**Home Work 3:**

**Intelligent Systems- Backpropagation and Auto-Encoder**

# 

# **Problem One**

## **System Description**

The aim of this program is to use backpropagation algorithm in order to train multi-layer neural network in correctly classifying grayscale images of numbers ranging from Zero to Nine.

Initially, the weights was initialized at random values between -1 & 1. The following sigmoid function

  was chosen. Derivative function is then equal to For one point, the data is propagated through the layers of the network, until the output is obtained at the output neurons. Finally, the delta value is tabulated at the outermost layer using the equation

).

For each output neuron the vector of deltas is found. According to the Gradient Descent Learning rule is To all output neurons, the weight changes are then applied; where is the input from neuron j of the previous layer to output neuron i with weight ,

is the corresponding neuron delta,

The momentum is depicted by is & The learning rate is depicted by

The deltas is calculated for one-layer at a time. The forward weights are used to tabulate the current layer deltas according to the formula

, The same learning rule as applied for the output neurons is then applied to the present/current layer neurons using newly calculated deltas, weights & inputs. For every point in the training subset this procedure is repeated.

For every layer until the 1st neuron layer, the above procedure is repeated.

### **System Implementation Configuration & Others**

The above algorithm can be configured to use a variable number of parameters. The parameters are in the form of 1) number of layers 2) number of neurons present per layer 3) relevant input files through the function parameters. In the implementation, I ended up using 1 hidden layer. Thus, a total of 2 layers was used which includes the output layer. After accounting for simulation time & performance, I used a hundred (100) neurons per hidden layer & also Ten (10) output neurons for each digit (Zero (0)- Nine (9)). After experimenting, it was observed that **a learning rate of 0.06** gave satisfactory speed of convergence with relatively small oscillation range. Also, a **momentum value of 0.6** was chosen to account for oscillations & stabilize the gradient descent. In order, to manage all the variables, & to make the system modular & expandable, a Perceptron object class was designed such that the weights & inputs to the perceptron, the sum, momentum, output, delta & learning rate were self-contained. It also had methods to update weight changes, tabulate outputs & inputs,

A subset of training set of size Hundred (100) is randomly selected & used for training for every epoch. For live testing, a subset of size hundred (100) is selected randomly from the test set & used. Thus, the progress over the training set & test sets are recorded. A test confusion matrix is tabulated over the whole test & training set as well as the final error rate by the end of training. The results are as depicted below. The stopping condition was to train for 450 epochs. The number of training epoch would be increased further if it was noticed that the error rate was not satisfactory or (below 0.10). **When the error rate was low enough I stopped the training.**

**Parameter Settings:**

I ended up running several individual iterations before deciding which parameter to choose for my model. Initially, I tried several parameters for my model. In the beginning, I ended up using **100 epochs** with **1000** random training points. As suggested in the problem, I ended up using stochastic gradient descent for each epoch. Firstly, to decide on how to select the number of hidden neurons in each layer, I had to run my algorithm few times. After, 5 different counts ranging from 100 to 300 over the range of 50; I plotted the final Testing & Training hit-rates. Now, please refer to Figure 2.a, you can clearly see that there is a sharp increase in the hit-rate at 150 neurons threshold; later Testing & Training hit rates settles down over a small range until 300 neurons. From this observation I decided to go with **150 hidden neurons**. Later, I iteratively improved my design & was able to implement with the design with **100 hidden Neuron**. I optimized my architecture.

To decide on the **learning rates (ηh, ηo)** for both hidden & output weights were tried over a range of values {0.01, 0.02, 0.02, 0.06, .. 0.1}. What I noticed is there was consistent increase in the learning rate which is depicted in Figure 2.b, & hence I decided the learning rate to be set at **0.1 which was an initial setup;** later I changed it into 0.06**.** For both the layers in the algorithm equal learning rates was considered. In my coding I even experimented, a learning rate of 0.06. **I found that it provided satisfactory speed of convergence with relatively small oscillation range.**

The **Momentum parameter (α)** was also tested for various range of 5 values {0.05, 0.1, 0.15, 0.2 & 0.25}. It’s observed that hit-rate of both Testing & Training sets peak at a value 0.15 which is observed from the figure 2.c.Initially I took 0.15 as Momentum parameter (α) later I used 0.6 Thus the value was selected as the final momentum step parameter. Finally, **Ten output neurons** was used to depict one of each digit were used in the output layer.

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Architecture 1** | **Architecture 2** |
| **Learning Rate** | 0.1 | 0.06 |
| **Momentum parameter** | 0.15 | 0.6 |
| **Hidden neurons** | 150 | 100 |
| **Hidden Layer** | 1 | 1 |
| **Output neurons** | 10 | 10 |
| **Computation time** | 16 hours | 2 hours |
| **Coding style** | For loops | Matrix multiplication |
| **Epoch at which error rate stabilized** | 400 | 20 |

