Neural Networks: Assignment 1

Spring semester 2019

Introduction

We will be applying various forms of classification algorithms to the famous MNIST data set. The data, a collection of handwritten digits, are 16×16 pixel images stored in the form of a vector $\mathbf{v} \in \mathbb{R}^{256}$. Once an algorithm (e.g., a $(256+1) \times 10$ weight matrix $\mathbf{W}$ in Task 4) has been developed using a set of training data we can determine its effectiveness in classifying provided testing data. With the exception of the Bayes rule classifier developed in Task 3 all of the algorithms in this assignment employ machine learning principles with various levels of sophistication: Task 2 uses a variant of the k-nearest neighbors classification scheme, Task 4 a single-layer perceptron, and Task 5 gradient descent algorithm. The classification algorithm that has achieved the most success with the MNIST data (see `https://arxiv.org/pdf/1202.2745.pdf`) had an accuracy of 99.77 %. However, that involved a convolutional neural net with significant preprocessing of the data so we will find that our relatively simple algorithms will have more modest success.

Task 1: Analyze distance between images

To begin this analysis we had to sort the training input data based on the output data, e.g. all of the vectors classified as 1's were stored so that the center and radius of that classification could be computed. The vector $\mathbf{x}_d$ representing the center of the cloud C_d was computed with the formula $\frac{1}{N}\sum_i \mathbf{x}_d^i$, where N is the number of vectors with the classification d in the training set and $\mathbf{x}_d^i$ is the ith vector in that classification. In order to find the radius of each cloud C_d we had to iterate through each vector in that cloud and find the distance $|\mathbf{x}_{i,d} - \mathbf{x}_d|$ that was largest for all of the vectors in that classification.

With the center and radius of each cloud C_d computed we had 10 ($d = 0, \ldots, 9$) spheres scattered throughout $\mathbb{R}^{256}$. To give a quantitative measure of how difficult two clouds were to separate we computed the distance between the ith and jth cloud center, $|\mathbf{x}_i - \mathbf{x}_j|$. The result made intuitive sense as this calculation revealed that 0's and 1's are the easiest to separate while 7's and 9's are the most difficult.

Task 2: Implement and evaluate the simplest classifier

Naïvely we would expect that it would be sufficient to check if a particular vector were within a cloud but that classification scheme would break down if there was overlap between the spheres. With that implementation the error between test input and output was over 80 %. Thankfully there is a straightforward classification scheme that is much more accurate. For every vector in the test input data set $\mathbf{x}_i$ we computed the distance to the center of each cloud C_d. Whichever distance was the shortest would then give the desired designation. In the case of the clouds C_7 or C_9, for example, a vector might lie within both but be considerably closer to one of the centers.

Figure 1 shows the confusion matrices computed using the training and testing data sets where the computed distances were Euclidean. As expected 0's and 1's were comparatively easy to distinguish. In both cases there were 7's and 9's that were misclassified as the other but the most common mistake in both training and test was classifying a 6 as a 0. In both the training and testing data sets the hardest digit to classify was 5; the most common misclassifications were 3 and 8, which makes intuitive sense. Using a data set on which the classification algorithm was not trained yielded a 20% error, which is remarkably worse than the 12% error achieved by LeCun et. al's one-layer NN in 1998.

Of course there are several ways to determine distances and so far we have implemented the simplest. Thankfully the `sklearn` library provides several metrics with which to compute the distance between the endpoints of two vectors. We tried the metrics 'euclidean', 'correlation', 'seuclidean', 'cosine', and 'manhattan'; we found that the only one that improved upon the Euclidean distance measurement was the 'correlation' metric; see `https://blogs.sas.com/content/iml/2018/04/04/distance-correlation.html` for more details.

Task 3: Implement a Bayes rule classifier

In order to implement a Bayes rule classifier, our first task was to come up with and implement a feature to discriminate between the digits. We decided to use the level of asymmetry as discriminating factor. We implemented this feature by calculating the absolute difference in contrast values between the left and the right side of every row per digit. Since every row consists of 16 pixels, the middle line was drawn between the seventh and eighth index, as Python indexing starts at zero. The total difference per row was calculated by taking the sum of the absolute differences at every index from the middle line. These absolute differences were calculated by subtracting the reverse of the array on the right side from the left side of the middle line.

