the center, better known as the variance.

2.2.3 Distance between centers

A last estimate of the difficulty of classifying a digit is by calculating the (Euclidian) distances between the centers of the 10 clouds

$$dist_{ij} = dist(c_i, c_j) = \sqrt{\sum_{k=1}^{256} (c_{i_k} - c_{j_k})^2} \tag{1}$$

for clouds $i, j = 0, 1, ..., 9$. Intuitively, we can say that if the distances between the centers are very large, the digits will be easier to distinguish.

The distances between all centers are displayed in figure 2. The displayed distance matrix is a symmetric matrix with a zero diagonal, as the distance from c_i to c_i is zero. We observe that the distance between centers 0 and 1 is the largest - thus that the centers of C_0 and C_1 lie the furthest away from each other - implying that these digits will be easiest to distinguish from each other. The centers that lie the closest to each other are c_7 and c_9. The most difficult distinction is therefore between digits 7 and 9.

$$\begin{pmatrix}
0. & 14.4 & 9.3 & 9.1 & 10.8 & 7.5 & 8.2 & 11.9 & 9.9 & 11.5 \\
14.4 & 0. & 10.1 & 11.7 & 10.2 & 11.1 & 10.6 & 10.7 & 10.1 & 9.9 \\
9.3 & 10.1 & 0. & 8.2 & 7.9 & 7.9 & 7.3 & 8.9 & 7.1 & 8.9 \\
9.1 & 11.7 & 8.2 & 0. & 9.1 & 6.1 & 9.3 & 8.9 & 7. & 8.4 \\
10.8 & 10.2 & 7.9 & 9.1 & 0. & 8. & 8.8 & 7.6 & 7.4 & 6. \\
7.5 & 11.1 & 7.9 & 6.1 & 8. & 0. & 6.7 & 9.2 & 7. & 8.3 \\
8.2 & 10.6 & 7.3 & 9.3 & 8.8 & 6.7 & 0. & 10.9 & 8.6 & 10.4 \\
11.9 & 10.7 & 8.9 & 8.9 & 7.6 & 9.2 & 10.9 & 0. & 8.5 & 5.4 \\
9.9 & 10.1 & 7.1 & 7. & 7.4 & 7. & 8.6 & 8.5 & 0. & 6.4 \\
11.5 & 9.9 & 8.9 & 8.4 & 6. & 8.3 & 10.4 & 5.4 & 6.4 & 0.
\end{pmatrix}$$

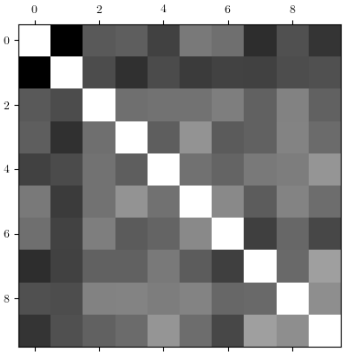

Figure 2: The distance between the centers of the training data and corresponding visualisation. The squares in the image range from black to white where white is a very small distance and black is a very large distance

$$\begin{pmatrix}
0. & 0. & 0. & 0. & 0.6 & 1.3 & 11.3 & 0. & 1.9 & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
1.5 & 0. & 0. & 4.5 & 4.5 & 0.5 & 1.5 & 2. & 3. & 0. \\
0. & 0. & 1.5 & 0. & 0.8 & 2.3 & 0. & 0.8 & 2.3 & 0.8 \\
0. & 6.6 & 0.8 & 0. & 0. & 0. & 2.5 & 0. & 0. & 12.3 \\
3.4 & 0. & 2.3 & 3.4 & 4.5 & 0. & 3.4 & 1.1 & 2.3 & 3.4 \\
6.6 & 2.6 & 3.3 & 0. & 1.3 & 0. & 0. & 0. & 0.7 & 0. \\
0. & 2.4 & 0. & 0. & 1.2 & 1.2 & 0. & 0. & 0.6 & 10.2 \\
0.7 & 1.4 & 0.7 & 6.9 & 1.4 & 2.1 & 0.7 & 0. & 0. & 2.1 \\
0. & 2.3 & 0. & 0.8 & 7.6 & 0. & 0. & 4.5 & 0. & 0.
\end{pmatrix}$$

