Neural Networks

Prof. W. Kowalczyk

Assignment Report

Assignment 1

1 Introduction

For this Assignment, we will develop and deploy different algorithms which should recognize handwritten digits. These digits are represented by images of 16x16 pixels as a vector of length 256 whose values range between -1 (black) and 1 (white). They are split into a training set of 1707 images as well as a test set comprising of 1000 images.

2 Task 1: Analyze distances between images

For the first task, we classified the images with a very simple classifier. For this, we interpret the set of all training images of an arbitrary digit as a "cloud" in the 256-dimensional space (hereafter "phase space" for brevity). Thus, we can calculate the center of that cloud, C_d, as the mean value of all respective pixels for all images depicting the same digit. Then, we can classify pictures based on their distance in the phase space to the centers of these 10 clouds whereby we assign to a picture the digit whose center is closest to the point of that image in the phase space.

First, we calculate the number of images depicting a digit and store these numbers in variable "n". Then we read in the coordinates of all pictures in the training set in the phase space, sum up the values for each coordinate for each digit in the "Sum" and then calculate the center of these clouds, C_d for all ten digits d based on the training set as well as the radius r_d of these clouds. This information is stored in "c" and "r". The latter is defined as the distance from the center of a cloud to its farthest member. Then, we compute the distance of every cloud's center to the centers of the other digits and store the values in "distancematrix". A larger distance between two centers imply that the two corresponding digits are written more different compared to a pair of two other digits whose centers are closer to each other. Table 1 gives the radii of the center for each digit as well as the distance to the closest other center and its corresponding digit.

The table shows that the digit 9 has the largest radius with 16.1 pixel with 0 having a slightly smaller radius. This implies that these two digits are looking more different than the other digits, i.e. were written more different. This in turn means that these two digits per se bear the highest danger to be confused with another digit. In contrast to that, the digit 1 has the smallest cloud with a radius of 9.5 pixel. The "ones" thus were written rather similar and should be easier to distinguish from other digits. The other digits have similar large clouds with a radius around 14 pixel. The two digits which are the hardest to distinguish are 7 and 9 whose centers are mere 5.4 pixels distant from each other. 1 is the digit whose center is the farthest from the other centers as its 9.9 pixels away from the center of 9. As we presumed from its comparable small radius, it is indeed the digit which is easiest to distinguish from the other digits. In total, we have four digits who have 9 as their closest neighbor (1, 4, 7 and 8) as well as three with 5 as their closest center (0, 3 and 6). As a small distance between two centers imply a high risk of confusion, we presume that 9 and 5 will most frequently be mistaken for a different digit and vice versa. The radius of the clouds (14 pixels in average) for all digits, save digit 1, is two to three times bigger than the distances between their centers clouds. The reliability of this approach is therefore highly debatable.

Digit	0	1	2	3	4	5	6	7	8	9
Radius of phase cloud [px]	15.9	9.5	14.2	14.7	14.5	14.5	14.0	14.9	13.7	16.1
Distance to closest center [px]	7.5	9.9	7.1	6.1	6.0	6.1	6.7	5.4	6.4	5.4
Corresponding closest Digit	5	9	8	5	9	3	5	9	9	7

Table 1: Table of the radii of the corresponding clouds for all ten digits as well as the distances to the closest digits.

3 Task 2: Implement and evaluate the simplest classifier

Now, we will develop a classifier with that approach. Our Classifier will calculate for every image the distance in the phase space to all ten cloud centers and then assign them that digit whose center is closest to the vector of that image. Afterwards, we will analyze the success of this approach.

