NANYANG TECHNOLOGICAL UNIVERSITY

CZ4042 NEURAL NETWORKS PROJECT 2

PREPARED BY:

NAME	MATRIC. NUM
DEKA AULIYA AKBAR	U1323056K
MUNDHRA SHREYAS SUDHIR	U1322112G

BACHELOR OF COMPUTER SCIENCE

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

TABLE OF CONTENTS

INTRODUCTION

For this project, we have been given a dataset of images containing digits from 0 to 9, which has already been divided into training and test set and we are required to find the most optimum model to train on our dataset. We are required to use two techniques for training, namely, Deep Convolutional Neural Networks and Auto-Encoders.

PART I - CONVOLUTIONAL NETWORK

1 Introduction

A convolutional neural network is a type of neural network which is generally used in applications where the size of the dataset is very large. Due to this, it is computationally very intensive to have all hidden layers fully connected. As a result, in a convolutional neural network, each neuron is only connected to a locally contiguous subset of nodes from the previous layer. It consists of two types of layers, convolutional layer and pooling layer. Each convolutional layer has multiple feature maps, where each feature map is a 2D arrangement of neurons used to calculate a single feature, whose value is a 2D matrix. Each convolutional layer is connected to a pooling layer, which is used to reduce the size of the feature maps in the previous layer by dividing it into disjoint regions and pooling each region to a single scalar value. This kind of network is widely used in computer vision for image and video recognition.

2 Methodologies

2.1 Strategies

In order to have a fair comparison, we make sure that we use the same training and test set for all the models. We plot the training accuracy against the number of epochs for each model and also the test accuracy against different values of each parameter. We choose the parameter values which give the highest test accuracy. Note that we do not use the training accuracy to choose the parameter values, since the training accuracy will be high although the test accuracy will be low if the model over-fits the training data. Due to this, such a model will not generalise well to new examples.

2.2 Assumptions

In order to remove the fluctuations of the training accuracy of our models across all iterations, we plot the training accuracy against the number of epochs instead of the number of iterations, so that we are able to better study our plots of convergence. However, since MATLAB only provides the mini-batch accuracy after each iteration and not the accuracy of the entire dataset after each epoch, we plot the mini-batch accuracy of the last iteration for each epoch against the number of epochs. Our assumption is that over a large number of epochs, this plot should approximate the plot obtained by plotting the accuracy of the entire training data against the number of epochs. However, this should not have an impact on the results we obtain, since we finally decide all the parameters of our model based on the Test Accuracy and not the Training Accuracy, which is still calculated for the entire test set.

2.3 Model Architectures

2.3.1 1 Convolutional and 1 Pooling Layer

In this architecture, we experiment with the best values for the learning rate, mini-batch size, momentum and decay term.

2.3.2 2 Convolutional and 2 Pooling Layers

In this architecture, we use the learning parameters obtained from Part 1 and experiment with the best values for the number of filters for both the layers.

2.4 Learning Algorithms

We use stochastic gradient descent with momentum for training the models. In this algorithm, the training set is divided into multiple batches of equal size and the weights are updated for one batch at a time. Backpropagation is used to implement gradient descent. Momentum is used for speeding up the convergence and ensuring that the algorithm does not oscillate for a long time near the optimum. Decay term is used to ensure that the model does not over-fit the training data.

3 Implementation

The file "cnnPreprocess.m" is used to randomly divide our dataset into training set and test set. The same division of data is used to experiment for all the models and architectures.

The file "cnnTrain1" is used to train the architecture for Part 1 for different learning parameters. The file "cnnTrain2.m" is used to train the architecture for Part 2 for different number of filters. We also have separate files for varying the parameters to make it more modular –

1. vary_lr.m: This file is used to experiment with different learning rates.
2. vary_batchsize.m: This file is used to experiment with different batch sizes.
3. vary_momentum_decay.m: This file is used to experiment with different combinations of momentum and decay terms.
4. vary_num_filters.m: This file is used to experiment with the number of filters for both the convolutional layers in Part 2

Once we have found the optimum parameters for both the architectures, we find their accuracy on the test set to evaluate how well the algorithm performs for those parameters.

Note that although we obtain plots of convergence for all the models we train our data on, we only include in this report, the plot for the model that has the highest Test Accuracy and the models whose parameter values are near the parameter values for that model.