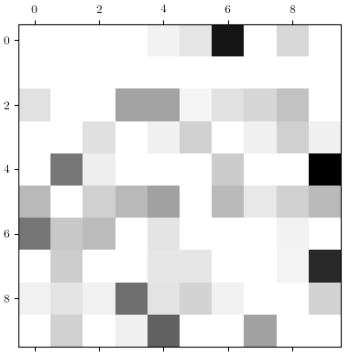

Figure 3: The confusion matrix of the training data and corresponding visualisation. The squares range from white to black where a white square visualizes a low percentage of wrongly classified digits and a black square visualizes a high percentage of wrongly classified digits.

$$\begin{pmatrix}
0. & 0. & 1.3 & 0.9 & 1.8 & 0.9 & 10.3 & 0.4 & 4.5 & 0.4 \\
0. & 0. & 0. & 0. & 0. & 0. & 0.8 & 0. & 0. & 0. \\
2. & 0. & 0. & 5.9 & 7.9 & 1. & 0. & 2. & 12.9 & 0. \\
3.8 & 0. & 3.8 & 0. & 1.3 & 10.1 & 0. & 0. & 1.3 & 2.5 \\
1.2 & 3.5 & 3.5 & 0. & 0. & 0. & 1.2 & 1.2 & 0. & 9.3 \\
5.5 & 0. & 0. & 10.9 & 5.5 & 0. & 1.8 & 0. & 0. & 7.3 \\
7.8 & 0. & 2.2 & 0. & 2.2 & 1.1 & 0. & 0. & 0. & 0. \\
0. & 3.1 & 1.6 & 0. & 7.8 & 0. & 0. & 0. & 0. & 9.4 \\
3.3 & 2.2 & 0. & 6.5 & 3.3 & 3.3 & 0. & 0. & 0. & 2.2 \\
0. & 5.7 & 0. & 0. & 9.1 & 0. & 0. & 5.7 & 2.3 & 0.
\end{pmatrix}$$

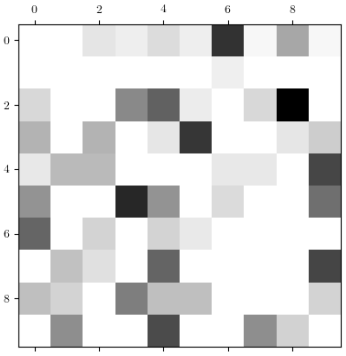

Figure 4: The confusion matrix of the test data and corresponding visualisation. The squares range from white to black where a white square visualizes a low percentage of wrongly classified digits and a black square visualizes a high percentage of wrongly classified digits.

Training data	[0 6], [4 1], [4 9], [6 0], [7 9], [8 3], [9 4]
Test data	[0 6], [2 3], [2 4], [2 8], [3 5], [4 9], [5 0], [5 3], [5 4], [5 9], [6 0], [7 4], [7 9], [8 3], [9 1], [9 4], [9 7]

Table 2: The pairs of digits [i j] in both the training and test data where i is misclassified as j in more than 5 $ of the cases.

2.2.4 Implementation of the classifier

We implement this smallest distance classifier to all images of the training set. We calculate the distance of an image to all clouds C_i and a digit is in this fashion classified as i when it has the smallest distance to the center of cloud C_i. When we compare this output to the designated output, we find that the accuracy of the training set is 86.4%. Consecutively we apply our smallest distance classifier to the test set and find a 80.4% accuracy.