**Note**: Used Architecture 2 in the end for problem 1.

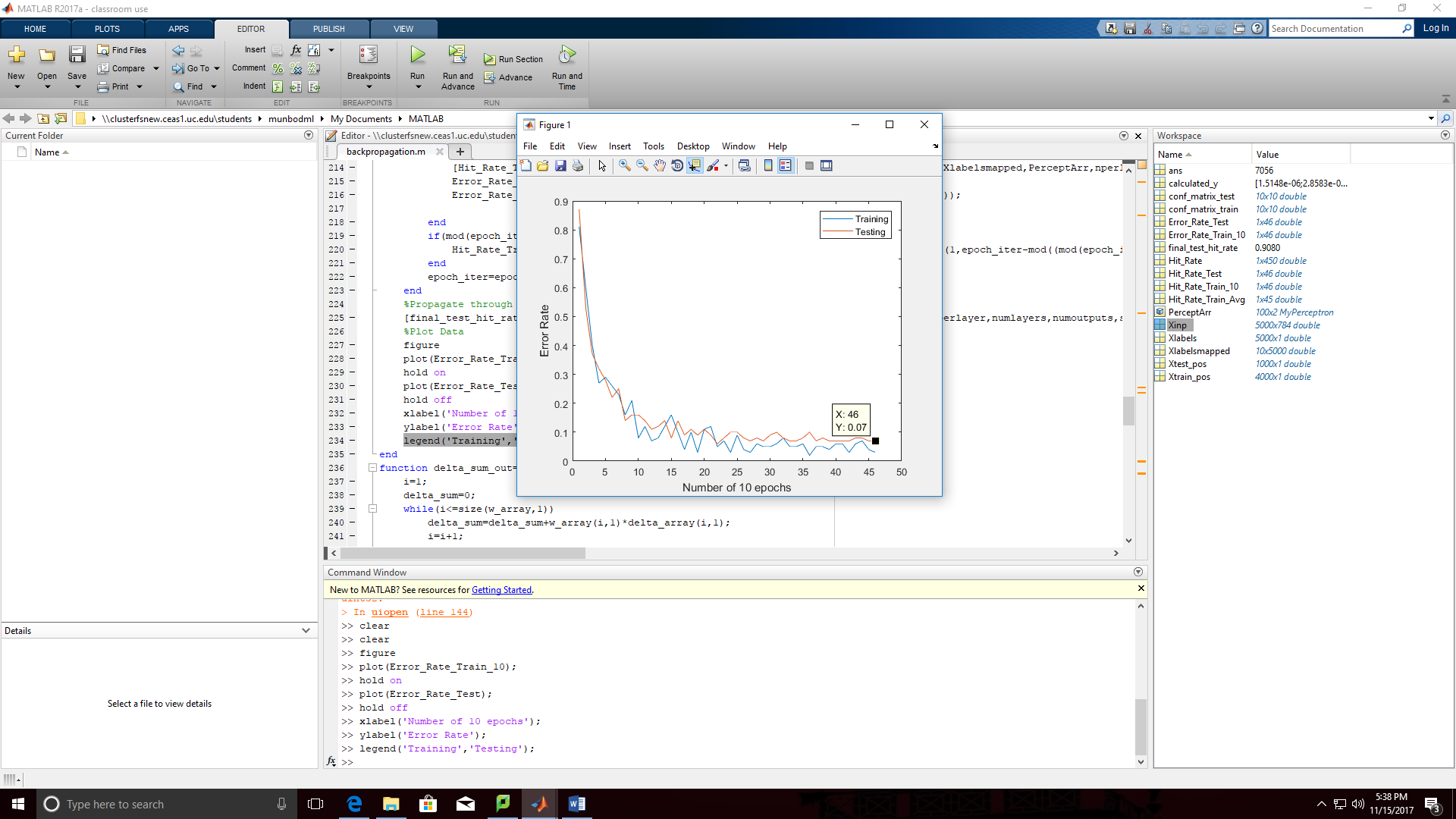
## *From the above table its clear how different parameters effects the overall performance of the architecture. When you change the number of neurons in hidden layers it effects the learning rate & momentum parameter.*

* The reason why we employ **momentum** is because the algorithm remembers the last step taken & adds some proportion of it to the current step. Hence, even if the algorithm is stuck in a flat region or some local minima; it can eventually get out & continue to move towards the true minimum.
* Thus in short: while performing gradient descent, how much the current situation affects the next step is tabulated by the **learning rate**. While how much the previous steps affect the next step to be decided/taken is done by the **momentum**.
* **Momentum helps accelerate gradients in the right direction**.

SGD has trouble navigating ravines. What is ravine? It’s a region or area, where the surface curves much more steeply in 1 dimension than the other. Ravines are common phenomena near local minima’s. SGD also tends to oscillate across the narrow ravine.

Hence the negative gradient will point down along one of the steep sides. Rather than point along a ravine towards an optimum.

## **Results**



*Figure 1: Plot of Error Rate Vs Number of Epochs (For every 10th epoch is calcualted)*

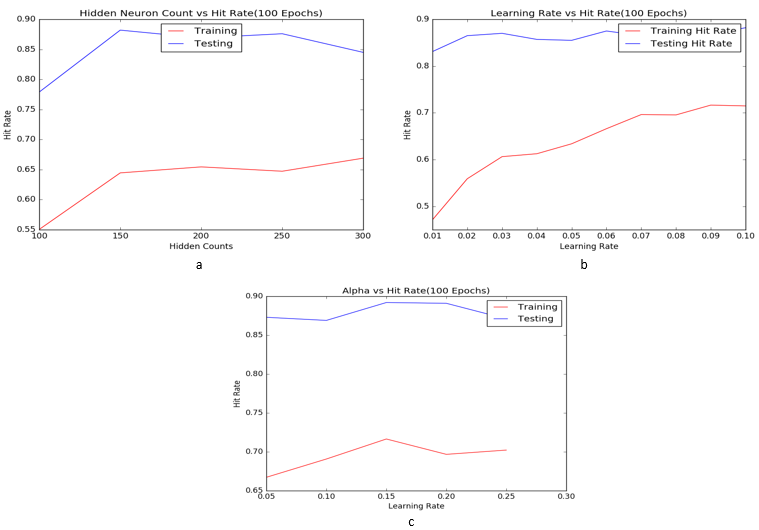


Figure *2*: (a) Plot of hit-rates against different hidden neuron counts over 100 epochs for both Testing & Training sets is tabulated. (b) Plot of Hit-rates against multiple learning rates (η) is tabulated for both Testing & Training sets. (c) Plot of Hit-rates are provided as a function of momentum (α) step is tabulated for both Testing & Training sets.

Table 1: Final Configuration Matrix between the actual & predicted labels for Training Data (Over All 4000 Training Points)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| **A**  **C**  **T**  **U**  **A**  **L** |  | **PREDICTED** | | | | | | | | | |
| **Digits** | **Zero** | **One** | **Two** | **Three** | **Four** | **Five** | **Six** | **Seven** | **Eight** | **Nine** |
| **Zero** | 357 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **One** | 0 | 453 | 2 | 3 | 0 | 0 | 2 | 1 | 0 | 0 |
| **Two** | 2 | 1 | 391 | 2 | 3 | 0 | 2 | 2 | 4 | 1 |
| **Three** | 0 | 0 | 2 | 397 | 2 | 0 | 0 | 1 | 1 | 2 |
| **Four** | 0 | 0 | 2 | 1 | 391 | 1 | 0 | 1 | 1 | 2 |
| **Five** | 0 | 1 | 0 | 1 | 3 | 343 | 2 | 1 | 2 | 2 |
| **Six** | 1 | 0 | 0 | 0 | 1 | 1 | 371 | 0 | 0 | 0 |
| **Seven** | 0 | 2 | 1 | 0 | 1 | 0 | 0 | 418 | 0 | 0 |
| **Eight** | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 397 | 0 |
| **Nine** | 1 | 3 | 0 | 1 | 2 | 4 | 2 | 3 | 4 | 395 |

