Neural Networks 2019 - 2020 Assignment - 1

Deadline March 22th 2020

1 Introduction

This paper examines the performance of different classification algorithms on a simplified version of the MNIST hand written numbers dataset. The dataset consists of 2707 images (1707 in the training-set and 1000 images in the test-set) with $16 \times 16 = 256$ pixels each. First, a simple distance based classification algorithm, utilizing different measures of distances, is applied to the data. Theory and procedure of the classification algorithm are discussed in detail. The simple distance based algorithm is then compared with a more sophisticated classification algorithm, a single layer multi-class perceptron. Here, two variants of the perceptron are implemented, utilizing two variations of the perceptron learning rule. Respective theory and training procedure of the perceptron are discussed in detail as well. Lastly, the perceptron as a classification algorithm is extended to examine a non-linearly separable problem, the XOR problem. Here, a multi-layer perceptron is used with gradient descent to train the network. As such, we examined the following topics in chronological order: Distance based classification algorithms, single layer perceptrons and linear separability, multi-layer perceptrons and gradient descent.

2 Analyzing Distances between Images

The purpose of the first task is to develop a general idea about clouds, which are collections of points in n-dimensional space. In this part of the assignment, measures about clouds of points will be introduced, with the aim of building a simple distance based classification algorithm for hand written digits. For each digit $d \in \{0, 1, ..., 9\}$ the respective cloud C_d in 256 dimensional space $\mathbb{R}^{256}$ is computed, where C_d consist of all training images that represent the digit d.
More specifically for each cloud of the 10 digits the following measures were computed:

- Center C_d: Mean of all coordinates of points within a digits' cloud C_d, where the center of each cloud is represented by a vector of means of all corresponding class observations. As such, there are 10 centers for each of the 10 digit classes.
- Radius r_d: Distance between a cloud center C_d and its furthest instance. The radius is a scalar. However, note that the clouds should not be expected to be spherical, nor that the observations are uniformly distributed in it. We'll come back to this in our analysis of the classifier.
- Number of points n_d : The number of images corresponding to each digit class.
- Distances between centers $d_{ij} = d(C_i, C_j)$: Euclidean distance between all cloud centers.

	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
Radius	15.89	9.48	14.16	14.74	14.53	14.45	14.03	14.90	13.70	16.13
Points	319	252	202	131	122	88	151	166	144	132

Table 1: Summary Table of Radii and Number of Points for each Digit Cloud C_d

From the summary in Table 1, it follows that the digit 0 has the most images and that the digit 5 the least. Furthermore, examining the radii, we can see that digit 9 has the largest radius and digit 1 the smallest. The radius can be seen as a measure of diversity of drawing a number. A higher radius means that one instance is far from the center of it's cloud, or drawn very differently. When a particular digit is drawn in a variety of ways, it is more difficult for a classifier to distinguish them correctly. Examining the distance matrix in Table 2, which represents the distances between each of the 10 cloud centers, it is apparent that the digits 0 and 1 are furthest apart from each other with a distance of 14.45 and the digits 7 and 9 are closest with a distance of 5.43. The distances between the cloud centers C_d indicate similarity between digits. Thus, digits 0 and 1 are the most dissimilar and digits 7 and 9 the most similar. Considering the shape of the numbers this is reasonable.

	0	1	2	3	4	5	6	7	8	9
0	0.00	14.45	9.33	9.14	10.77	7.52	8.15	11.86	9.91	11.49
1	14.45	0.00	10.13	11.73	10.17	11.12	10.61	10.74	10.09	9.93
2	9.33	10.13	0.00	8.18	7.93	7.91	7.33	8.87	7.08	8.89
3	9.14	11.73	8.18	0.00	9.09	6.12	9.30	8.92	7.02	8.35
4	10.77	10.17	7.93	9.09	0.00	8.00	8.78	7.58	7.38	6.01
5	7.52	11.12	7.91	6.12	8.00	0.00	6.70	9.21	6.97	8.26
6	8.15	10.61	7.33	9.30	8.78	6.70	0.00	10.89	8.59	10.44
7	11.86	10.74	8.87	8.92	7.58	9.21	10.89	0.00	8.47	5.43
8	9.91	10.09	7.08	7.02	7.38	6.97	8.59	8.47	0.00	6.40
9	11.49	9.93	8.89	8.35	6.01	8.26	10.44	5.43	6.40	0.00

Table 2: Distance matrix between each digit cloud center C_d

Furthermore, the distances between cloud centers give an indication of how well a classifier can discriminate the digits. Since the proximity of centers gives an indication of similarity it also gives an indication of discriminability. That is, digits whose centers are far apart can be easily discriminated from each other, while digits whose centers have small distances between each other are harder to classify.