We generate a 10×10 confusion matrix for the training and test data with elements c_{ij} containing the percentage of digits i that were misclassified as digit j. This matrix gives insight into classes that are difficult to separate. These matrices are given and visualised in fig. 3 and fig. 4.

By setting a 95 % accuracy boundary we find that the pairs of digits in table 2 are difficult to calculate for the training set. The misclassified pairs are also given for the test set in this table. We see indeed that for both the training and test set, the 7 and 9 were hard to distinguish but we also see some other pairs of digits that had a lot of misclassifications. These misclassification pairs are quite intuitive as the digits often resemble each other.

Besides, we observe a higher accuracy and a lower misclassification percentage for the training data compared to the test data. This is obvious as the centers of clouds of our 'shortest distance' classifier are calculated using the training data. This is in general the case.

2.2.5 Different distance measures

In order to see if we can get a better accuracy using a different distance measure, we apply our simplest distance classifier again to the test set, using the `scikit-learn` package [5]. We find that the *correlation* distance measure provides the best accuracy on the test set, namely 80.6%, see table 3.

Bray-Curtis	68.2%	Correlation	80.6%	Jaccard Needham	22.4%	Minkowski	80.4%	Sokal-Sneath	22.4%
Canberra	44.4%	Cosine	79.9%	Kulsinski	22.4%	Rogers-Tanimoto	22.4%	squared Euclidian	80.4%
Chebyshev	78.4%	Dice	22.4%	Manhattan	72.1%	Russell-Rao	22.4%	standardized Euclidian	22.4%
City-block	72.1%	Hamming	12.1%	Matching	22.4%	Sokal-Michener	22.4%	Yule	22.4%

Table 3: Accuracies of different distance metrics

The correlation distance between two vectors u and v is defined as [6]:

$$[h]dist(\bar{u}, \bar{v}) = 1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2} \tag{2}$$

and is a measure of dependence between two vectors. If it is zero, the vectors are completely independent. Overall, there is a wide spread in the performance of the different distance measures. The achieved accuracies range between 22.4 and 80.6 %.

2.3 Bayes Rule classifier

We would like to investigate if we can get a better accuracy by implementing a feature of our own and thereby distinguishing between two digits. The feature we choose to investigate is the projected width of the image and we distinguish between the digits 0 and 1. We expect to achieve a high accuracy using these digits. We obtain the projected width by summing over the y-axis of our 16x16 reshaped image, only using pixels with a value higher than zero, and then calculate the length of this projection.

First, we calculate the projected width of all the images in the training data belonging to the two clouds and discretize those into 8 bins. With this amount of bins, each bin is sufficiently populated, allowing us to calculate the probability histogram, see fig. 5. The probability histogram tells us what the probability is that a certain projected width corresponds to the digit 0 or 1. The decision boundary lies at a projected width of 6.5 pixels and is also shown in the histogram.

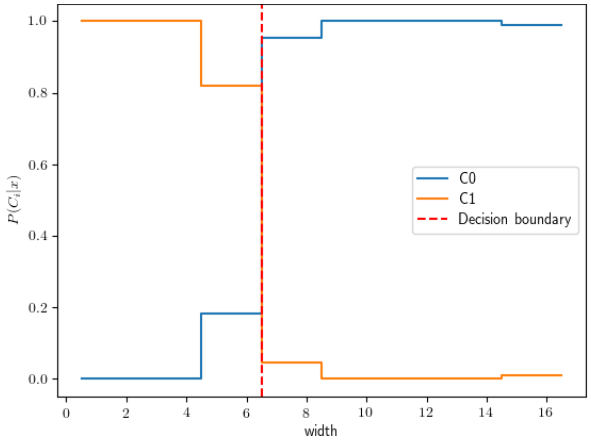

Figure 5: Histogram of the posterior probabilities of the chosen feature in clouds C_0 and C_1.

