Universiteit Leiden
The Netherlands

Statistical Science for the Life and Behavioral Sciences

Deep Learning and Neural Networks

Assignment 1 - Neural Network from scratch

Authors:
Freek van Geffen
Justin Kraaijenbrink

Student number:
s2633256
s2577984

Instructor: Dr. W.J. Kowalczyk

March 11, 2021

Table of Contents

1 Methods and Results

1.1 The MNIST Data set

The Modified National Institute of Standards and Technology (MNIST) data set is a data set with images of handwritten digits. In this assignment, we worked with a simplified version of the MNIST data set. It contains 2707 digits represented by vector of 256 numbers that represent 16x16 images. The value of each pixel is between -1 and 1. The data set is divided into a training set (1707 images) and a test set (1000 images).

The objective of this assignment is to develop and evaluate several algorithms for classifying images of handwritten digits d, where $d = 0, 1, ..., 9$. For each digit d a cloud of points in 256 dimension space, C_d, exists. The number of points n_d is different for each of the digits. In order to gain insights about the MNIST data, for each digit cloud C_d, the center c_d and the radius r_d were computed based on the images from the training data.

Centers The center, c_d represents a 256-dimensional vector of means overall coordinates that belong to C_d. The calculated centers are illustrated in Figure 1. The graphic illustration of the centers shows that some of the centers represent a digit very clear, e.g. digit 1. Nevertheless, some centers are vaguer, e.g. digit 2 and digit 5.

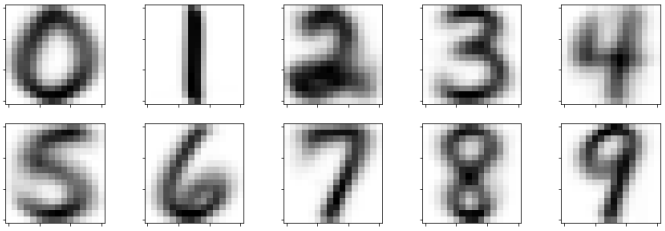


Figure 1: The centers c_d, which represent the means over all coordinates of all images in a cloud C_d.

Radii The radius, r_d, is defined as the biggest distance between the center of C_d and points from C_d. Table 1 shows the number of points that belong to each of the digit clouds and the radius of each of clouds. Digit cloud 1 has the lowest radius and digit cloud 9 has the highest radius. The result that digit cloud 1 has the lowest radius is consistent with the clear illustration of the center in Figure 1. This would suggest that digit 1 is easier to classify than digit 9. The number of points that belong to C_d ranges from 88 to 319, and the radii do not incorporate that information. This makes conclusions based on the radii less reliable.

Table 1: The number of images n_d and radius r_d for every digit cloud C_d of the training set.

Cloud C_d	Counts n_d	Radius r_d
0	319	15.893
1	252	9.481
2	202	14.169
3	131	14.745
4	122	14.534
5	88	14.452
6	151	14.032
7	166	14.909
8	144	13.706
9	132	16.139

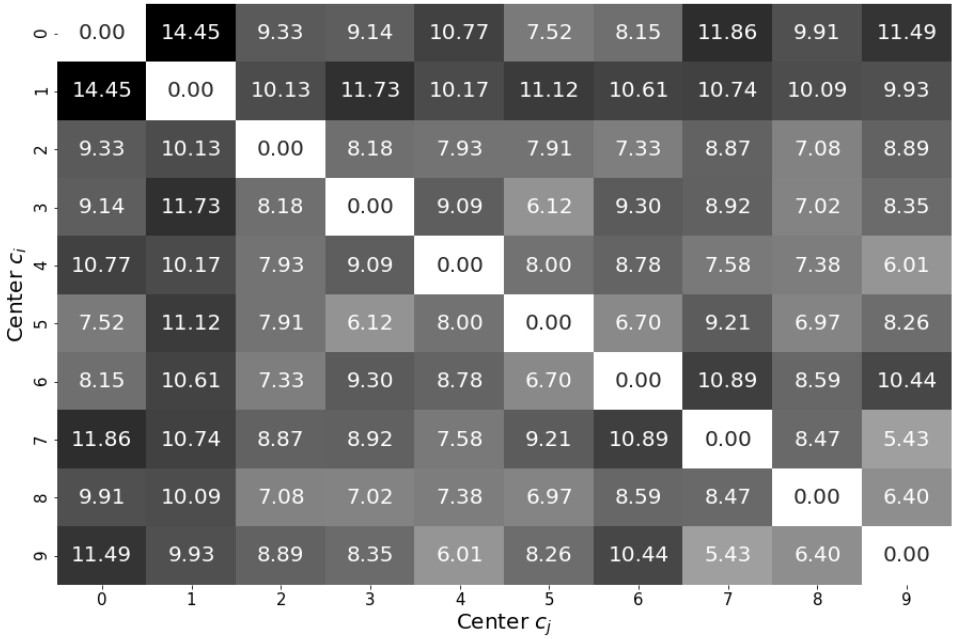


Figure 2: The Euclidean distances between center c_i and center c_j, for i, j = 0, 1, ..., 9.

