Neural Networks 2018

Assignment 1

Group 20

March 14, 2018

1 Introduction

The establishment of *Pattern Recognition* in the early 20^{th} century was mainly used in statistics, but the evolution of the society from the industrial to its postindustrial phase and the advent of computers have led to a more demanding practical applications, capable of dealing automatically with the increased amount of data produced (Theodoridis and Koutroumbas, 2009). Nowadays, *Pattern Recognition* makes up a significant part of *Machine learning* applications and *Artificial Intelligence* systems that are able to cope with a large variety of problems, such as machine vision, character recognition, mining for Bio-medical data, where different *patterns* need to be distinguished among others in the data and be classified (Theodoridis and Koutroumbas, 2009). The basic idea behind these systems relies on the use of one or more classifiers that are trained on a set of data where the class of each object is known based on specific features and then the classifier(s) are applied on raw data and classify them according to *a priori* knowledge.

Automated *handwritten numeral recognition* has attracted an intense interest in research and development in the last decades showing a remarkable improvement in *machine learning* and *artificial intelligence* (Teow and Loe, 2002) (Chmielnicki and Stąpor, 2011). Nevertheless, the conceptual knowledge gained by humans in rather simple problems requires a huge amount of data in order to make machines to perform similarly to humans (Lake, Salakhutdinov, and Tnenbaum, 2015) (Chmielnicki and Stąpor, 2011). This is due to the different representations of the same object which human minds perceive in an uncompetitive manner unlike the machines do, even though sophisticated algorithms have been developed. Each person has its own writing style which consequently makes the automated classification of the characters an extremely complex task for machines to deal with accurately (Giuliordi, Lillo, and Pena, 2014) (Lake, Salakhutdinov, and Tnenbaum, 2015) (Chmielnicki and Stąpor, 2011).

In this assignment we build a computer vision system based on Neural Networks to correctly identify numerical characters on a simplified version of the well studied *MNIST* dataset (Cohen et al., 2017). The MNIST dataset was first introduced by Y. LeCun, L. Bottou, Y. Bengio and Haffner (1998) which was soon established as benchmark dataset for classification tasks in the scientific community, so that accuracy in prediction among different approaches can be compared (Cohen et al., 2017). Although the original dataset is consistent of both handwritten digits and characters, in this study we are going to use a simplified version (in terms of resolution and size) of the *MNIST* dataset which contains only 2,707 instances for digit classification.

In the following sections we first build a simple classifier capable of distinguishing digits based on the distance between the images. Next, the classifier is enriched by another feature of the digits in order to improve the classification accuracy. This feature is the 'amount of ink' that is printed out to draw each digit. A Bayesian classifier is used to differentiate two digits based on this feature. Next, we introduce a *multi-class perceptron* and study the accuracy on the MNIST dataset. In the last sections we build a simple *Gradient Descent* algorithm, first for the age-old XOR problem, and finally for the MNIST dataset.

2 Simplest classifier: distance between images

There are many ways that someone could approach arithmetic digit classification. One such a way is by clustering digits into ten groups (clouds) according to their label and then build an algorithm that distinguishes the different numbers based on the distance between the digit in question to the ten clusters. In this case the higher the score or the distance in our case, the more dissimilar the digits are from the corresponding cluster. One way to correctly distinguish the digits is to compute the center of each of the ten digits in the dataset and then assign each sample to the cluster with the closest distance.

Our dataset for this task consists of 1,707 samples of the digits 0-9 where each of instance is represented by 256 pixels. Table 1 shows the distribution of the ten clouds in the dataset. Before calculating the centers of each cluster, a (10,256) *numpy* array is created to accommodate them. Then the mean of each digit and for each of its 256 dimensions is computed for all our examples by using the *mean* function over the examples.

Table 1: Number of samples per cloud

Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Samples	319	252	202	131	122	88	151	166	144	132

Once the centers of each sample are computed, we can easily calculate the distance between the center to the vector of each image by subtracting each of this points from the center of the image and then take the *absolute* value. Then this measure can be used to assign the images into one of the ten clouds based on the lowest distance. The distance of the center of each image to its more distant pixel is defined as the *radius* of the image. The measure used to calculate the distances is the *Euclidean* distance which is the square root of the sum of the squared distances from the center to any other point in the image as it is shown in Equation 1.