First, we calculate the coordinates of the centers of all 10 digits as described above. Then we go through the data again and calculate for all images the distance to all 10 centers with four different metrics (Euclidean, Manhattan, Cosine and Correlation) and store them in the lists “disteuclid”, “distmanhattan”, “distcosine”, “distcorrelation”. The shortest distance for each metric, image and metric is then stored in “trainpredictioneuclid”, “-manhattan”, etc. The Euclidean metric calculates a distance between two vectors with the well-known Pythagorean Theorem. Using the differences of two vectors $\vec{v}$ and $\vec{w}$ in two coordinates x and y this means: $d_{Euclid} = \sqrt{(v_x - w_x)^2 + (v_y - w_y)^2}$. The Manhattan metric calculates a distance between two points as the absolute sum of the differences in the respective coordinates: $d_{Mahattan} = |v_x - w_x| + |v_y - w_y|$. A Cosine metric computes the similarity between two vectors, i.e. the angle θ between them. Thereby, it is independent from their lengths: $d_{Cosine} = cos(\theta) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| \cdot |\vec{w}|}$. Then, we classify (“predict”) the image as that digit whose center has the shortest distance in phase space to the image. Afterwards, we calculate a confusion matrix called “confusion_matrix” with “confusionmatrix” from sklearn.metrics. Then for all four metrics we compute the number of images successfully assigned and store that information in “testcountereuclid”, “-manhattan” etc. Then the ratio of that number to the total number of images ´stored in “n” gives the success rate of the method. This process is then repeated for the test set.

Now, we will describe the results of the classifier on the training set. When we take a look at the resulting confusion matrices (which can be seen as output of the code), we can see that all four matrices most often confuse a 6 in an image which they wrongly classify as a 0. The Manhattan metric confused the most images in that regard: from 319 images classified as 0, there were 248 correctly classified and a total of 57 images showing a 6 were wrongly classified as a 0. As expected from Task 1, a 1 is very easy to distinguish: in training, every time the code predicted a 1, it was correct (this was true for all four metrics). In contrast to that, 1 and 9 are the two digits which were wrongly classified most often. In the case of 1, it was often mistaken as a 4 by the Euclidean and Correlation metric and very often as a 2 by the Manhattan metric. 9s were often mistaken for a 4 (especially by the Manhattan metric) or as a 7. In Task 1, we observed that the centers of 7 and 9 are closest together. Here, this is reflected by the aforementioned high rate of a 9 being mistaken as a 7. However, vice versa this is not true: All four metric don’t show a higher than usual amount of 7s wrongly classified as a 9.

Now, we will discuss the results from the test set. In general, the results are quite similar to those obtained from the training set, but the confusion generally is higher. Again, all four metrics often mistook a 6 as a 0 but the Manhattan metric now did not fail significantly more often than the other. Again, nearly every time a 1 was called it was correct. At the same time, an image of a 1 was very often mistaken for other digit, especially for a 2 or a 4. The method seems to be quite bad at predicting a 2: in about a third of the cases images classified as a 2 were actually another digit, often an 8. Lastly, let's talk about the success rates of the algorithm with the four metrics. The rates of algorithm with Euclidean, Cosine and Correlation metrics ran on both sets are higher than that of the Manhattan metric. The highest accuracy rates obtained were produced by the Euclidean and the Correlation metric on the training set: >86%! Naturally, the success rates in general are significantly lower for the test set.

Metric	Euclidean	Manhattan	Cosine	Correlation
Training Set	86.4	76.6	86.1	86.7
Test set	80.4	72.1	79.9	80.6

Table 2: Table of success rates in % of our algorithm with four different metrics run on the training set (with which the coordinates of the centers in phase space were calculated) as well as the test set of images.