Distances Subsequently, the Euclidean distances between the centers of the 10 clouds were computed, see equation 1. These distances show which digits are easier to distinguish. Figure 2 shows the distances between each of the centers. The distance between the centers of digit 0 and digit 1 is the largest distance, which implies that digit 0 and digit 1 are the easiest pair of digits to distinguish. The distance between the centers of digit 7 and digit 9 is the smallest, which suggests this pair of digits is the most difficult to separate.

$$dist_{ij} = dist(c_i, c_j) = \sqrt{\sum_{i=1}^{256}(c_i - c_j)^2} \quad \text{for i, j} = 0, 1, ..., 9 \tag{1}$$

1.2 Simple Distance-based Classifier

An algorithm to classify handwritten images of digits is the simple distance-based classifier. The simplest distance-based classifier calculates the distances between an image and the 10 centers. The center with the closest distance defines the label of the image. The centers are calculated based on training images. In this task, we implemented a simple distance-based classifier to classify images of the MNIST dataset.

Evaluating the classifier The simple-distance based classifier, with Euclidean distances, correctly classified 86.4% of the images of the training set and 80.4% of the images of the test set. Figure 3 shows the confusion matrices of the training and test set. As expected, digit 1 is the easiest digit to classify correctly. Digit 5 is one the most difficult to classify correctly. It has the lowest percentage of correctly classified observations in the training set and the second lowest percentage in the test set. Digit 2 has the lowest accuracy in the test set. These findings tie well with the findings based on the clarity of the illustrated centers. The expectation that digit 9 is one of the hardest to classify, based on the size of its radius, does not find any substantial evidence in Figure 3.

The most occurring misclassification in the training set is classifying digit 4 as a 9. The most occurring misclassification in the test set is classifying digit 2 as an 8. Based on the shortest distance analyse, we expected that digit 7 and digit 9 are hard to distinguish. The confusion matrices show some evidence that digit 7 and digit 9 are difficult to distinguish. Based on the training set, 10% of the 7's are classified as a 9 and 5% of the 9's are classified as a 7. Based on the test set, 9% of the 7's are classified as a 9 and 6% of the 9's are classified as a 7.

Most of the digits have a higher training accuracy than test accuracy, which is obvious as the centers are calculated based on the training data. Only the digits 4 and 6 have a higher test accuracy than their training accuracy. It is an interesting result that digit 4 and 6 have a higher test accuracy than their training accuracy.

