Neural Networks 2019

Assignment 1

March 13, 2019

Introduction

We report on the development and testing of a number of basic algorithms to classify digits from a simplified version of the MNIST database (Lecun et al., 1998). The MNIST database comprises a large collection of low-quality images of hand-written numerals (60,000 training items and 10,000 test items), taken from a diverse group of people, including employees of the American Census Bureau and high-school students. Because the recognition of digits is an archetypal example of a classification problem, the MNIST database is a popular resource for the training and testing of machine-learning algorithms. For the human brain, recognising an arbitrary digit is a trivial task that can be executed flawlessly in no time. However, due to the large variability among handwritings, it is not straightforward to achieve comparable performance with a computer. Sometimes, a hand-written 4 may look devilishly similar to a 9, a 3 may be confused with an 8 and a sloppy 6 may resemble a 0.

Over the past two decades, a multitude of methods were devised to classify MNIST digits with the highest possible accuracy. The majority of these rely on *neural networks* – architectures consisting of logical processing units, known as perceptrons, which are connected to each other just like nerve cells in the human brain. In 2012, a convolutional neural network achieved an accuracy of 99.77% (Ciresan, Meier, and Schmidhuber, 2012) on the MNIST test set, making it the current record holder. Please refer to the MNIST website[1] for the complete ranking and the associated publications.

In this report, we explore a number of simple methods to classify MNIST digits. Each image from the dataset can be represented by a vector in n-dimensional space, with n the number of pixels. In task 1 and 2 we attempt to distinguish between digits by evaluating the "distances" between different images. In task 3, we develop a Bayes-Rule classifier, which exploits the number of filled pixels as a feature to discriminate between two digits. In task 4, we set out to develop a single-layer perceptron algorithm and we demonstrate that this simple neural network is capable of achieving accuracies well over 80%. In task 5, we provide a glimpse of more advanced algorithms by simulating an XOR gate with a multilayer network and trying our hand on a multilayer network for the simplified MNIST database.

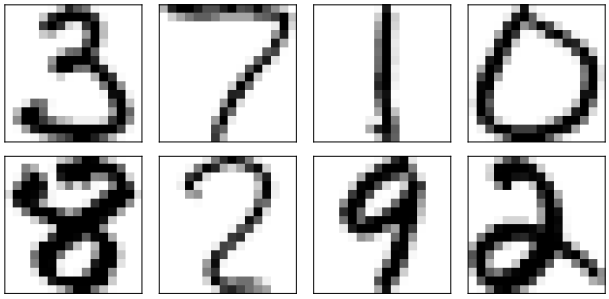

Figure 1: Eight images sampled from the training set. The black pixels have values +1 and the white pixels have values −1. The values of the gray pixels range in between.

The Dataset

Our dataset comprises a training set and a test set, which contain 1,707 and 1,000 images respectively. Each image consists of $16 \times 16 = 256$ pixels. There are two other files, which contain lists of the *true* digits (from 0 to 9) that are shown in each of the images. A perfect classification algorithm will correctly identify these *true* digits in all of the images. Please refer to Table 1 for a detailed overview.

For computational simplicity all images can be mapped onto a vector $\vec{x}$ (i.e. a point) in 256-dimensional space, where each component represents the value of one pixel in the image. In other words, the projection of this vector onto the i-th axis is equal to the value of the i-th pixel. Figure 1 shows eight arbitrary images from the training set.

[1] `http://yann.lecun.com/exdb/mnist/`

Table 1: Overview of the number of images per digit contained in the training set and test set respectively. Percentages are mentioned between brackets.

Digit	# in training set	# in test set
0	319 (18.7%)	224 (22.4%)
1	252 (14.8%)	121 (12.1%)
2	202 (11.8%)	101 (10.1%)
3	131 (7.7%)	79 (7.9%)
4	122 (7.1%)	86 (8.6%)
5	88 (5.2%)	55 (5.5%)
6	151 (8.8%)	90 (9.0%)
7	166 (9.7%)	64 (6.4%)
8	144 (8.4%)	92 (9.2%)
9	132 (7.7%)	88 (8.8%)
Total	1,707 (100%)	1,000 (100%)

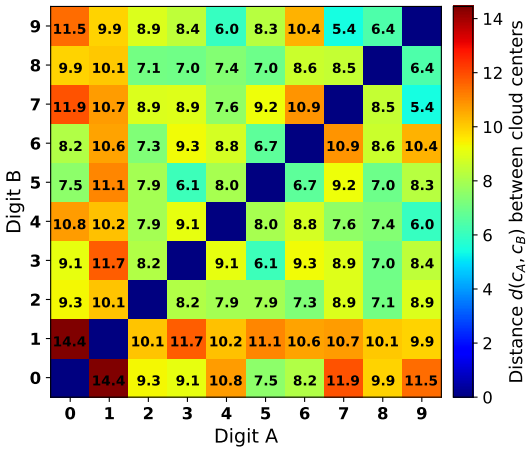

Figure 2: Overview of the distances $d(C_A, C_B)$ between the clouds of two digits A and B, as computed from the training set. The values are specified in the colour bar. Naturally, the values on the diagonal are zero, because the distance from a point to itself is zero.

Task 1

When vectorising all images from the training set, each digit d (from 0 to 9) corresponds to a cloud C_d of points $\vec{x}$ in 256-dimensional space. We compute the center of each cloud by taking the component-wise average of all vectors that make up the cloud. In addition, one can define the radius of each cloud as the largest distance between the center of C_d and any of the points in C_d. We also compute the frequency by which each digit occurs in the training set (see Table 1).