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

Table 2 depicts the radii as they have been computed for the ten clouds of our dataset. The radius is here defined as the biggest distance between the center of the cluster and the points from that cluster. It seems that digit 1 has the largest radius, which indicates that there is an example of digit 1 that is very dissimilar from most of the other digits that are labeled as 1.

Table 2: The maximum radius from the center for each digit

Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Radius	16.754	23.461	17.403	18.568	19.343	16.881	17.434	19.975	18.269	19.811

Next, we compute the distances of the centers of the clusters we created earlier to each individual number to have a clear insight of the distances in our dataset. To do so, we create a 10×10 distance matrix where we subtract the pairwise distances between each cloud from the ten digits as it is shown in Table 3.

Table 3: Pairwise distances between each cloud from the individual digits

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0									
1	14.45	0								
2	9.33	10.12	0							
3	9.14	11.73	8.17	0						
4	10.76	10.17	7.93	9.08	0					
5	7.51	11.11	7.90	**6.11**	8.00	0				
6	8.15	10.61	7.33	9.30	8.78	6.69	0			
7	11.86	10.74	8.87	8.92	7.58	9.21	10.88	0		
8	9.90	10.08	7.07	7.02	7.38	6.96	8.58	8.46	0	
9	11.48	9.93	8.88	8.35	**6.01**	8.25	10.44	**5.42**	6.40	0

Looking at the Table 3 we can conclude that the digits that are the most difficult to be correctly identified are those with the lowest distance from one cloud. Digits '7' and '9' are the most difficult with distance 5.42, followed by digit '9' and '4' with distance of 6.01 while the third more difficult digits to recognize are '5' and '3' with distance of 6.11. The easiest digits to be identified according to our algorithm are the digit '1' and '0' which are the most distant ones, with distance of 14.45.

To evaluate the performance of our system, we implement it both in the training and the testing sets and compute the accuracy in all the classified digits. Table 4 shows the percentages of correctly classified digits both in the training and the testing sets and also the difference between those two, while Tables 5 and 6 represent the confusion matrices for the training and the testing set respectively.

In the training set the digits which were the most difficult to be classified correctly, as it shown in the confusion matrix at the Table 5, is the digit '5' with 76.1% accuracy, followed by digit '4' with 77.8%. The third most difficult was digit '2' with 82.6%. Looking deeper at the confusion matrix, we can see that '5' was equally misclassified as a '0','3','4','6' and '9' while '4' in most of the cases was misclassified as a '9' and in some cases as

Table 4: Percentage of correctly classified digits in the training set, testing set and the percentage point difference betweet those two.

Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Training set	84.9	100	82.6	91.6	77.8	76.1	85.4	84.3	84	84.8
Testing set	79.4	99.1	68.3	77.2	80.2	69.1	86.6	78.1	79.3	77.2
Difference	5.4	0.9	14.3	14.4	2.3	7	1.2	6.2	4.6	7.5

Table 5: Confusion matrix for the training data

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

an '1'. Number two was mostly perceived as a '3' or a '4'. However, the classification accuracy differs in the testing set. The most difficult to identify was digit '2' with 68.3% , followed by digit '5' with 69% and third digit '9' with 77.2%. It is important at this point to mention the differnce in performance between to two datasets. As it is shown in Table 4, digits '2' and '3' were those with the highest misclassification rate in contrast with the the training set.

Table 6: Confusion matrix for the test data

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

Our initial assumption was that digit '7' and '9' were the most difficult to be classified correctly according to the results in the training set. Nevertheless, looking at the results of the testing set, we notice that this does not hold. Our explanation for difference in results is that digit '2' and '3' are more delicate in the individual handwritten style, which has as a consequence the two digit in some cases to look similar to each other confusing our system whether it is a '2' or a '3'. In order to find the optimal distance measure that yields the best accuracy in our system, we implemented all the measures available in the *sklearn.metrics.pairwise.pairwisedistances*[1] package. Figure 1 illustrates the 18 different measures and their accuracy rates in the testing set. The bar chart shows that the most accurate measures are the *Euclidean*, *squared Euclidean*, *correlation* and *Minkowski* distance all showing a 79.4% accuracy. This makes sense for the *Euclidean* and *squared Euclidean* and *Minkowski* (p=2) measure at least, since these are three different names for exactly the same measure. And it seems that correlation is the only actual different measure that performs with the same accuracy as the others.

