Report 1

Deep Learning and Neural Networks

Yizhen Dai (s2395479),
Anne Liebens (s1838458),
H.C. (Bram) Otten (s2330326)

September 27, 2021

Contents

1 Introduction

In this assignment, we developed and evaluated several algorithms for image classification: classification based on distance (task 1 and 2), multi-class perceptrons (task 3), and the simple perceptron algorithm for linear separability (task 4). The data we used are images from the famous MNIST data set, which consists of 2707 handwritten digits. Each image consists of a vector of 256 numbers (ranging from -1 to 1) that represents a 16x16 image. The data is further split into train and test sets (with 1707 and 1000 images respectively).

We also implemented a network that learns the XOR operation via gradient descent (task 5).

2 Methodology

2.1 Classification based on distance (task 1, 2)

In this task, the average value of the 256 pixels in every training image for every digit is considered to be a sort of digit template, hereafter called the center of a cloud per digit. For each new (test) image, the Euclidean distance between its pixel values and the cloud centers can be calculated. The proposed algorithm classifies the new image as the digit whose cloud's center is closest.

This distance-based algorithm was evaluated by 0/1-loss using the test set. 0/1-loss is the ratio of incorrectly classified digits out of the total data set.

2.2 Multiclass perceptron (task 3)

In this task, a single layer multi-class perceptron training algorithm was implemented. There are 10 possible outcomes. As the input vector x was appended with an additional vector of ones, the dimensions of the weight matrix were 257 x 10. To determine the value of the 10 output nodes for a certain example, the weight matrix was multiplied with the input vector. The output node with the highest value (highest activation) determined the class of the input instance.

Weights were initialized at random. Three algorithms with different weight update rules were created:

- The first algorithm is referred to as the simple multi-class perceptron. For the simple multi-class perceptron, the weight matrix was updated as long as misclassifications were found. This was done by choosing one misclassified instance and updating the weights connected to the output nodes that were activated more than the target output node. The weights of these nodes were decreased by the value of the input node: $w_{\text{output node}} = w_{\text{output node}} - x$. The weights connected to the target output node were also updated, but

instead of decreasing their values, they were increased by x.

- The second algorithm is the Hebbian perceptron, as the update rule of this algorithm is inspired by Hebb's rule. In the Hebbian perceptron, the weights were initialized with a uniform distribution on $[0, 1]$ and were updated as follows for wrongly classified examples: $w_{\text{output node}} = w_{\text{output node}} + (\text{target} - \text{output})x$ where the target was 1 for the weights connected to the output node that should have been activated the most and 0 for the other nodes and x the input node for a wrongly classified example. As we were working with a target output of 0 and 1, it was necessary to add 1 to all the training inputs to make sure they were positive and then normalize them so that they would only take on values between 0 and 1. To ensure that the output would fall between 0 and 1, the outcomes were normalized as well.

  During the implementation of the Hebbian perceptron algorithm it was found that updating the weights until all training examples were classified correctly took a very long time. Therefore the weights were considered to be converged when less than 5% of the training set was misclassified. As overfitting is often a problem in classification algorithms, this loss of accuracy on the training set was not considered to be a problem.

- A third algorithm was written to improve the Hebbian perceptron. In this algorithm only nodes that had higher activation than the target node were updated with Hebb's update rule. The normalization procedures were not used for this implementation. This approach, as a more deliberate implementation of Hebb's update rule as described by Géron, is shown below:

  – For nodes that had higher activation than the target node, the "target" value was set to the value of the output node that should have been activated the most.

  – For the node that should have been activated the most, the "target" was set to the value of the node that was actually activated the most.

  – Additionally, $(\text{target} - \text{output})x$ was scaled by a distance value: the difference between the maximum and minimum activation.

  The weight updates were continued as long as there were misclassified training examples.

2.3 Linear Separability (task 4)

In a different manner than in task 3, we are going to train binary classification models instead of using a multi-class perceptron algorithm. To determine if the set of all images of the digit i is linearly separable from the set of all images of the digit j, for i,j in $0, 1, ..., 9$ with $i \neq j$, we used Cover's Theorem. Cover's Theorem in high dimensional spaces gives the estimation of the chance of linear separation for a randomly labeled set of N points in d-dimensional space:

- If the number of points in d-dimensional space is smaller than $2 \times d$, they are almost always linearly separable
- If the number of points in d-dimensional space is bigger than $2 \times d$, they are almost always *not* linearly separable

Then we implemented the simple perceptron algorithm and applied it to all pairs of sets of images from classes i and j in the training set. Further, we used the same algorithm to linearly separate the set of all images of i (for i in $0, 1, ..., 9$) from all remaining images.