Once the centers of each cloud C_d are known, there exists an intuitive way to classify an arbitrary image by computing the distance between its vector $\vec{x}$ and all cloud centers. The digit contained in this image is *most likely* the digit whose cloud center lies closest to the vector of the image.

Figure 2 shows the distances between each of the cloud centers, as derived from the training set. We find that the distance between the cloud centers of the pair (0, 1) is the largest. Since these digits have mutually very different shapes (regardless of the handwriting, most pixels that are black for a 0 are white for a 1 and vice versa; see Figure 1), this is a logical result. On the contrary, we find that the distance between the cloud centers of 7 and 9 is the smallest and therefore it *seems* that this pair of digits will be most difficult to separate. This also makes sense intuitively, because a sloppy 9 may look like a 7. Other pairs for which the distances are small are (4, 9) and (3, 5).

Although Figure 2 shows that there are clearly some outliers in the distances between the cloud centers, the majority of the values do not differ by more than a factor 2. This seems to indicate that most digits are more or less equally difficult to differentiate between.

It should be noted that the distances between cloud centers do not tell the entire story. In principle, two digits whose clouds lie close together may still be quite well separable when the clouds are compact and their points are scattered over a small volume. In this respect, the cloud radius as defined previously provides a bit more insight. However, its value is of limited use, because a single extremum (i.e. the maximum distance) does not tell anything about the *bulk* behaviour of the points in C_d and the shape of the cloud. Hence, a better observable would be the variance of the distances between all points in C_d and its center.

Task 2

Having established what the distance between the centers of different clouds C_d is, we now classify all digits in the training set and the test set. This is done by simply calculating the distance from an image vector $\vec{x}$ to all cloud centers and classifying that image as being the digit to whose cloud center the distance is smallest.

It is striking how well this straightforward classification scheme works. For the training set, we achieve an accuracy of 86.35% (i.e. 86 out of 100 images are classified correctly), while the accuracy on the test set amounts to 80.40%.

Figure 3 shows the 10×10 confusion matrices that were obtained. These depict what percentage of the digits A (in training set and test set respectively) are classified as being a digit B. Within both sets, not all digits occur with equal frequency and therefore we choose to plot percentages, rather than absolute numbers. This allows for a better comparison between different pairs of digits.

Based on the distances between the cloud centers found in Task 1, one expects that it is most difficult to separate the pair (7, 9). Indeed, it turns out that for the training set 10.2% of the images containing a 7 are recognised a 9. Other major misclassifications are 0 being recognised as a 6 (11.3%) and 4 being recognised as a 9 (12.3%). From Figure 2, we can see that the distances between these digits are also relatively small. As noted previously, the degree of scatter exhibited by the cloud points also determines how well one can

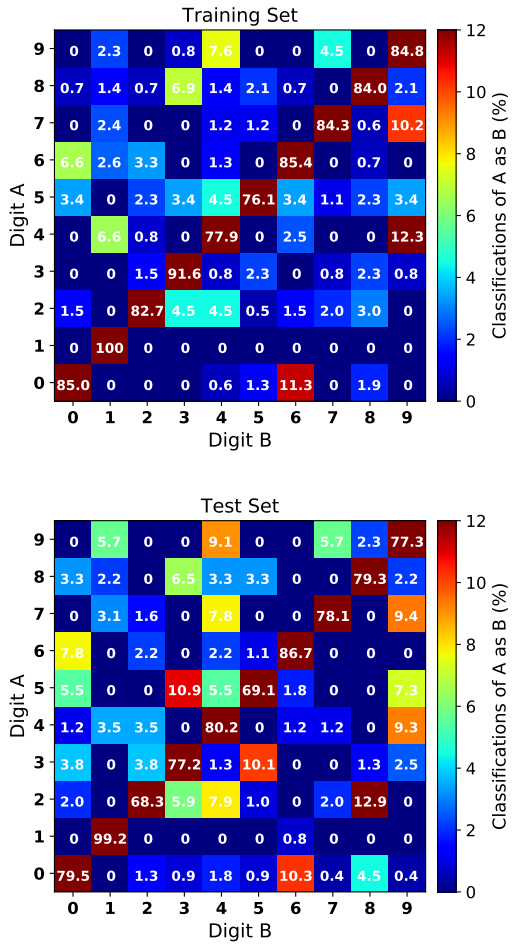

Figure 3: Confusion matrices for the Euclidean distance classifier, as obtained for the training set (**top**) and the test set (**bottom**). Each element of the matrix shows what percentage of the digits A, on the vertical axis, was classified as being B, on the horizontal axis. The elements in each row add up to 100%, as each image will be always be classified as something.

distinguish between a pair of digits, but this is *not* reflected in the distance between the cloud centers.

Figure 3 also hints at the limited scope of the training set. If the training set were a proper representation of the test set, the two confusion matrices would have looked very similar. Instead, the confusion matrix of the test set indicates that quite a number of digit pairs, among which (4, 9), (7, 9) and (0, 6), but also (3, 5) and (2, 8) are competing for the highest misclassification rate. The pair (2, 8) is particularly interesting. In the training set, a 2 is classified as an 8 in only 3% of the cases, while this percentage is 12.9% for the test set. Apparently, the 2's from the test set tend to look more similar to an 8 than the 2's from the training set.

From the diagonals of the confusion matrices, we can also infer which digit is most difficult to classify. In the training set, this appears to be 5 (in 'only' 76.1% of the cases a 5 is recognised as a 5). In the test set, 2 and 5 are most difficult to identify (the associated percentages are 68.3% and 69.1% respectively). On the other hand, it is by far the easiest to classify 1. In the training set, 1 is recognised as itself in 100% of the cases, while we record only a single misclassification in the test set.