[1] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html

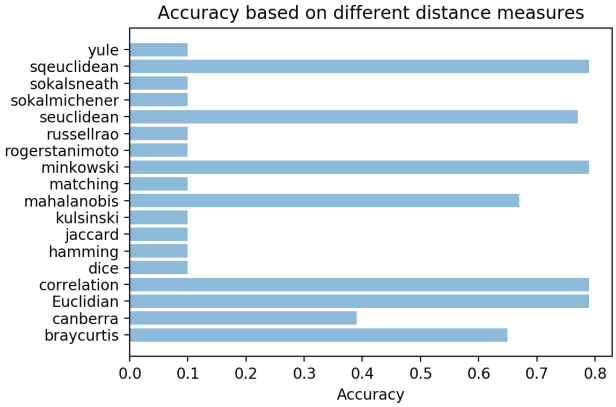


Figure 1: Accuracy of the 18 different distance measure in the testing dataset.

3 Bayes Rules Classifier

After using the clustering algorithm described in the previous section, we employ a different representation of our data to distinguish the difficult cases (e.g. a 5 or 7). For simplicity, in this section, we will only look at the classification problem between digit 5 and 7.

Possibly the easiest feature to think of is the number of pixels that are written on. This is easily calculated by taking the sum of all the pixel-values in the images. We will call this feature the 'amount of ink'. One would expect digit 5 to score on average higher than digit 7, since digit 5 requires more ink, or more pixels that are written on. Figure 2 shows the distribution of the 'amount of ink' in the training data. It can be readily appreciated from this figure that, as we expected, the distribution of the amount of ink used for digit 5 is more to the right than the distribution for the amount of ink used for digit 7. The feature values are negative because most pixels are not drawn on, and these have value -1, so when an image has no 'ink' on it, the feature will have value -256.

We then find the probability $P(C|X)$ that a digit with feature value X is class C, using *Bayes*' rule:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}. \tag{2}$$

The probability that a given class C (i.e. digit 5 or 7) has a feature value X, $P(X|C)$, can be found by simply counting the number of cases in each bin, and dividing by the total number of cases. For digit 5 and 7, this gives a similar distribution to the histogram shown in Figure 2. The probability $P(C)$ can be found by counting all the cases that the class is C and dividing by the total number of cases. This gives us the following values: $P(5) = 0.35$ and $P(7) = 0.65$, indicating that there are almost twice as much 7's in the training data as 5's. Finally, the probability $P(X)$ can simply be found by taking the number of cases in each bin for both digits, summing these up, and dividing by the total number of cases. The probability is shown in Figure 3a, in which easily becomes apparent that the probability $P(X)$ is indeed the sum of the individual histograms.

Finally, the posterior probability $P(C|X)$ is plotted in Figure 3b. The cutoff between where we would predict a digit to be 5 or 7 is shown to be at a feature value of -102.

Now, we can use this probability to classify cases on the test set, by calculating the feature value for all cases, and checking Figure 3b for the posterior. The digit is then classified as the class with the highest posterior probability, i.e. when the feature value is below -102, it will be classified as a 7, and when the feature value is above -102, it will be classified as a 5. When applying this method to the training set, we get an accuracy of 78% and when applying this method to the test set, we get an accuracy of 68%.