In order to obtain the posterior probabilities, we used the Bayes rule:

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)} \tag{3}$$

We take $P(X) = 1$ (the probability that the image indeed has a projected width) and calculate the prior probabilities $P(C_i)$ as the number of images in one category (digit 0 or 1) divided by the total number of images in both categories. The class-conditional $P(X|C_i)$ can be calculated from the discrete projected widths of the training data.

Using the obtained decision boundary, we can apply our Bayes Rule classifier to the test data of the two clouds: we calculate the projected width of all these images and assign a class (digit 0 or 1) to each image depending on whether the value is below or above the decision boundary. As can be concluded from the histogram, if the projected width is higher that 6.5, the image will be classified as digit 0.

For the training data set, our Bayes Rule classifier reaches an accuracy of 99.3% and for the test data set this is 98.8%. However, on a different pair of digits, it would be significantly more difficult to distinguishes them well and therefore to enable the algorithm to classify the data with a high accuracy.

2.4 Multi-Class Perceptron Algorithm

We have observed that the accuracy using our 'simplest distance classifier' was maximum 80.6% on the test data. We have improved this accuracy by implementing a Bayesian classifier to distinguish two digits, and we would now like to reach a high accuracy on classifying all digits. We can do this using a Neural Network, where we will train the network to recognize our handwritten digits of the MNIST data set.

In order to do so, we will implement a single-layer multi-class perceptron algorithm. The design of our network is depicted in figure 6. We use 256 input nodes $\boldsymbol{x}$ for the pixels of our image plus one input node inserted at the beginning, which is set to 1, for the bias.

$$\boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{256} \end{pmatrix} \tag{4}$$

For every image we define a desired output 10-vector $\boldsymbol{f}$ using the label y for every image

$$\boldsymbol{f} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_9 \end{pmatrix} \text{ with } f_i = \begin{cases} 1, & \text{if } f_i = y \\ 0, & \text{else} \end{cases} \tag{5}$$

This desired output vector will be useful for later on.

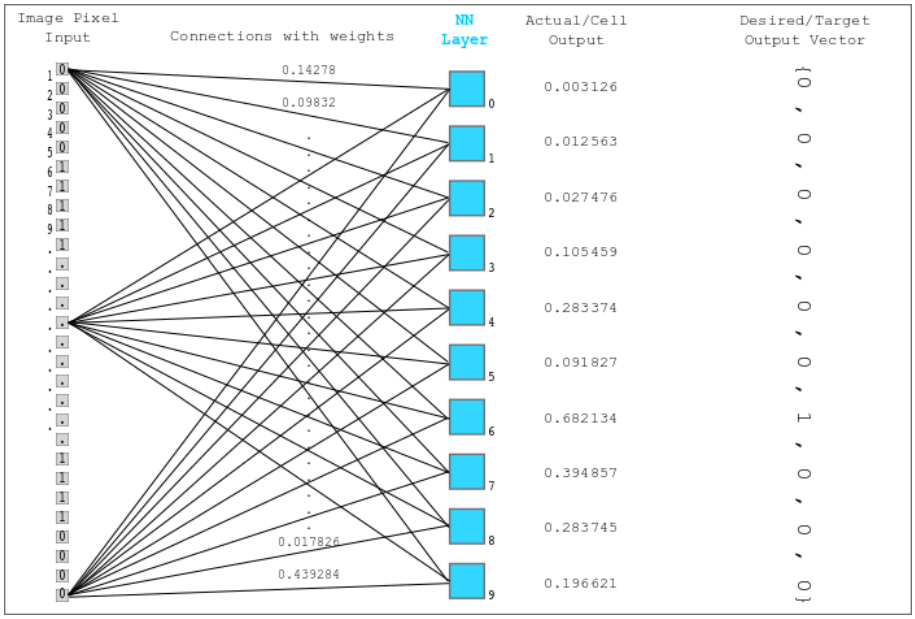

Figure 6: Design of our network on which we apply the multi-class perceptron algorithm. Source: [3]