Table 2: Percentage of misclassified images for different distance measures on both the training data and the test data.

Distance measure	Training error	Test error
Euclidean, L2	13.65%	19.60%
Cosine	13.95%	20.10%
Manhattan, L1	23.43%	27.90%

Up to now, we only considered the Euclidean distance measure to compute distances between image vectors $\vec{x}$ and the centers of the clouds. However, there also exist other distance measures, which lead the algorithm to perform differently. Table 2 shows the error rates ($= 100\% -$ accuracy) that we obtain on the training set and test set respectively, using a number of distance measures. We find that the Euclidean and cosine distance measures lead to similar errors, with the Euclidian distances performing marginally better on the test set. The larger error for the Manhattan distance is not surprising. Given its definition as the sum of absolute values of the component wise differences, the distance between two points may seem artificially enhanced. This in turn will lead to more misclassification and thus a higher error.

Task 3

We set out to develop a Bayes-Rule classifier which is able to discriminate between pairs of digits based on Bayes' theorem. As a feature, we use the number of non-white pixels N_{pix} in the image to identify the digit. That is, the number of pixels with a value different from -1. To discriminate between two digits, we seek the posterior probability $P(d|N_{\text{pix}})$. This is the conditional probability that the image contains the digit d given that it has N_{pix} non-white pixels. Subsequently, we select the digit for which this value is the highest. According to Bayes' theorem,

$$P(d|N_{\text{pix}}) = \frac{P(N_{\text{pix}}|d)P(d)}{P(N_{\text{pix}})}. \qquad (1)$$

Here, $P(N_{\text{pix}}|d)$ is the likelihood that the image contains N_{pix} non-white pixels when it corresponds to the digit d. Furthermore, $P(d)$ is the prior probability, which is proportional to the number of occurrences of d in the training set. In other words, it is the probability of finding the digit d without any further knowledge about N_{pix}. Finally, $P(N_{\text{pix}})$ serves as a normalisation factor, but can be ignored here as we are only interested in the *ratio* between posteriors.

From the training set, we select all images that contain digits A and B respectively. For both digits we construct histograms of 256 bins, where the i-th bin contains the number of images for which $N_{\text{pix}} = i$. It can be motivated that these histograms are immediately proportional to the posterior distributions we are seeking.

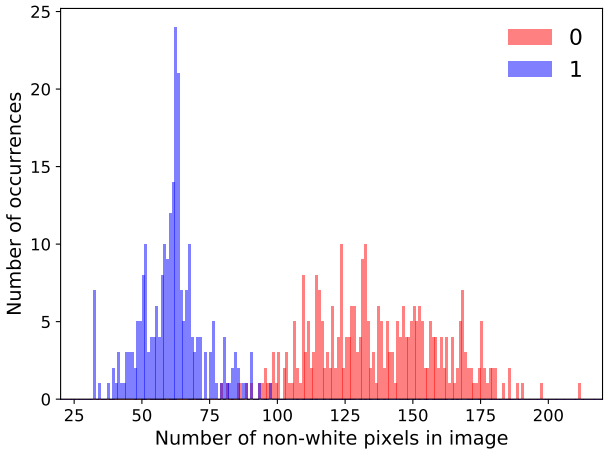

Figure 4: Histograms of the number of non-white pixels for the digits 0 and 1, as obtained from the training-set images. Both histograms are non-normalised versions of the posterior $P(d|N_{\text{pix}})$, as motivated in the text.

By construction, the histograms are non-normalised versions of the likelihood $P(N_{\text{pix}}|d)$ and when integrating over them, one obtains the total number of occurrences of the digit in the training set. This implies that the histograms are also proportional to the prior $P(d)$.

Figure 4 shows the histograms obtained for the pair (0, 1). Suppose we now draw a random image from our dataset that either contains a 0 or a 1. We classify this image by counting the number of non-white pixels N_{pix} and computing the ratio

$$r = \frac{P(0|N_{\text{pix}})}{P(1|N_{\text{pix}})} = \frac{P(N_{\text{pix}}|0)P(0)}{P(N_{\text{pix}}|1)P(1)}. \qquad (2)$$

In other words, r is the ratio between the bin heights of both histograms at N_{pix}. When $r > 1$, the digit is classified as 0 and when $r < 1$ the digit is classified as 1. When r is exactly unity, we randomly classify the digit as being either 0 or 1.

Figure 5 shows the accuracies we achieved for all pairs of digits (A, B) on the training set and the test set respectively. Because the histograms are only based on the training set, the accuracies on the training set are naturally higher than those on the test set.

For both sets, it turns out that the digit 1 can be separated relatively well from the others. This makes intuitive sense, because an image with a 1 typically has fewer non-white pixels than other digits. Therefore, the corresponding histogram is less likely to substantially overlap with the histogram of another digit. For the pair (0,1) in particular, the accuracy is 99% on the training set and 94% on the test set.