4 Results and Analysis

4.1 Model with 1 Convolutional and Pooling layer

4.1.1 Learning Rate

The other learning parameters are set to the following values –

1. *Maximum Epochs = 25*
2. *Batch Size = 128*
3. *Momentum = 0.1*
4. *Decay = 0.0001*

We experiment for the following values of the learning rate:
[0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3]

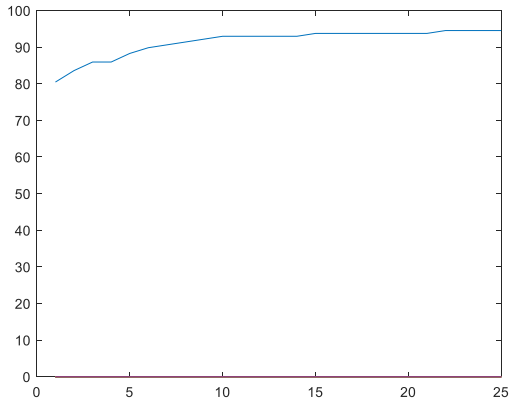


Figure 1. Training Accuracy VS Epochs (Learning Rate = 0.001)

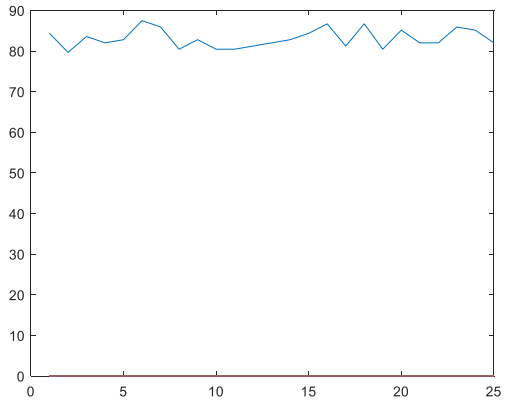


Figure 2. Training Accuracy VS Epochs (Learning Rate = 0.003)

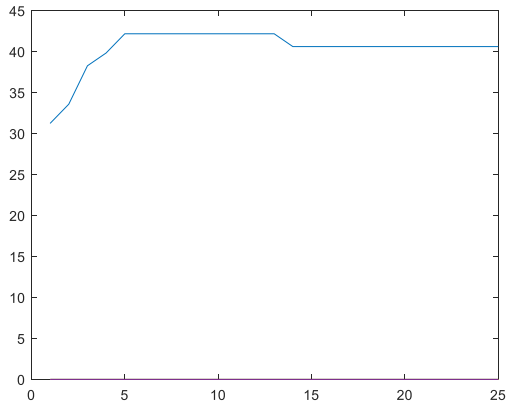


Figure 3. Training Accuracy VS Epochs (Learning Rate = 0.01)

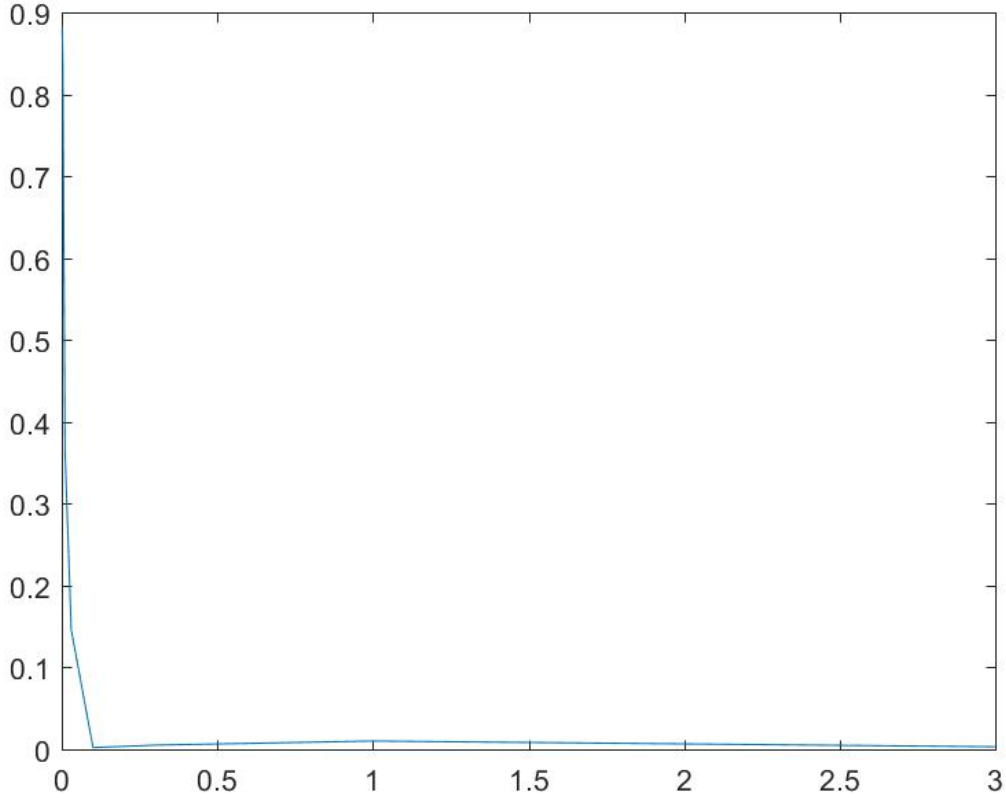


Figure 4. Test Accuracy VS Learning Rate

It can be seen from the above graphs that the Training Accuracy as well as the Test Accuracy decreases after a learning rate of 0.001. This is because if the learning rate is too high, it might oscillate and take a longer time to reach the optimum, or even diverge from the optimum. Hence, we choose a learning rate of 0.001 for our algorithm.