By calculating an average asymmetry value for all of the digits in the training set, we could get an overview on the amount of (a)symmetry in the digits, as provided in table 1. Most of the digits show expected asymmetry values. For instance, since the digit one is just one straight line in most cases we would expect a low asymmetry value, and for digits 2, 4

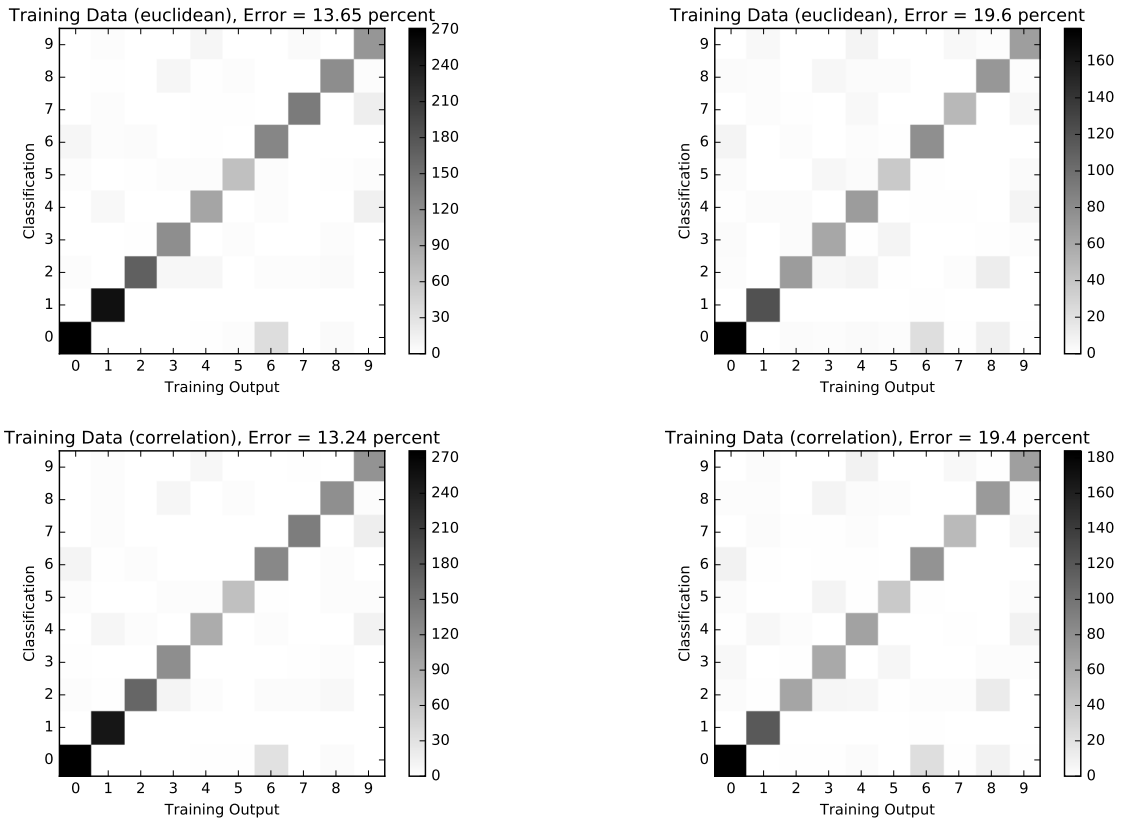

Figure 1: **Task 2**. Using the centers of each cloud C_d computed with the training data we found these confusion matrices for the training and testing data, respectively. The distance metric used for the top two matrices was Euclidean. The metric used for the bottom two matrices was the so-called correlation distance.

or 5 we would expect a higher asymmetry value as they have different shapes on both sides of the middle line. Looking at the data, this roughly seems to be the case. As the task was to keep the classifier simple, we had to use just two digits to discriminate between. To ensure good classification and to optimize our results, we chose to use digits one and five, since their average asymmetry value had the biggest difference.