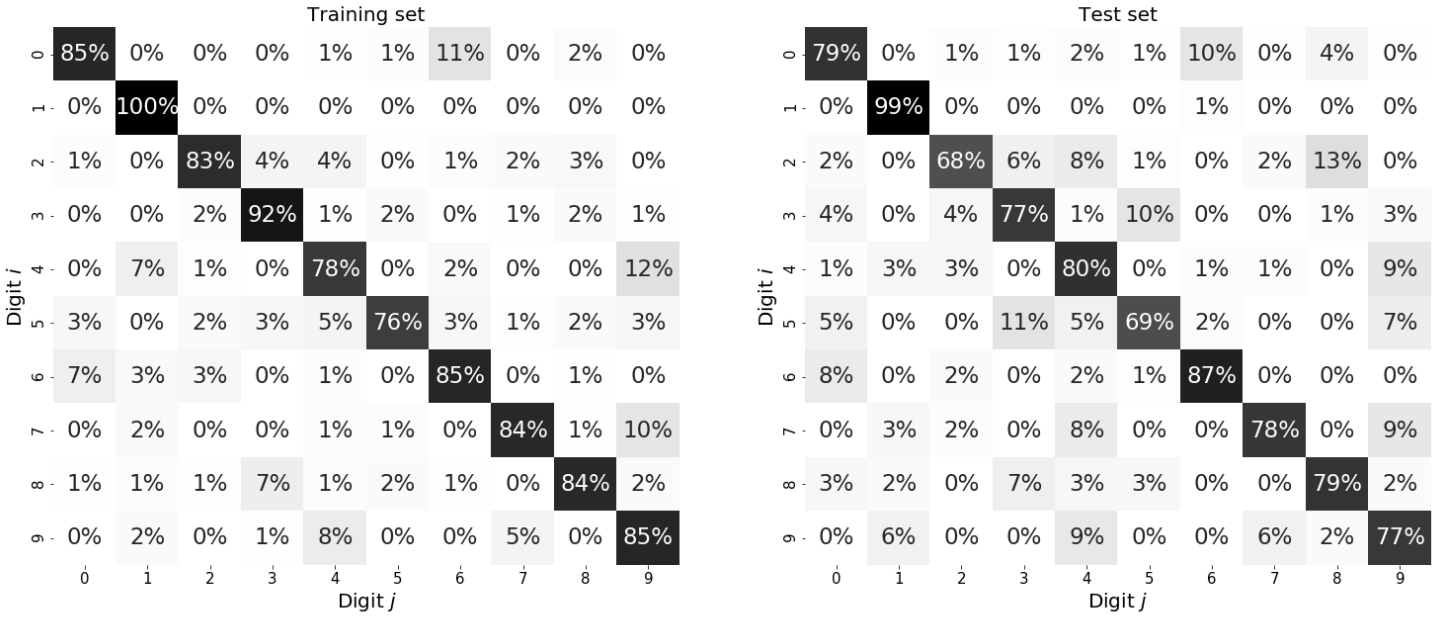


Figure 3: Confusion matrices which contain the percentage of digits i that are classified as j. The left confusion matrix shows the results from the training set, the right confusion matrix shows the results of the test set.

Comparison distance measures The results discussed in the previous paragraph are classified using the Euclidean distance. Other distance measures were compared using the `scikit-learn` package. Table 2 shows the distance measures and their accuracy. The correlation distance is the best performing distance measures. The worst performing distance measures are the Cityblock distance, L1 and the Mahalanobis distance.

1.3 Multi-class perceptron algorithm

With the Simple Distance-based Classifier, we obtained a maximum accuracy of 80.6% when using the correlation distance meausure (Table 2). Neural networks have proven to be an excellent machine learning method for classifying handwritten digits, with accuracies above 99% (Ciresan et al., 2010, e.g.,). Although Ciresan and colleagues made use of a six layer perceptron, the excellent performance of such a neural network gives rise to the question whether we can improve the accuracy of the Simple Distance-based Classifier by utilizing a neural network instead.

To do so, we implemented a multi-class perceptron algorithm with only an input layer and an output layer. The input layer consisted of $256 + 1 = 257$ nodes: 1 for each pixel of a 16×16-image and an

Table 2: Classification accuracies for each method of distance measurement used in the Simple Distance-based Classifier algorithm.

Method	Accuracy (%)
Cityblock	72.1
Correlation	80.6
Cosine	79.9
Euclidean	80.4
L1	72.1
Mahalanobis	72.1
Minkowski	80.4
SEuclidean	79.0
SqEuclidean	80.4

additional node for the bias. The output layer consisted of 10 nodes, one for each digit.

Setting up the network When reading in the data, it was already in the right format: a 1707 $\times$256 table, with for every training instance (rows) the values of each pixel of the image (columns). In order to improve speed performance of the network, we appended this matrix with a column vector (1707$\times$1) of only ones for the bias, such that we can use matrix multiplications later on.

Additionally, we also have set up three functions in order to propagate information forward into the network. First of all, we created a function to initialize weights, where the weights were stored in a 257$\times$10 network. The initialization was standard normal random, because this strategy yields values between -1 and 1 around 0. This prevents from obtaining too large values, which may cause exploding weights. Another function we wrote was used to create values for the output nodes. This was simply the dot product of the training data appended with the biases and the weights, yielding another matrix of 1707$\times$10 ((1707$\times$257)$\times$(257$\times$10)). For each training instance, we now have 10 activation values, one for each digit. To obtain the predicted digit, we made the third function, which takes the maximum value of the 10 output activations and finds the corresponding digit. This was done using the `numpy argmax`-function.

Training the network With all the necessary set-up work done, we could train the network. This was done using the following steps:

1. Initialize weights at random
2. While there are misclassified training examples
   (a) Select a misclassified example
   (b) Update the nodes that are more activated than the node of the desired output by **-x**: $weights = weights - x$
   (c) Update the node of the desired output by **+x**: $weights = weights + x$
   (d) Leave all other nodes unchanged
3. End while-loop

In order to keep up the accuracy during each training iteration, we added an additional function to the training algorithm, which divided the number of misclassifications by the total number of training instances.

Evaluating the network We can describte the performance of training the network in terms of number of iterations to converge and in terms of runtime, where runtime is based on a Dell XPS, with 16GB RAM and a Intel Core i7-9750H CPU processor. Results for two different weight initialization strategies

Table 3: Performance in terms of number of iterations and runtime for different runs of training the multi-class perceptron algorithm.