Generally, the learning rate for stochastic gradient descent needs to be lower than for batch gradient descent, since the gradient for stochastic gradient descent is noisier. However, this noise might also help escape local minima in certain situations and converge to a local minima which is closer to or same as the global minima.

4.1.2 Batch Sizes

The other learning parameters are set to the following values –

1. *Maximum Epochs = 25*
2. *Learning Rate = 0.001*
3. *Momentum = 0.1*
4. *Decay = 0.0001*

We experiment for the following values of the batch size:
[16, 32, 64, 128]

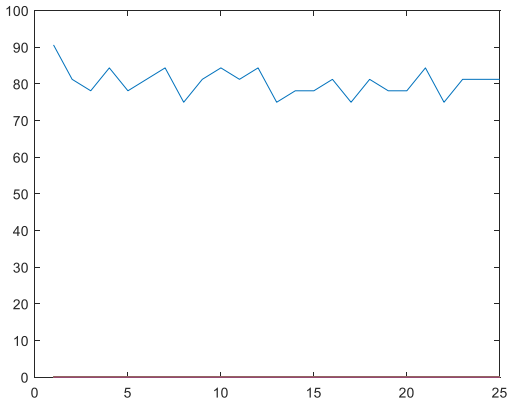


Figure 5. Training Accuracy VS Epochs (Batch Size = 32)

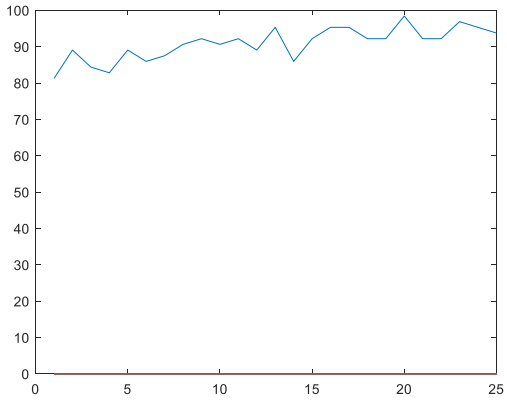


Figure 6. Training Accuracy VS Epochs (Batch Size = 64)

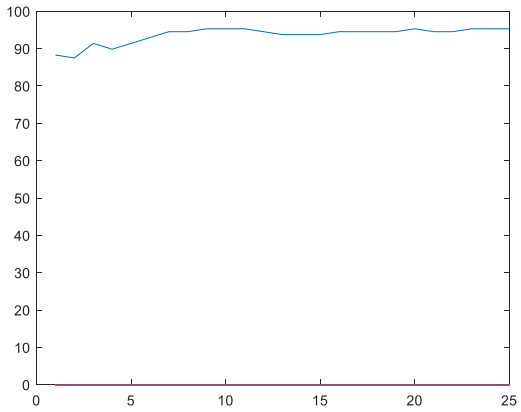


Figure 7. Training Accuracy VS Epochs (Batch Size = 128)