There also exist pairs for which our Bayes classifier does not work. For the test set in particular, accuracies occasionally drop below 60% (whereas a totally random classification would yield 50% accuracy on average). A striking example is the pair (4,6), for which the accuracy is even lower than 50%, which means that more than half of the images are misclassified. This can be explained by the fact that images of a 4 and

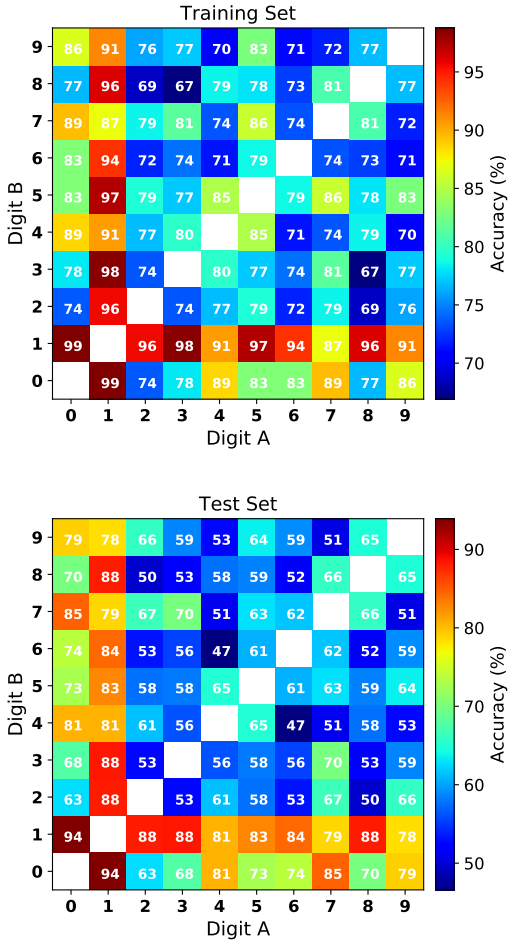

Figure 5: Overview of the accuracies obtained with our Bayes-Rule classifier on the training set (**top**) and the test set (**bottom**), for all for all pairs of digits (A, B). Note the different colour scale of both panels.

a 6 have a roughly equal number of non-white pixels, which makes the histograms overlap almost completely. Therefore, the ratio r lies very close to unity all the time. Additionally, the priors that we use are based on the relative occurence of the digits in the test set and Table 1 shows that these are not exactly the same for the test set.

Task 4

No matter how well the above described methods work, none of them are considered neural networks. A neural network is a computational system build up of perceptrons, with unlimited complexity. Here we will consider a very simple, single layer neural network and apply it to the MNIST data set.

Perceptrons are computational units which convert a given number of inputs into a single output. The inputs can be outputs from other perceptrons or input data provided by the user. The inputs are weighted by a set of weights and the perceptron can use different functions to convert the weighted inputs into an output. The goal of perceptron learning is to tune the weights

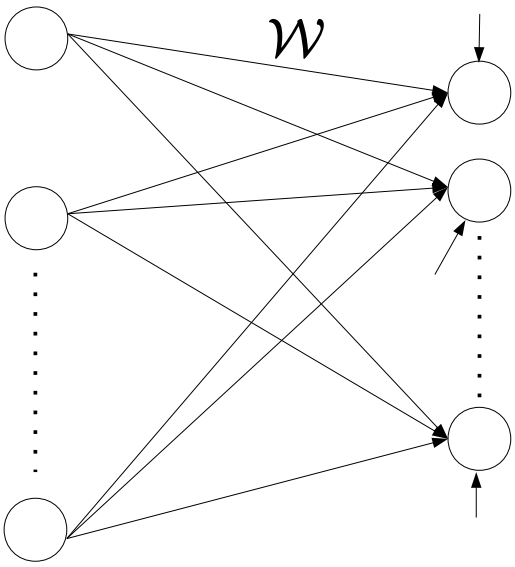

Figure 6: Graphical depiction of the single layer network. The column on the left represents the input data (256 pixels per image) and the right column denotes the output layer, with 10 nodes in total. The arrows show the connections between both layers, as well as the biases. This results in 256+1 = 257 tuneable parameters per output node.

such that input data is classified correctly. Apart from data inputs, a perceptron typically takes a bias. This bias is also a trainable parameter just like the weights and adding it empirically shows improved performance.

In the specific application of the MNIST data set we build a network consisting of 10 output nodes, one for each digit (see Figure 6). Each node has 256 inputs from the data. To each image vector $\vec{x}$ we manually append a 1 as the 257th component, such that the weight of this 'pixel' takes the role of a bias term. The altogether 257 inputs are weighted, resulting in a 257×10 weight matrix: one weight for each connection between the input to one of the output nodes, plus a bias for each of the output nodes.

The node which, upon being presented with an input image, produces the highest activation is the output digit of the network.

$$\vec{a} = \mathcal{W} \cdot \vec{x} \tag{3}$$

Where $\vec{a}$ is the activation of the output neurons, $\mathcal{W}$ the weight matrix (here of shape 257 × 10) and $\vec{x}$ the input vector of the image. The network is trained by sequentially presenting it with a single image from the training set. If the network does not classify it correctly the weights of those nodes which produce higher activation than the correct output are amended. The update rule used is the perceptron learning rule:

$$\mathcal{W}' = \mathcal{W} - \eta\vec{x} \tag{4}$$

Here η is the learning rate which dictates not only how fast the network learns, but also the scale on which

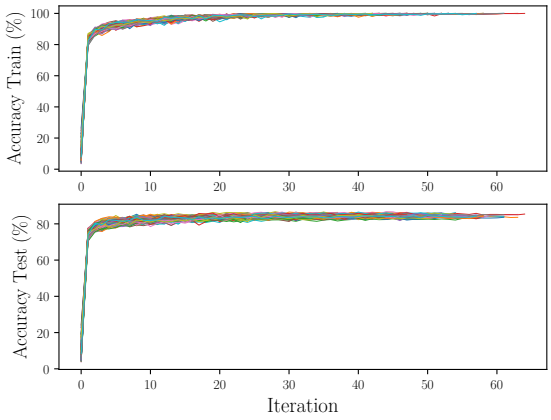

Figure 7: Accuracy achieved on the training set as a function of iterations for 100 runs. Note that after an initial steep increase the network struggles to improve. Though a few runs take up to 70 iterations, most finish around 50.