The simple perceptron algorithm for binary classifications was implemented following the following steps:

1. $d = \begin{cases} 1 \text{ when } y = i; \\ -1 \text{ otherwise} \end{cases}$ where y is the target value.
2. Initialize weight w randomly.
3. Activation $= \sum(w * x) + b$, where x is the training data and b is the bias/intercept.
4. $\hat{d} = \begin{cases} -1 \text{ when activation} \leq 0; \\ 1 \text{ when activation } > 0. \end{cases}$
5. If $d \neq \hat{d}$, then $w = w + \eta \times d \times x$, where η is the learning rate.

2.4 The XOR network and gradient descent (task 5)

The loss function used here is the mean squared error (MSE), defined as $1/4 \sum_{i=1}^{4} (y_i - d_i)^2$, where y_i is prediction corresponding to X_i and d_i is the desired prediction for X_i with $i \in 1, 2, 3, 4$. ($X = ((0, 0), (0, 1), (1, 0), (1, 1))$ and $d = (0, 1, 1, 0)$ – we do not split into a train and test set.)

The activation (φ) functions used here are the sigmoid function (more correctly the logistic function), rectified linear unit (ReLU), and hyperbolic tangent (TanH) functions. Their equations and derivative terms are shown below :

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}; \frac{\partial \text{sigmoid}(x)}{\partial x} = \text{sigmoid}(x)(1-\text{sigmoid}(x));$$
$$\text{TanH}(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}; \frac{\partial \text{TanH}(x)}{\partial x} = 1-\text{TanH}(x)^2;$$
$$\text{ReLU}(x) = \begin{cases} 0 \text{ for } x \le 0; \\ x \text{ for } x > 0; \end{cases} \quad \frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 0 \text{ for } x \le 0; \\ 1 \text{ for } x > 0. \end{cases}$$

The XOR network has two input nodes, a single hidden layer with two nodes (y_1 and y_2), and one output node (y).The structure of the network is shown in figure 1.

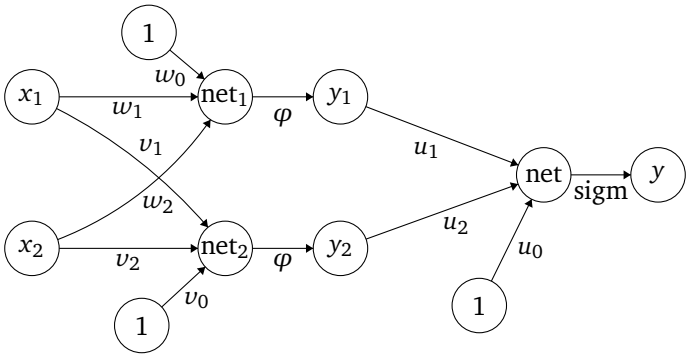


Figure 1: structure of our XOR neural network. Note that the activation function between the hidden and output layer is always the sigmoid function.

The following formulas illustrate the process in details. There are nine weights in total.

Given an input ($(x_1, x_2) = X_i$ for $i \in \{1,2,3,4\}$), weights ($u_0, u_1, u_2, v_0, v_1, v_2, w_0, w_1, w_2 \in \mathbb{R}$), and activation function (a φ from above), a prediction y_i is made (using forward propagation) as follows:

$$y_{i,1} = \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i});$$
$$y_{i,2} = \varphi(v_0 + v_1 x_{i,1} + v_2 x_{2,i});$$
$$y_i = \text{sigmoid}(u_0 + u_1 y_{i,1} + u_2 y_{i,2}).$$

The idea behind gradient descent is to update the weights in the direction that reduces MSE most. The number of updates performed is an arbitrary number generally large enough for convergence. For any weight x, $x_{\text{after update}} = x - \eta(\partial \text{MSE}/\partial x)$ with some learning rate $\eta \in \mathbb{R}_{>0}$. That partial derivative is the hard part; let us make it more concrete with a weight connecting the hidden to the output layer (viz. the weight between y_1 and net (which is $\varphi^{-1}(y)$)):

$$\begin{aligned} \frac{\partial \text{MSE}}{\partial u_1} &= \frac{\partial}{\partial u_1}\frac{1}{4}\sum_{i=1}^{4}(y_i - d_i)^2 = \frac{1}{4}\frac{\partial}{\partial u_1}\sum_{i=1}^{4}(\varphi(u_0 + y_{i,1}u_1 + y_{i,2}u_2) - d)^2 \\ &= \frac{2}{4}\sum_{i=1}^{4}(\varphi(u_0 + y_{i,1}u_1 + y_{i,2}u_2) - d) y_{i,1} \frac{\partial \varphi(u_0 + y_{i,1}u_1 + y_{i,2}u_2)}{\partial u_1} \\ &= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i) y_{i,1} \frac{\partial y_i}{\partial u_1}. \end{aligned}$$

Note: the loss function we are using is MSE instead of the total squared error.