Then, we introduce a single layer with 10 nodes (one for each digit). We initialize a random weights matrix drawn from a uniform distribution between 0 and 1 for all connections between the 10 nodes in the hidden layer and the 257 input nodes.

$$\boldsymbol{W} = \begin{pmatrix} w_{0,0} & \dots & w_{0,256} \\ \vdots & \ddots & \vdots \\ w_{9,0} & \dots & w_{9,256} \end{pmatrix} \tag{6}$$

For every node j, we calculate the discriminant z_j which is the dot product between the weights for a node W_{ji} and the pixels of an image x_i:

$$\boldsymbol{z} = \begin{pmatrix} z_0 \\ \vdots \\ z_9 \end{pmatrix} = \boldsymbol{W} \cdot \boldsymbol{x} = \begin{pmatrix} w_{0,0}x_0 + w_{0,1}x_1 + ... + w_{0,256}x_{256} \\ \vdots \\ w_{9,0}x_0 + w_{9,1}x_1 + ... + w_{9,256}x_{256} \end{pmatrix} \tag{7}$$

Subsequently, we define an activation function which returns the index of the element with the largest value in our vector $\boldsymbol{z}$:

$$a = \texttt{argmax}(\boldsymbol{z}) \tag{8}$$

As our output is already binary we simply take a as the output of our algorithm and call it $\hat{y} = a$. Thus, we can compare the predicted output $\hat{y}$ to the desired output y. Furthermore, we define a 10-vector $\boldsymbol{g}$ for every image in the same way we defined the desired output vector $\boldsymbol{f}$:

$$\boldsymbol{g} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_9 \end{pmatrix} \text{ with } g_i = \begin{cases} 1, & \text{if } g_i = \hat{y} \\ 0, & \text{else} \end{cases} \tag{9}$$

Lastly we define a stepsize $\eta = 1.0$ which can be interpreted as the learning rate.

As long as any of the predicted outputs $\hat{y}$ are not equal to desired outputs y (so, as long as there are misclassified samples) we perform the following steps:

1. We pick a random misclassified image r.
2. We update the weights using the output of the previous steps for image r (such as $\boldsymbol{f}$ and $\boldsymbol{g}$):

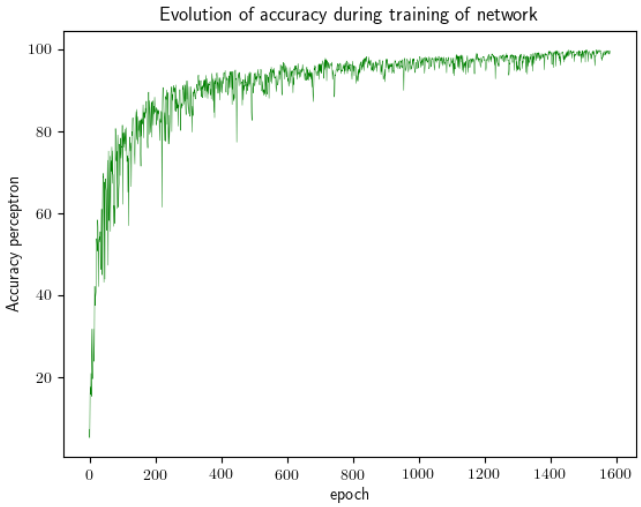

Figure 7: The accuracy per run for the training set during training our neural network using the perceptron algorithm.

$$\boldsymbol{W_{new}} = \boldsymbol{W} + \eta(\boldsymbol{f} - \boldsymbol{g}) \cdot \boldsymbol{x}^\top \tag{10}$$