By discretizing all of the samples for these digits into ten bins of equal width, a more detailed insight into the asymmetry values could be created by means of a histogram as presented in figure 2.

In order to use this feature in a Bayesian classifier, we had to calculate probabilities $\mathbf{P}(\mathbf{C}_1|\mathbf{X})$ and $\mathbf{P}(\mathbf{C}_5|\mathbf{X})$ where $\mathbf{C}_1$ is the probability of a digit being 1 and $\mathbf{C}_5$ the probability of a digit being 5. These probabilities could be obtained using the Bayesian rule with the following probabilities that were easier to obtain:

$$P(\mathbf{C}|X = x) = \frac{P(X = x|\mathbf{C}) \times P(\mathbf{C})}{P(X = x)} \tag{1}$$

Using this Bayesian rule we could predict for all the digits in the test set whether it was more likely to be a one or a five. The highest probability would decide if the digit was

Table 1: **Task 3**. Asymmetry value overview

Digit	Asymmetry value
0	42.6001
1	13.9671
2	76.3558
3	70.8656
4	62.6055
5	77.4272
6	55.0181
7	58.5029
8	55.5739
9	45.1624

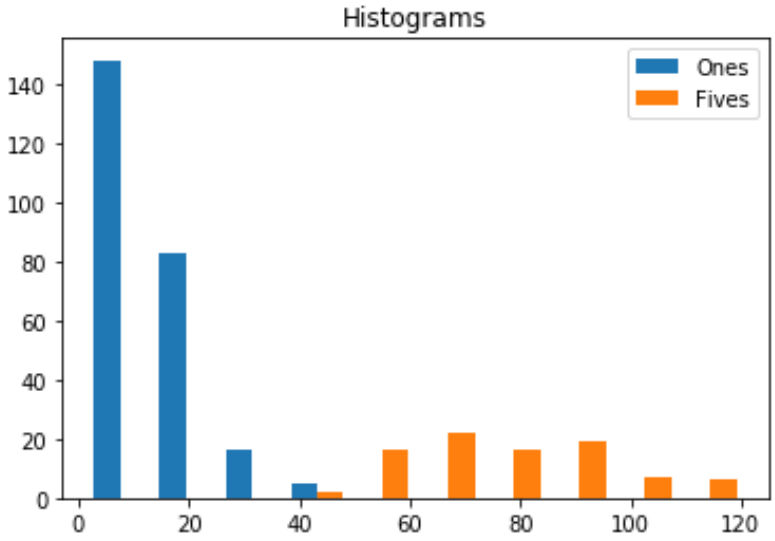

Figure 2: **Task 3**. Asymmetry values for ones and fives

classified as either one or five. When comparing our predictions to the actual output of the test set we found the results as shown in table 2.

Table 2: **Task 3**. Accuracy of our Bayesian classifier

Correctly classified ones	114/121
Correctly classified fives	55/55
Total accuracy	0.9602

Task 4: Implement a multi-class perceptron algorithm

The theory behind our perceptron algorithm is fairly straightforward. We needed to set up a Python class, `perceptron`, that would create a matrix of weights such that each column vector $\mathbf{C}$ contained a bias w_0 (zeroth element) and the weights vector $\mathbf{w}$ indicative of that digit. To create an object of this nature the following inputs would need to be provided by the user: the dimensionality of the training vectors (in the case of the MNIST data, $\mathbb{R}^{256}$), training step size η, and number of iterations I. By using an activation

$$f(\mathbf{C}, \mathbf{x}) = w_0 + \mathbf{w} \cdot \mathbf{x}, \tag{2}$$

$$Activation = \begin{cases} 1 & f(\mathbf{C}, \mathbf{x}) \leq 0, \\ 0 & f(\mathbf{C}, \mathbf{x}) > 0, \end{cases} \tag{3}$$

we were able to determine whether or not a particular input vector $\mathbf{x}$ could be reasonably classified by a particular digit's weights. We began by randomly initializing the weight matrix and then appending each column vector in the following way:

$$w_0 = w_0 + \eta \times Activation, \tag{4}$$

$$\mathbf{w} = \mathbf{w} + \eta \times Activation \times \mathbf{x}. \tag{5}$$

These equations were implemented in a `while` loop such that the weights were updated I times. With this formulation the weights are only adjusted if there isn't a match between the input vector and that particular digit.