the potential landscape is explored. High η means the network learns fast, but because it takes such large steps an optimal configuration of the weights might never be found. Contrary, small η means that even though optima will likely be found, it might take infinitely long to find them given the small step size. Empirically people have found a value of $\eta = 10^{-2}$ tends to work best. We employ this value. The weights are updated for all weights vectors belonging to nodes producing higher activation than the true output. To improve the probability of the correct output being selected, not only are the weights for these wrong nodes lowered, the weights for the true output node is increased:

$$\mathcal{W}' = \mathcal{W} + \eta\vec{x} \tag{5}$$

Training is considered to be completed when the accuracy on the training set is 100%. This takes on average about 50 loops through the entire training set. After completing this training, the network is applied to the test set. Even though the network has been trained such that it is impeccable on the training set, this will not be true for the test set.

The accuracy as function of iterations is shown in Fig. 7. A dramatic increase in accuracy during the first few iterations is observed, after which it slowly continues rising. The largest variation happens after around 10 iterations, later on the variation again decreases. This makes intuitive sense, at first improvement is relatively easy and all runs rapidly increase to $\sim 80\%$ accuracy. After this initial rise it becomes more difficult to improve, in some runs the network learns quickly and in others much more slowly, however after 30 iterations the quickest runs start completing reaching 100% accuracy and the rest continues a slow rise until completion. This makes the variation in the models shrink. However, it can be observed that the variation in the accuracy on the test set is more stable. After the initial increase in variation at approximately the same point as for the

training set, a diminishing is only seen due to runs completing and dropping out. Since we decided to stop runs when 100% accuracy on the training set is achieved, the number of lines drops towards the end.

Interestingly, the accuracy on the test set barely increases after the initial rise. Whereas the training set accuracy continues to slowly rise from $\sim 80\%$, no such rise is discernible in the test set. This suggests that after the initial training, further fine tuning the weights on the training set has no beneficial effect on the accuracy for the test set.

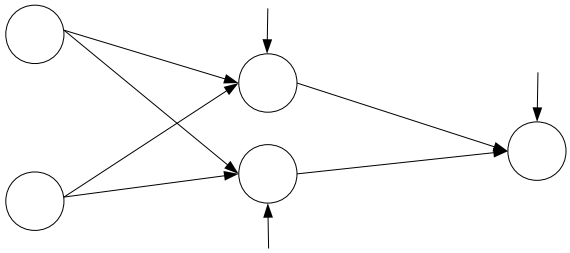

Figure 8: Graphical depiction of the XOR network

Task 5

In the previous section we considered a single layer neural network in which the weights are updated whenever the network produces the wrong output. A different approach is to use a simple gradient descent algorithm. Here the network is tested on a training set and the mean squared error(MSE) between the *true* result and the network result is computed. The weights are then updated by computing numerical gradient of the MSE.

$$\nabla_{ij} = \frac{network(w_{12}, ..., w_{ij}, ...) - network(w_{12}, ..., w_{ij} + \varepsilon, ...)}{\varepsilon} \tag{6}$$

$$\mathcal{W} = \mathcal{W} - \eta\vec{\nabla} \tag{7}$$

Again η is the learning rate, which we will now vary. ε is the gradient descent step size, which from experiments is found to be best set to $\varepsilon = 10^{-3}$. $\mathcal{W}$ is the weights matrix and $\tilde{\nabla}$ the gradient of the weights matrix. Note that in the case described here the weights matrix is 1 dimensional and thus simply a vector.

This numerical computation of the gradient descent should result in a weights matrix which provides an ever decreasing MSE and thus an increasing accuracy of the network. This algorithm is applied to the XOR network, because it is one of the simplest problems which cannot be solved by a single layer network. Table 3 lists what an XOR network should produce as output given the two inputs.

Given that single layer perceptron networks can only achieve linear separability, the solution to the XOR problem requires a multi-layer network. The simplest version of which consist of two input nodes, two hidden nodes and a single output node(see Figure 8). Both input nodes are connected with both hidden nodes, which are both connected with the output node. All non-input nodes also have a bias. This means the weights vector consist of 9 elements, 6 of those elements provide weights to connections between nodes and 3 provide weights to the biases. It now pays to think carefully about the functions governing the output of the perceptrons. Before we took this output to simply be the dot product of the inputs and the weights, but better solutions are available. Some of the most often used functions are the sigmoid, ReLu and tanh:

Table 3: The expected output of an XOR network

Input1	Input2	Ouput
0	0	0
1	0	1
0	1	1
1	1	0

$$sig(t) = \frac{1}{1 + e^{-t}} \tag{8}$$

$$ReLu(t) = max(0, t) \tag{9}$$

$$tanh(t) \tag{10}$$

The sigmoid crosses the y-axis at 0.5 and is bounded to be within 0 and 1. It gives an alternative to the simple sum in that it provides a steeper slope at the class boundary making fewer decision ambiguous. The ReLu function is often used because only positive values get activated directly proportional to the weighted input of the node. The tanh is in many ways similar to the sigmoid but returns values on the interval -1 to 1, while crossing the y axis at 0 which might make it easier to differentiate between positive and negative inputs. Recently, the ReLu function has found a large following, but it is not immediately clear whether it will produce the best results for our network. Given that we want final outputs to be either 0 or 1 depending on the input, the sigmoid might make more sense.