4 Task 3: Implement a Bayes Rule classifier

Here, we will use Bayesian statistics to classify the images. For this, we will select a certain feature X of the data based on which we will try to distinguish between two digits. We decided to distinguish the digits 1 and 8 based on the area of the images that is colored, i.e. the sum of the pixel values (hereafter dubbed "activation sum"). We expect this to work quite well since a written 8 in general covers more space than a written 1. The implementation is simple. While looping through all images of the training data showing either a 1 or an 8, we add the sum of all pixel values in that image to the lists "activation1" and "activation8", respectively. These two lists represent the probabilities $P(X|C)$ for C=C1 or C8. Now we use Bayes' Theorem and compute the posterior probability:

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

Where $P(C|X)$ is the posterior probability density function (PDF) and represents the probability that an image with value X for the feature belongs to class C (which in our case can be digit 1: C1 or digit 8: C8), $P(X|C)$ is the likelihood and represents the probability that a class C has a certain value X for the feature. $P(C)$ is the prior probability that an image belongs to class C. P(X) ensures the normalization of the posterior PDF.The prior $P(C)$ is basicly equal to the frequency of the digits in the sets sets and can be seen in table 3. We calculate $P(C|X)$ and thus $P(X|C)$ as distributions per bin of activation sum and plot the resulting $P(X|C)$ for the training set in figure 1 between the values 0 and -250. $P(C8|X)$ is much wider distributed around its mean at S≈-110 than $P(C1|X)$, which may be a source for an increased chance of wrongly classifying an 8 as a 1. $P(C1|X)$ in contrast is much more confined around its mean at S≈-180. Now we need a boundary with which we will determine of an image shows a 1 or an 8. In a perfect world, we simply would calculate the value of the activation sum where the continuous distributions of both digits reach the same count of images. However we are working with discrete distributions, so the posterior PDF's of the two classes will not be exactly equal

at any X but they will be very close to one another around the boundary. Since the data is not very well sampled, we simply set the boundary to the X where the two distributions come very close to each other. We do this in looping through all values in $P(C1|X)$ and once the value of $P(C1|X)$ at an activation is less than 0.0008 (This value ensures the correct boundary) away from the respective value of $P(C8|X)$, we set the boundary as the bin of this activation sum. Thus, we set the decision boundary at **-141.7**.

In this next step, we use Bayes Theorem to classify all images of the test data showing a 1 or an 8, to either class $C1$ or $C8$ using their activation sum X. If $X < boundary$ then the digit is classified as digit 1 and if $X > boundary$ then it is classified as digit 8. For that, we go through those images of the test data showing a 1 and calculate the resulting activation sum and store that information in "activation1". If this value is smaller than the boundary, we classify this image as an 1 since the low activation sum indicates that only a comparable small area of the image was being painted. If the value is higher than the boundary, we classify it as an 8. This is repeated for all test images depicting an 8. We store this decision together with the actual digit for both digits. Then, we compare the predicted values (our decision) to the actual digit and list the accuracies in list 4.

The results are quite good for both data sets with success rates ranging from 80% to up to 87%. Of course the accuracy drops by a few percentage points from the training to the test set - however, the success of classifying an 8 is slightly higher in the test data set (87%) than in the training data set (86.8%). Meanwhile, the success drops significantly more for the digit 1 from 86.9% for the training set to 80.1% for the test set. In general, images are being misclassified when they are situated at the end of the corresponding distribution, thus on the wrong side of the boundary. In 1 they can be seen as purple bars. When compared to the much simpler classifier in task 1 and 2, this algorithms achieves a significantly worse accuracy considering we are only trying to classify 2 digits compared to the previous 10 digits.

	Total # of 1's and 8's	# of 1's	# of 8's	$P(C1)$ [%]	$P(C8)$ [%]
Training Set	396	252	144	63.6	36.3
Test Set	213	121	92	75.2	43.2

Table 3: Table of $P(C)$ for both digits and both data sets.