Run	Normal Weights Iterations	Runtime (sec)	Uniform weights Iterations	Runtime (sec)
1	270	8.122	272	8.316
2	281	8.954	286	8.828
3	303	9.498	235	7.392
4	381	12.182	323	10.297
5	274	8.682	318	9.994
6	272	8.681	291	9.509
7	262	8.651	308	9.736
8	312	9.896	302	9.633
9	353	11.154	284	9.104
10	270	8.836	239	7.809

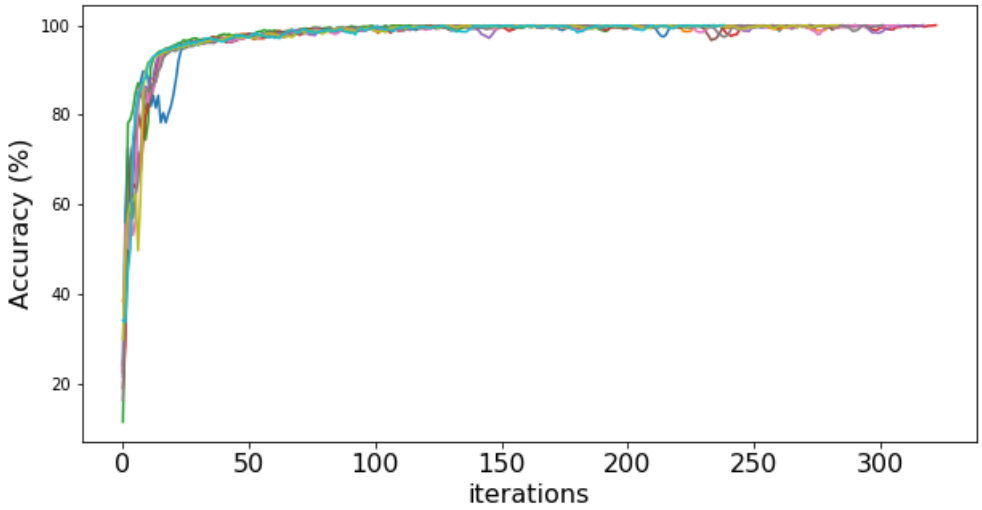


Figure 4: Development of accuracy per iteration of training the neural network, for 10 different runs.

(standard normal and uniform) are displayed in Table 3. The only thing we can conclude is that our first neural network seems pretty stable!

Another meausure of performance is the accuracy. In Figure 4, we see that for each run of training the multi-class perceptron the development of accuracy is relatively the same. All runs make a huge improvement in accuracy within the first 30 epochs of training, and develop further to 100% within the next 300 iterations. Another indication that training our network has worked out nicely!

The last and most rigorous test of performance is not based on the training set, but on the test set. Again, we defined a function, with `test_in`, weights and `test_out` as inputs. The weights are obtained from training the network. With 10 different runs of training, we obtained accuracies ranging from 86.0% to 87.3%, (mean = 86.7%, standard deviation = 0.5%). Not as good as Ciresan and colleagues, but at least better than the Simple Distance-based Classifier!

Comparision with Géron The last part of this task was to compare our algorithm with the algorithm from the textbook of Aurélien Géron (Géron, 2019). Where in our algorithm we looked at the activities of nodes that were higher than the activity of the desired output, and updated the weights to those nodes downwards, and the weights to the desired node upwards, the update rule of Géron is slightly different.

The nodes with activities that are lower than the activity of the desired output node, are leaved untouched. Here, all nodes are updated according to the following formula: $w_{i,j}^{nextstep} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$. All nodes are updated, so even the nodes with lower activities. This is definitely less efficient, since the nodes that are already lower than the desired output node are updated upwards, since $y_j - \hat{y}_j$ will be positive. Another pitfall of this algorithm is exploding weights, because $y_j - \hat{y}_j$ will generally be fairly bigger than 1. Since 257 weights are summed, this will cause the activities to explode.

Based on the above two reasons, we did not manage let the algorithm of Géron converge on the training set, but with an educated guess we can say that Géron's algorithm would have performed worse.

1.4 Linear separability

Single-layer perceptrons try to find suitable values for the weights such that training data are correctly classified. Geometrically, perceptrons try to find a hyperplane which separates the training data of different classes. Whenever there exists such a hyper-plane, classes are called linearly separable (Goodfellow, Bengio, and Courville, 2016). To determine whether the images of digit i are linearly separable from the images of digit j, for i, j in $0, 1, ...9$, such that $i \neq j$, we could try to answer this question with a theoretic approach and a computational approach.

Theoretical approach The theoretical approach applies Cover's Theorem, which can be used to calculate the probability that a randomly labeled set of N points in d-dimension space, is linearly separable according to the following equation:

$$F(N, d) = \begin{cases} 1, & \text{when } N \leq d + 1 \\ \frac{1}{2^{N-1}} \sum_{i=0}^{d} \binom{N-1}{i}, & \text{when } N \geq d + 1 \end{cases} \tag{2}$$

Applying this equation would results in a probability greater than 0.99 when testing for linearly separability between digit 1 and digit 7 ($F((252 + 156), 256) \geq 0.99$).

More general, the Cover's theorem tells us that two digits i and j are linearly separable if the sum of n_i and n_j is smaller than two times the number of dimensions. This would imply that all combinations of digits are linearly separable except the pairs 0 and 1 and 0 and 2.