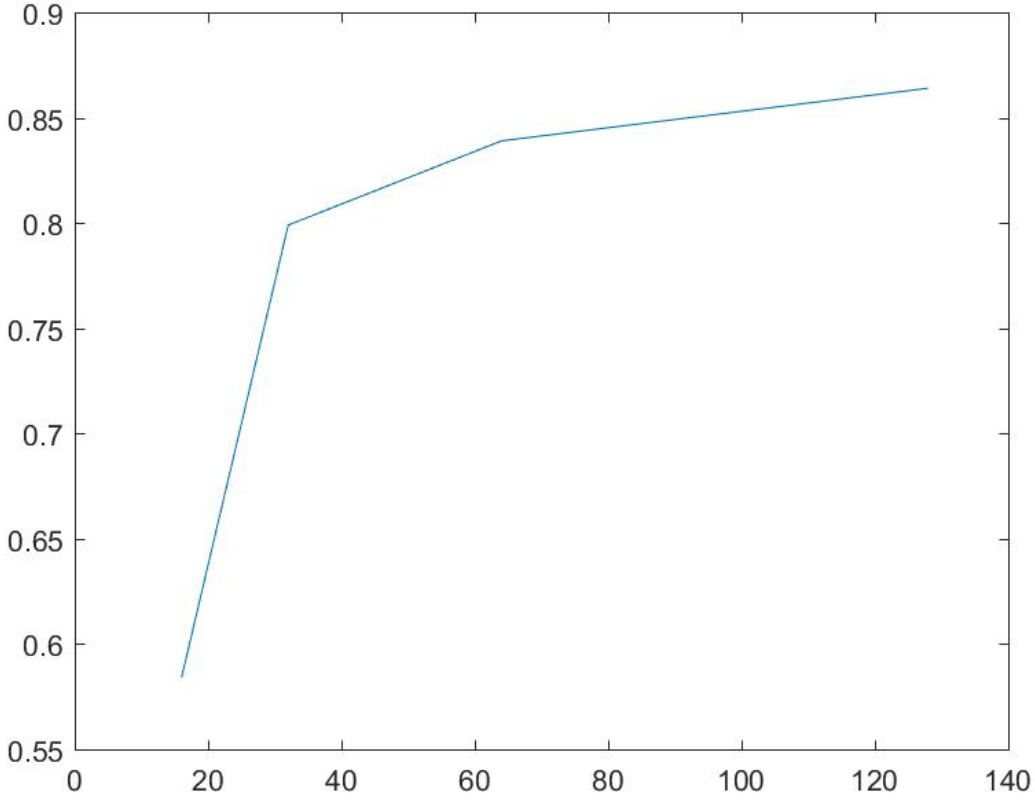


Figure 8. Test Accuracy VS Batch Size

If the size of the training dataset is very large, the computation of the gradient for the entire dataset is very computationally intensive and slow. At the same time, this can also lead to over-fitting. Hence, we divide the data into mini-batches in stochastic gradient descent and the weights are updated over mini-batches of training patterns.

As the size of the mini-batch decreases, the error function for stochastic gradient descent becomes more and more noisy. Hence, we can see lot of fluctuation in the Training Accuracy in the above figures as the batch size decreases.

As we can see from the figures, the Training Accuracy for batch sizes of 64 and 128 was higher compared to the other batch sizes for our training dataset. Although the Training Accuracy for batch sizes of 64 and 128 was almost the same, it can be seen from Figure 8 that the Test Accuracy was increasing with increasing batch sizes and the Test Accuracy for a batch size of 128 was higher than for a batch size of 64. This means that our model generalises better to new data samples with a batch size of 128 compared to a batch size of 64. Hence, we choose a batch size of 128 for our algorithm.

4.1.3 Momentum and Decay

The other learning parameters are set to the following values –

1. *Maximum Epochs = 25*
2. *Learning Rate = 0.001*
3. *Batch Size = 128*

We experiment for all combinations of the following values of Momentum and Decay:
Momentum = [0.1, 0.3, 0.9]
Decay = [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5]

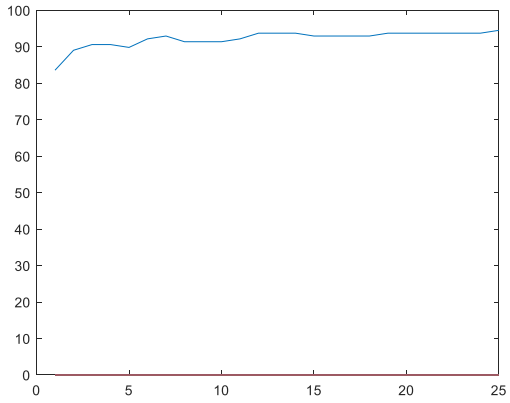


Figure 9. : Training Accuracy VS Epochs (Momentum = 0.1, Decay = 0.0005)

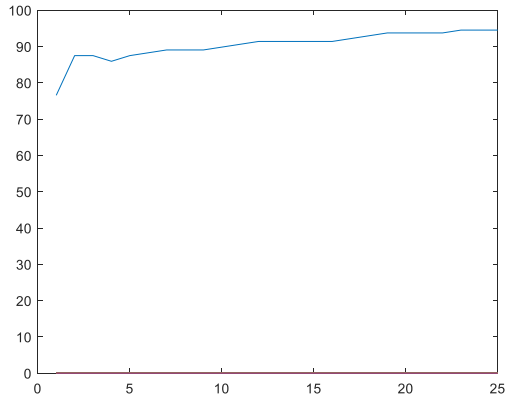


Figure 10. Training Accuracy VS Epochs (Momentum = 0.1, Decay = 0.001

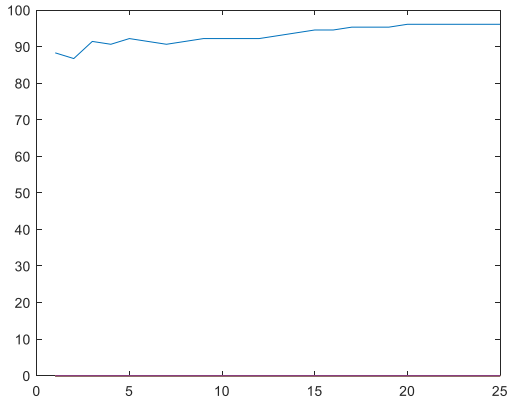


Figure 11. Training Accuracy VS Epochs (Momentum = 0.1, Decay = 0.005)