Digit	1	8	1 and 8
Training Set	86.9	86.8	86.9
Test Set	80.1	87.0	83.1

Table 4: Accuracy in % of the Bayes classifier

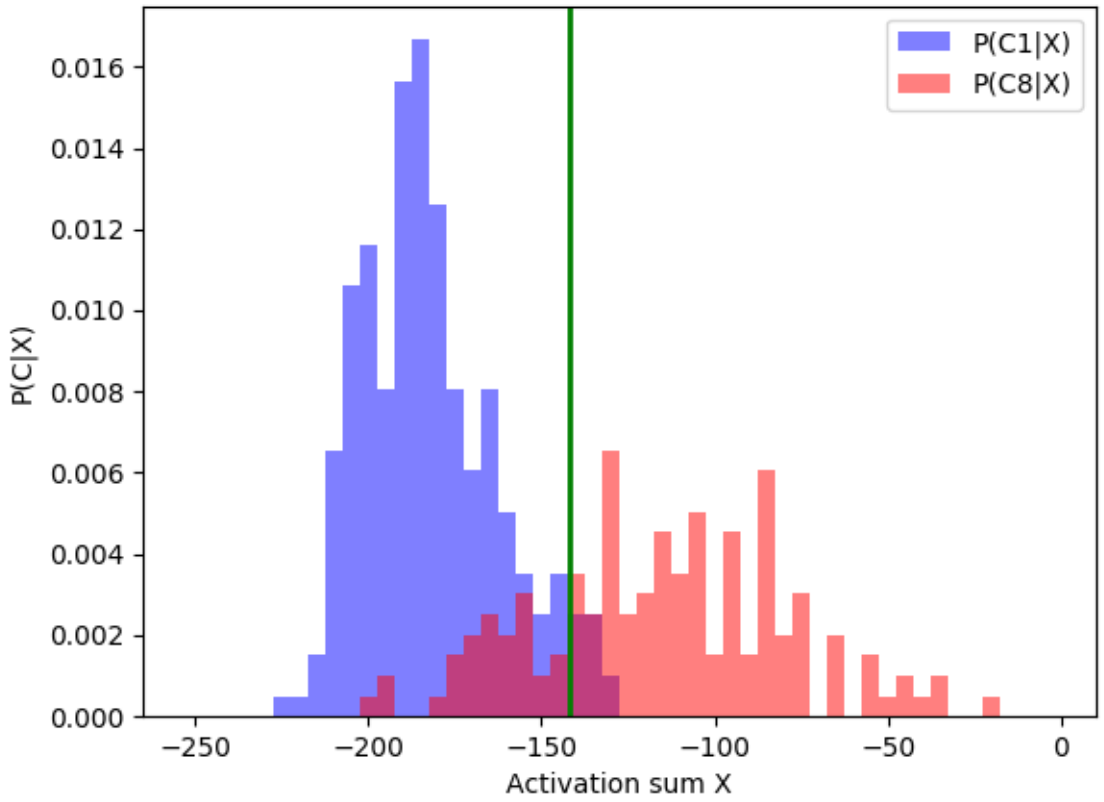

Figure 1: Histogram of the activation sum for all images showing a 1 (blue, purple on the **right** of the boundary) or an 8 (red, purple on the **left** of the boundary) with the decision boundary (green) at X=-141.7.

5 Task 4: Implement a multi-class perceptron algorithm

In this task, we will build a multi-class perceptron training algorithm with which we will train a single layer perceptron to classify digits. The latter will have 10 nodes, corresponding to the 10 digits, with each node being connected to every of the 16x16=256 perceptrons and thus having 256+1 inputs as well as 1 output. After the training on the training set, we will run the algorithm on the test set to test its success.