The other weights between the hidden and output layer differ only slightly in an application of the chain rule:

$$\frac{\partial \text{MSE}}{\partial u_0} = \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)\frac{\partial y_i}{\partial u_0};$$
$$\frac{\partial \text{MSE}}{\partial u_2} = \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i) y_{i,2}\frac{\partial y_i}{\partial u_0}.$$

The partials of weights in the first layer are computed a bit differently. Here is a representative one in more detail again:

$$\begin{aligned}
\frac{\partial \text{MSE}}{\partial w_1} &= \frac{\partial}{\partial w_1}\frac{1}{4}\sum_{i=1}^{4}(y_i - d_i)^2 = \frac{1}{4}\frac{\partial}{\partial w_1}\sum_{i=1}^{4}(\varphi(u_0 + y_{i,1}u_1 + y_{i,2}u_2) - d)^2 \\
&= \frac{1}{4}\frac{\partial}{\partial w_1}\sum_{i=1}^{4}(\varphi(u_0 + \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})u_1 + \varphi(v_0 + v_1 x_{i,1} + v_2 x_{2,i})u_2) - d)^2 \\
&= \frac{2}{4}\sum_{i=1}^{4}\varphi(u_0 + \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})u_1 + \varphi(v_0 + v_1 x_{i,1} + v_2 x_{2,i})u_2 - d)x_{i,1} \ldots \\
&= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)x_{i,1}\frac{\partial y_i}{\partial w_1}\frac{\partial \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})}{\partial w_1} \\
&= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)x_{i,1}\frac{\partial y_i}{\partial v_2}\frac{\partial y_{i,1}}{\partial v_2}.
\end{aligned}$$

And here are the others:

$$\begin{aligned}
\frac{\partial \text{MSE}}{\partial w_0} &= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)\frac{\partial y_i}{\partial w_0}\frac{\partial \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})}{\partial w_0}; \\
\frac{\partial \text{MSE}}{\partial w_2} &= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)x_{i,1}\frac{\partial y_i}{\partial w_1}\frac{\partial \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})}{\partial w_2}; \\
\frac{\partial \text{MSE}}{\partial v_0} &= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)\frac{\partial y_i}{\partial v_0}\frac{\partial \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})}{\partial v_0}; \\
\frac{\partial \text{MSE}}{\partial v_1} &= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)x_{i,2}\frac{\partial y_i}{\partial v_1}\frac{\partial \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})}{\partial v_1}; \\
\frac{\partial \text{MSE}}{\partial v_2} &= \frac{1}{2}\sum_{i=1}^{4}(y_i - d_i)x_{i,2}\frac{\partial y_i}{\partial v_2}\frac{\partial \varphi(w_0 + w_1 x_{i,1} + w_2 x_{2,i})}{\partial v_2}.
\end{aligned}$$

We experimented with a few learning rates for each of the aforementioned activation functions.

We also tested how results are effected by the initialization of the weights. We tried three initialization techniques:

- Uniform initialization: The weights will be sampled from a uniform distribution over the interval [-2.5, 2.5].
- Glorot's initialization strategy: The first six weights (to the hidden layer) are sampled from $N(0, 1/6)$, while the last three (to the output layer) come from $N(0, 1/4.5)$.
- Xaivier's initialization strategy: All weights are sampled from $N(0, \sqrt{2/6})$.

The uniform initialization is also used in a "lazy" alternative to gradient descent: trying out the classification performance of the randomly initialized weights.

3 Results

3.1 Classification based on distance (task 1, 2)

First, we looked at the radii of the clouds as shown in table 1. The radius of a cloud c_d for digit d is defined as the maximum of the (Euclidean) distances between the center of c_d and a training image representing digit d.

Digit	0	1	2	3	4	5	6	7	8	9
Radius	10.0	3.3	9.8	9.3	7.5	11.9	8.3	6.1	7.3	8.1

Table 1: radius per digit cloud – the radius of a cloud c_d for digit d is defined as the maximum of the (Euclidean) distances between the center of c_d and a training image representing digit d.

Next, we can look at the (Euclidean) distances between the clouds, shown in figure 2.

We expect digits to be misclassified as belonging to a relatively similar cloud relatively often. The relatively similar clouds appear to be 3-5, 4-9, 7-9, and 8-9.

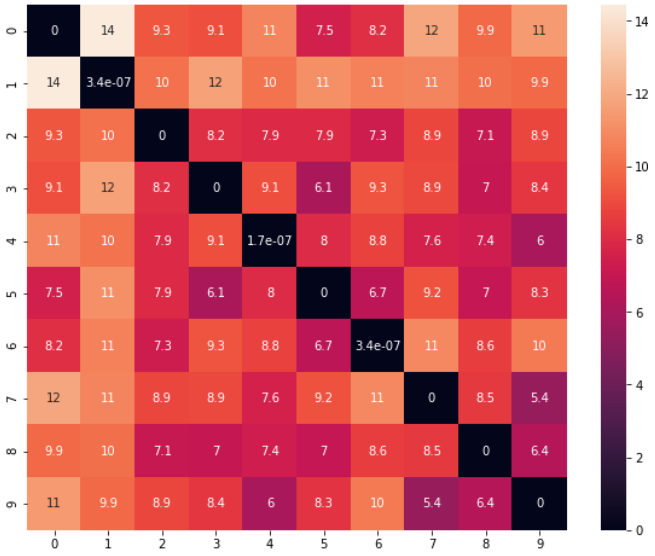


Figure 2: (Euclidean) distances between digit clouds. Darker values indicate smaller numbers / less distance.