2.1 Implement and Evaluate the Distance Based Classifier

Description of the Distance Based Algorithm

This section will examine a simple distance based classification algorithm. The idea behind the algorithm is very simple: examine the distances between the point and each cloud center C_d and classify the point with the digit d as belonging to the closest cloud center. Classification results can then be evaluated by computing the confusion matrix, a symmetric matrix, where columns represent true digit labels and rows represent the classified digit labels.

Euclidean Distance based Algorithm

The above mentioned algorithm is implemented on the MNIST handwritten digits set. First the 10 class centers C_d for each digit d are computed using the training set, which subsequently serve as cloud center estimates for classification on the test-set. Then the distances between each of the 1707 points (training-set) from the class centers are calculated. Distances between centers and points are measured using the Euclidean distance:

$$d(x_i, C_d) = \sqrt{\sum_{i=1}^{n} (x_i - C_d)^2} \tag{1}$$

Each handwritten digit is classified to the digit class center to which it has the minimum distance to. The resulting classification on the training set can be examined in the confusion

matrix represented in Table 3. As mentioned in the section above, the columns of the

	0	1	2	3	4	5	6	7	8	9
0	271	0	3	0	0	3	10	0	1	0
1	0	252	0	0	8	0	4	4	2	3
2	0	0	167	2	1	2	5	0	1	0
3	0	0	9	120	0	3	0	0	10	1
4	2	0	9	1	95	4	2	2	2	10
5	4	0	1	3	0	67	0	2	3	0
6	36	0	3	0	3	3	129	0	1	0
7	0	0	4	1	0	1	0	140	0	6
8	6	0	6	3	0	2	1	1	121	0
9	0	0	0	1	15	3	0	17	3	112

Table 3: Confusion Matrix for Classification on the Training-Set

confusion matrix represent true digit labels, rows represent classified digit labels and the diagonal represents the number of correctly classified labels. From the table it follows that the algorithm is not able to distinguish well between zero and six, since six and zero are frequently (36 and 10 times) confused with each other during classification. This can be expected due to the similarity of their shape, the small distance of 8.15 between the class center of zero from six, and the large radius of C_0. The algorithm also struggles with correctly classifying sevens and nines, due to the small distance of 5.43 between both class centers. From the distance matrix in Table 2, we observe that the class center for zero C_0 is close to that of six C_6, similarly C_7 is close to C_9, which verify these conclusions.

Since the classification accuracy of a model trained and tested on the same data is overoptimistic as a performance reference, the confusion matrix for the predictions on the test-set using the centers from training-set is shown in Table 4. In other words, we computed the class centers on the training-set, and then classified the test-set using these class centers. In the table we observe that zeros are often (23 times) misclassified as sixes and that the

	0	1	2	3	4	5	6	7	8	9
0	178	0	2	3	1	3	7	0	3	0
1	0	120	0	0	3	0	0	2	2	5
2	3	0	69	3	3	0	2	1	0	0
3	2	0	6	61	0	6	0	0	6	0
4	4	0	8	1	69	3	2	5	3	8
5	2	0	1	8	0	38	1	0	3	0
6	23	1	0	0	1	1	78	0	0	0
7	1	0	2	0	1	0	0	50	0	5
8	10	0	13	1	0	0	0	0	73	2
9	1	0	0	2	8	4	0	6	2	68

Table 4: Confusion matrix for classification on the test-set

digit 9 is often misclassified as a four (8 times) or seven (6 times). These results are in line with the conclusions drawn from the dsitance matrix in Table 2 and coincide with the expectations drawn about the similarity between digits based on the distance matrix.

Alternative Measures of Distance

In this section we examine our distance based classifier using alternative measures of distance, again using the minimum distance as our class prediction. A popular alternative to the Euclidean distance is the Mahalanobis distance, which takes the variance co-variance

matrix into account, considering details of the data structure. Distances between centers and points are measured using the Mahalanobis distance formula:

$$d(\mathbf{x_i}, \mathbf{c_d}) = \sqrt{(\mathbf{x_i} - \mathbf{c_d})^T \mathbf{S}^{-1} (\mathbf{x_i} - \mathbf{c_d})} \tag{2}$$

In the equation above, $\mathbf{S}$ represents the variance co-variance matrix of the data. An interesting property of the Mahalanobis distance is that if the co-variance matrix S is the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. Among the Eucledian distance and the Mahalanobis distance other measures of distances are also examined in the summary Table 5.

The summary Table 5 shows, that the Euclidean distance outperforms other measures on the test-set and has a classification accuracy of 0.80. However, the Mahalanobis distance has the highest classification accuracy of 0.96 in the training-set. This is to be expected, since the co-variance matrix S in the Mahalanobis distance takes dependencies between $\mathbf{x}$ and $\mathbf{C}_d$ into account. However, considering dependencies does not result in a higher classification accuracy than the Euclidean distance in the test-set. This is because the dependencies between training and test set are not the same, as C_d and the training set are dependent. Furthermore, it should be noted that main advantage of the Mahalabonis distance is in modelling normally distributed data. Here, we have no reason to assume that our data follows a normal distribution.