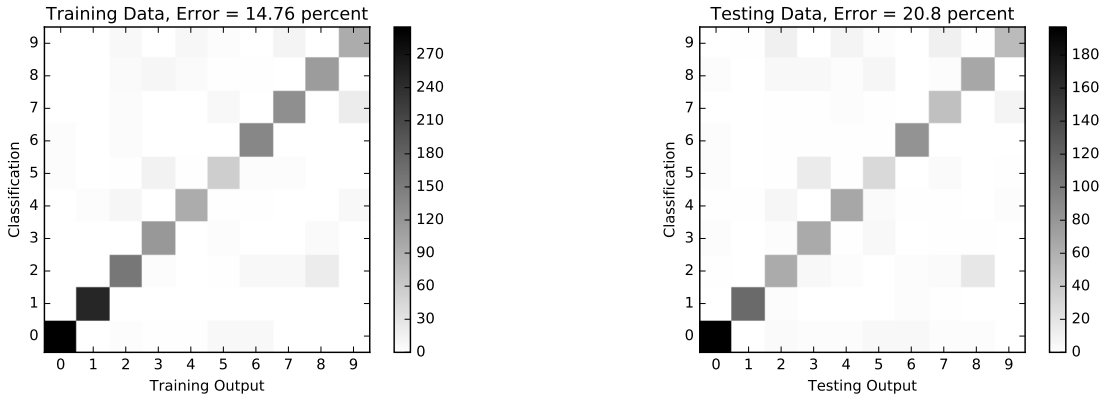

Figure 3: **Task 4**. Results of applying the single-layer perceptron to the MNIST training and testing data, respectively.

Once the weights matrix is determined using the training data we must determine the accuracy of the perceptron. For each input vector $\mathbf{x}$ we compute $f(\mathbf{C}_d, \mathbf{x})$ where $\mathbf{C}_d$ is the column vector associated with the digit d. Geometrically speaking this determines the distance

that a particular input vector lives from the hyperplane separating d and `not` d. It stands to reason that we should classify $\mathbf{x}$ based on which hyperplane it is furthest from. Once we have classified each member of the training set using this method we can create a confusion matrix comparing the training outputs with what the perceptron came up with, and in turn see how the perceptron performs on testing data. The single perceptron classified both the MNIST training and testing data with about the same success as the distance-based classifier. Both algorithms are built upon the idea of separating possible outcomes in the phase space where the input data fits. The key difference is the distance-based classifier which relies upon finding a so-called center C_d for each digit that serves as a sort of ideal representation of that digit in the form of the inputs while the perceptron creates separating hyperplanes in the same phase space that are built by iteratively rewarding classified inputs and correcting the weights once a misclassification is encountered.

Increasing I appears to have a harmful effect on the accuracy of the single-layer perceptron which would imply that the classes C_d for digits $d = 0, \ldots, 9$ are not separable. If we wanted to improve the accuracy of the perceptron we could employ a pocket algorithm of some kind, which would have the additional benefit of being less computationally expensive (as weights in the pocket are not checked every time).

Task 5: Implement the gradient descent algorithm

Our objective for task 5 was to implement the gradient descent algorithm into a neural network that finds the solution to the exclusive or (XOR) problem. The neural network should consist of two input nodes, two hidden nodes and one output node. To simulate the XOR gate, inputs where only one of the two input nodes was activated should (1,0 or 0,1) return an activated output node. Otherwise, the output should be zero.