The algorithm obtains a 0/1-loss score of 0.196 on the test set. This means $(1 - 0.196) \times 100\% = 80.4\%$ of test images were correctly classified as the digit they represent.

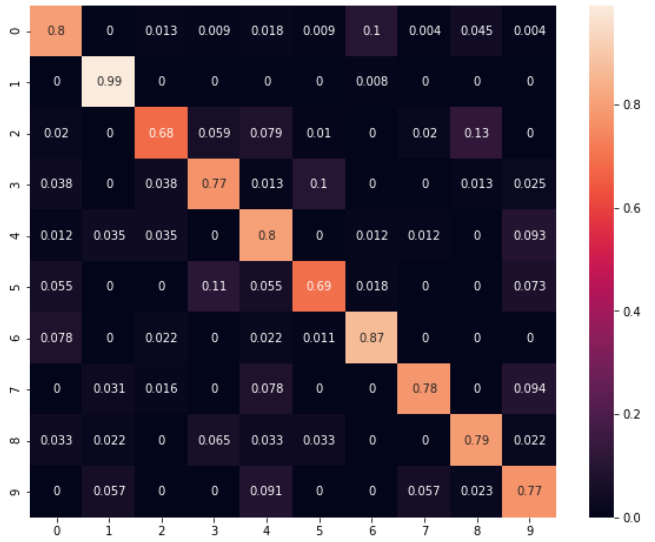


Figure 3: [proportion of test digits classifacations using distance-based algorithm. Vertical axis indicates true value, horizontal axis the predicted value. Darker values indicate lower proportions.

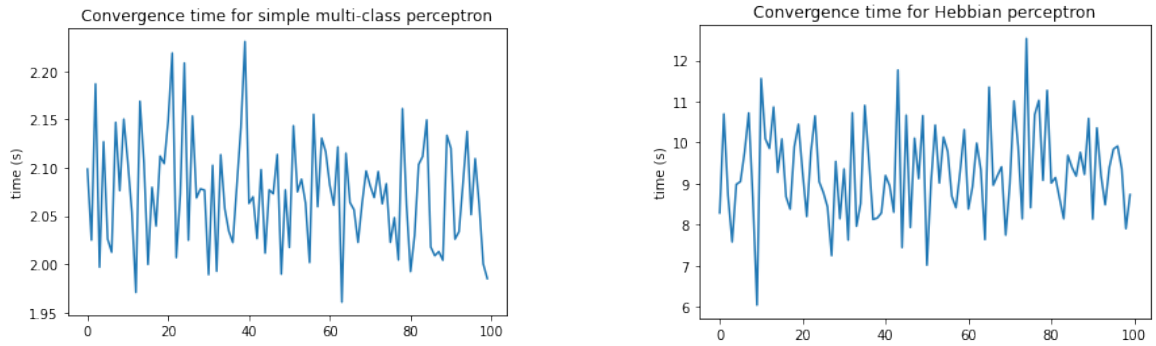
In accordance with expectations, digits are relatively often classified as digits with similar clouds, as shown in figure 3. The pairs with relatively similar cloud centers described above (3-5, 4-9, 7-9, and 8-9) indeed are relatively often classified as each other by the algorithm. 0-6 and 2-8 are mildly surprising, but also had similar cloud centers. Note the asymmetry of 2-8 and 8-2: the 2s in the test set are often confused for an 8 but none of the 8s is confused for a 2.

3.2 Multiclass perceptron (task 3)

For all three perceptron algorithms, the convergence speed and accuracy were tested.

The simple multi-class perceptron and Hebbian perceptron algorithms were run 100 times and the results are shown in figure 4. The average convergence speed of the simple multi-class perceptron, seen in figure 4a was 2.074 seconds, with a standard deviation of 0.057 seconds. Figure 4b shows the results of the convergence experiment for the Hebbian perceptron. The average convergence speed of the Hebbian perceptron was 9.283 seconds, with a standard deviation of 1.131 seconds. This means that the Hebbian perceptron is much slower than the simple multi-class perceptron. Additionally, the weight convergence for the Hebbian perceptron stops as

soon as 95% of the training instances is correctly classified, while for the simple multi-class perceptron 100% accuracy can be reached for the training set. Therefore, the training process of the simple multi-class perceptron could be considered to be better.