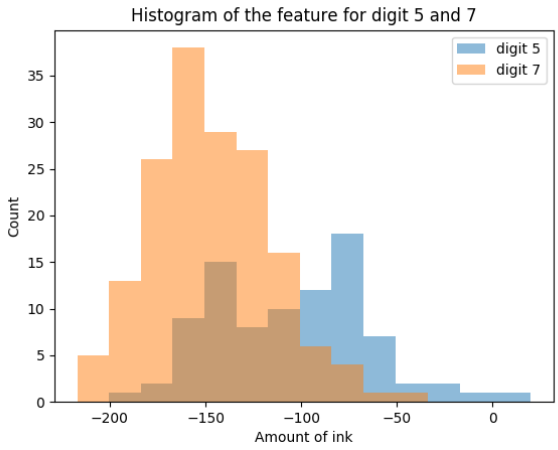
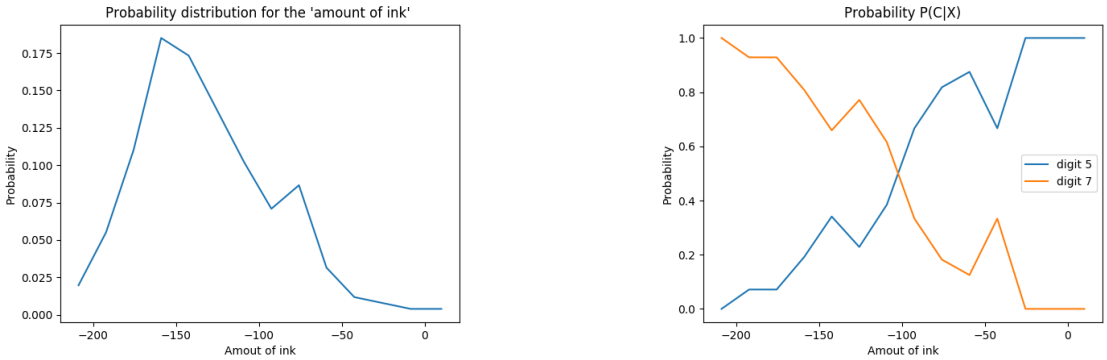


Figure 2: Distribution of the 'amount of ink' for digit 5 or 7 on the training data.



(a) Probability for any observed 'amount of ink'

(b) Probability for the posterior

Figure 3: The probability to observe a feature value $P(X)$ and the posterior probability $P(C|X)$.

4 Multi-class perceptron

A multi-class perceptron is an extended version of the simple perceptron, which is a binary classifier. The perceptron takes as input n variables $(x_1, ..., x_n)$, computes the weighted sum of these inputs to a 'node', after which a certain activation function (e.g the Heaviside step function) determines whether the value in the node becomes either a 0 or a 1. In this way, a certain example i with features $(x_1, ..., x_n)$ can be classified using n parameters.

In case there are more than two possible outcomes, for example in digit classification, we can generalize this approach by essentially stacking multiple of these perceptrons on top of each other. If we stick as much perceptrons as there are classes then each output node j $(0 < j < 10)$ can be viewed as giving a probability of some sort that the given class is digit j. Classification can then be performed by taking as the predicted class the node with the highest output (i.e. the argmax). This process is visualized in Figure 4.

Implementing this is fairly straightforward: we start by initializing a matrix w of random weights between 0 and 1, which has shape (257,10) corresponding to the number of input nodes and the number of output nodes. This is because the weighted sum from the input to the output can be realized as a matrix multiplication involving the weights matrix w and the input vector X. Mathematically, with output node vector A, this means $A = w^T \cdot X$. We transpose w because a dot product with a matrix of shape (10,257) and a vector of shape (257,1) provides us with an output vector of shape (10,1), as is required. This matrix multiplication is executed for each example in the training set, the predicted digit $\hat{y}$ is taken as the argument maximum of the output vector and the weights are adjusted according to the following steps:

1. Check if the output $\hat{y} = argmax(j)$ is the same as the example label y. If so, continue to the next example, else go to step 2.

2. Increase the weights that are connected to the required output node (y). Decrease the weights that are connected to the output node given by the network $(\hat{y})$. Mathematically: $w_j = w_j \pm X_j * 0.05$.