In order to implement this we created several functions to break the objective up into smaller tasks. First, we created a function that uses the input values and the given weights to calculate the output:

$$\text{xor}_{net}(x_1, x_2, weights) \tag{6}$$

To be able to train the network in such a way that it gives the desired output for all combinations of input values, we trained the weights with use of the Gradient Descent algorithm which is a iterative optimization algorithm to find the minimum of a function. In this case, we used the algorithm to minimize the Mean Squared Error (MSE), an error measure defined as the squared difference between the real output value and the desired output value for a given set of input values, divided by the number of output values:

$$MSE(weights) = \frac{1}{4}\Big(\Big(0 - \text{xor}_{net}(0,0,weights)\Big)^2 + \Big(1 - \text{xor}_{net}(0,1,weights)\Big)^2 \tag{7}$$

$$+ \Big(1 - \text{xor}_{net}(1,0,weights)\Big)^2 + \Big(0 - \text{xor}_{net}(1,1,weights)\Big)^2 \Big). \tag{8}$$

In order to minimize the MSE, the gradient descent algorithm iteratively updates the weights in the direction in the weight space such that that the MSE is lowered as much as possible. This direction follows from the calculation of the MSE gradient. Weights are adjusted with just a small fraction of the gradient, bound by the value of the learning rate η, also referred to as step size. This process is repeated until convergence into a minimum for the MSE, if this minimum is equal to the global minimum, the network performs optimally with the right weights. The probability of reaching this optimal outcome depends on the initialization of the weights and the value of the learning rate. If the learning rate is set too low it will take too long to converge as opposed to when the learning rate is set too high, which will result in skipping over the optimal distribution of weights.

In addition to the initialization of the parameters, the choice of activation function is crucial in whether the neural network will reach success or not. The activation function is, as the name suggests, a function that determines when a node will be activated when given the incoming values.

To gain insight in the performance of the neural network with gradient descent we have performed several experiments using three different learning rates, two different initialization of weights and three different activation functions. In all the experiments we let the neural network train for 4000 iterations. The initial weights were either drawn from a normal distribution $\mathbf{N}(0,1)$ or a uniform distribution $\mathbf{U}(-1,1)$. As activation functions, we have experimented with the sigmoid function, the rectified linear unit function and the hyperbolic tangent function:

Sigmoid activation function:

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{9}$$

Rectified linear unit activation function:

$$ReLu(x) = \max(0, x) \tag{10}$$

Hyperbolic tangent activation function:

$$tanh(x) = \frac{\exp(-x) - \exp(x)}{\exp(x) + \exp(-x)} \tag{11}$$

With our experiments we could gather the following results when using the sigmoid, rectified linear unit and hyperbolic tangent activation functions as included respectively in figures 4, 5 and 6. In all figures, the upper three plots are results from experiments with a normal distribution of initialized weights, whereas the lower three plots are from experiments with a uniform distribution of initialized weights.

When interpreting the results for the sigmoid activation function in figure 4 it can be stated that the learning curve is steeper when using a larger learning rate. In other words, the MSE is minimized more quickly when the learning rate is increased. Also, running the