Computational approach This approach assumes that the convergence of a perceptron can only be accomplished if two classes are linearly separable (Raschka, 2015). Hence, the perceptron was trained multiple times except with training data only from digit i and j, for i, j in $0, 1, ...9$, such that $i \neq j$. This resulted in 45 trained models and all of them converged, which implies that all combinations of images are linearly separable.

Digit 7 and 9 had the most iterations (91) to convergence which implies that these are the hardest digits to distinguish. This finding is in line with the finding based on the simple distance-based classifier. The digits 1 and 7 had the lowest amount of iterations (5) to convergence, which implies that these are the easiest digits to distinguish. This finding is not in line with results from the simple distance-based classifier, which argued that digits 0 and 1 are the easiest digits to distinguish. A possible reason for this difference could be the different number of observations that are in the training set. Table 7 (Appendix) shows the number of iterations that were needed for convergence and the number of observations that were used to train the perceptron for each digit pair.

Subsequently, the computation approach was used to determine whether the set of all images of i, for i in $0, 1, ..., 9$, are linearly separable from all remaining images. Hence, 10 perceptrons were trained with all training observations but the classification of images where $y \neq i$ were modified to k, where $k \in [0, 9]$ and $k \neq i$.

This resulted in 10 trained models which all converged. The model of image $i = 1$ had the lowest amount of iterations for convergence, 98. This result is in line with our previous findings which implied that digit 1 is the easiest to distinguish. Furthermore, the model of image $i = 2$ had the highest number of iterations for convergence, 619. This finding is also in line with our previous findings which implied that the digit 2 is one of the hardest digits to distinguish. An unexpected finding is that the model for image $i = 5$ only needed 181 iterations to convergence, but our previous findings suggested that this was also a difficult digit to distinguish. However, digit 5 only has 88 observations in the training set, which

could cause this relatively low amount of iterations. Table 8 (Appendix) shows for each set of images i the number of iterations that were needed for convergence.

1.5 XOR-network and Gradient Descent Algorithm

For the last task of this assignment, we implemented the XOR network with backpropagation, using the Gradient Descent Algorithm. A XOR-network has two inputs, taking values of either 0 or 1 and must output a single value, also either 0 or 1. The specific configurations are summarized in Table 4.

Table 4: All input combinations and corresponding output result

X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Note that it is not possible to use a Single Layer Perceptron such as we did at task 3, because the four possible configurations are not linearly separable. It is in no way achievable to draw a single straight line to separate the points that must output 0 ($X_1 = X_2 = 0$ or $X_1 = X_2 = 1$) from the points that must output 1 ($X_1 = 0, X_2 = 1$ or $X_1 = 1, X_2 = 0$). This is visualized in Figure 5.

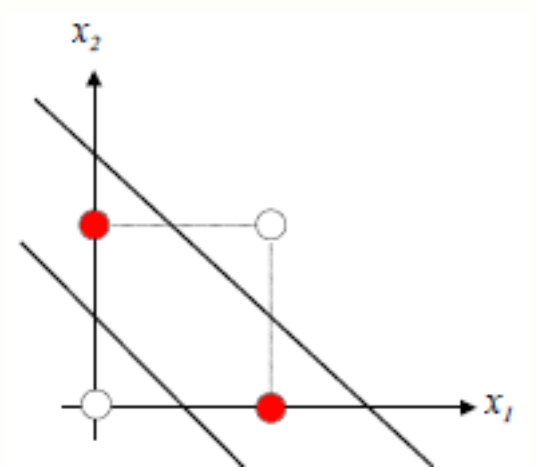


Figure 5: XOR-problem visualized

Since it is not possible to use a Single Layer Perceptron, we implemented a Multi Layer Perceptron with one hidden layer, as shown in Figure 6. The XOR-network consists of an input layer with two nodes plus an additional bias, one hidden layer of two nodes plus an additional bias and an output layer with just one node. There is also some bias, which is added to the nodes in the hidden layer and to the node in the output layer.

Setting up the network Before being able to train the network, some set up work had to be done first. In order to manage comparing different weight initalization strategies, we created three different weight functions: standard normal, uniform and uniform between -1 and 1. We also defined four activation functions and their derivatives: sigmoid, tanh, relu and softplus. An overview of the formulas of the activation functions and corresponding derivatives is displayed in Table 5.