Subtracting the desired outputs $\boldsymbol{f}$ from the actual outputs $\boldsymbol{g}$ will cause the nodes to be updated correctly. For example, in the case of a 0 that is misclassified as a 2:

$$\boldsymbol{f} - \boldsymbol{g} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \\ \vdots \\ 0 \end{pmatrix} \tag{11}$$

$$(\boldsymbol{f} - \boldsymbol{g}) \cdot \boldsymbol{x}^\top = \begin{pmatrix} 1 \\ 0 \\ -1 \\ \vdots \\ 0 \end{pmatrix} \cdot \begin{pmatrix} x_0 & x_1 & \dots & x_{256} \end{pmatrix} = \begin{pmatrix} x_0 & x_1 & \dots & x_{256} \\ 0 & 0 & \dots & 0 \\ -x_0 & -x_1 & \dots & -x_{256} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \tag{12}$$

3. Calculate $\hat{y}$ and $\boldsymbol{g}$ again for all images and repeat above steps as long as the desired output of all images in not equal to the actual output of the images.

For the training set, an accuracy of 100% should be reached if we repeat it until there are no misclassified samples anymore. The accuracy per run or cycle is plotted in figure 7. We see that the accuracy increases very fast in the beginning, and asymptotically reaches the 100%. However, the algorithm is not very fast. It takes more than 1600 runs to complete.

Using the weights calculated in the previous step, we can now classify our images in the test set. Applying our optimal weights to the test set, we receive an accuracy of 87.1%.

A big drawback, however, is that the perceptron algorithm is unable to handle vectors that are linearly non-separable and this limits its use in some classification problems. An example where this is the case, is the exclusive-or (XOR) function; a perceptron algorithm cannot learn to predict the right outcomes for each vector. Therefore, we will implement the gradient descent algorithm to look at this XOR-problem in the next section.

2.5 Gradient Descent Algorithm

In the backpropagation method, the weights of a network are adjusted by calculating the gradient of the loss function, that maps the difference between the predicted and desired output. We will

implement the gradient descent algorithm on a neural network that is trained to learn the XOR function, see Table 4. The network we created consists of an input layer with two nodes, a hidden layer with two nodes as well and an output layer with one node. We add the bias nodes and thus need 9 weights in total. This network is shown in figure 8.

Input vector	Desired output
(0,0)	0
(1,0)	1
(0,1)	1
(1,1)	0

Table 4: The input vectors and corresponding desired output values for the XOR-problem.

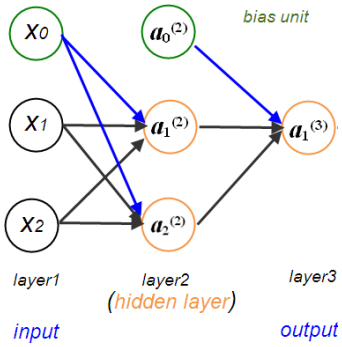

Figure 8: A visualisation of the neural network that we used for the XOR-problem. X_1 and X_2 are the input nodes and X_0 is the bias node for the input layer. In the hidden layer there are also two nodes plus a bias node and the $a_i^{(j)}$ are the activations for node i in layer j. As can be seen, there are 9 connections in the network and therefore 9 weights. Source of this image: [1]

The input of the hidden layer is calculated using the discriminant function

$$\boldsymbol{z}^{(2)} = \boldsymbol{w}^{(1)} \cdot \boldsymbol{x}^{(1)} \tag{13}$$

where $\boldsymbol{w}^{(1)}$ is a 2x3-matrix of weights between the input layer and hidden layer and $\boldsymbol{x}^{(1)}$ are the inputs. As indicated in figure 8, the non-input layers use an activation function. We will use the sigmoid function $sigm$, hyperbolic tangent $tanh$ and linear rectifier $relu(x) = max(0, x)$ as activation functions and test their abilities. The activation function g is applied to the hidden layer:

$$\boldsymbol{a}^{(2)} = g(\boldsymbol{z}^{(2)}) \tag{14}$$

and a bias node $a_0^{(2)}$ is added to this layer. The output is then calculated by performing the previous steps again:

$$\boldsymbol{a}^{(3)} = g\left(\boldsymbol{z}^{(2)}\right) = \text{sigmoid}\left(\boldsymbol{w}^{(2)} \cdot \boldsymbol{a}^{(2)}\right) \tag{15}$$