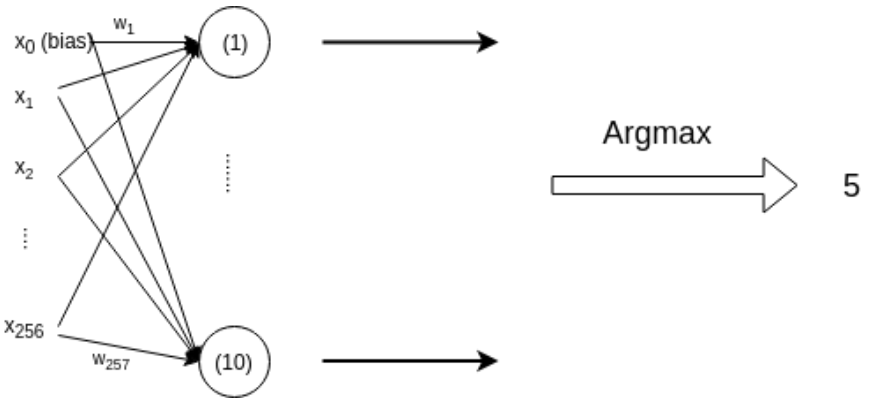
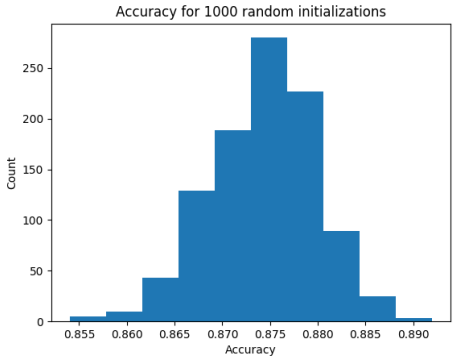
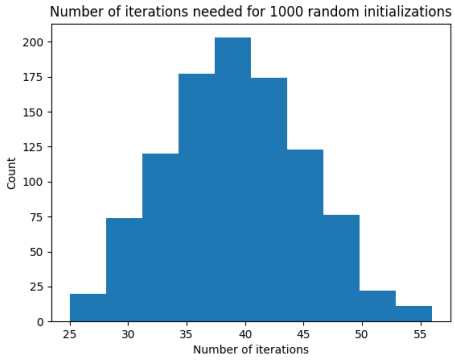


Figure 4: Schematic representation of the multi-class perceptron indicating the 257 inputs, 10 outputs, and the classification based on the argument maximum.



(a) The accuracy on the test set. The mean of the accuracy is found to be 0.874.



(b) Number of iterations over the entire training set needed to achieve an accuracy of 100%.

Figure 5: Distribution of accuracy and number of iterations over the training set for a 1000 random initializations of the multi-class perceptron.

These steps can be repeated until a satisfactory number of correctly classified examples is achieved. We have chosen to repeat these steps while the network is still finding incorrectly classified examples in the training set.

It takes only about 40 iterations over the entire set to reach an accuracy of 100% on the training set. We then use these adjusted weights to classify the digits in the test set and reach an accuracy of about 87%.

Since this model is dependent on the initial weight configuration, the values given in the previous paragraph differ for different run of the program. This is, however, not a drastic difference as can be viewed in Figure 5. After a 1000 random initializations, the perceptron achieves a mean accuracy of 0.874 on the test set, which requires a mean amount of loops over the training set of 39.

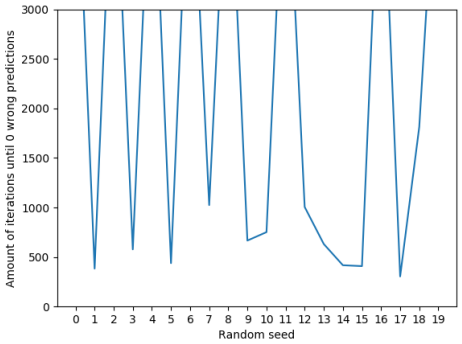
5 Gradient descent

Finally, we will apply (a simple version of) the gradient descent algorithm on the *MNIST* data set. Before implementing this, we start by creating a network that learns the XOR function. This network consists of two inputs, x_1 and x_2, two hidden nodes and one output node. Adding a bias to the input and hidden layer gives us 9 weights between these nodes, or 9 tuneable parameters. The *sigmoid* activation function is used for the hidden and output nodes usisng Equation 3,

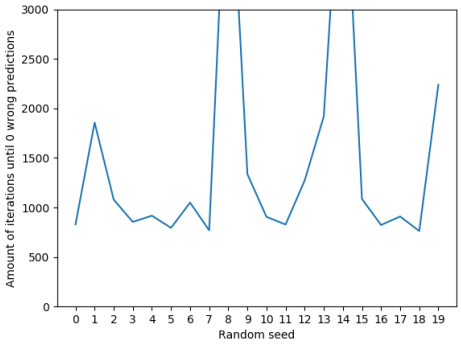
$$\sigma(x) = \frac{e^x}{e^x + 1}, \tag{3}$$

and finally, a translated *Heaviside* step function is used to convert the output to a 0 (if the output is below 0.5) or a 1 (if the output is above 0.5).