The code works as follows. After it reads in the data in the training set and stores the information in "x_train"(i.e. a vector $\vec{x}$), we initialize random weights and biases for all 10 digits to obtain the first (random thus imprecise) prediction in array "weights" (i.e. a matrix $\hat{w}$). In total, this amounts to 10x(256+1)=2570 weights for ten digits. Furthermore we initialize a vector "y_train" ($\vec{y}$) for the result of the prediction. Then we make the first prediction by performing a matrix multiplication:

$$\vec{y} = \sigma(\hat{w} \cdot \vec{x} + \vec{b})$$

This first prediction will probably be inaccurate. Then, we check which images were misclassified and store that information in "misclassified". Now we train the algorithm. For all images,

the weights in every line corresponding to another digit than the actual digit are being reduced by the values of the image while the weights in the line corresponding to this digit are being raised by the values of the image. When this update is done, we again let the algorithm predict the digits in the training data stored in "x_train", write the results into "y_train" and update the list of misclassified images, "misclassified". We do this and update the weights and biases until this list is empty, i.e. until no images are being misclassified anymore and thus reached an accuracy of 1.0 on the training set. This concludes the training of the algorithm. Afterwards, we run the algorithm on the test data set with the final weights and biases to predict the digits as described above. We store the information of the predicted and the actual digit for all images and calculate the accuracy of the algorithm.

The total runtime of the algorithm amounts to 75 to 90 seconds (depending on the device used) and the algorithm needs between 1750 and 1950 steps to be fully trained on the training set i.e. to be ready for operation. With the thus acquired configuration, the algorithm achieved an accuracy of 87% to 88% on the test set. This is quite good for such a simple neural network (no hidden layers), albeit the very first and simplest algorithm reached a comparable accuracy.

6 Task 5: Implement the Gradient Descent Algorithm

In this section we will discuss how we built and trained a neural network that simulates an exclusive or (XOR) gate. More specifically our goal was to build a neural network consisting of two input nodes, two hidden notes and one output node. If the values of the two input nodes (x_1, x_2) are equal to (0,1) or (1,0) the output of the output node should be 1 and for an input of (0,0) and (1,1) the output should be 0. The output of the neural network is a function of the input values and the weights:

$$XORNET(x_1, x_2, weights)$$

To build a neural network like this we need to train the neural network in such a way that the weights in the neural network give the desired output for the specified input values. To do this we will use the Gradient Descent Algorithm. This requires us to define a cost function, which in our case will be the Mean Squared Error (MSE). The MSE for a given set of weights in the network is equal to the square of the difference between the output value and the desired output value for the specified input values divided by the total number of output values. Mathematically this looks like this:

$$\begin{aligned} MSE(weights) = \frac{1}{4}[&(0 - XORNET(0,0,weights))^2 + (1 - XORNET(0,1,weights))^2 \\ &+ (1 - XORNET(1,0,weights))^2 + (0 - XORNET(1,1,weights))^2] \end{aligned}$$

For randomly initialized weights this MSE will be larger than zero and in order to get a smaller MSE we will update the weights in the steepest direction in 'weight space' that lowers the MSE. This direction can be found by computing the gradient of the MSE. We will only adjust the weights by a small fraction of the gradient in its direction as the gradient only applies locally at the computed position. This small fraction is also called the step size or learning rate η. After having updated the weights we will recompute the MSE and repeat the process until the weights have reached a (local) minimum for the MSE. If this local minimum is also the global minimum we will have converged to the weights and network we wanted. This however is not necessarily the case and our success will depend on how we initialize the weights, what learning rate we use and which activation function we use to compute the output of each hidden node and output node. We have run the training algorithm for the XORNET for different learning rates, different initialization of weights (namely drawn from a normal distribution with mean 0 and variance 1 or drawn from a uniform distribution between -1 and 1) and different activation functions (sigmoid, tanh, ReLu). The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

The tanh function is defined as:

$$\tanh(x) = \frac{\exp(x) - xp(-x)}{\exp(x) + exp(-x)}.$$

And the ReLu function is defined as:

$$ReLu(x) = \max(0, x).$$

The obtained results for the sigmoid activation function can be found in figure 2. It seems that a larger learning rate converges quicker to the global minimum of the MSE. The top row are

the learning curves for initialization drawn from normal distribution and the bottom row for a uniform distribution, it also seems that drawing from a uniform distribution seems to be better for quick convergence.