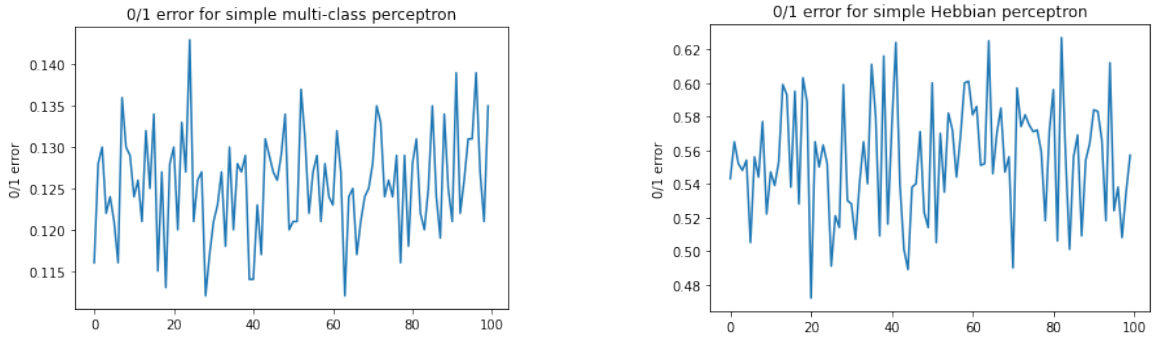
(a) the simple multi-class perceptron.

(b) the Hebbian perceptron.

Figure 4: convergence speed on the training set.

The accuracy could be tested by comparing the provided labels of the test set to the labels produced by the algorithms. The simple multi-class perceptron converges to weights that result in no misclassifications for the training set, however, the average number of misclassifications for the test set lies at 125.48 with a standard deviation of 6.263. This means that more than 10% of the 1000 test instances is misclassified, which is also illustrated in Fig. 5a. This figure shows the 0/1 error for the misclassifications.

For the Hebbian perceptron, figure 5b shows 0/1 error of the classifications for the Hebbian perceptron. Accuracy on the training set is not as high for the Hebbian perceptron as for the simple multi-class perceptron (95% versus 100%), however, it was expected that less accuracy on the training set might lead to better generalization on the test set. Figure 5b shows that this is not the case: the 0/1 error oscillates around 0.5, which means that more then half of the test examples is misclassified, which can be considered to be an undesirable result.



(a) simple multi-class perceptron.

(b) Hebbian perceptron.

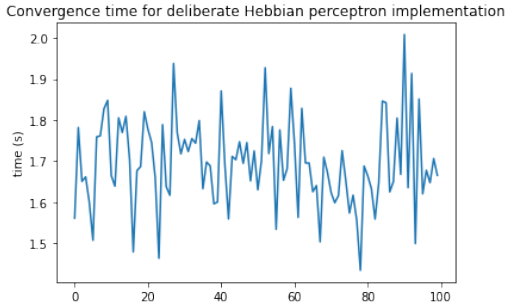
Figure 5: 0/1 error on the test set.

Because the Hebbian perceptron algorithm performed poorly, informal experimentation was done with an adaptive learning rate for the training set. Different learning rates and adaptive learning rates were tried. For the adaptive learning rate, the number of iterations after which the learning rate was changed was varied as well. The number of misclassifications after each weight update was monitored. The increasing adaptive learning rate worked best: it was found that the number of misclassifications kept oscillating between 55 and 80 with an occasional drop, where 45 misclassifications was the lowest number recorded. The number of iterations after which the learning rate was increased did not matter much for the convergence speed. However, the Hebbian perceptron implementation that was attempted first reached a similar oscillation effect, meaning that the convergence was not affected by the learning rate and therefore further testing with the learning rate was not considered.

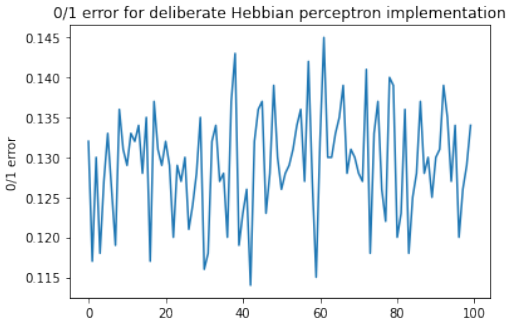
The poor performance of the Hebbian perceptron could be due to the fact that all weights are updated and not just the weights connected to output nodes that were activated more than the target output node. These extra weight updates can break the performance of the network on the training examples that were updated correctly before: it could be that because all the weights are adjusted the output values change too much, which leads to a situation where examples that were classified correctly before are misclassified with the new weights. It could also be that because all of the weights are updated, the weights will at some point diverge so much that increasing and decreasing their values will not change the output enough to reach different conclusions for the classification.

Figure 6 shows results for convergence speed and classification experiments for the more deliberate implementation of the Hebbian perceptron, which was created to improve the Hebbian perceptron, each repeated 100 times. Average convergence speed was 1.694 seconds, with a standard deviation of 0.106. On average, 129.25 images were misclassified, with a standard deviation of 6.676. Figure 6b shows the 0/1 error for the deliberate implementation of the Hebbian perceptron and it can be seen that the performance of this algorithm is similar to the performance of the simple multi-class perceptron.