To make our output binary, we apply a step function over the output activation, this gives the prediction from the algorithm:

$$\hat{\boldsymbol{y}} = \begin{cases} 1 & \text{for } \boldsymbol{a}^{(3)} > 0.5 \\ 0 & \text{for } \boldsymbol{a}^{(3)} \leq 0.5 \end{cases} \tag{16}$$

Subsequently, we define the error or loss function, mse, that calculates the mean squared error that the network makes on the four input vectors and desired outputs:

$$mse = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{17}$$

where N is the number of data points (4 for the XOR function), $\hat{y}_i$ the predicted output value of the network and y_i the desired output value for data point i.

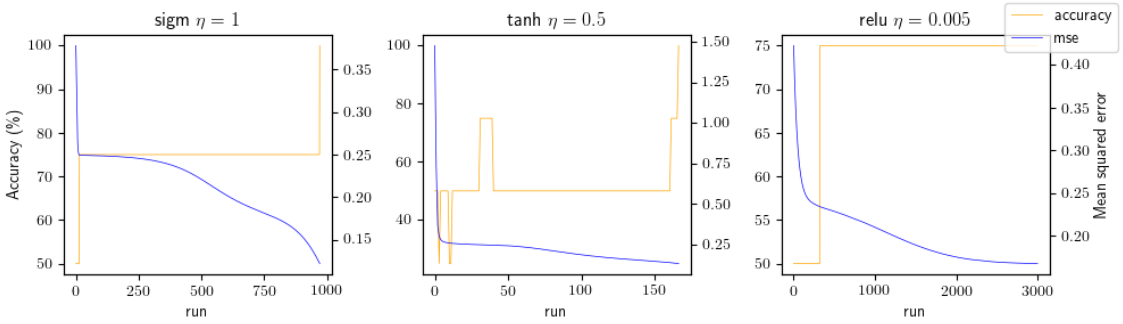

Figure 9: Accuracy and mean squared error for the gradient descent algorithm applied on the XOR-network

The gradient of this *mse* function is approximated by calculating the values of:

$$\nabla mse_i = \left(mse\left(w_0, w_1, ..., w_i + \varepsilon, ..., w_8\right) - mse\left(w_0, w_1, ..., w_i, ...w_8\right)\right)/\varepsilon \tag{18}$$

for $i \in 0, 1, .., 9$ and $\varepsilon = 10^{-3}$. This gradient function ∇mse returns the vector of partial derivatives of the *mse* function over each i^{th} component of the weights vector. The gradient of the *mse* indicates where the sum of squared errors is minimized.

To implement the gradient descent algorithm, we first initialize random weights. As long as the predicted outputs $\hat{\boldsymbol{y}}$ are not equal to the desired outputs $\boldsymbol{y}$ we update the weights:

$$\boldsymbol{w}_{new} = \boldsymbol{w} - \eta \nabla mse\left(\boldsymbol{w}\right) \tag{19}$$

The initialization of η and the weights are very important. When the width of the absolute minimum of the *mse* is comparable to η, the algorithm may never find the optimal weights. When η is too small, it may get stuck in a local minimum and never get out. However, [7] suggest that local minima do not exist for the XOR-function, only saddle points exist, thus η could never get too small. Furthermore, the performance of our algorithm will differ for differently initialized weights. In some cases, we may find that negative weights will be useful.

We have plotted the accuracy and mean squared error for each run in figure 9. We find that the sigmoid function works best for $\eta = 1$ and weights initialized from a uniform distribution between 0 and 1. We find that the hyperbolic tangent function is faster with the same initializations. It is the fastest and works best for $\eta = 0.5$ and weights initialized from a normal distribution between -1 and 1. Furthermore we find that the *relu* function does not reach a 100% accuracy within 3000 runs. For the relu function, we initialize weights from a normal distribution around 0 and with scale 0.2 and we set $\eta = 0.005$. Overall, we find that *tanh* has the best performance.