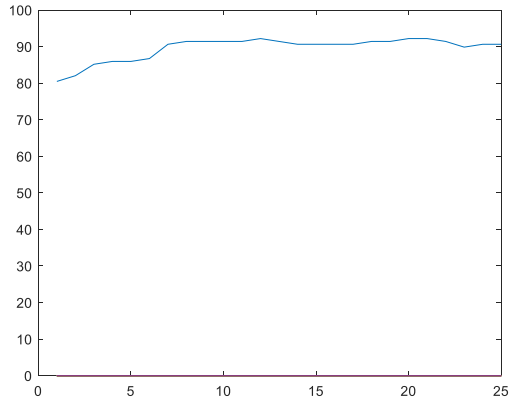


Figure 12. Training Accuracy VS Epochs (Momentum = 0.3, Decay = 0.001

		Decay							
		0.0001	0.0005	0.001	0.005	0.01	0.05	0.1	0.5
Momentum	0.1	0.8760	0.8690	0.8850	0.8740	0.8660	0.8790	0.8620	0.8810
	0.3	0.8630	0.8460	0.8700	0.8400	0.8520	0.8610	0.8790	0.8770
	0.9	0.5940	0.5950	0.5730	0.6190	0.6600	0.6940	0.6420	0.6880

Figure 133: Test Accuracy for different combinations of Momentum and Decay

Although the Training Accuracy for a momentum of 0.1 and decay of 0.005 was slightly better than for a momentum of 0.1 and a decay of 0.001, the Test Accuracy for a momentum of 0.1 and decay of 0.001 was the highest which means that it generalises very well for new data samples. Hence, we choose a momentum of 0.1 and decay of 0.001 for our learning algorithm.

The error function we use in stochastic gradient descent might often have many shallow ravines, where we might get trapped instead of converging to the optimum. The addition of the momentum term helps us in taking slightly larger steps so that we can escape these ravines. At the same time, the decay term ensures that our weights are closer to zero so that the model we obtain after training has less variance and does not over-fit the training data. However, the decay term can also not be too high else it will lead to under-fitting of the training data, which will lead to low Training Accuracy as well as low Test Accuracy.

4.1.4 Recommended Model

Based on the above analysis, we would recommend to use the following learning parameters –

1. *Learning Parameter = 0.001*
2. *Batch Size = 128*
3. *Momentum = 0.1*
4. *Decay = 0.001*

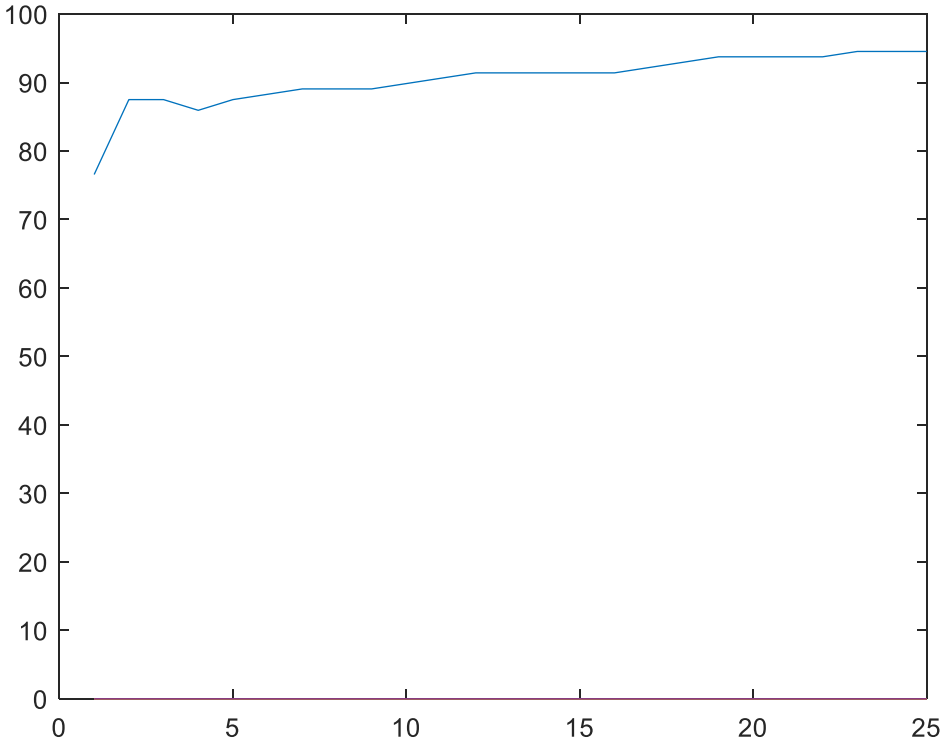


Figure 14. Training Accuracy VS Epochs for the Recommended Model

Figure 14 shows the plot of convergence for training obtained for this model. We got a Test Accuracy of 88.50% after training with this model.

4.2 Model with 2 Convolutional and Pooling layers

4.2.1 Number of filters

The learning parameters are set to the following values –

1. *Maximum Epochs = 20*
2. *Learning Rate = 0.001*
3. *Batch Size = 128*
4. *Momentum = 0.1*
5. *Decay = 0.001*

We experiment for all combinations of the following values of number of filters in both the Convolutional Layers:

Convolutional Layer 1 = [20, 30, 40, 50, 60, 70]
Convolutional Layer 2 = [20, 30, 40, 50, 60, 70]