	Training-set	Test-set
Cosine	0.861	0.799
Euclidean	0.864	0.804
Manhattan	0.776	0.721
Mahalanobis	0.961	0.721
Chebyshev	0.842	0.784

Table 5: Summary of the Simple distance based classification Algorithm on the training and test set, utilizing five distance measures

3 Implementation of a Multi-Class Perceptron Algorithm

In this section we implemented two different versions of a multi-class perceptron training algorithm that aim to correctly classify the image vectors corresponding to their given label. Therefore, both versions of the algorithm belong the class of supervised training algorithms that try to approximate some function $y = f^*(x)$ which maps an input x to a category y by defining a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$ and learning the values of the parameter θ that results in the best approximation.

The implemented single layer multi-class perceptron consists of 10 nodes corresponding to the 10 digits d_i (for i in $\{0, 1, ..., 9\}$). Each node takes an image vector from the MNIST data-set plus the bias as input. Thus, the output layer consists of 10 outputs, one for each node. A schema of the perceptron can be seen in Figure 1. The output of the perceptron is calculated as

$$\hat{\mathbf{y}}_j = \phi(\mathbf{W^T x_j}) \tag{3}$$

for j in $\{1, 2, ..., N\}$ where:

- $\mathbf{x_j}$ represents the j-th input image vector which has 257 entries, where the first entry is always 1.
- $\mathbf{W}$ represents the weight matrix of size $(257, 10)$, where each column denotes the weights that belong to the node of the digit d_i.
- ϕ denotes the activation function which is applied to all outputs of the 10 nodes.

- $\hat{\mathbf{y}}_\mathbf{j}$ represents the vector of 10 output activations for the j-th input vector.
- N is the total number of image vectors in the data set.

For the output layer, the applied activation function is the softmax function which is defined as:

$$\phi(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \tag{4}$$

where z is a vector of the inputs to the outout layer. Here, we have 10 output units, so there are 10 elements in z, where i indexes the output units, so $i = 1, ..., K$. The softmax function limits the range of each output unit to be between 0 and 1. But it also divides each output such that the total sum of the outputs is equal to 1. Consequently, the outputs can be interpreted as probabilities.[1] The predicted category is the output node with the highest activation $\hat{y}_j = max\{\phi(\hat{\mathbf{y}}_j)\}$.

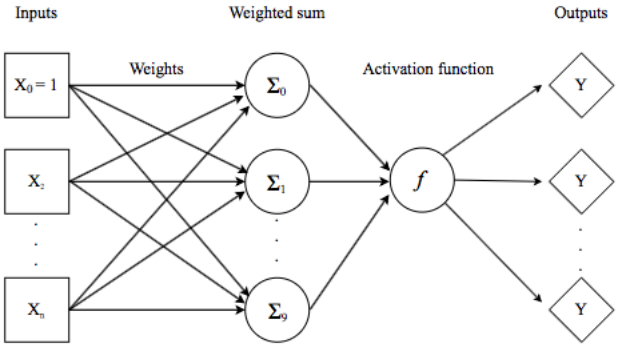

Figure 1: Single layer multi-class perceptron schema. Inputs are represented as squares, weighted sums as circles, weights as arrows from inputs to the weighted sums, the activation function f as a circle and outputs Y as diamonds.

The weights are initialized randomly using a uniform distribution in the range $[-1, 1]$. The weights are updated using two different algorithms. The first algorithm can be seen in the first grey box ("Algorithm 1"). Each iteration of the algorithm, the outputs of the network are calculated one instance at a time as described above. For each misclassified example (x_j, c_i) the weights are updated according to the rules in the box. By applying these update rules, the weights that would have led to the correct classification are getting reinforced by adding the input vector to the weights. Conversely, the weights that led to incorrect classifications and higher activations are reduced by the same input vector. The algorithm runs until there are no more misclassified examples.

[1]Note: The use of softmax as our activation function on the output layer differs from the motivation of using activation functions in multi-layer networks, where they allow the network to approximate non-linear functions. Despite the use of the function, our single-layer network still learns a linear decision surface.

Algorithm 1: Update rule from Lecture 2, Slide 36

For each misclassified example (x_j, c_i), where $\hat{y}_j \neq y_j$.