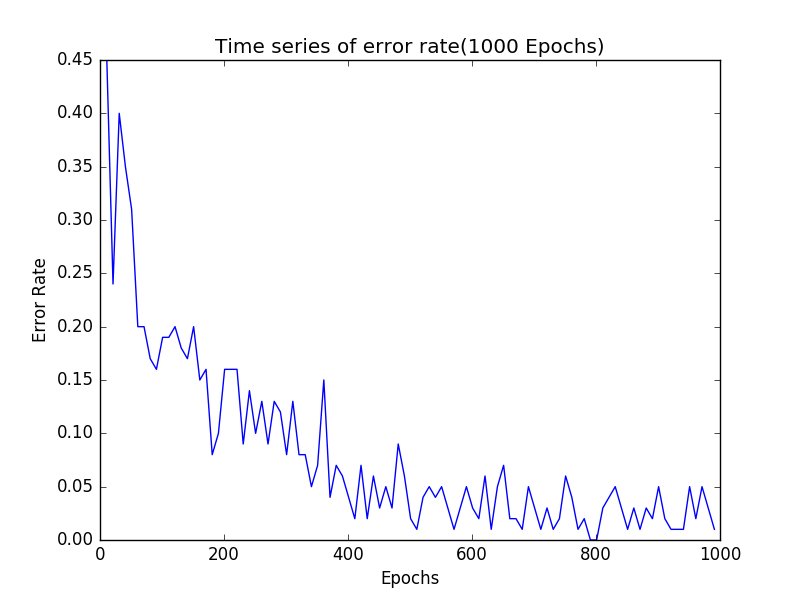
Table 2: Final Configuration Matrix between the actual & predicted labels for Testing Data (Over All 1000 Testing Points)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
| **A**  **C**  **T**  **U**  **A**  **L** |  | **PREDICTED** | | | | | | | | | |
| **Digits** | **Zero** | **One** | **Two** | **Three** | **Four** | **Five** | **Six** | **Seven** | **Eight** | **Nine** |
| **Zero** | 96 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **One** | 1 | 108 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Two** | 1 | 2 | 105 | 1 | 3 | 1 | 2 | 5 | 2 | 1 |
| **Three** | 0 | 0 | 2 | 91 | 0 | 0 | 0 | 3 | 0 | 0 |
| **Four** | 0 | 0 | 0 | 0 | 102 | 0 | 0 | 0 | 0 | 3 |
| **Five** | 1 | 0 | 0 | 2 | 1 | 91 | 0 | 1 | 3 | 0 |
| **Six** | 1 | 1 | 0 | 0 | 1 | 1 | 80 | 0 | 1 | 0 |
| **Seven** | 0 | 2 | 1 | 0 | 4 | 0 | 0 | 80 | 0 | 1 |
| **Eight** | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 76 | 0 |
| **Nine** | 0 | 2 | 0 | 2 | 3 | 0 | 0 | 1 | 2 | 97 |

## **Analysis of Results**

I used the parameters from architecture 1 which is mentioned above. The final network was trained using **stochastic gradient descent** procedure which has **online learning** mode enabled.

In the end 100,000 points was used in total it compromised hundred random training points (100) which was chosen from training set present in each epoch & thousand epochs was used overall (1000) (thus the final count was 100 \* 1000 = 100,000 points was used in total). The activation functions of every neuron (either zero/one target value) is determined by the **Sigmoid** function. I made sure to include the weights of bias terms was included & trained along with the other weights. The error rates observed were plotted for every 10 epochs as depicted in Figure 3. This error rate was for Architecture 1. I used for loop in my coding style which slowed my runs. The overall time was around 16 hours.



**Figure 3**

In the image above we can clearly see a sharp decline in the error rate during the first 30 epoch & gradually the error rate is headed towards zero. We do notice that after five hundred epochs the error “flat lines” & starts to oscillate in a range suggesting most of the learning has been completed by then. The oscillating pattern of the hit-rate could be explained by the fact that each epoch works on a random subset of the training set. The hit-rates were tabulated using the threshold approach were the operating parameters H = 0.75 & L = 0.25 such that is considered a match for = 1 if ≥ H & is considered a match for = 0 if ≤L. Once the network was trained the testing set (1000 points) were used to evaluate the performance. A hit-rate of **92.6 %** was observed with max-threshold approach used to tabulate the hits.

**In Figure 1, the architecture 2 was implemented**. Over a 100 points we can see that there is a steady decrease in the error rate as the system is learning at each epoch. In the end; a test error rate 0.0910 is observed which is close to the test error rate of 0.069 during training stage of the last epoch. The above observation was observed during final testing over all the thousand test set. It’s observed from the Figure 1 that training set has error rate lower than the testing set this behavior is consistent; which is a right observation.

From, the training confusion matrix, Table 1, which is for architecture 2

A **Confusion matrices** was generated for both Testing & Training sets to further evaluate the performance. Confusion matrix for the training set of Four thousand (4000) points is depicted in Table 1. **Most digits/points were correctly classified with a hit-rate of 96.7% this is shown by the large diagonal values in the confusion matrix.** Digit Nine has the **most** mis-classifications (17) followed by digit two (16)this shows that both of them **shared** features with all other digits; this was possible thanks to the mis-classifications obtained from the confusion matrix. After digits Nine & Two it’s digit 8 which had highest proportion of incorrect classification when measured in descending order. The similarity in the orientations or shapes of the digits Nine <-> five, Two <-> eight explains why the most individual miss-classifications were observed between the above mentioned digits. While the testing confusion matrix, is available in Table 2. This table depicts that the numbers five, eight & seven have the highest proportion of incorrect classification when measured in descending order. Number eight is misclassified mostly as number five in both the testing & training matrices. The number five is also most frequently misclassified as number 8. Number Nine is mostly mistaken with number four in both Testing & Training cases & vice versa.

The confusions are raised may be due to the similarly shaped clusters.