It can be concluded that the more deliberate Hebb's rule algorithm is the fastest algorithm and only has three misclassifications more than the simple multi-class perceptron. Further testing would need to be done to determine whether the classification power of the simple multi-class perceptron is significantly better than the classification power of the deliberate Hebb's rule perceptron.



(a) convergence speed on the training set.



(b) 0/1 error on the test set.

Figure 6: convergence speed and misclassifications for deliberate Hebbian perceptron implementation.

3.3 Linear Separability (task 4)

The space dimension in our case is 256 since each digit is made of 256 pixels. Based on Cover's Theorem, we compare the number of total data points and $2 \times d = 512$.

For any two-digit pair (i, j), the Cover's Theorem shows that (0,1) and (0,2) are almost always not linearly separable if two digits are randomly distributed in the space. Other combinations are almost always linearly separable. However, with a learning rate of 0.1, the simple perceptron algorithm can linearly separate all the digit sets. The 0/1-loss for all the combination are shown in figure 7. We can see the 0/1-loss reaches 0 within 100 epochs except for pair (7,9). The reason that Cover's Theorem did not show that pairs (0,1) and (0,2) are linearly separable is that the Theorem is only an estimate for randomly distributed points while the distribution for our data points are not random. Also, the total number of points for (0,1) and (0,2) are 571 and 521, very close to the threshold (512).

For any digit and the remaining digits, the Cover's Theorem shows that they are almost always not linearly separable. However, the simple perceptron algorithm can linearly separate every digit from the remaining digits. The 0-1 errors for all the combination are shown in figure 8. We can see the 0/1-loss reaches 0 for digit 0,1,3,5,6,7,9 within 1000 epochs. Again, the reason is that the theorem is an estimate for randomly distributed points, which is not our case.

3.4 The XOR network and gradient descent (task 5)

We trained the network with quite a few combinations of parameters – the number of initialization strategies times the number of learning rates times the number of activation functions times the number of random seeds tried out. The results, with 30000 epochs and seed of 42, are shown in Table 2.

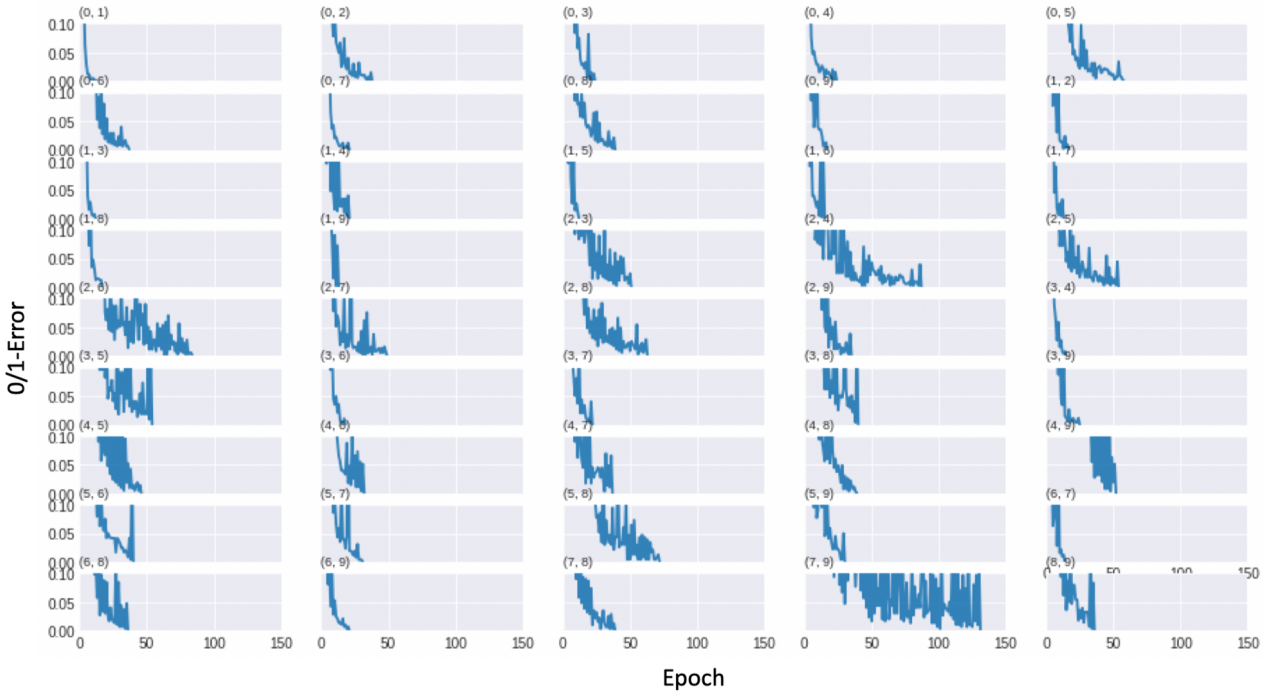


Figure 7: 0/1-Error for all the digit pairs in training dataset. The upper left tuple in each small figure shows the digit pair for that small figure.