- Update weights of those nodes whose activation is higher than c_i with

$$w^{t+1} = w - x$$

- Update weight of node c_i with

$$w^{t+1} = w + x$$

- Leave all remaining weights unchanged

Algorithm 2: Perceptron learning rule from Géron's Textbook

$$w_{i,j}^{t+1} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

- $w_{i,j}$ is the connection weight between the i-th input neuron and the j-th output neuron
- x_i is the i-th input value of the current training instance
- $\hat{y}_j$ is the output of the j-th ouput neuron for the current training instance
- y_j is the target output of the j-th output neuron for the current training instance. In our case this is a vector, where all values are 0, except the node corresponding to our target class which is 1.
- η is the learning rate

The algorithm in the first box ("Algorithm 1") can be seen as a special case of the algorithm in the second box ("Algorithm 2") where $\eta = 1$ and instead of using $(y_j - \hat{y_j})$ the sign function $sgn(y_j - \hat{y_j})$ is applied. Therefore, the algorithm in the second box can be seen as the general perceptron updating rule. In particular, this means that all weights in Algorithm 1 are updated by the whole input vector, whereas in Algorithm 2 every weight vector is updated only by the factor $\eta(y_j - \hat{y_j})$ of the input vector. The input vector is distributed across the weight vectors according to the output probabilities which sum to 1. Therefore, Algorithm 2 is more likely to converge faster with a constantly decreasing number of misclassified examples. Conversely, Algorithm 1 is more likely to converge with a more fluctuating number of misclassified examples since the difference between old and new weights is larger compared to Algorithm 1.

By training both algorithms on the training set and using the updated weights to evaluate the accuracy of the algorithms on the test set and then repeating this process 50 times, we get an expected value of the convergence speed on the training set and an expected accuracy of the classifier on the test set. A comparison of both training processes is shown in Figure 3. The training process of Algorithm 1 is shown in blue, Algorithm 2 is represented by orange. The solid lines represent the mean number of misclassified examples in a particular epoch of the training. Additionally, we plotted the area between the 97.5th and 2.5th percentile of misclassified examples in each epoch of both algorithms. As expected, the number of misclassified examples is decreasing faster with Algorithm 2 and it converges

earlier on average. Furthermore, it can be seen that the 97.5th percentile of Algorithm 1 is almost always a bit higher, especially in the beginning of the process between epochs 0 and 20 and in the end of the training process between epochs 40 and 50. From the two boxplots of Figure 4 it becomes more apparent that Algorithm 2 converges faster on average than Algorithm 1. The convergence range of Algorithm 1 is between 32 and 55 epochs with a median of slightly above 40 epochs. In contrast, the convergence of Algorithm 2 ranges from 30 to 50 epochs with a median of approximately 37 epochs. If we compare the predictive accuracy on the test set, it can be seen that both algorithms have a high accuracy between 86% and 89% on average, although Algorithm 2 performs a bit better having a higher range and a slightly higher median of 87.5%. The results are shown in Figure 5. A comparison of the runtime of both algorithms is shown in Figure 6. The median runtime until convergence on the training set for Algorithm 1 is between 5 and 6 seconds. The median runtime for Algorithm 2 is only 3 seconds and therefore it is almost twice as fast as Algorithm 1.

We conclude that both versions of the single layer multi-class perceptron converge within a few seconds and not more than 55 epochs of updating the weights. If we repeat the experiment 50 times, each time initializing the weights uniformly between -1 and 1, the resulting weights lead to an expected accuracy between 86% and 89% on the test set. Overall, Algorithm 2 performs slightly better having a faster convergence and runtime and a higher accuracy on the test set. The reason for the better performance of Algorithm 2 is the slightly modified updating rule, which distributes the input values across all weight vectors according to the factor $(y_j - \hat{y_j})$ which sums to 1. Therefore, the difference between the old and new weights is smaller compared to Algorithm 1, where the difference is always the whole input vector.

4 Linear Separability of the Data Set

From the results of the previous sections we already know that the sets of images from the training set are linearly separable. The implemented multi-class perceptron algorithm terminates after a finite number of iterations, separating all classes. Therefore, the 10 sets of image vectors are linearly separable and the generalized perceptron convergence theorem holds.

The function counting theorem of Cover gives the probability that a randomly labeled set of N points in d-dimensional space (in general position) is linearly separable. It follows from the theorem that if $N < 2 \times d$ the points are almost always linearly separable. If we suppose the training set to be a randomly labeled data set in general position, then for example, with classes 1 and 7 we have $252 + 166 < 2 \times 256$. Thus, it would be very likely that these classes are linearly separable. This holds for all other combinations of classes as well, except classes 0 and 1 where $N > 2 \times d$. In this case, linear separability of the two classes would not be very likely. The same situation applies if we would want to separate one class from all remaining classes, since we would have $1707 > 2 \times 256$ for every class. Therefore, if we assume the training set to be a randomly labeled set of N points in general position, it follows from Cover's theorem that linearly separating one class from the remaining classes is almost always not possible. However, the assumption of the training set being a randomly labeled set of N points in general position does not hold. Since the data set consists of vectors that represent color intensity values of images of 10 digits we know already that these vectors are not randomly labeled and that vectors of the same digit are likely to be more similar than vectors of different digits. In fact, just by looking at the data without knowing what it represents, we can observe that almost all of the entries have a lot of -1's at the beginning and in the end. Consequently, there is some covariance in the data and the assumption of general position becomes unlikely. Due to this underlying patterns in the data we can expect the perceptron algorithm to converge after a finite number of iterations, separating all classes. The results of our implemented

perceptron algorithm for separating all combinations of classes is shown in Table 6. Indeed, the algorithm converges for all combinations of classes after no more than 10 iterations if we apply it once to every combination, each time initializing the weights uniformly in the range $[-1, 1]$. With the given set of weights, the algorithm converged for some combination of classes already after 2 iterations. Separating classes 2, 3 and 4 took the most iterations, indicating that these image vectors are more similar and thus harder to separate, given the initialized set of weights.