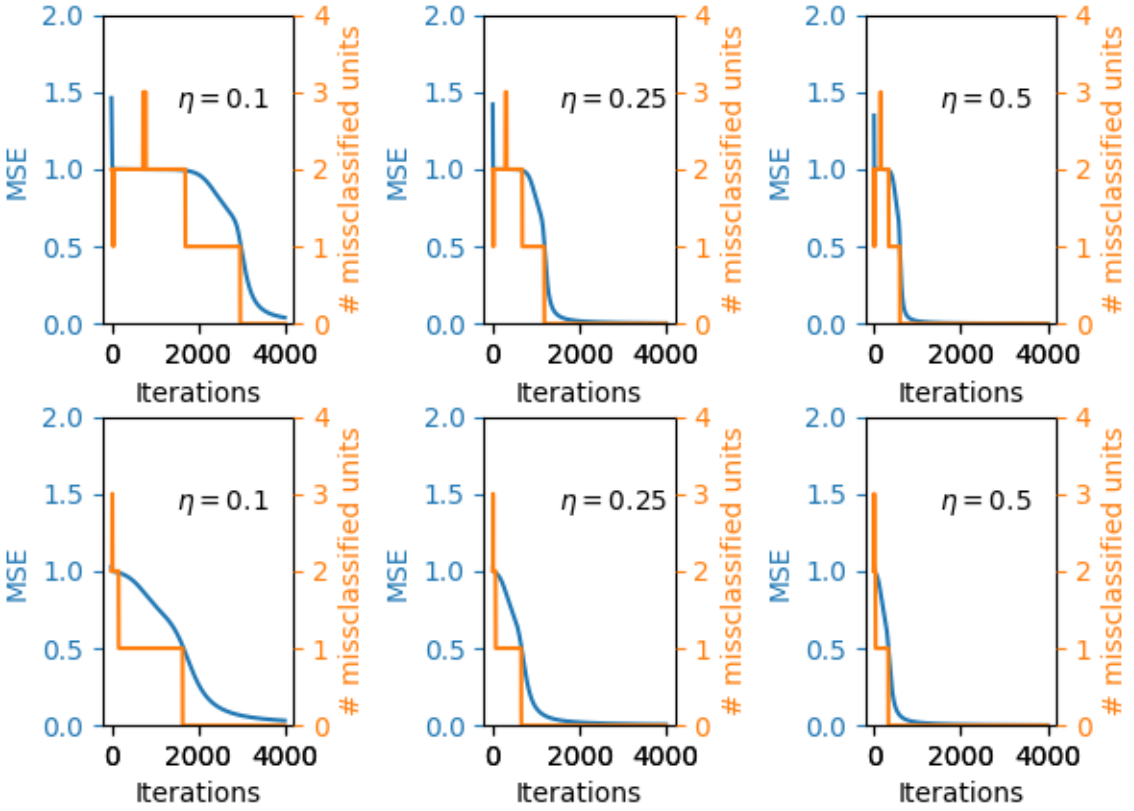

Figure 2: The Learning curves of our neural network when using the sigmoid activation function. The figures from left to right are plotted for increasing learning rate: $\eta = 0.1$, $\eta = 0.25$ and $\eta = 0.5$ respectively. For the top row the initial weights are drawn from a normal distribution with a mean of 0 and a variance of 1. For the bottom row the initial weights are drawn from a uniform distribution between -1 and 1.

The results for the tanh activation function are shown in figure 3. What can be seen in the bottom left and center plots is that for certain initial weights the network converges very rapidly to the global minimum, much faster than the sigmoid activation function. However what can also be seen is that it can get stuck in a local minimum in the top three plots without any signs of coming out of it. The bottom right plot shows why its potentially risky to pick a learning rate that is too large: the algorithm doesn't converge, but bounces between two minima. This is similar to the $f(x) = x^2$ analogy where if we just take the derivative $2x$ with learning step $\eta = 1$ then starting at $(x = 1, y = 1)$ the gradient descent algorithm will never converge to $(0, 0)$ but will bounce between $(-1, 1)$ and $(1, 1)$ forever.