By plotting the MSE against the number of performed updates, we see how fast and smooth the algorithm (does not) converge. There are too many of these plots to show, so only a selective representation is displayed in figure 9.

We are not surprised to see that the learning rate correlates with convergence rate. The TanH generally converges faster, more consistently, and to a lower MSE than the sigmoid. The ReLU fails to classify more than two of the four samples correctly in our experiments. The uniform weight initialization is clearly favorable for the sigmoid activation function, while the Glorot initialization strategy seems to make the ReLU work better. The Xaivier initialization strategy does not seem to work particularly well in these experiments.

It takes around 12000 random uniform initializations ([-2.5, 2.5]) to find a set of weights that correctly classifies all samples using the sigmoid activation function (this is that “lazy” approach, without gradient descent).

4 Conclusion

- The distance-based classification algorithm performed surprisingly well for its simplicity – it only uses average pixel values in a training set and classifies 80.4% of test images as the true digit.
- The multi-class perceptron performed best when only weights are updated for nodes that are activated more than the target node. Additional improvements for convergence speed include taking the target output into account when updating the weights and scaling with a distance measure. Accuracy on the test set lies between 87.1% and 87.4%, where the improvements for convergence speed resulted in slightly less accuracy, however, further testing is needed to investigate whether this difference is statistically significant. It was found that when the weights for all output nodes are updated, not all training examples will be classified correctly, possibly because too many weights are changed and the output nodes of previously correct classifications change so much that they are misclassified with the new weights. The comparison of the digit classification algorithms is shown in table 3. The simple multi-class perceptron works better than the distance-based classifier, which is not surprising since it is a more complicated algorithm.
- In task 4, we explore the linear separation for the images. The simple perception algorithm can linearly separate all the digit pairs and each digit from the remaining digits. Cover’s Theorem can roughly estimate the chance of two point sets being linearly separable if they are randomly distributed in the space. This is not our case since our data points are not random. Therefore, we cannot use Cover’s Theorem to predict linear separation.
- The proposed neural network structure is sufficient for solving the XOR problem (task 5). Whether correct weights are learned depends on choice of activation function, learning rate, number of iterations, and weight

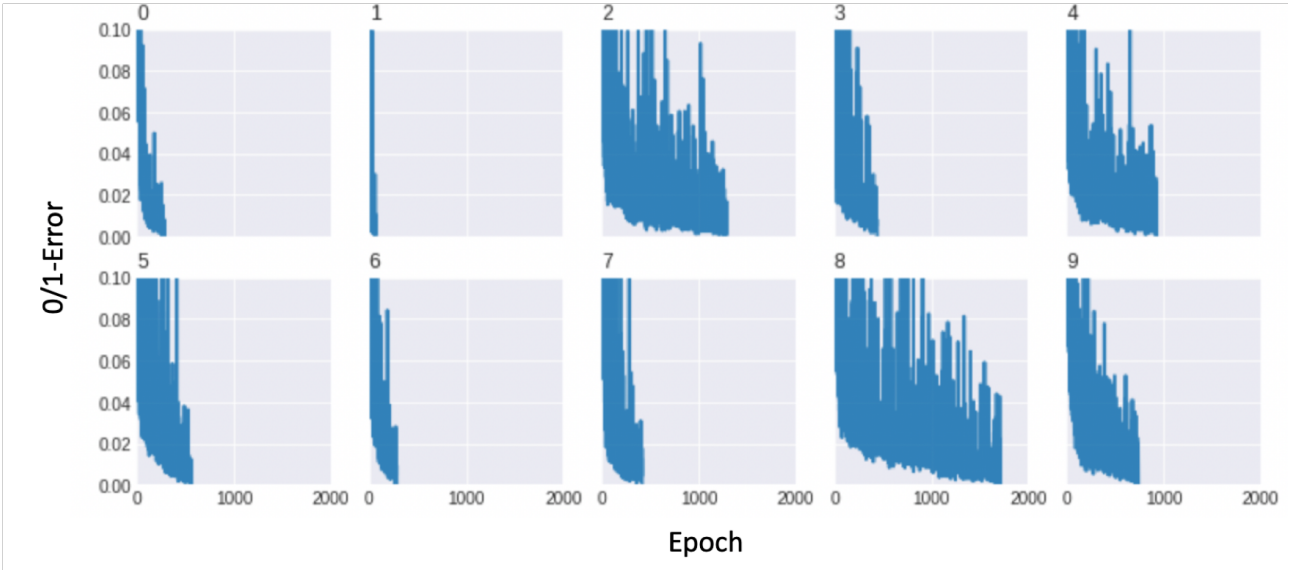


Figure 8: 0/1-Error for each digit in training dataset. The upper left digit in each small figure shows the digit we are classifying from the remaining digits for that small figure.