“**Max-threshold**” method was used to generate the confusion matrix for training data. Table 2 depicts the confusion matrix for a testing set 1000 (thousand) points.

In the testing set, most mis-classifications were observed in digit two (sixteen) – consistent with training set – followed by digit eight (thirteen) & digit nine(ten). Individual mis-classifications were highest between the digits two <-> seven, seven <-> four. These digits having similar edges can explain the reason for miss-classifications.

# **Problem Two**

## **System Description & Implementation**

I have used similar configuration with respect to parameters as question 1. The only change was except how the outputs are being calculated; outputs are the representation of inputs themselves. Hence, for this assignment there are seven eighty-four output neurons (784). For this problem I have used Architecture 2 which has 1 hidden layer with hundred neurons. 0.06 & 0.6 which corresponds to the learning rate & momentum same parameter setup as in question one was used. The loss function was used to tabulate every epoch & stored at epoch 0, & every 10 epochs. After training, the loss function was evaluated over the whole training set & the whole testing set. Then, both loss functions (training over 4000 points, & testing over 1000 points) were scaled down to a 100 points for comparison purposes. The total training error & testing error were also binned according to the digit label. At the end of training, the weight vector for each of the 100 hidden neurons was plotted as a 28 by 28 grayscale image to get the features for which each neuron will be triggered the most. The results are depicted below.

**Parameter Settings:** Parameters required to execute the learning step we chosen by performing several individual runs. Initially I used Architecture 1 with **1000 epochs** each with **100** random training points for each epoch (stochastic gradient descent) were used for this purpose. **100** hidden neurons have been used, consistent with the earlier problem. **784** output neurons were used the output layer to reconstruct every bit corresponding to the 28\*28 input patterns. In order to select learning rates (ηh, ηo) for both hidden & output weights were tried over a range of {0.2, 0.01, 0.03, 0.05} value. Higher range values were used in the selection procedure (compared to the earlier problem) by considering the complexity of the re-construction problem. The J2 loss function **accumulated** across all the points was tabulated for every parameter value in the selection procedure. Plots of both Testing & Training errors from Figure 3.a show a gradual decrease in the error with the lowest training error achieved at **0.15**. Refer to Figure 6

Similar to the earlier problem, the learning rates of both the layers were chosen to be equal. The momentum parameter (α) was also tested for over a range of 5 values {0.05, 0.1, 0.15, 0.2, & 0.25}. From figure 3.b it could be seen that the squared-error in the training set was its lowest when **α=0.2**. refer to figure 6. **Later I ended up using Architecture 2 which was 0.06 & 0.6 which corresponds to the learning rate & momentum same parameter setup as in question one was used**