Using these functions and 9 weights initialized uniform random on the range -1 to 1, we train our network to represent an XOR function. While training the mean square erro(MSE) and number of misclassified inputs is counted at each iteration. The training is performed with a range of different values of the learning rate.

In Figure 9 an example of the progress for 20 different runs for 4 different learning rates using the sigmoid function is shown. Given that the iteration axis is displayed in logscale, it becomes clear the speed of convergence is widely different. This is further indicated in Figure 10. The lines indicating different learning rates are clearly separated and all shows similar trends. After an initial decline in the MSE they encounter a difficult area where the progress stalls. This is also present in the right plane where the number of misclassified items fluctuates rapidly between 1 and 2 in the corresponding area. Apparently the network is quite unstable at this point and

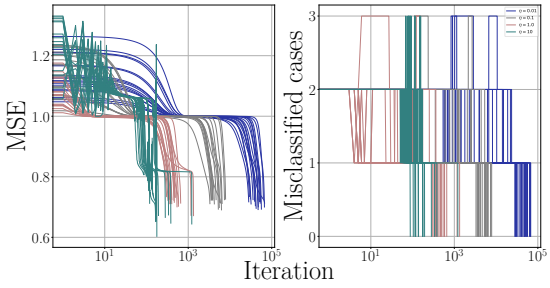

Figure 9: Progress for a Sigmoid activation for different learning rates. The left panel shows the MSE, whereas the right indicates how many inputs are misclassified. Green: $\eta = 10$, Red: $\eta = 1$, Grey: $\eta = 0.1$, Blue: $\eta = 0.01$

Table 4: Number of non-converging runs for different learning rates and activation functions

Learning rate	Activation Function		
	Sigmoid	ReLu	Tanh
0.01	2	14	0
0.1	4	6	0
1	2	20	0
10	5	20	20

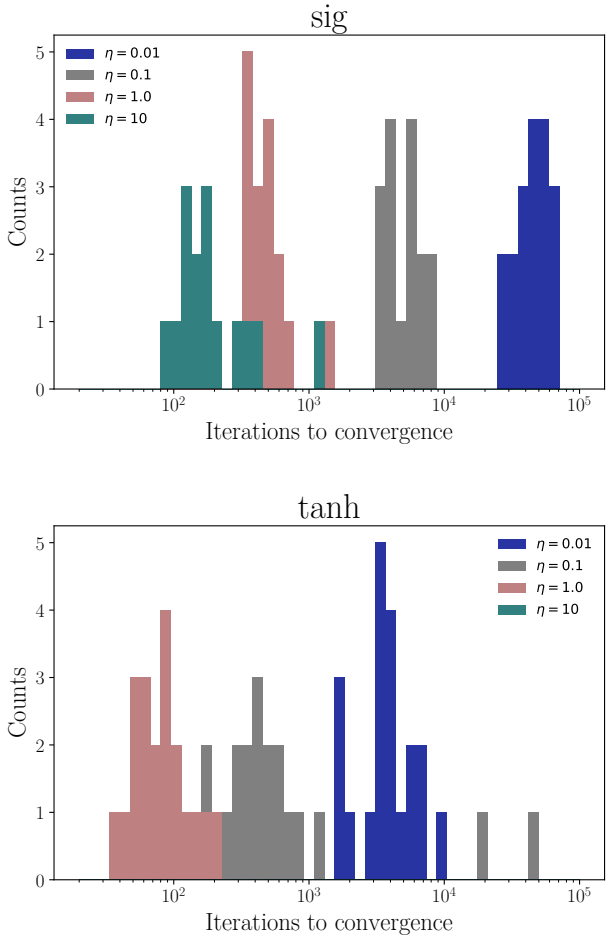

Figure 10: Iterations to convergence for the sigmoid and tanh activation functions

the learning is indecisive to which weights are most beneficial to change, leading to a rather erratic phase. After this stall the MSE rapidly drops again until the run is truncated at the point where all items are correctly classified. At this point the MSE ~ 0.7. Interestingly not all runs converge within 10^5 iterations, extending leads to no further convergences either. The stalling phase where all runs go through seems to simply never end in some cases and the weights cannot be optimised. The number of cases where this happens varies widely with activation function and learning rate. Table 4 provides an overview. The largest learning rate converges very poorly, also apparent from the wild behaviour of the green lines in Figure 9. The ReLu function seems to be not fit for this type of network, clearly demonstrated by the bad convergence properties for almost all learning rates. In contrast, the *tanh* converges extremely well for all but the highest learning rate.

From Figure 10 it seems clear that the tanh function also delivers faster convergence for all learning rates. For the highest learning rate the *tanh* no longer converges (also see: Table 4). Given the otherwise perfect and fast convergence for all other learning rates, this suggests there is a fine line between fast convergence and no convergence at all. It might be dangerous to explore that area to closely as it could potentially be very unstable under slight changes in the data. Furthermore, we note that the different learning rates for the sigmoid are much clearer separated compared to the tanh where we see a fair amount of overlap.

The MNIST dataset

Now that we have demonstrated the basics of the gradient descent algorithm and the power of multilayer networks, we endeavour to try again on the MNIST data set. Wondering whether we can improve our result by using a multilayer network, several differences with the XOR network come to mind. Whereas the XOR network only contained 9 weights, a network suitable for the MNIST data such as a 256-30-10 network contains 8050 weights(see Figure 11). Furthermore, the XOR only had four possible inputs on which to train, the MNIST network will have 1707. This seems quite daunting and it is only fair to question whether the approach used on the XOR network will still work.