We implement this network by creating a function that has as input x_1, x_2 and the vector $\vec{w}$ consisting of 9 weights, initialized from the standard normal distribution. The input is converted to a vector $\vec{X}_1 = [+1, x_1, x_2]$ and multiplied by the first 6 weights reshaped to a matrix of shape (3,2), analogously to the way the multi-class perceptron propagated its input. The hidden layer is then a vector of shape (2,1), and the activation function is applied. Another bias node is added, and the three outputs from the hidden layer ($\vec{A}_1$) are then propagated to



(a) Weights initialized from the standard normal distribution.



(b) Weights initialized randomly between 0 and 1.

Figure 6: Amount of iterations needed until the network has successfully learned the XOR problem with two different initialization techniques, for different random seeds. Values above 3000 indicate the network is stuck and will not learn the problem (within 3,000 iterations).

the final output node using the last three weights from the weight vector $\vec{w}$. The activation function is applied again to receive the output value A_2, and the step function is applied to receive the predicted class $\hat{y}$.

The error function is then implemented, which will be the mean squared error (*MSE*) made by the network on the 4 possible input vectors: $[0, 0]$, which should return 0, $[0, 1]$, which should return 1, $[1, 0]$, which should also return 1, and $[1, 1]$, which should return 0. The *MSE* is now defined as

$$1/4 * \sum_{i=1}^{4} (y_i - \hat{y}_i)^2, \tag{4}$$

where y_i is defined as the target value corresponding to the input (x_1,x_2) for the four different input cases. The gradient of the mean squared error is computed numerically by the definition of the derivative

$$f'(x) = \lim_{h \to \infty} \frac{f(x+h) - f(x)}{h}. \tag{5}$$

That is, we calculate the derivative of the *MSE* with respect to each of the 9 weights by substituting $f(x)$=MSE($\vec{w}$) and $h = 10^{-3}$ in Equation 5, giving

$$\frac{\partial MSE}{\partial w_i} = \frac{MSE(\vec{w} + \langle 0, ..., h, ..., 0 \rangle) - MSE(\vec{w})}{h}, \tag{6}$$

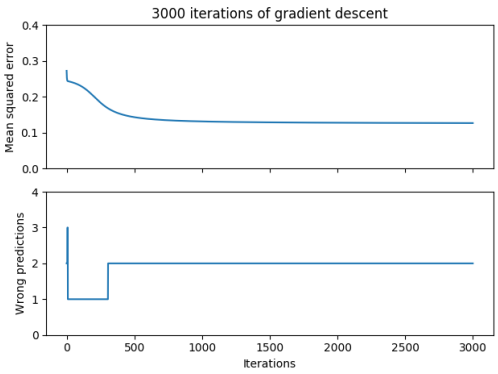
where we set $h = 10^{-3}$ on the i-th element of the vector. Then we update the weights according to the gradient descent algorithm: $w_i = w_i - \eta * \frac{\partial MSE}{\partial w_i}$, where η is the learning rate.

Since this problem has perfect training data, we will say the network has 'learned the problem' if it has an accuracy of 100% on the training examples, or 4 out of the 4 examples correct. It is redundant to run the network again on a test set, because the examples will be exactly the same as the training set.

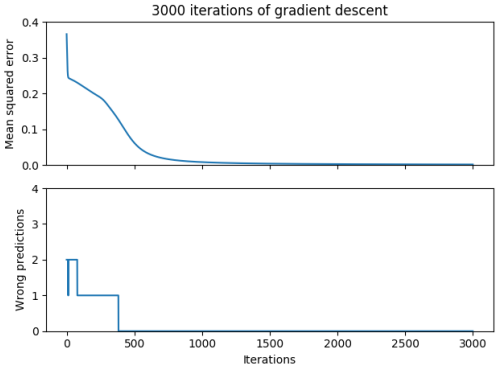
After the implementation, we run the program for 3000 iterations of gradient descent, initially for a learning rate of 1.0. We quickly find that the initialization of the weights has a great impact on whether the algorithm will learn this problem or not. Figure 6a shows the amount of iterations needed for 20 different initialization seeds for the random weights. This shows that about half the time, the initial weights are chosen such that the network does not learn the problem. This was tested with 30,000 iterations on random seed 0, after which the network was still stuck at 2 wrong predictions, so it is safe to say the network is stuck in a local minimum in about half of the cases.