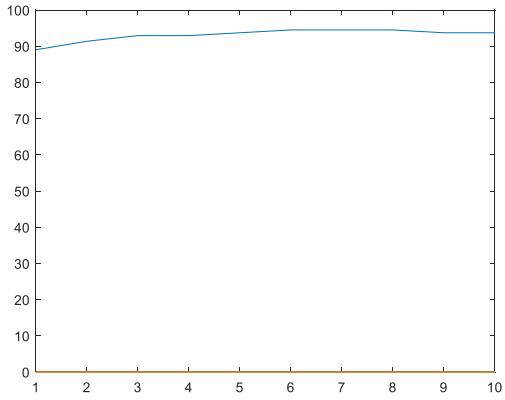


Figure 15. Training Accuracy VS Epochs (Number of filters = [50, 50])

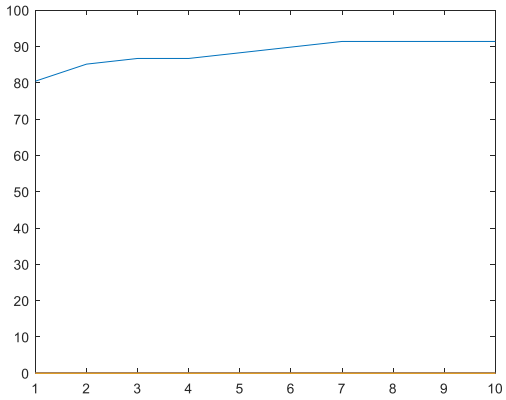


Figure 16. : Training Accuracy VS Epochs (Number of filters = [50, 60])

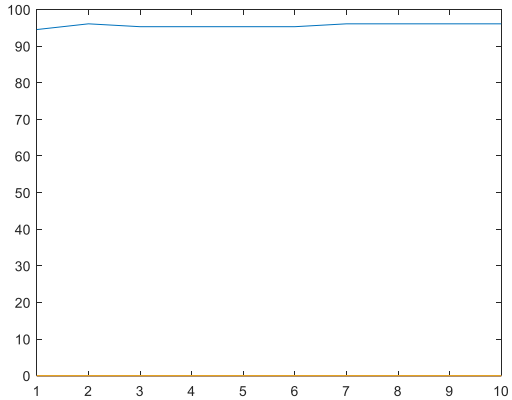


Figure 17. Training Accuracy VS Epochs (Number of filters = [50, 70])

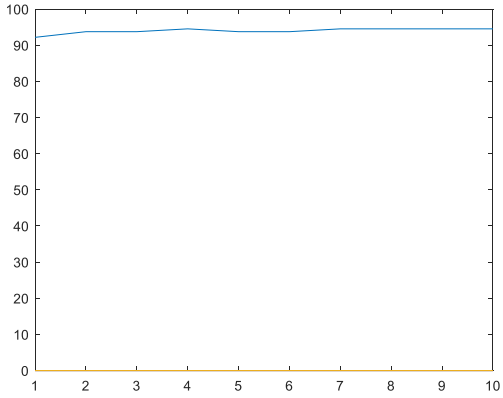


Figure 18. Training Accuracy VS Epochs (Number of filters = [40, 60])

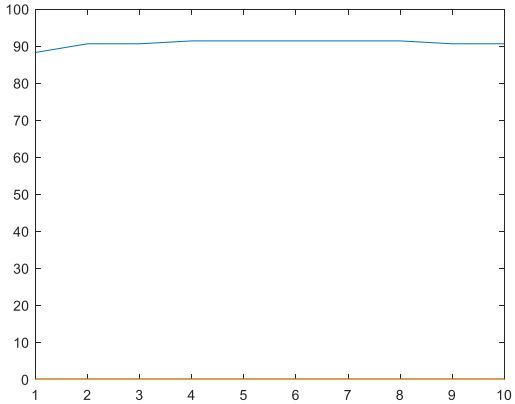


Figure 19. Training Accuracy VS Epochs (Number of filters = [60, 60])

		Number of filters in Convolution Layer 2					
		20	30	40	50	60	70
Number of filters in Convolution Layer 1	20	0.8940	0.8990	0.8840	0.8960	0.8960	0.8860
	30	0.8940	0.8980	0.8950	0.8900	0.8830	0.8990
	40	0.8930	0.8860	0.8900	0.8930	0.8900	0.8910
	50	0.8910	0.8960	0.8900	0.8970	0.9030	0.8820
	60	0.8930	0.8930	0.8920	0.8880	0.8920	0.8910
	70	0.8990	0.8910	0.8970	0.9010	0.8990	0.8950

Figure 20. Test Accuracy for different number of filters in Convolutional Layer 1 and Convolutional Layer 2

Although the Training Accuracy with 50 filters in the first Convolutional Layer and 60 filters in the second Convolutional Layer is lower compared to some of the other models, the Test Accuracy for that model is the highest which means that it generalises well to new data samples while the other models slightly over-fit the training data. Hence, we choose a model with 50 filters in the first Convolutional Layer and 60 filters in the second Convolutional Layer.