The forward propagation of information through the network was done using matrix multiplications wherever possible. The function `xor_net()` has as input an numpy array with two entries ($[X_1, X_2]$), weights (vector of length 9), and the activation function that must be used. The weights were split up

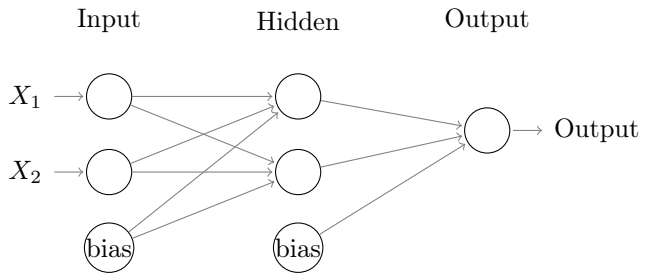


Figure 6: XOR-network

Table 5: Overview of different activation functions and corresponding derivatives

Activation function	Formula	Derivative
Sigmoid	$\frac{1}{1+\mathrm{e}^{-x}}$	$x \cdot (1-x)$
Tanh	$\frac{2}{1+\mathrm{e}^{-2x}} - 1$	$x(1-\tanh x^2)$
Softplus	$\log(1+\mathrm{e}^{x})$	$\frac{1}{1+\mathrm{e}^{-x}}$
RELU	$\begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$

into two separate vectors of length 6 and 3, which were used to compute the activations for nodes in the hidden layer and output layer respectively. Activations for nodes in the hidden layer were computed using matrix multiplication of the weights vector and another matrix, where the actual activities were obtained by applying the activation function on $[H_1,\ H_2]$:

$$\begin{bmatrix} w_0 & w_1 & w_2 & v_0 & v_1 & v_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ X_1 & 0 \\ X_2 & 0 \\ 0 & 1 \\ 0 & X_1 \\ 0 & X_2 \end{bmatrix} = \begin{bmatrix} H_1 & H_2 \end{bmatrix}$$

The activities of the nodes in the hidden layer were propagated forward to the output node, by first appending $[H_1,\ H_2]$ with a one for the bias, and subsequently performing a matrix multiplication with the weight vector of length 3 and the hidden layer activity vector:

$$\begin{bmatrix} u_0 & u_1 & u_2 \end{bmatrix} \begin{bmatrix} 1 & H_1 & H_2 \end{bmatrix}^T = \begin{bmatrix} Y \end{bmatrix}$$

The activation function was applied on Y, and when this result was bigger than 0.5, the classification of the output was 1. If the output of the classification was smaller than or equal to 0.5, the classification was 0. With that, the forward propagation in the XOR-net was finished!

We also defined three other functions:

- `mse()`: A function for the Mean Squared Error (MSE) for each run of the four possible input arrays ([0, 0], [0, 1], [1, 0], [1, 1]).
- `grdmse()`: A function to compute all partial derivatives, one for each weight in the network. For the weights that lead into the hidden nodes, this was done using different formulas. For example, the formula for $\frac{\delta E}{\delta w_1}$, the weight from X_1 to the first node in the hidden layer, was $\frac{\delta E}{\delta w_1} = (\hat{y} - d)\phi'(\text{output activity})u_1\phi'(\text{hidden activity 1})x_1$. Another example is for $\frac{\delta E}{u_2}$, the weight from the second hidden node to the output node. There we used the formula $\frac{\delta E}{\delta u_2} = (\hat{y} - d)\phi'(\text{hidden activity 2})$. All other partial derivatives were obtained in a similar way.

· `update_weights()`: This function was defined to be able to update the weights according to the following rule: $w_{new} = w - \eta w_{grd}$, where η is a learning rate and w_{grd} the vector of nine partial derivatives computed with `grdmse()`.

With all this set up work being being done, we could finally train our XOR-network!

Training the network To train the network, we exploited a comparable approach as used for task 3, but with slight differences. Instead of training while there are misclassified examples, we used a number of iterations as criterion for the while-loop. This is because we didn't know beforehand whether each combination of weight initialization strategy, activation function, and learning rate, would converge and find weights such that every training instance would be classified correctly.

We fed four possible training instances to the network - [0, 0], [0, 1], [1, 0] and [1, 1] - and computed subsequently the partial derivatives and the updated weights. Four each four training instances we computed the MSE, such that we could keep track of the error during training.

This whole process of feeding four inputs to the network, computing partial derivatives for each training instance, updating weights and computing the MSE was captured in a function, with as input the four training instances, a learning rate, the activation function and it's derivative, initial weights and the maximum number of iterations. This was done to make evaluating the network more convenient, by easily tuning several hyperparameters.

Evaluating the network Figure 7 visualizes the progression of both the accuracy on classifying the four training instances and the MSE over the several iterations. The four panes represent the results for the different activations functions. The learning rate is accustomed to the corresponding activation function, and are based on some trail-and-error process of different combinations of activation function and learning rate. However, some learning rates makes perfect sense, for example the learning rate of the Tanh-activation function($\eta = 0.5$). When evaluating the formulas, we see that Tanh $\approx 2 \cdot$ Sigmoid.