## **Results**

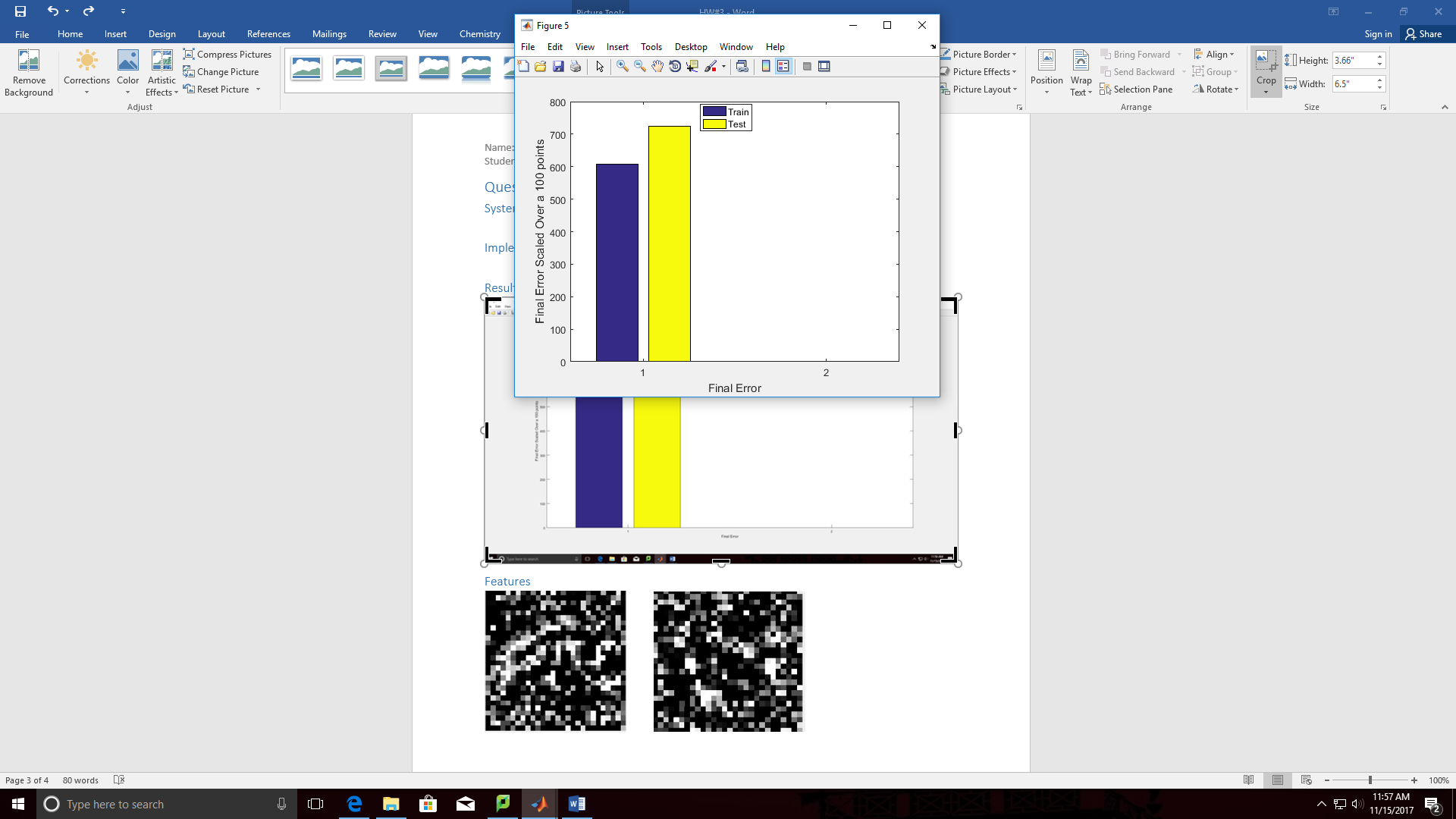


Figure 2: Final Error Squared for Testing & Training Data Scaled Over 100 Points

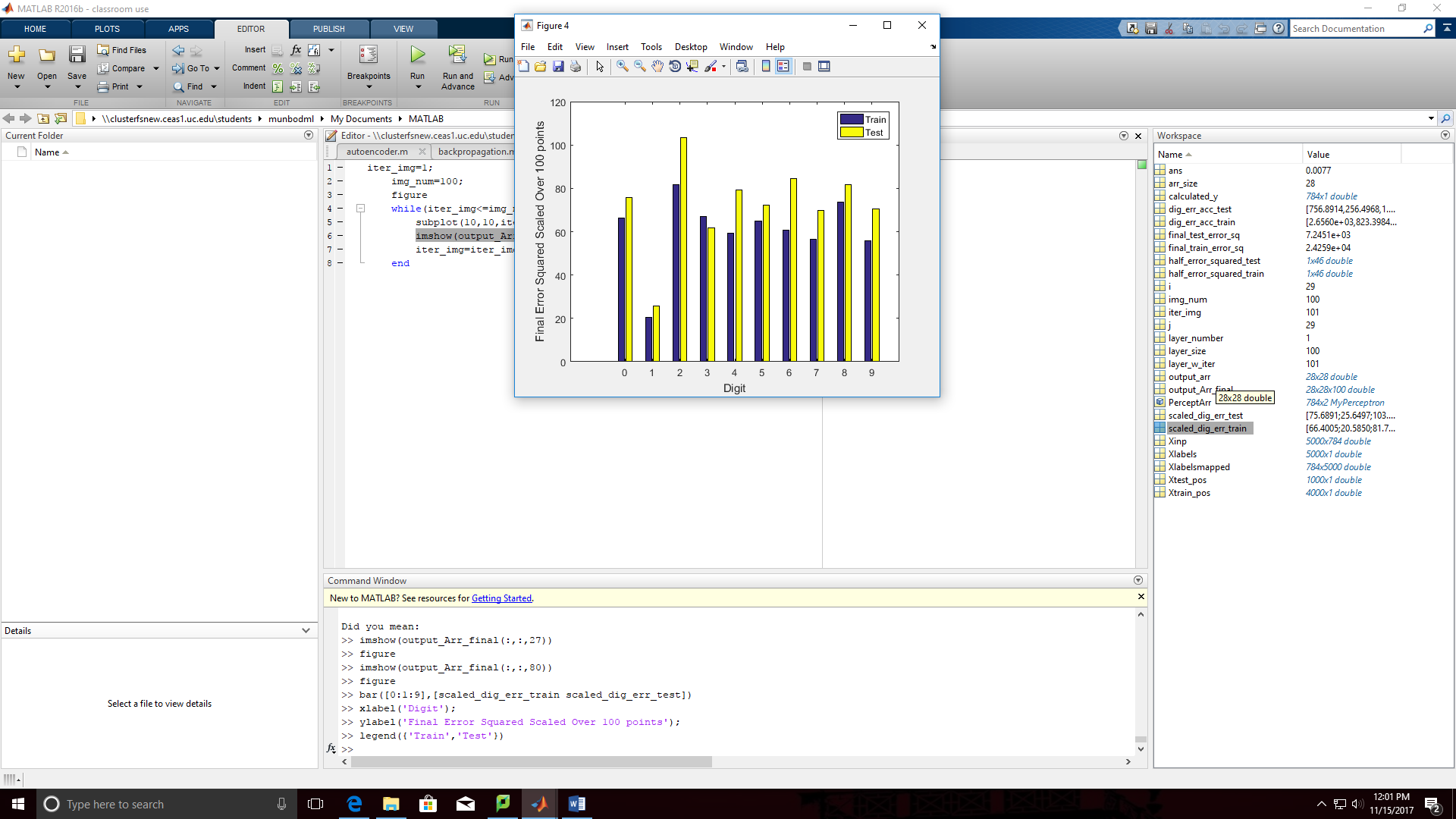


Figure 3: Final Error Squared for Testing & Training Data Scaled Over 100 Points v/s Digit Label