	0	1	2	3	4	5	6	7	8	9
0	0	4	3	4	3	7	4	5	6	2
1	4	0	2	2	2	3	3	2	2	3
2	3	2	0	10	10	6	8	9	6	4
3	4	2	10	0	2	7	3	3	9	4
4	3	2	10	2	0	5	5	6	7	4
5	7	3	6	7	5	0	5	4	8	4
6	4	3	8	3	5	5	0	4	7	4
7	5	2	9	3	6	4	4	0	4	9
8	6	2	6	9	7	8	7	4	0	4
9	2	3	4	4	4	4	4	9	4	0

Table 6: Number of Epochs until Convergence for each Combination of Classes i and j, for i, j in $\{0, 1, 2, ..., 9\}$ where $i \neq j$

The results for separating one class from the remaining classes can be seen in Table 7. Again, the algorithm converges after a finite number of iterations, separating all classes. The weights were initialized as before and the algorithm was applied only once again for every class. Separating class 1 from all other classes was the fastest with only 5 iterations of updating the weights. Separating class 2 from all remaining classes was the slowest with 94 iterations.

After applying the simple perceptron algorithm to all combinations of classes and each class against all remaining classes, we conclude that the sets of image vectors from the training set are linearly separable from each other. Cover's function counting theorem does not apply here, because the training set is neither randomly labeled, nor in general position, otherwise the perceptron algorithm would probably not converge after a finite number of iterations. Only because there are some underlying patterns in the data, the perceptron algorithm is able to linearly separate the classes.

0	1	2	3	4	5	6	7	8	9
22	5	94	23	62	31	19	30	74	46

Table 7: Number of epochs until convergence for separating each class from all remaining classes.

5 Implementation of the XOR Network

5.1 Finding XOR-Weights using Gradient Descent

The purpose of this task was to extent the Single-Layer-Perceptron to a Multi-Layer-Perceptron (MLP). The MLP unlike the SLP can solve the exclusive-or (XOR) classification problem, a prominent problem which illustrates the SLP's weakness of being unable to classify non-linearly separable data. The XOR problem, as shown in Figure 8, is a logical function, which has two inputs x_1 and x_2 and one output. Outputs are ex-

pected to be True = 1 when one (and only one) of the inputs is False = 0. In other words, if the number of True's is odd, then the XOR function should output True = 1.

x_1	x_2	Outputs
0	0	0
0	1	1
1	0	1
1	1	0

Table 8: Truth table of x_1 XOR x_2

A perceptron with two input nodes, one hidden layer with two hidden nodes and one output node was used to solve the XOR problem. Additionally, an input bias and a hidden bias was used. Two different activation functions, tanh and sigmoid, where examined. First, We initialized the weights at random using a uniform distribution on the interval $[0, 1]$. In order to ensure that the loss function of the algorithm converges to a minimum, gradient descent is applied. The loss function is needed to quantify the errors of the neural network in terms of predictions and is used for evaluation. Here the mean squared error, which measures the squared distance between predicted and true values is used. We define our loss function as follows:

$$L(y, \hat{y}) = L(y, f(x, \mathcal{W}) = (y - f(x, \mathcal{W}))^2 \tag{5}$$

Where $f(x; \mathcal{W})$ is the function learned by the neural network and $\mathcal{W}$ is the collection of weights. As f is defined as a series of compositions, we use the chain rule of differentiation to find the partial derivatives w.r.t. the individual weights, which, in turn, are the elements of the gradient. For the gradient descent, we divided our squared loss by two, for easier differentiation. As $x/2$ is a monotonous function, this doesn't change the local minimum to which our weights converge. From equation 5 it is easy to see that the first element in our chain of derivatives is

$$\frac{\partial L(y, \hat{y})/2}{\partial y} = y - \hat{y} \tag{6}$$

which we then have to multiply by the derivative w.r.t. to the weights of the layers. These derivatives are the elements that make up the gradient $\nabla f(x, \mathcal{W})$.

The gradient descent algorithm then descents along the loss landscape using the negative of the gradient. Once the slope is equal to zero, a minimum has been reached and the algorithm converges. The algorithm thus computes the local gradient of the error functions with respect to the weights and updates them to descent down the direction of the gradient, minimizing the loss function until it reaches a minimum.