		Uniform init.		Glorot init.		Xaivier init.	
Learning rate	Activation func.	Missed	MSE	Missed	MSE	Missed	MSE
0.1	sigmoid	0	0.03	2	0.25	3	0.25
0.5	sigmoid	0	0	2	0.25	2	0.25
1	sigmoid	0	0	2	0.24	3	0.25
2	sigmoid	0	0	2	0.24	2	0.25
0.1	TanH	0	0	0	0	0	0
0.5	TanH	0	0	0	0	0	0
1	TanH	0	0	0	0	0	0
2	TanH	0	0	0	0	0	0
0.1	ReLU	1	0.17	0	0	1	0.17
0.5	ReLU	1	0.17	0	0	1	0.17
1	ReLU	1	0.17	0	0	1	0.17
2	ReLU	1	0.17	0	0	1	0.17

Table 2: results of training our XOR network with some combinations of learning rates, activation functions, and initialization strategies. The number of updates is 30000 and the seed (for weight initialization) is 42.

initialization. The TanH activation function works best with a relatively small learning rate (in the order of 0.1) and number of updates when compared with the sigmoid and ReLU activation functions – it also works well regardless of the initialization function used. The Xavier initialization strategy appears inferior to the others.

0/1 Error	distance-based model	multi-class perceptron
Train Data	0.136	0
Test Data	0.196	0.125

Table 3: the 0/1 error of distance-based model and multi-class perceptron on MNIST data.

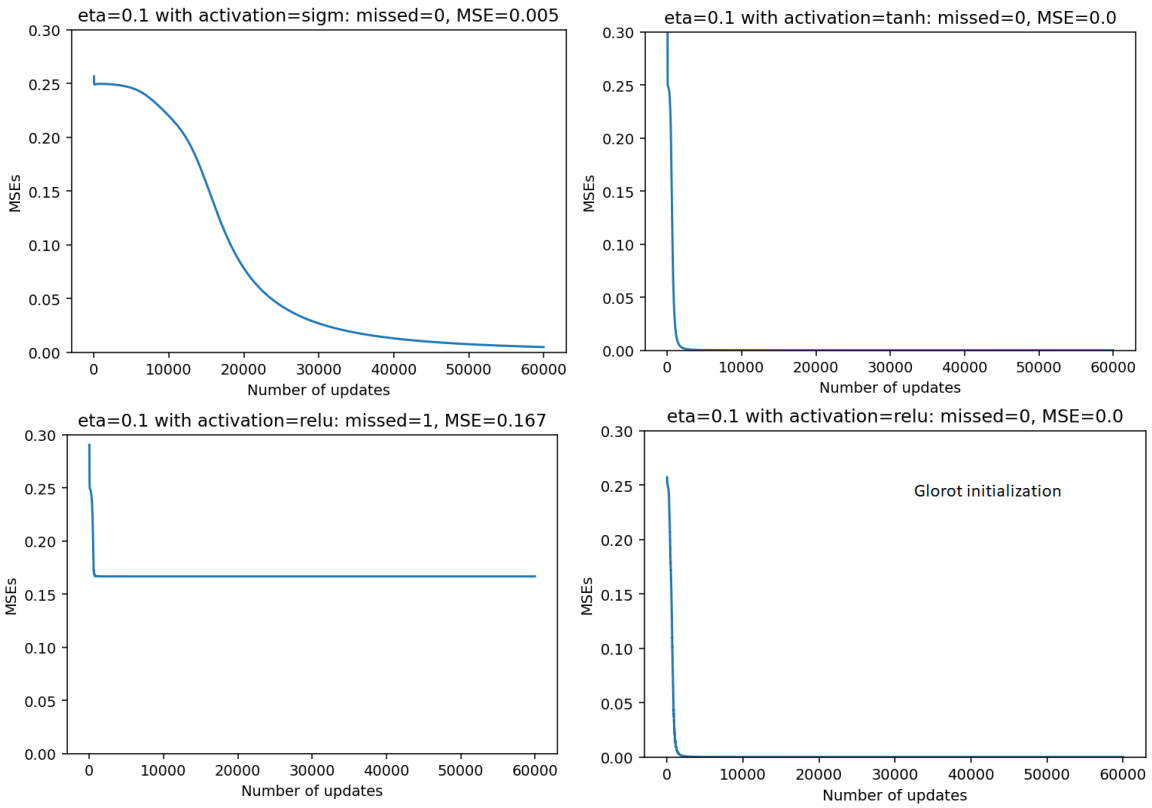


Figure 9: some MSEs plotted against number of updates with random uniform weight initialization in first three cases, Glorot initialization in lower right plot. Seed 42.