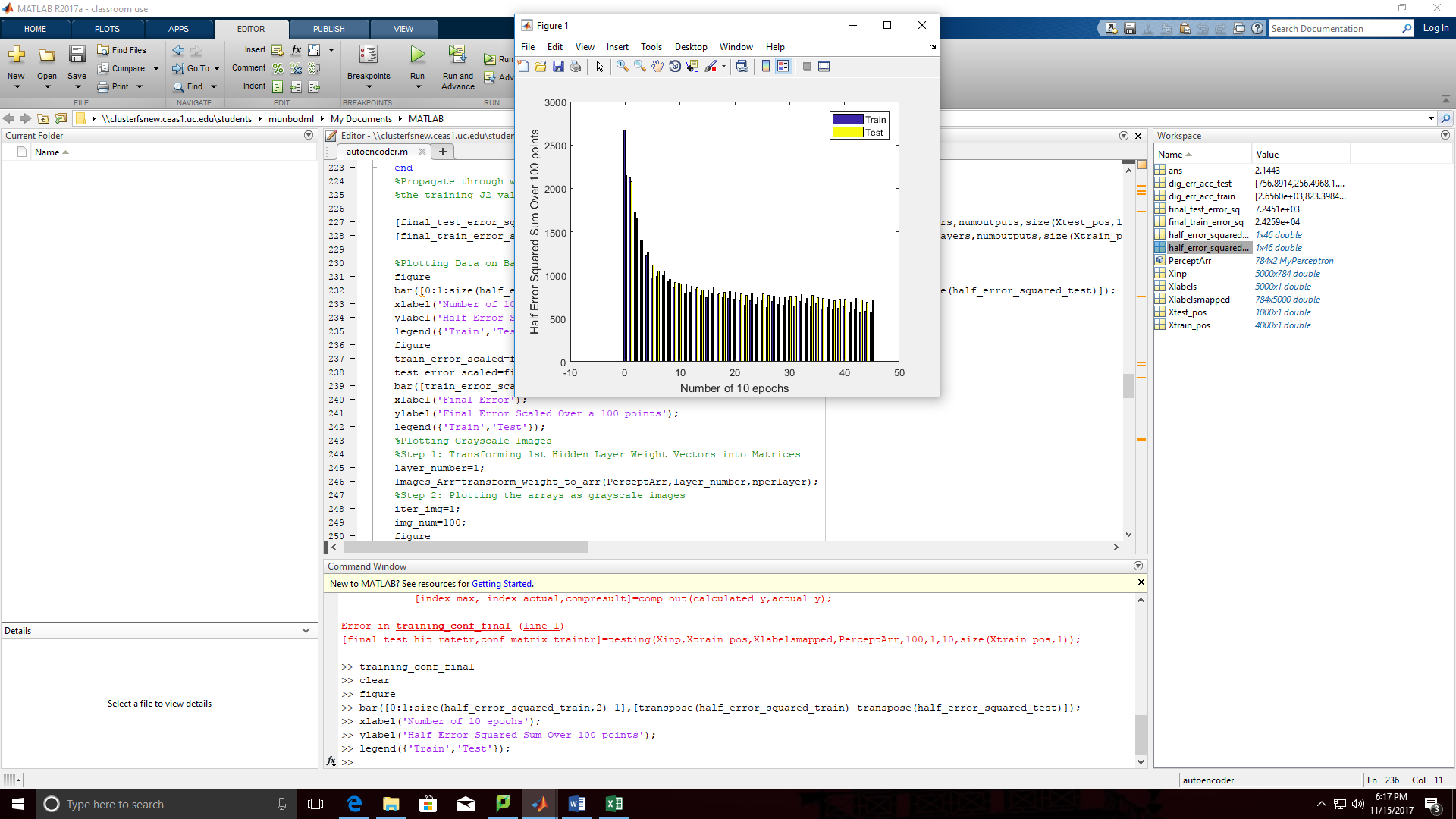


Figure 4: Final Error Squared for Training & Testing Data Scaled Over 100 Points