4.2.2 Recommended Model

Based on the above analysis, we would recommend to use the following number of filters –

1. *Number of filters in first Convolutional Layer = 50*
2. *Number of filters in second Convolutional Layer = 60*

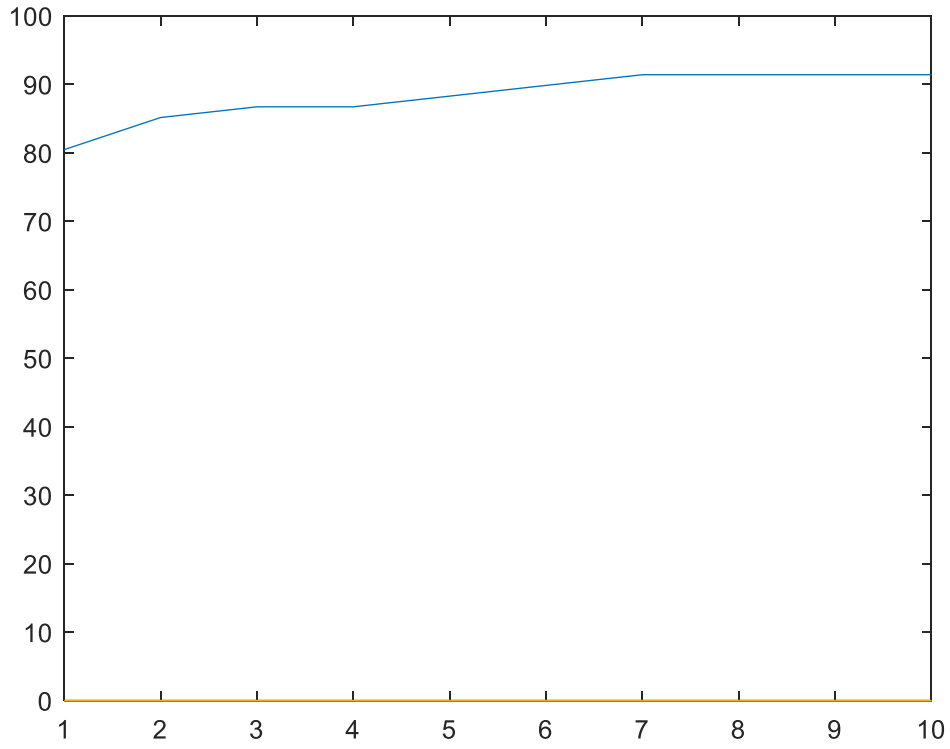


Figure 21. Training Accuracy VS Epochs for the Recommended Model

On training with this architecture and the learning parameters obtained from the first model, we get a Test Accuracy of 90.30%, which is higher than the Test Accuracy of 88.50% obtained in Part 1. Hence, it is better to train with this architecture instead of the architecture obtained in Part 1, while still using the learning parameters from Part1.

However, it is not always true that the Test Accuracy of the model will increase if we increase the number of convolutional layers. In general, although the Test Accuracy of the model might increase initially on increasing the number of convolutional layers, the Test Accuracy might actually start decreasing if we keep increasing the number of convolutional layers. This is because addition of each convolution layer to the model reduces the number of input features to the fully connected layers.

5 Conclusion

The kind of model we choose is dependent on our dataset and the distribution of the data in our dataset greatly impacts the accuracy and efficiency of our model. Hence, it is essential to shuffle the data randomly before training to reduce the chances of having an unfortunate split or of having entire mini-batches of highly correlated examples.

We experiment with many parameters like the learning rate, mini-batch size, momentum and decay term in order to find the optimum model for our dataset. Each parameter has a different purpose and can impact our accuracy on the training and test set in different ways. We can also extend this project to experiment with more parameters like the ratio in which the dataset is divided into train set and test set, the dimensions of the kernel, dimensions of the pooling regions, function used for pooling and so on. However, we do not experiment with these parameters for this project as they are outside the scope of this project.

PART II - AUTOENCODERS

1 Introduction

An autoencoder is an unsupervised learning algorithm that attempts to reconstruct its input, i.e. set the target values to be equal to the inputs $y^{(i)} = x^{(i)}$. The internal representation of an autoencoder encodes the input features into a feature vector in the hidden layer and learns the weights for these hidden layers by learning the approximation to the identity function $h_{W,b}(x) \approx x$.

The identity function places a constraint in the network. By limiting the number of neurons in the hidden layer interesting structure of the data can be learned from the autoencoders. Hence, in the case where the dimension of the hidden layer n_h is (< n), the autoencoder obtain a lower dimensional representation of the input signals, that is it performs feature extraction algorithm that learns the important features of the data. This representation can be then feed into other algorithms, such as softmax algorithm for classification problem.
In this experiment, we aim to utilise the Autoencoder in order to use neural network as content addressable memories. The project uses hand-written digit images provided by the MNIST database for the dataset. Each image is a black and white (bilevel) images that is made up of 28x28 pixels (784 pixels). The total size of ~5000 training images and ~1000 testing images have been selected for the project.