The first important change compared to the XOR network is the error function. The XOR function worked with a simple MSE, that won't work for discrete classification. Instead, we opt to use the cross entropy as an error function.

$$E = \log(p_m) + \log(1 - p_{!m}) \tag{11}$$

$$p_m = \frac{a_m}{\sum(a)} \quad p_{!m} = \frac{\sum(a_{!m})}{\sum(a)} \tag{12}$$

Here p_m is the probability of selecting the correct class m given the activation a_m of node m and the activation

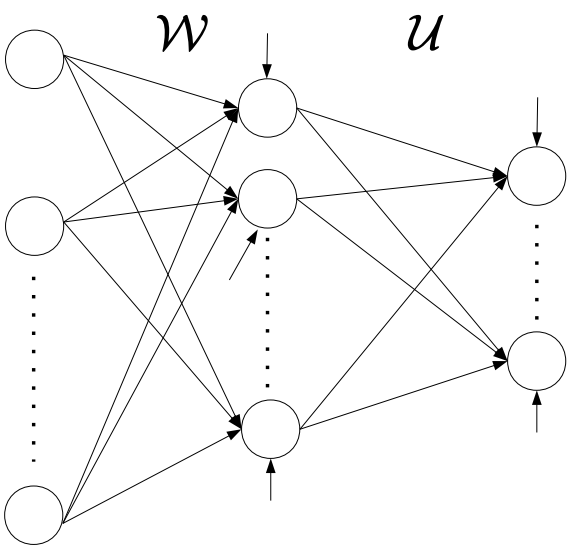

Figure 11: Graphical representation of the multilayer MNIST net with the two weight matrices $\mathcal{W}$ and $\mathcal{U}$

of all other nodes. Similarly, $p_{!m}$ is the probability of selecting a node which is not m. This error function is minimized precisely when the probability to select the correct class m becomes unity.

Secondly, some experimentation reveals that even when making full use of matrix multiplications this code will take several hours to complete compared to the mere seconds for the single layer network from Task 4. The method of numerical gradient descent is one of the major issues. Trial steps in all 8050 dimensions following eq. 6 are of equal size. This makes it unlikely to find a successful solution in such a high dimensional case. These and several other points make us shift the approach slightly to implement analytic back-propagation.

Implementing analytic back-propagation should be significantly faster as it can be expressed in matrix multiplications instead of tedious loops. It will also implicitly and automatically differ step sizes in all dimensions. The crux is to find the derivatives for the error function with respect to weights between the hidden layer and output nodes, and weights between the input and hidden layer. Once these are computed implementation is straight forward. The derivatives as used in the implementation are presented in eq. 13 and 14

$$\begin{aligned}\frac{\partial E}{\partial u_{11}} &= (1-\phi_m)\left[1-\frac{\phi_m}{\sum(\phi)}\right] y_1 \\ &+ \frac{\phi_{!m}(1-\phi_{!m})}{\sum(\phi_{!m})}\left[1-\frac{\phi_{!m}}{\sum(\phi)}\right] y_1\end{aligned} \tag{13}$$

$$\begin{aligned}\frac{\partial E}{\partial w_{11}} &= (1-\phi_m)\left[1-\frac{\phi_m}{\sum(\phi)}\right] u_1\hat{\phi}(1-\hat{\phi})x_1 \\ &+ \frac{\phi_{!m}(1-\phi_{!m})}{\sum(\phi_{!m})}\left[1-\frac{\phi_{!m}}{\sum(\phi)}\right] u_1\hat{\phi}(1-\hat{\phi})x_1\end{aligned} \tag{14}$$

Here u_{11} is an element from the matrix $\mathcal{U}$ which contains the weights connecting the 30 hidden nodes and one bias with the 10 output nodes. The weight w_{11} is from the matrix $\mathcal{W}$ which contains the weights connecting the 256 inputs and one bias with the 30 hidden nodes. ϕ is the sigmoid function with the same convention as before regarding the subscripts m and $!m$. The notation used here is shorthand for:

$$\phi = \phi(u_{11}y_1 + u_{21}y_2 + ... + u_{30,1}y_{30} + u_{31,1}b_1)$$
$$\hat{\phi} = \phi(w_{11}x_1 + w_{21}x_2 + w_{256,1}x_{256} + w_{257,1}\hat{b}_1)$$

In this y_i are the outputs from the hidden layer and x_j are the input values from the image vector. The b and $\hat{b}$ are the biases for the output and hidden layer respectively. The hat notation for the sigmoid functions is to easily differentiate between the functions for the output layer and for the hidden layer.

Using these derivatives the network is supplied with uniform random weights on the interval (-1, 1) and updated according to eq. 7. The convergence is exceedingly fast compared to the numerical gradient descent of Task 5. Figure 12 shows the accuracy as function of the number of iterations for different learning rates for 20 runs. What stands out immediately is how only the smallest learning rate proves successful within 50 iterations, after which all runs are cut short. The largest learning rate shows no progress at all, whereas the intermediate value seems to only progress very erratically and slowly.

The trend for the lowest learning rate seems very similar to that displayed by a single layer network. Note however, the accuracy on the test set is now 88.7% on average instead of 84.3%. The turn in the training set is also much smoother, delivering all runs to nearly 100% train accuracy after a limited number of runs. In the single layer network this turn was sharper and occurred at much lower accuracy after which all models still needed to continue fine-tuning.