v/s Digit Label

## **Features**

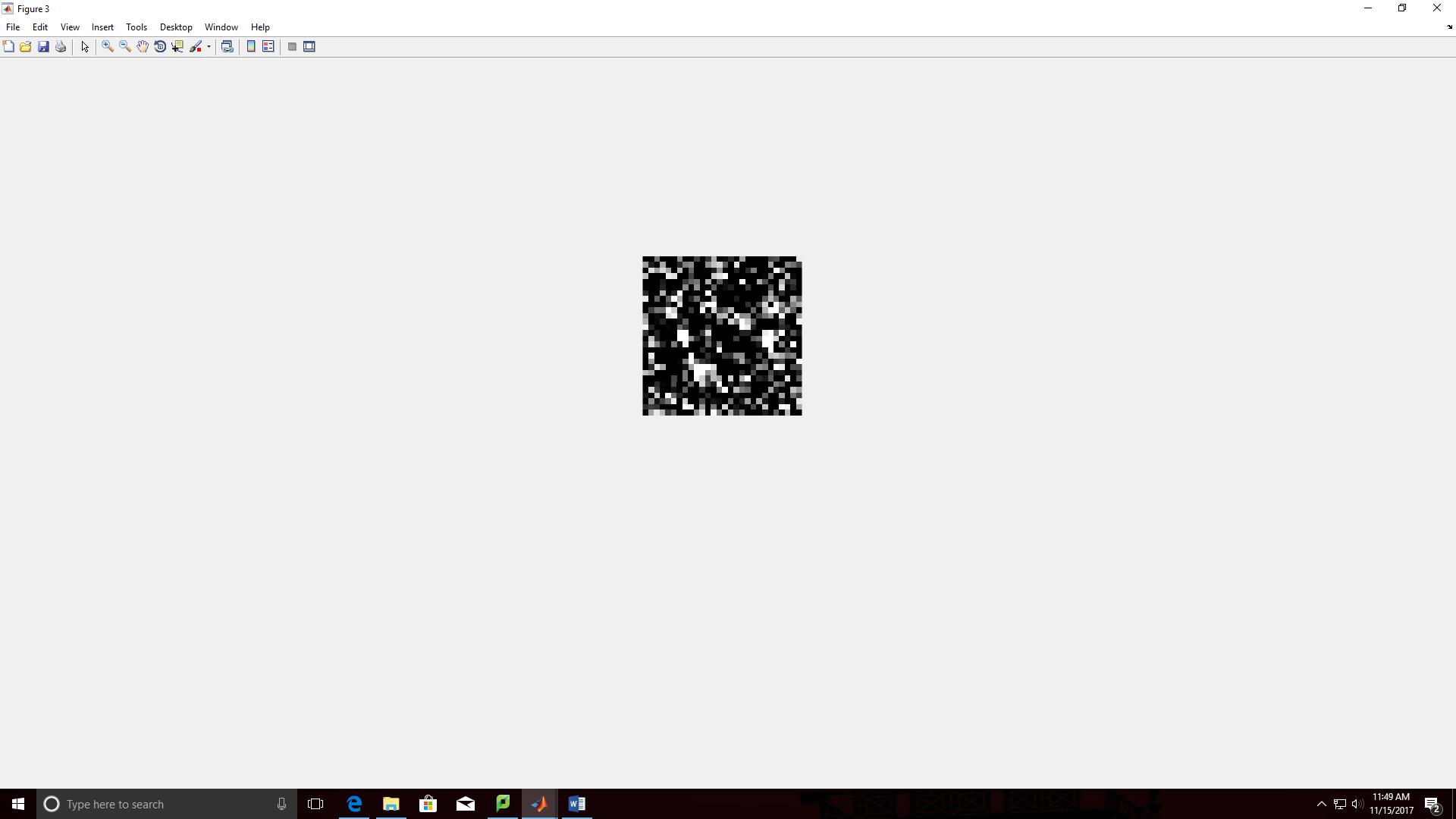
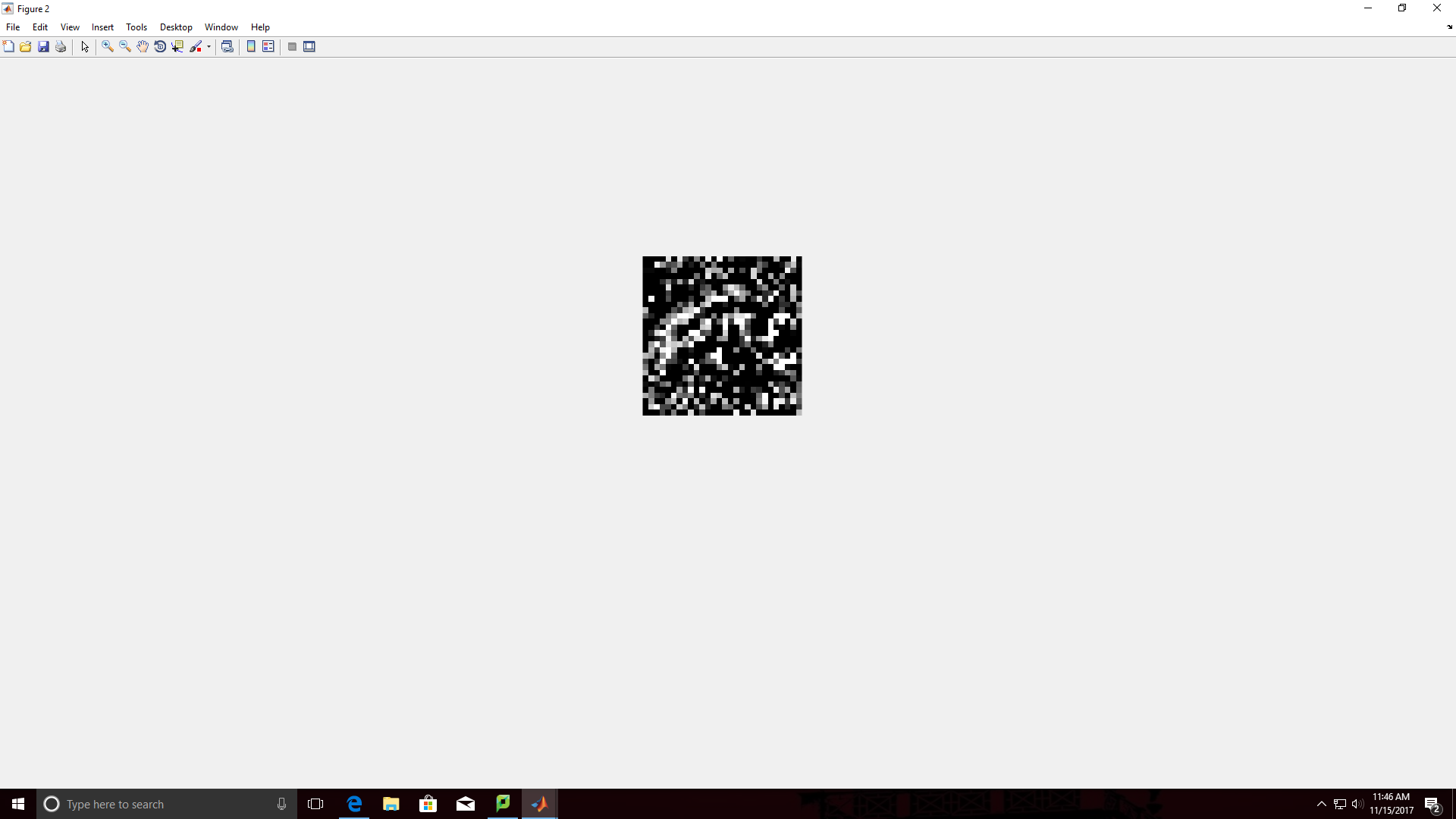