The results are shown in Figure 7 and Figure 8. In Figure 7 we can see that on average, Tanh reduces the MSE in less epochs for all stepsizes except $\eta = 1$. For $\eta = 1$, the network with tanh fails to converge but instead jumps around wildly. The reason for the faster convergence and the instability is the bigger range of the derivative of tanh, as:

$$\frac{d}{dx} \tanh x = \operatorname{sech}^2(x) = 1 - \tanh^2 x \qquad \text{and}$$
$$\frac{d}{dx} S(x) = \frac{d}{dx} \frac{1}{1 + e^{-x}} = \frac{e^{-x}}{(1 + e^{-x})^2}$$

The derivative of the hyperbolic tangent therefore has the range $[0, 1]$, while the derivative of the sigmoid function has the range $[0, 0.25]$. The use of the hyperbolic tangent therefore leads to a faster updating of the weights, which reduces the number of steps necessary. For $\eta = 1$ the size of steps steps however prevented convergence which led to the instability. Furthermore, we can see in both functions that the algorithm seems to be stuck at a saddle point of the loss landscape for some time (the flat parts in the graph), until it again converges faster again. This is the case, as a linear decision surface that incorrectly classifies one of the four data-points is easy to find, while the nonlinear XOR problems takes significantly more iterations. Lastly, it is also worth paying attention to the shaded area around the mean lines. The upper bound of the area is again the 97.5th percentile of 50 iterations for

a given step. The lower bound is the 2.5th percentile. In other words, 95% of our iterations fall into the shaded area. We can see that our 50 iterations are relatively close to each other, and that for the stepsize of $\eta = 0.5$ for the tanh function all of our iterations converged in under 300 epochs. From the area we can see that the algorithm with the Sigmoid-Activation function with $\eta = 1$ converges in about 1000 epochs for all of our 50 iterations and that the algorithm with Tanh converges in about 500 epochs for $\eta = 0.5$. However, MSE was not the only statistic we tracked, as lastly the error count is at least equally important. The two statistics are different in so far as it is possible to achieve 0 errors before the MSE reaches zero, as the prediction is simply the rounded value of the output node.

In Figure 8, which shows the number of misclassifications, we can see a similar pattern with respect to the different stepsizes than we did in the Figure showing the MSE: Almost all networks with the hyperbolic tangent activation make zero prediction errors after 500 Epochs using a stepsize of $\eta = 0.5$. Our experiment therefore shows that tanh is the preferred activation function for our model. As it leads to a better model in less stepsizes.
Consequently, we continued to analyze tanh with different random weight initialization methods. We used the uniform distribution twice, with the intervals $[0, 1]$ and $[-1, 1]$. The results can be seen in Figure 9. Unsurprisingly, we can see that the larger interval leads to a large variety in errors the networks starts out with, but the mean is also significantly higher. Between 10 and 100 epochs the two intervals give very similar results, but then they diverge again. This is surprising, as one would expect that after a certain number of epochs the weights of the two functions would be the same despite different initialization. Furthermore, we know that for our XOR-Problem some of our weights have to be negative. Nevertheless does our algorithm converge faster when all weights are initialized in the interval $[0, 1]$.

5.2 Bruteforcing XOR-Weights

Lastly, we brute-forced the weights of the XOR network, i.e. we randomly created weights until our network would not produce any classification errors. From theory we know that some of our weights have to be negative, we therefore have to use a distribution where numbers under zero have a probability unequal to zero. We therefore decided to use the uniform distributions on the interval $[-2, 2]$. For our data, a XOR network would not require the bias terms. The use of the bias terms increases the count of random weights from six to nine. Not using the bias terms would therefore decrease the numbers of iterations necessary to find working weights. However, for reasons of comparability we sticked to the set-up we used in the first part of the task. In Figure 2 we can see that the number of epochs until correct weights were found varies widely. While one of our 50 iterations was able to make 0 prediction errors after 178 attempts, most of them needed between 20,000 and 40,000 attempts. The expected convergence time of our gradient descent algorithm is therefore significantly faster than the random weights initialization.

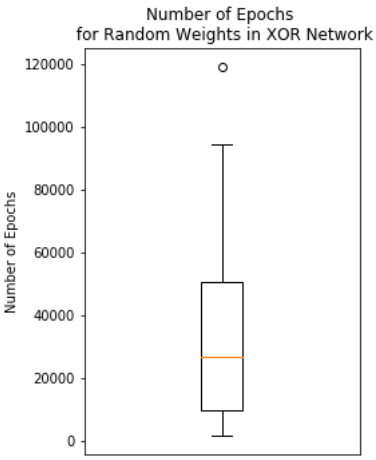

Figure 2: Number of epochs for Completely Random Weights for XOR Network for 50 iterations.