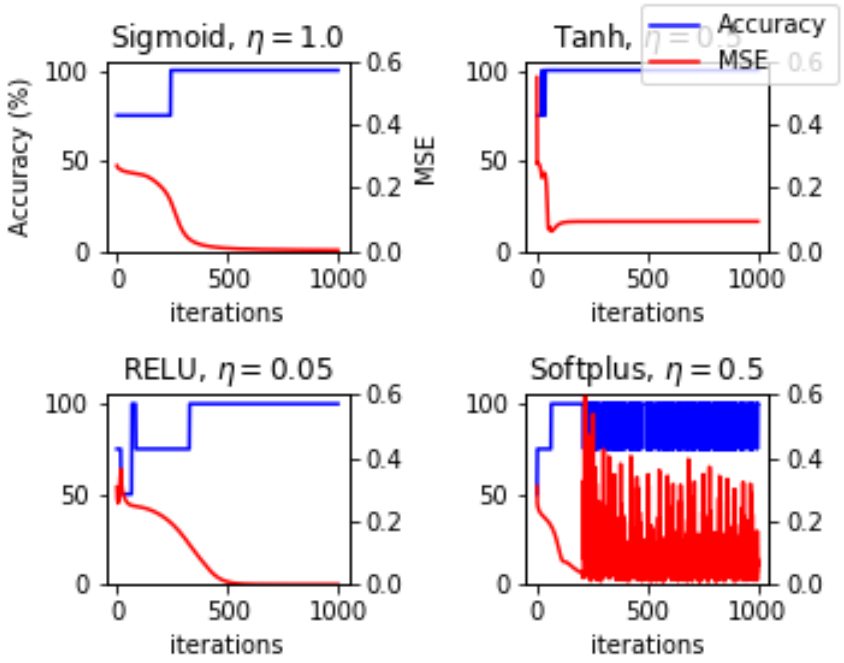


Figure 7: Progression of accuracy and MSE over iterations for the four different activation functions.

Some remarkable observations: all four trained networks seem to converge and obtain perfect accuracy, which means that each training instance is correctly classified. However, Softplus is constantly shifting between 75% and 100% accuracy, and the MSE exhibits an insane pattern from iteration 200 on. This

brings about the thought of training the XOR-network with Softplus activation to a maximum of 200 iterations.

Another striking result is that the MSE of the Sigmoid and RELU activation functions drop to approximately 0 after iteration 500, while the MSE of Tanh stabilizes around 0.1 as soon as 100 iterations are run. This makes the suggestion that the network with Sigmoid and RELU are more confident in classifying two different inputs as 1 and two ditto inputs as 0.

The third and last notable result is that the Sigmoid function appears to deliver the most stable results. At some point around 100 iterations, the RELU function achieves optimal accuracy, to drop to only 75% accuracy for the next 300 iterations, to finally reach optimal accuracy again. The Sigmoid function does not show this drop in accuracy.

Based on these results, the Sigmoid function seems to achieve the best performance when training the network. When zooming in on the Sigmoid function, we can compare different weight initialization strategies. In Figure 8 the accuracies and MSEs are displayed for standard normal weights, uniform weights [-1, 1] and uniform weights [0, 1].

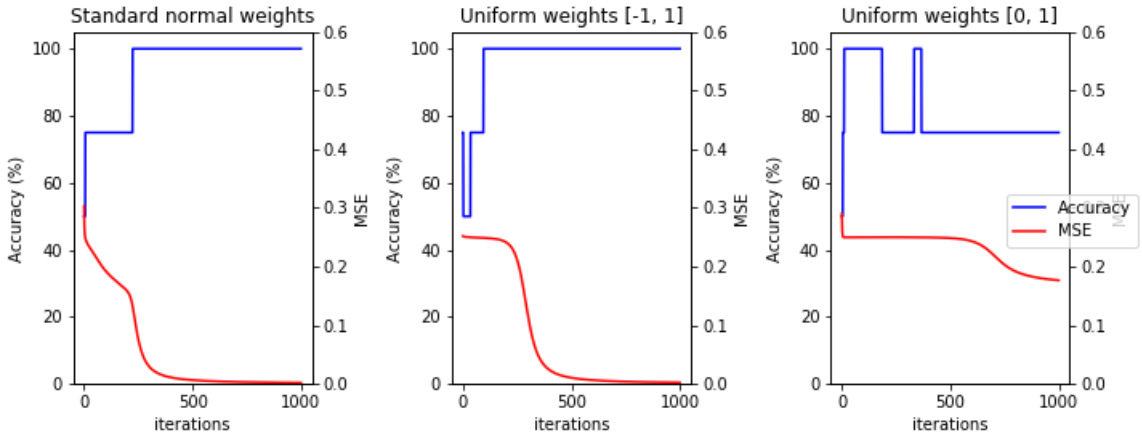
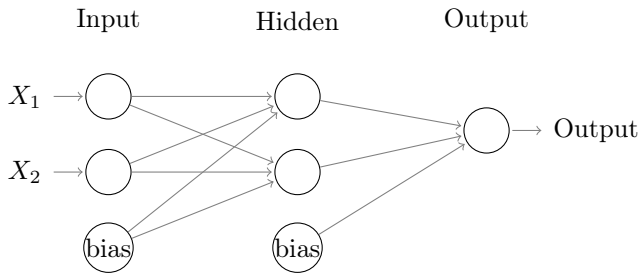


Figure 8: Progression of accuracy and MSE over iterations for the three different weight initialization strategies.