Figure 5: Features for Two of the Hundred Hidden Neurons

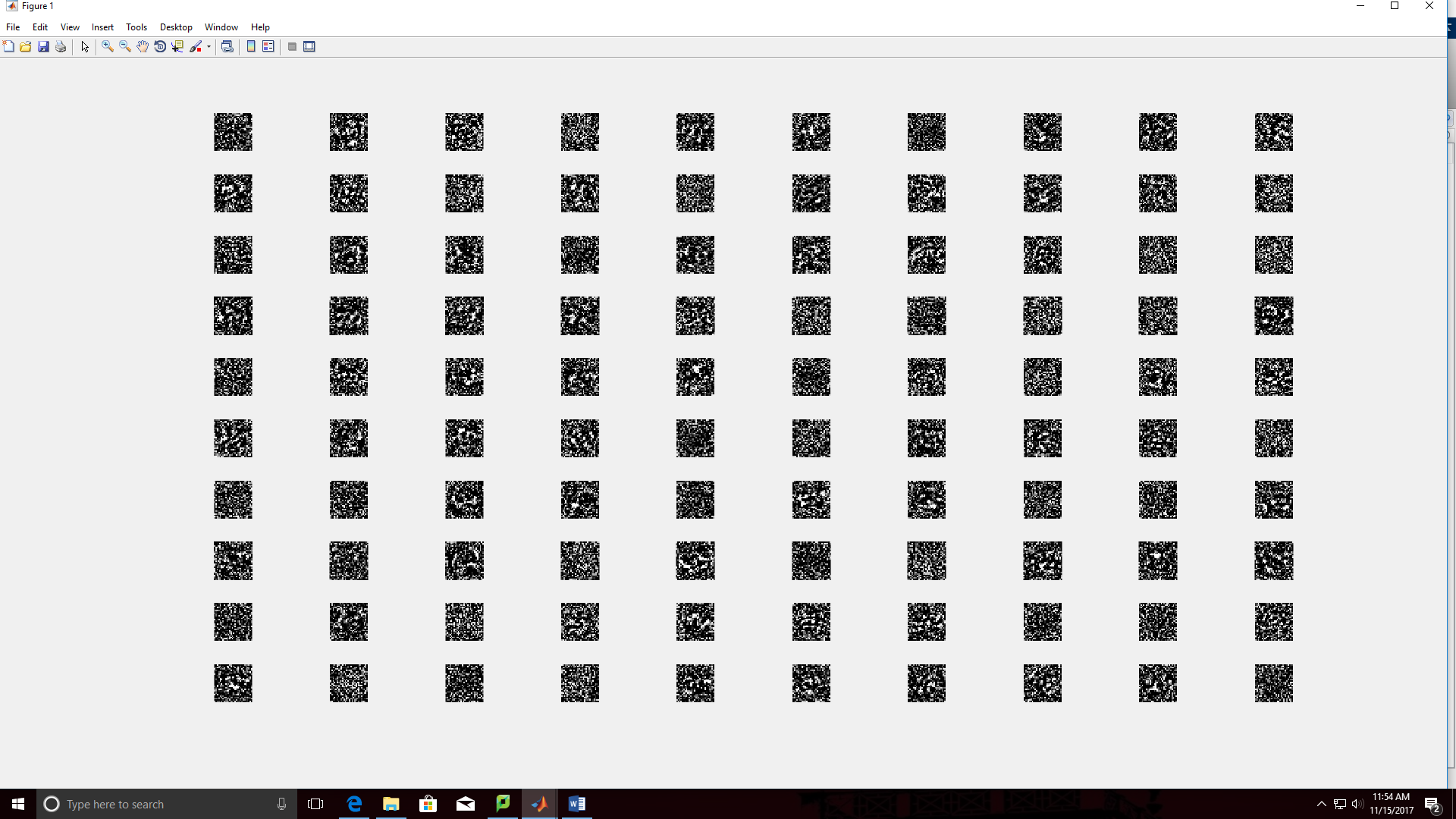


Figure 5: Features for All of the Hundred Hidden Neurons

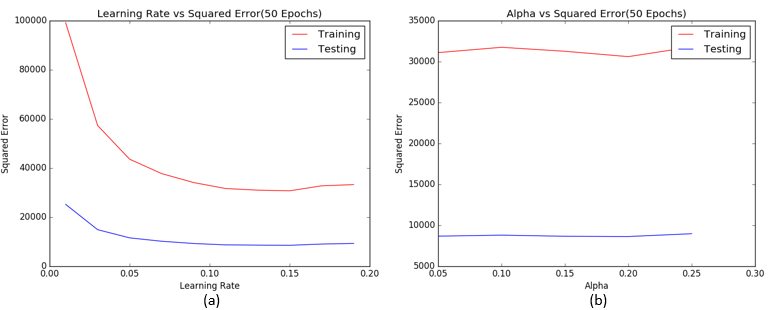


Figure 6: (a) Plot of learning rate (η) vs accumulated squared error over 50 epochs with 100 points per epoch. (b) Plot of squared error as a function momentum step size (α) over 50 epochs

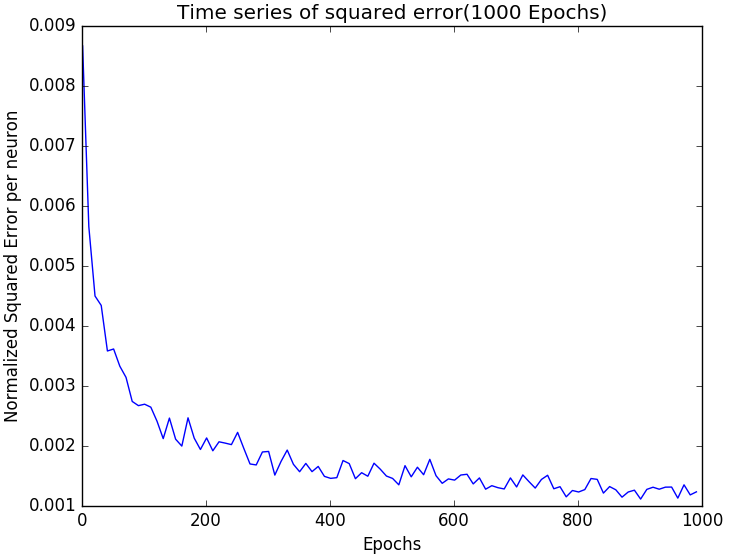


Figure 7: Time series plot of squared (J2) error rates for every 10 epochs over 1000 epochs of training

## **Analysis of Results**

In Figure 2, the total final training loss value (2.4348\*10^4) over Four thousand (4000) points & the total final test loss value (7.2541\*10^3) are shown scaled over a hundred points by multiplying by hundred & dividing by the training or test size. As can be seen, the final testing loss value over hundred points (725.41) is higher than the training loss value over hundred (100) points (606.48). When the training loss value over hundred (100) points is divided by (100\*784), the error squared/dimension is approximately 0.0076 which is acceptable considering that the range is 1.00 (it is approximately 0.7% of the range).

In Figure 3, we can observe the distribution of the final total scaled over loss value over the ten digits ranging from zero to nine (Zero-Nine). The figure depicts for both Testing & Training. The digit two gives us the highest error the error.

**In Figure 4** via the time series data set of Testing & Training loss values are depicted as calculated over a 100 (hundred) points. Generally, there is a decrease in the value of the total error as the number of epochs increases & training occurs.

**In figure 5,** we can observe that the numbers five & Six are visible. That means that for those 2 numbers the neurons associated with them generate a strong response hence its features are captured. For most numbers we will observe blurriness the reason is because most hidden neurons do not specialize in identifying the correct features for 1 digit. Most features for the hidden neurons when plotted are not too distinct/sharp. Figure 5 depicts the features for all the hundred hidden neurons. Figure 5, it is a plot of all the features which has been learnt by the hidden layer basically it indicates the features learnt by each of these neurons ranging from color intensities to different shapes. The relatively stable & low error rate indicates that most of the features have been captured though the features are not very distinctive in the above figure.

Over **50 (fifty)** **epochs** with hundred (100) random training points/epoch was trained using stochastic gradient descent algorithm using the above selected parameters in my final network. I made sure to include the weights for the bias terms & trained along with the other weights. For every neuron with (Zero/One (0/1) target values) sigmoid function was used as the activation function. In Testing & Training errors were accumulated & normalized using the respective sizes to tabulate the errors for each point.

From figure 2, when compared to the test set the normalized error over training set is relatively lower. This could be because that the training was done on the training set. Additionally, the training error was tabulated & normalized by both the number of neurons & number of points. The resultant value which the error per neuron is plotted for every 10 epochs as shown in Figure 7. The error/neuron sharply decreases over the 1st 100 (hundred) epochs & flat-lines after 300 (three hundred) epochs & visibly stabilizes by the (five hundred) 500th epoch; which is similar to the earlier problem. Furthermore, training & test set error for each digit was tabulated to assess the respective features learnt.