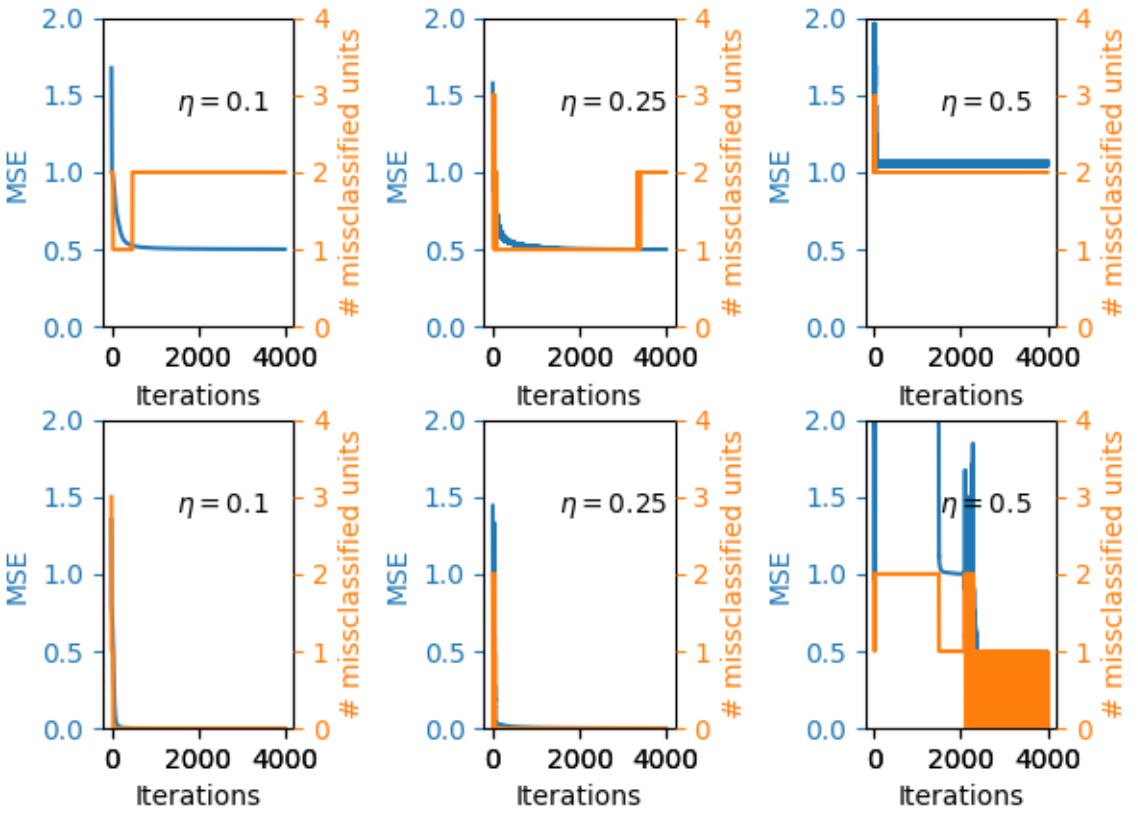

Figure 3: The Learning curves of our neural network when using the tanh activation function. The figures from left to right are plotted for increasing learning rate: $\eta = 0.1$, $\eta = 0.25$ and $\eta = 0.5$ respectively. For the top row the initial weights are drawn from a normal distribution with a mean of 0 and a variance of 1. For the bottom row the initial weights are drawn from a uniform distribution between -1 and 1.