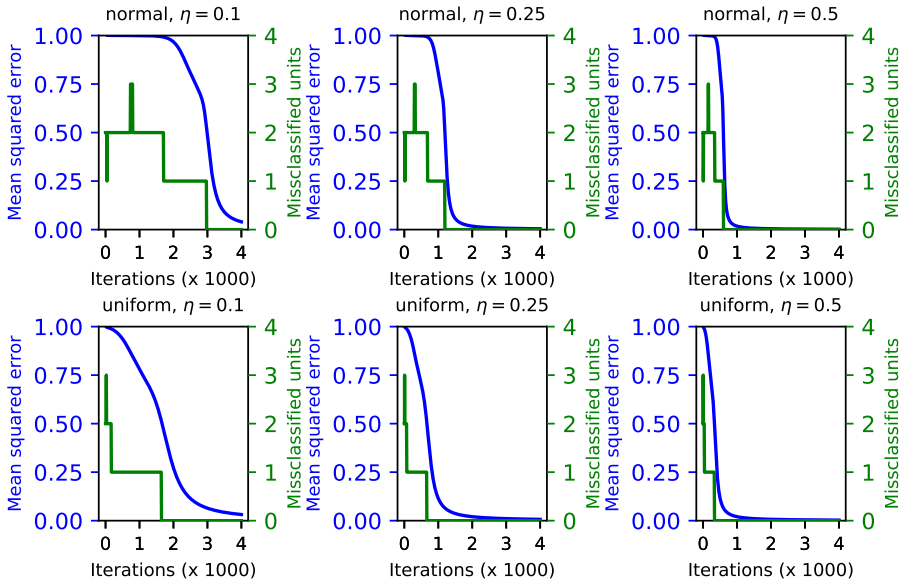

Figure 4: **Task 5**.Results regarding the sigmoid activation function. The top row consist of the results for normal weights distribution, bottom row showing results for uniform weights distribution.

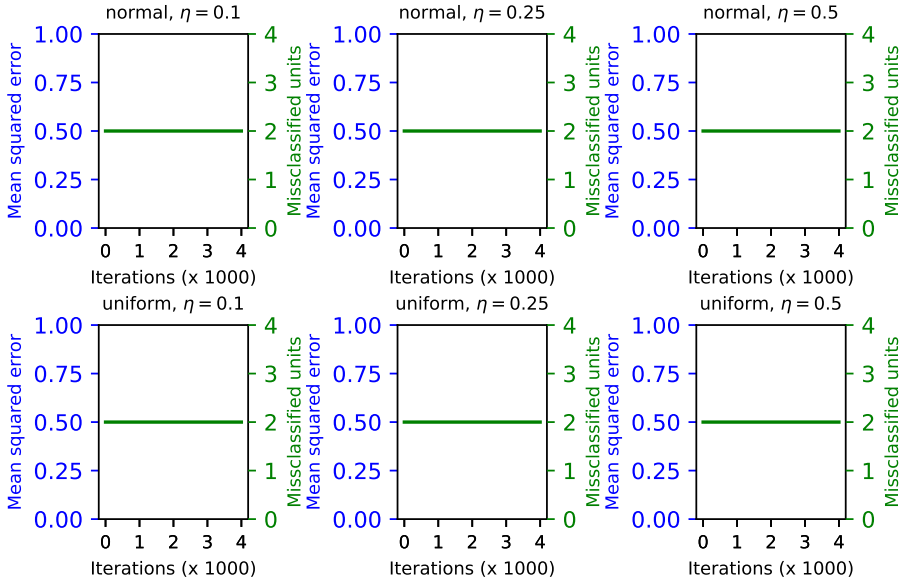

Figure 5: **Task 5**.Results regarding the ReLu activation function. The top row consist of the results for normal weights distribution, bottom row showing results for uniform weights distribution.

neural network with weights drawn from uniform distribution seems to converge to the minimum MSE faster than with weights drawn from a normal distribution.

The first impression on the results from the neural network with rectified linear unit activation function in figure 5 is that something went wrong, as the results seem to be the same for all different settings of the parameters and the MSE does not seem to decrease at all. This is, however, not the case since the gradient of the relu activation function is zero when x ≤ 0, resulting in never updating the weights. It would be possible to get rid of this problem by setting a very small learning rate, although it could still happen. A better solution would be to use a slightly different function such as $max(0.2x, x)$ to prevent the gradient from being zero.