Conclusion

We have implemented several measures to classify the limited MNIST database with 1707 training images and test 1000 images. Distance based measures were first employed to elucidate the difference and similarities between the digits. Strikingly, this very simple approach already achieved 86.35% accuracy on the training set using the Euclidean distance measure. We learned how certain digits, i.e. (0,1) are easily separable whereas other, i.e. (0, 6) are much more difficult. A Bayesian classifier was subsequently employed on the number of non-white pixels. Finding $P(d|N_{\text{pix}})$ we could classify the images based on the number of non-white pixels. Again (0,1) was most easily separated, whereas most other digits proved more challenging. Given the classifier in use, this makes intuitive sense. To improve the accuracy, a single layer neural network was employed as a next step in complexity. This yielded a whopping 100% accuracy on the training set and managed to

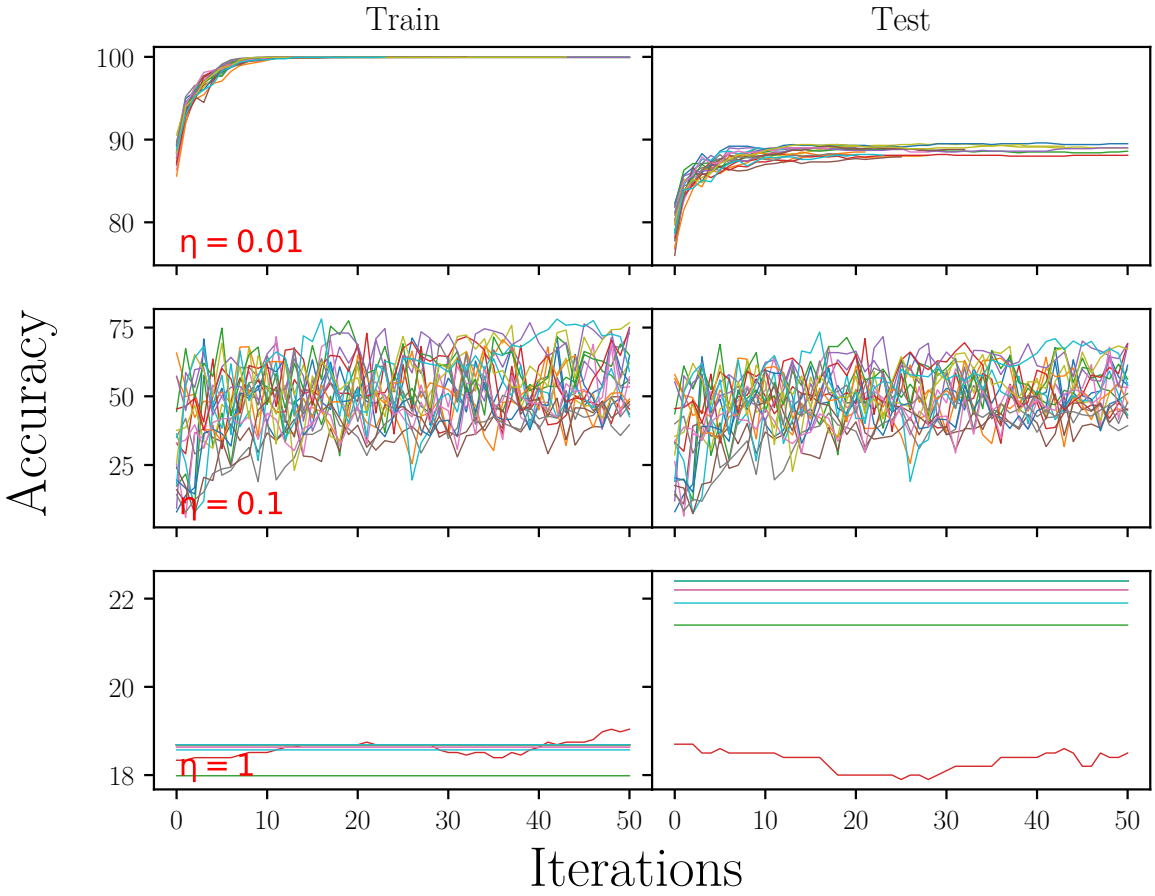

Figure 12: Progress on the MNIST data set using a 256-30-10 network with back-propagation for different learning rates in 20 runs. Clearly small learning rates work best, and both train and test set converge quickly to their final values.

classify 84.3% of the test digits correctly. In the spirit of Neural Networks we now fully ignore any previous knowledge about our data set and do no longer differentiate in performance between different pairs of digits. To present a simple multilayer network we change tack slightly and train an XOR network, trying different activation functions and learning rates it was trained using numerical gradient descent. The sigmoid and tanh functions compete for the top spot and thought the tanh wins on convergence speed there are indications of instability for the egregious learning rates used. A sigmoid function might be a slightly slower, but safer choice if extensive testing is not an option. This approach of multilayer networks was finally applied to the limited MNIST database implementing back-propagation instead of numerical gradient descent. For small learning rates convergence is exceedingly fast and an average accuracy of 88.7% is achieved on the test set, annihilating all other methods and clearly proving the why such networks are commonly used.

In conclusion, we attempt various methods of classifying a limited MNIST database while demonstrating the basics of neural networks. Unsurprisingly, a multilayer network with back-propagation produces the best results.

References

Ciresan, Dan, Ueli Meier, and Jrgen Schmidhuber (2012). "Multi-column deep neural networks for image classification". In: *IN PROCEEDINGS OF THE 25TH IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR 2012*, pp. 3642–3649.

Lecun, Y. et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.