Appendix

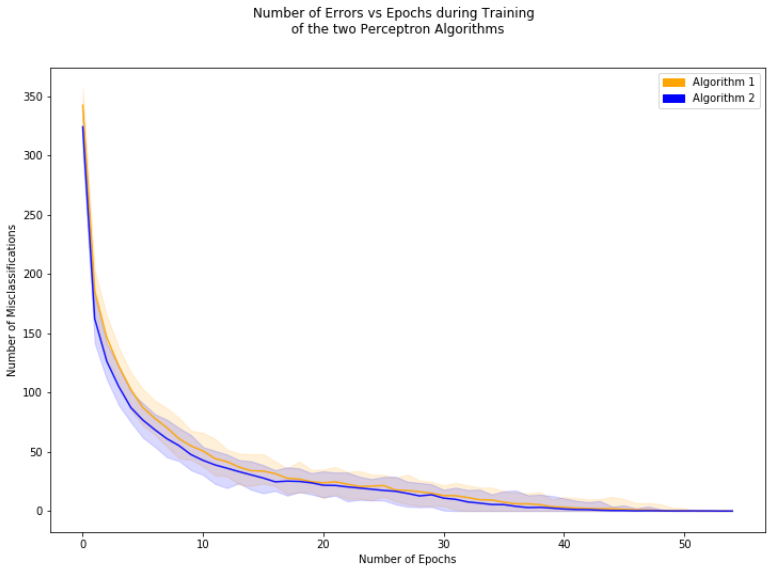

Figure 3: Number of Misclassifications vs. Epochs for the two Algorithms of the Multi-Class Perceptron. The line shows the mean over 50 iterations. The upper bound of the shaded area is the 97.5th percentile of these 50 iterations. The lower bound the 2.5th .

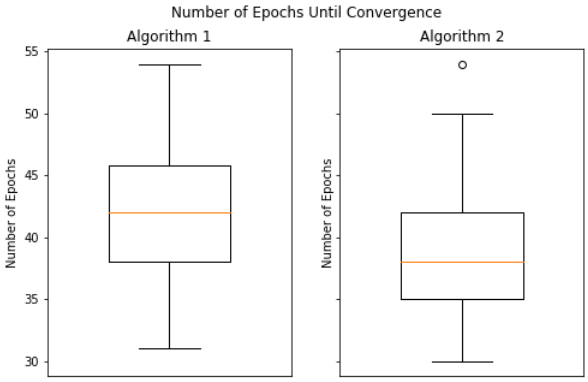

Figure 4: Boxplots epochs before convergence for the two multiclass perceptrons for 50 iterations.

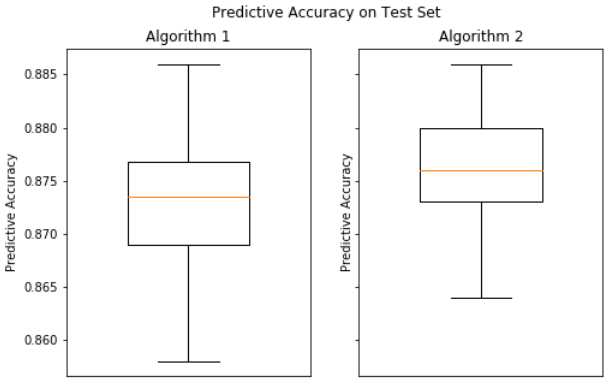

Figure 5: Boxplots of the prediction accuracy on the test set for the two multiclass perceptrons for 50 iterations.

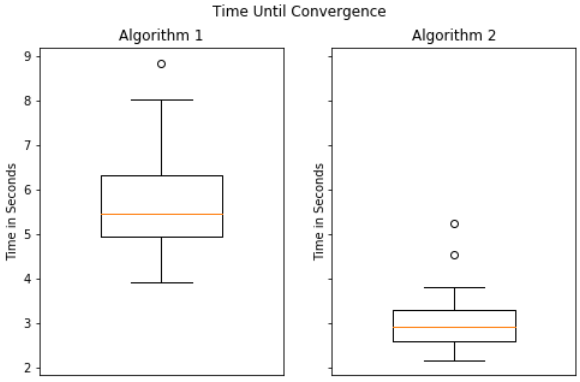

Figure 6: Boxplots of the Time until convergence in seconds for the two multiclass perceptrons for 50 iterations.