3 Conclusion & Discussion

In this assignment, we implemented several algorithms that could classify the simplified images of handwritten digits of the MNIST data set. First, we used the shortest (Euclidian) distance classifier (simplest classifier) to classify our handwritten images as digits. We calculated the centers c_i of clouds C_i for digits i and made predictions about how well our classifier would work based on the distance between the centers and the points furthest away from the center. To classify the images, the shortest distance between that image and all clouds was used. This algorithm achieved accuracies of 86.6% and 80.4% on the training and test data respectively. We also implemented other distance measures and the *correlation* measure resulted in the maximum accuracy of 80.6% on the test set.

In order to achieve higher accuracies, we implemented a Bayes rule classifier. To keep it simple, we used only two digits: 0 and 1. The feature we chose to use, was the projected width of the written digits. Intuitively, we expected this to be a clear, distinctive feature for these digits. The Bayes rule was used to obtain a posterior probability histogram including a decision boundary. The implementation of this algorithm resulted in a 99.3% and 98.8% accuracy for the training and

test data respectively; a high accuracy as we expected. However, implementing this method on multiple digits, would involve too many complex features and is nearly impossible.

Then, we constructed a simple multi-class, single layer perceptron algorithm. This neural network with no hidden layers could be trained on the complete training data set to obtain the weights w_i that provide a 100 % accuracy on the training data. We apply the optimized weights on the test data and achieved a 87.1% accuracy. However, we find that 20 % of the weights have values above 10, which is a clear sign of overfitting of the algorithm on the training data. This could be prevented by setting a maximum number of runs, or punishing large size weights as is done with regularization. Furthermore, an important disadvantage of the perceptron algorithm, is that it can only handle linearly separable vectors.

In the last task we use the Gradient Descent Algorithm applied to the XOR-function, which is linearly non-separable. We use a network with one hidden layer and propagate the gradient of the mean squared error between the predicted and desired output back to the input of the network and in this way minimize the errors. We achieve a 100% accuracy on XOR-functions. However, different performance levels can be achieved using different activation functions and different learning rates. We find that the hyperbolic tangent is the best activation function and find a fast learning rate for this function.

As we can see from these algorithms, the computer can learn to classify the handwritten digits well, almost as well as any human could. Even the simpler classifiers reach decent accuracies and the more sophisticated networks can be near perfect. With more complicated networks, computers can learn to do tasks way more difficult than the one discussed in this assignment.

References

[1] K. Hong. Neural networks with backpropagation for xor using one hidden layer. http://www.bogotobogo.com/python/python_Neural_Networks_Backpropagation_for_XOR_using_one_hidden_layer.php. Accessed: 2018-03-13.

[2] Y. Lecun et al. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2018-03-12.

[3] M. Lind. Simple 1-layer neural network for mnist handwriting recognition. https://mmlind.github.io/Simple_1-Layer_Neural_Network_for_MNIST_Handwriting_Recognition/, 2015. Accessed: 2018-03-12.

[4] Oxford-Dictionaries. *Oxford Dictionary of English*. Oxford University Press, 2010.

[5] Scikit-learn. *sklearn.metrics.pairwise.pairwise_distances*. http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html, 2017. Accessed: 2018-03-12.

[6] Scipy-community. scipy.spatial.distance.correlation. https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.distance.correlation.html, 2014. Accessed: 2018-03-12.

[7] I. Sprinkhuizen-Kuyper and E. Boers. Blum's local minima are saddle points. http://liacs.leidenuniv.nl/assets/PDF/TechRep/tr94-34.pdf. Department of Computer Science, Leiden University.