The results for the ReLu activation function are shown in figure 4. At first glance it seems like there is something going wrong as the MSE never decreases, however what we see here is the dying ReLu problem. A large enough gradient going through a ReLu neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. This is because the gradient of $ReLu(x) = max(0, x)$ is 0 for x<0, which means that our gradient descent algorithm will never update the weights. This problem could be remedied by picking a smaller learning rate, but even with a smaller learning rate the problem can still show up. A solution to this would be to use an altered activation function like $max(0.1x, x)$, because this will allow a chance for the neuron to recover since the gradient is never 0.

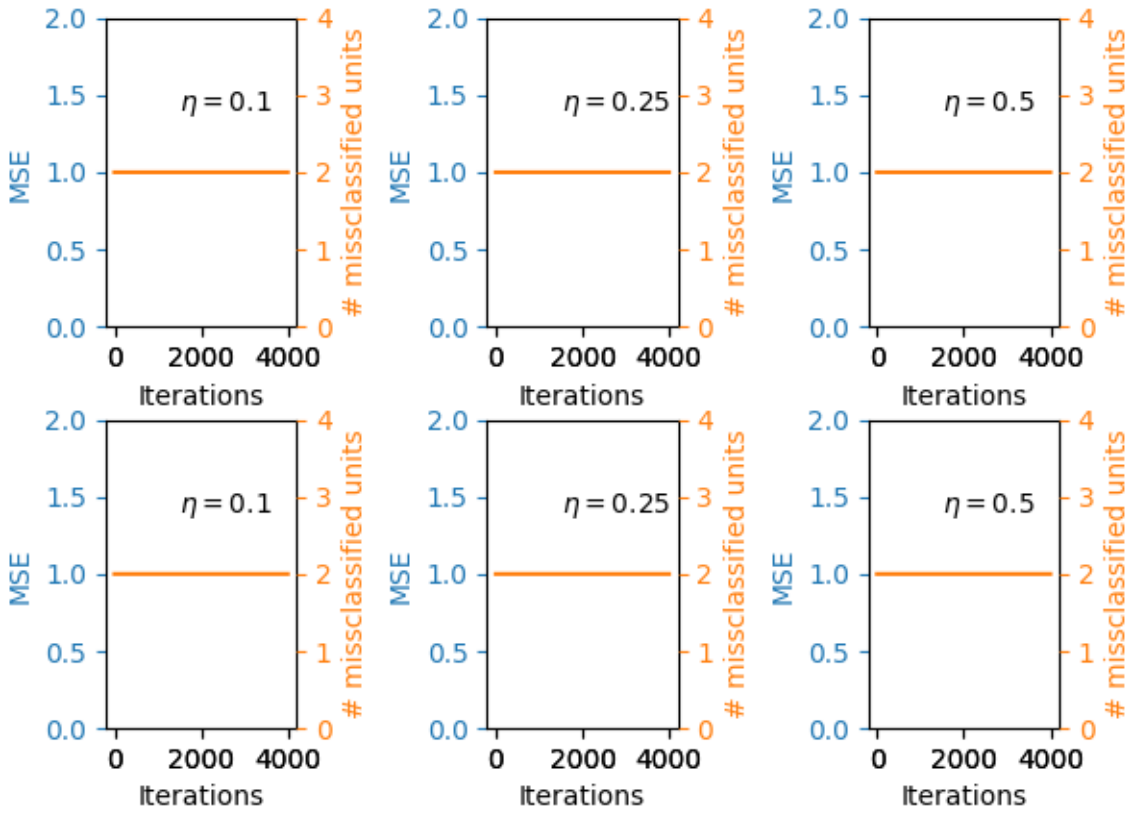

Figure 4: The Learning curves of our neural network when using the ReLu activation function. The figures from left to right are plotted for increasing learning rate: $\eta = 0.1$, $\eta = 0.25$ and $\eta = 0.5$ respectively. For the top row the initial weights are drawn from a normal distribution with a mean of 0 and a variance of 1. For the bottom row the initial weights are drawn from a uniform distribution between -1 and 1.