Another initialization scenario that can be chosen, is not setting our initial weights from the standard normal distribution, but taking random initial weights between 0 and 1. This seems to work more often for these particular seeds, only getting stuck in a local minimum in two of the cases. Interestingly, in the cases where the network learns the problem, the network does take about 300 to 400 more iterations to succesfully classify the XOR problem than with the succesful random initializations from the standard normal distribution.

Figure 7 shows the *MSE* and number of wrongly classified examples during the iterations of the gradient descent algorithm, for weights picked randomly from the standard normal distribution. Figure 7a shows a seed that does not converge within 3000 iterations, and Figure 7b shows a seed that does converge. These figures are the prime example of the importance of the initial weight values.



(a) Random seed 0, not converging or very slowly.



(b) Random seed 1, converging within 500 iterations of gradient descent.

Figure 7: Mean squared error and number of wrongly classified examples for standard normally distributed weights for (unsuccessful) random seed 0 and (successful) random seed 1.

6 Gradient descent in the MNIST

In this task we are asked to build a gradient descent algorithm to learn the weights and biases of the network constructed from the *MNIST* dataset and correctly classify the digits. The network in request at the first layer of perceptrons consists of 256 input nodes plus one node for the bias, 10 hidden nodes plus one node for the bias in the second layer and 10 output nodes for each individual digit (0-9). The procedure that is followed in this task is similar with what was done in the previous task modified for the network in question. The *sigmoid activation* function is first computed for all the hidden nodes using Equation 3 and then, they are multiplied by the corresponding weights that have been initialized randomly.

The same procedure is applied for the output nodes by using the values obtained in the hidden layers updating the weights. Once the weights are calculated the *Heaviside* is appplied to update the output values to 0 when the output value is lower than 0.5 and to 1 when the output value is greater or equal to 0.5. The *mean squared error* and the *derivatives* of the weights are calculated using equations 6 and 5 respectively.

The algorithm was implemented for different learning rates, namely 2, 5, 8 and 10. The first thing to note is that the algorithm is very slow when implemented like this, because there are now $257 * 10 + 11 * 10 = 2680$ weights that have to be updated using Equation 6, which means one by one putting each of these weights through the network and calculating their derivative. 5000 iterations of gradient descent took about 2.3 days of computing time. The most effective (minimal *MSE*) learning rate was found to be 10, which resulted in a mean squared error of 0.019160, and 26 misclassified digits, out of the 1707 in the training set. The *MSE* and number of wrongly classified digits versus the number of iterations of gradient descent can be viewed in Figure 8. It shows that the *MSE* quickly converges to a small number, after which the network does not improve a lot anymore. Interestingly, at about 300 iterations there is a wave pattern, which is probably due to the large learning rate of the algorithm, which makes it overshoot a couple of times when updating the weights. This can also be viewed in Figure 9, where the figure shows that the learning rate of 8.0 is actually converging faster than the learning rate of 10, indicating that a learning rate of 8 overshoots the updated weights less often.

In the end, the learning rate of 10 finds the smallest *MSE* after 5000 iterations, and the learned weights are used to classify the test set. While an accuracy of 98% was achieved on the training set, an accuracy of 84% is achieved on the test set. This corresponds nicely to the multi-class perceptron results, and, if the network had kept training to achieve an accuracy of 100% on the training set, we would converge to the same results as in the previous section.

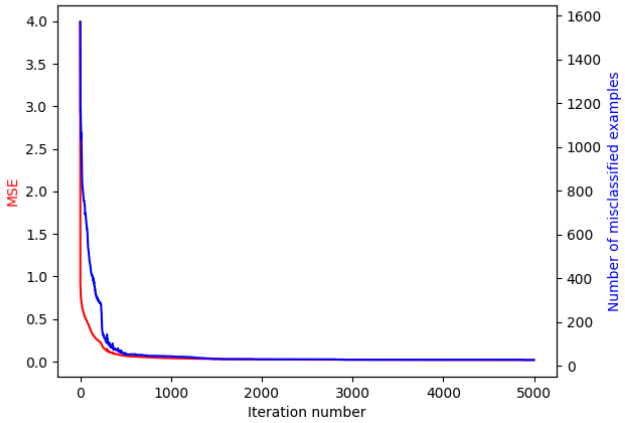


Figure 8: The mean squared error and number of incorrectly classified digits versus the number of iterations of gradient descent for a learning rate of 10.