What we see is actually what we could have expected beforehand. The results of standard normal weights and uniform weights [-1, 1] are quite comparable. The convergence to 100% accuracy takes place after roughly the same number of iterations and the MSE drops to approximately 0. The comparable results make perfect sense, because the initialized values will be close together. For standard normal weights you expect more values that are close to 0, and there could be also some values that are bigger than 1 or smaller than -1. For uniform values between -1 and 1, this distribution will be more even between -1 and 1.

The real difference is for weights that are uniformly distributed between 0 and 1. Regarding accuracy, we find far from optimal results, with some wiggling between classifying 3 out of 4 and 4 out of 4 training instances correctly. In the long run, from 450 iterations onwards, the XOR-network only classifies 75% correctly. Another remarkability is the MSE, which abundantly clearly does not drop towards 0. The behavior of the network with this weight initialization strategy can be expounded by having a closer look at the intuition behind the different weights. To succesfully classify each of the four training instances, there must be negative weights in the network. Have a look at the network again:



When we set the bias for H_1 to -1.5, the bias for H_2 to -0.5, the bias for the output to -0.5 and the weight from H_1 to the output to -1, the XOR-problem is solved! With the uniformly distributed weights between 0 and 1, we clearly do not allow for negative values. Therefore, there will be no negative weights at the start of training the network, considerably hindering the learning process.

The general conclusion of this task, is that training the XOR-network is very well possible with a Multi Layer Perceptron, using the Sigmoid activation function, initializing weights at random or uniformly between -1 and 1, and a learning rate of $\eta = 1$.

Comparison with 'lazy approach' Another approach to finding weights that computes the XOR function is generating random weights. Table 6 shows the number of iterations that were needed until the XOR function was computed by randomly generated weights. Generating weights with the uniform distribution [0, 1] did not compute the XOR function in 1,000,000 iterations. Weights generated with the standard normal function seem to compute the XOR function faster than weights generated with the uniform function [-1, 1].

Table 6: The number of iterations that were need to compute the XOR function using weights that were generated random. The weights were generated using the standard normal function and the uniform function [-1, 1]. The experiment was repeated 10 times. After 1,000,000 iterations the experiment was stopped and the value 'NA' was given.

	1	2	3	4	5	6	7	8	9	10
Standard normal	57807	1764	50339	15260	156168	47840	134697	69066	12648	2195
Uniform [-1, 1]	123619	357608	220692	50352	46798	NA	NA	732084	NA	309315

References

Ciresan, Dan Claudiu et al. (2010). "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition". In: *CoRR* abs/1003.0358. arXiv: 1003.0358. URL: http://arxiv.org/abs/1003.0358.

Géron, Aurelien (2019). *Hands-on machine learning with Scikit-Learn and TensorFlow :* Second edition. Includes QR code. Beijing: O'Reilly,

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press.

Raschka, Sebastian (2015). "Single-layer neural networks and gradient descent". In: *Dosegljivo: https://sebastianraschka. com/Articles/2015_ singlelayer_ neurons. html*.

Appendix

Table 7: The number of iterations until convergence and size of the training set ($n_i + n_j$) for digit i and j, for i, j in $0, 1, ..., 9$, such that $i \neq j$

i	j	N	iterations
0	1	571	25
0	2	521	37
0	3	450	29
0	4	441	26
0	5	407	19
0	6	470	24
0	7	485	24
0	8	463	30
0	9	451	29
1	2	454	11
1	3	383	20
1	4	374	12
1	5	340	17
1	6	403	46
1	7	418	5
1	8	396	24
1	9	384	7
2	3	333	27
2	4	324	59
2	5	290	28
2	6	353	64
2	7	368	84
2	8	346	44
2	9	334	25
3	4	253	9
3	5	219	21
3	6	282	10
3	7	297	17
3	8	275	24
3	9	263	40
4	5	210	16
4	6	273	28
4	7	288	20
4	8	266	33
4	9	254	26
5	6	239	16
5	7	254	22
5	8	232	23
5	9	220	12
6	7	317	8
6	8	295	26
6	9	283	13
7	8	310	26
7	9	298	91
8	9	276	40

Table 8: The number of iterations until convergence for digit i and k, for i in $0, 1, ..., 9$ and $k \neq i$.

i	iterations
0	105
1	124
2	579
3	210
4	639
5	118
6	119
7	241
8	487
9	286