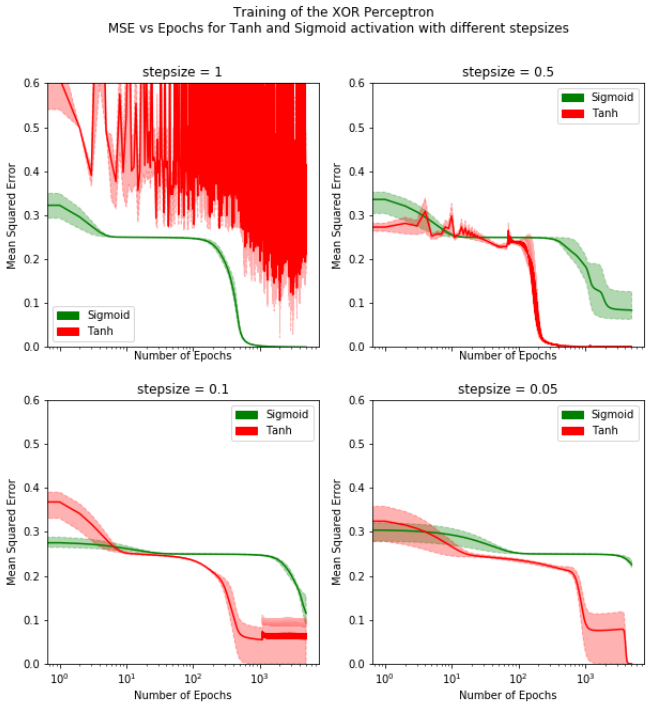

Figure 7: MSE vs. Epochs for Tanh and Sigmoid Activation functions with stepsize $\eta \in \{0.1, 0.2, 0.5, 1\}$. The line shows the mean over 50 fittings with different random initializations. The upper bound of the coloured area is the 97.5th percentile of the errors achieved in these 50 fittings, the lower bound is the 2.5th percentile.

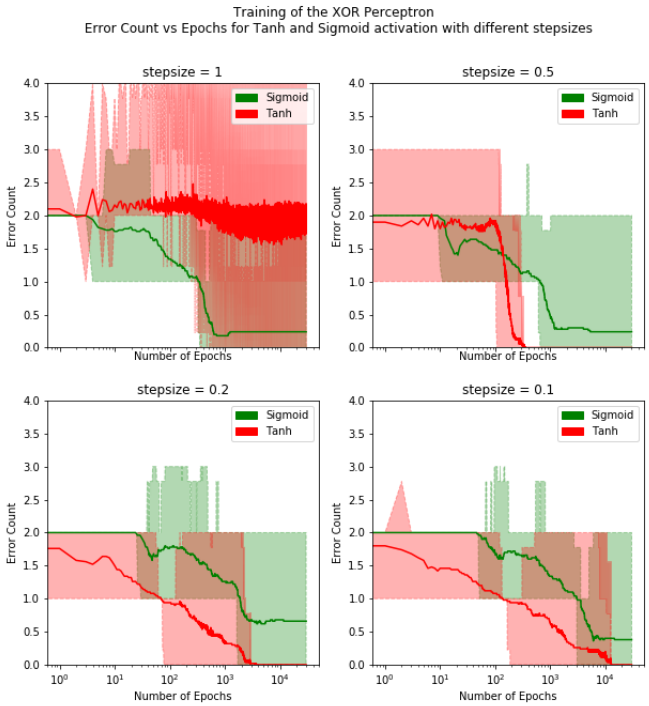

Figure 8: Error count vs. Epochs for Tanh and Sigmoid Activation functions with stepsize $\eta \in \{0.1, 0.2, 0.5, 1\}$. The line shows the mean over 50 fittings with different random initializations. The upper bound of the coloured area is the 97.5th percentile of the errors achieved in these 50 fittings, the lower bound is the 2.5th percentile.

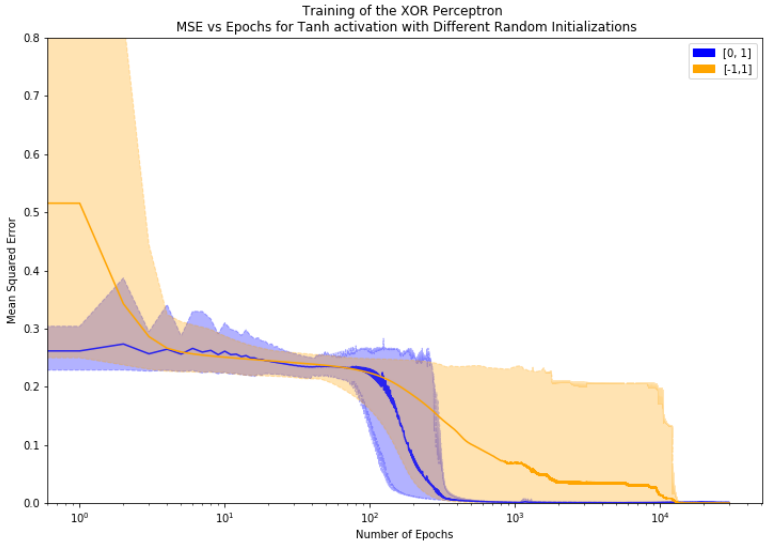

Figure 9: Comparison of Initialization on two random Intervals for tanh activation function for 50 iterations.