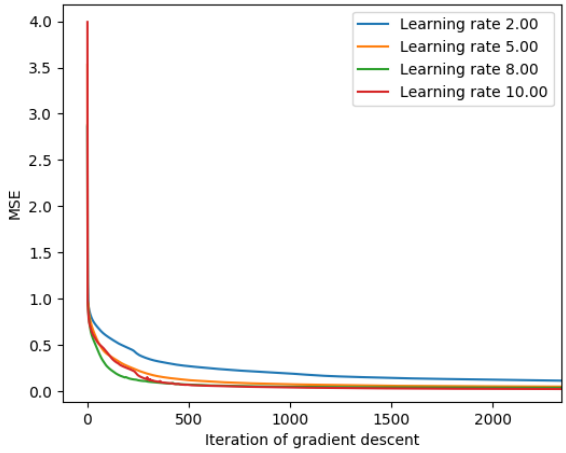


Figure 9: Mean squared error of the network on the training data, using learning rates of 2,5,8,10.

7 Conclusion

In this report we presented a number of different approaches that one can deal with to accurately classify handwritten digits. Initially, we introduced a simple classifier applied on the *MNIST* dataset based on the distance between each image and the 10 clusters, displaying 85.15% accuracy on the training data and 79.45% on the testing set. Next, we built a Bayes classifier to distinguish between the difficult cases, in particular digit 5 and digit 7. Using as a feature from the image its 'amount of ink' and calculating the probability that the image is a 5 or a 7. This method yielded an accuracy of 78% and 68% on the training and the testing set respectively. We also implemented a multi-class perceptron with 256 + 1 input nodes and 10 output nodes, which shows 100% accuracy on the training set and 87% on the testing set after only 40 iterations. In addition, a gradient descent algorithm was implemented on the *MNIST* dataset. The algorithm learns the XOR function, by using the *sigmoid* activation function and the Heaviside function, in under 500 iterations if the weights are chosen fortunately. In the same way, the *Gradient Descent* on the *MNIST* dataset is computed for 256+1 input nodes, 10 hidden nodes and one output node. This classifies all cases in the test set with an accuracy of 84%.

All in all, looking at the results, it is important to highlight the performance of the multi-class perceptron in contrast with the gradient descent algorithm. The multi-class perceptron needs only 40 iterations to update the weights in the net and absolutely correctly classify all the examples in the training set while our gradient descent algorithm after 3,000 iterations shows an accuracy of 'only' 98% on the training set. It seems that for this simple classification problem, a multi class perceptron is the most efficient and most accurate way to solve this problem, given that we have limited computing time.

References

Chmielnicki, Wiesław and Katarzyna Stąpor (2011). "Investigation of Normalization Techniques and Their Impact on a Recognition Rate in Handwritten Numeral Recognition". In: *Schedae Informaticae* 19.-1, pp. 53–77. ISSN: 0860-0295. DOI: 10.2478/v10149-011-0004-y.

Cohen, Gregory et al. (2017). "EMNIST: Extending MNIST to handwritten letters". In: *Proceedings of the International Joint Conference on Neural Networks*, pp. 2921–2926. DOI: 10.1109/IJCNN.2017.7966217.

Giuliordi, Andrea, Rosa Lillo, and Daniel Pena (2014). "Handwritten Digit Classification". In: *Universidad Carlos III De Madrid*.

Lake, Brenden M., Ruslan Salakhutdinov, and Joshua B. Tnenbaum (2015). "Program Induction". In: *Science* 350.6266, pp. 1332–1338. DOI: 10.1126/science.aab3050.

Teow, Loo-Nin and Kia Fock Loe (2002). "Robust vision based features and classification schemes for off line handwritten digit recognition". In: *Pattern Recognition* 35, pp. 2355–2364. DOI: 10.1016/S0031-3203(01)00228-X.

Theodoridis, Stergios and Konstantinos Koutroumbas (2009). "Pattern Recognition". In: *Elsener* 4th Edition.

Y. LeCun, L. Bottou, Y. Bengio and P. Haffner (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2323. ISSN: 00189219. DOI: 10.1109/5.726791. arXiv: 1102.0183.