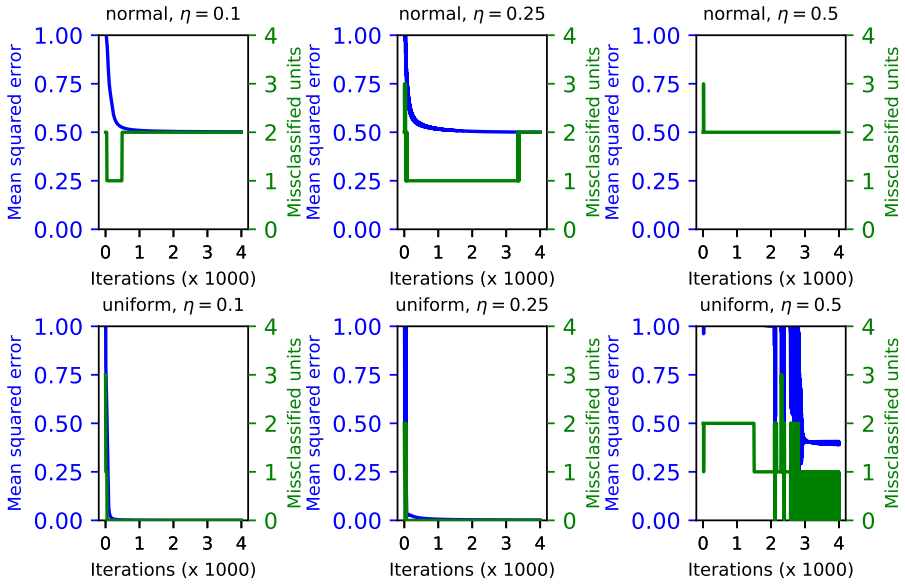

Figure 6: **Task 5**.Results regarding the tanh activation function. The top row consist of the results for normal weights distribution, bottom row showing results for uniform weights distribution.

As for the experiments with the tangent activation function as shown in figure 6, there are some interesting results. When using the normal distribution to initialize weights, the gradient descent seems to get stuck in a local optimum as it converges to a MSE value of 0.5. However, when using the uniform distribution, it converges very fast to the minimum MSE value, much faster than when using the sigmoid function. When using a larger learning rate, these results show the problem that can exist when using a learning rate that is too high, as the algorithm skips over the optimal solution, jumping between local minima without converging.

Conclusions

Our programs were successful in classifying either handwritten digits from MNIST data set or the XOR problem in proportion to their complexity. Task 1, the algorithm built to classify digits based on the degree to which they differed from the "ideal" version of each digit based on the training data represented in $\mathbb{R}^{256}$, did not leave a lot of room for optimization. We explored placing the data in a non-Euclidean space but this provided only marginal improvements both in getting the training data correct and analyzing a previously unseen data set. This can be attributed to the fact that the previously defined cloud radius is of the same order as the separation distance between cloud centers; it is entirely possible that many of the misclassified digits were within the radius of two different clouds.

The Bayes classifier as implemented in task 3 was very successful in classifying digits after training on the training set when limited to a classification task between two digits. As we chose the digits that were found to be least similar in terms of asymmetry, the classifier is expected to perform worse when classifying between other digits than the 1 and 5 that were used. Another limitation to this classifier is that it is expected to perform poorly when classifying multiple digits, as some of the digits have similar asymmetry values.

Task 4 was in large limited by the fact that it was only a single-layer neural network. The resulting hyperplanes distinguishing between "digit X" and "`NOT` digit X" had no curvature, which is well known to be not very accurate (and fails entirely in some cases such as the XOR problem). Increasing the number of network layers would in principle improve the accuracy of the classifier. Additionally, an optimum number of weight adjustment iteration needs to be established to prevent overfitting the network on the training data. With far more iterations to adjust the weights the accuracy on the training set would converge to 100% but its success in classifying the testing set would diverge. We settled on 100 iterations at a learning rate of $\eta = 0.01$ and found it had comparable success to the distance-based classifier. In principle one could make the classifier a function of number of iterations and learning rate in order to find the optimal value for each parameter; this would only take a few lines worth of adjustments to our program.

In task 5 we were able to train a neural network with one hidden layer using the gradient descent algorithm to provide the right solutions to the XOR problem. Using a sigmoid activation function provides stable results, as it converged via a gradation to the optimal solutions in all of our experiments. We also found that the hyperbolic tangent activation function is very useful to quickly solve the XOR problem when using a relatively small learning rate if the initial weights are drawn from a uniform distribution.