1.1 Algorithm

The training process of an autoencoder is based on cost function. The cost function measures the error between the input x and its reconstruction at the output. An autoencoder is composed of an encoder and a decoder. The encoder and decoder can have multiple layers. If the input to an autoencoder is a vector x, then the encoder maps the vector x to another vector z, as follows:

$$z^{(1)} = h^{(1)}\left(W^{(1)}x + b^{(1)}\right)$$

Figure 22. Encoder

where the superscript (1) indicates the first layer, $h^{(1)}$ is a transfer function for the encoder, W is the weight matrix and b is the bias vector. Then, the decoder tries to reconstruct the input by mapping the encoded representation z into an estimate of the original input vector, x, as follows:

$$\hat{x} = h^{(2)}\left(W^{(2)}x + b^{(2)}\right)$$

Figure 23. Decoder

where the superscript (2) represents the second layer. $h^{(2)}$ is the transfer function for the decoder.

The reconstructed input and the input values are then compared using a cost function, which uses Mean Squared Error (MSE). This MSE is then minimised during the backpropagation learning, so that the weights for the autoencoder that yield representations close the inputs can be obtained.

$$E = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(x_{kn} - \hat{x}_{kn}\right)^2$$

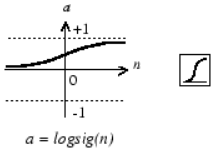
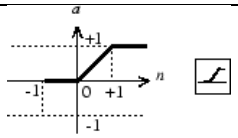
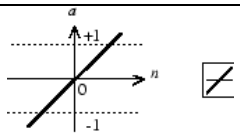
Figure 24. Mean Squared Errors

1.2 Autoencoder Parameters

1.2.1 Transfer Functions

Transfer functions are used to calculate a layer's output from its net input. The transfer functions are used by the neurons to determine neuron activation. For example, with sigmoid

function, a neuron is active when the output is close to 1 and inactive when the output is close to 0.

Log-Sigmoid Transfer Function	Saturating Linear Transfer Function	Linear Transfer Function
 a = logsig(n) Range: (0 1)	 a = satlin(n) Range: [0 1]	 a = purelin(n) Range: [0 1]
logsig(n) = 1 / (1 + exp(-n))	a = satlin(n) = 0, if n <= 0 n, if 0 <= n <= 1 1, if 1 <= n	a = purelin(n) = n

1.2.2 Sparsity Constraints

When the size of hidden units is large, sparsity constraints can be imposed on the input data to explore interesting structures of the inputs. The activation transfer functions can determine whether a neuron is active or inactive. Sparsity constraint can be established to the AutoEncoder by applying a regularizer, a function of the average output activation value of a neuron. The average activation of a hidden neuron i is defined as:

$$\hat{\rho}_i = \frac{1}{n}\sum_{j=1}^{n} z_i^{(1)}(x_j) = \frac{1}{n}\sum_{j=1}^{n} h\left(w_i^{(1)T}x_j + b_i^{(1)}\right)$$

Figure 25. Sparsity Constraints

We would like to enforce the constraint: $\rho_i = \rho$, where ρ is the sparsity parameter. ρ is usually a small value close to 0. Lower ρ encourages the autoencoder to learn a representation, where each neuron in the hidden layer fires to a small number of training examples so that each neuron specializes by responding to features that is exist in a small subset of the training samples.

Sparsity regularizer is used to enforce a constraint on the sparsity of the output from the hidden layer. Sparsity can be established by adding a regularization term that takes a large value when the average activation value, ρ_i of a neuron i is not close to its desired value ρ. Here, we used Kullback-Leibler divergence to compute the regularization term.

$$\Omega_{sparsity} = \sum_{i=1}^{D^{(1)}} KL(\rho||\hat{\rho}_i) = \sum_{i=1}^{D^{(1)}} \rho \log\left(\frac{\rho}{\hat{\rho}_i}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_i}\right)$$

Figure 26. Sparsity Proportion

In matlab, this $\Omega_{sparsity}$ is enforced by the *SparsityProportion* parameter.

1.2.3 Cost Function

$$E = \underbrace{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(x_{kn} - \hat{x}_{kn}\right)^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{weights}}_{\substack{L_2 \\ \text{regularization}}} + \beta * \underbrace{\Omega_{sparsity}}_{\substack{\text{sparsity} \\ \text{regularization}}}$$

Figure 27. Cost Function of an Autoencoder

In order to enforce sparsity constraints and avoid overwitting, we added sparsity regularization and regularization to the cost function. As seen in the formula above, β is the coefficient to control the sparsity regularization term. In matlab, the values of β can be set using the *SparsityRegularization* parameter.

2 Methodologies

2.1 Data Pre-processing

Before training and testing, the train and test data was first shuffled using the 'autoencoderPreprocess.m', which generates matlab objects for the data images and labels used during the training and testing. After the preprocessing, we will generate the following matlab objects:

- *dataTrain.mat*: 1x5005 cells, with 28x28 matrix in each cell. Every cell in the matrix corresponds to each pixel in the image and has a value within the [0 255] range. This range represents the intensity of the pixel, with 0 as total absence (black) and 255 as total presence (white). This image data will be used as the input to train the network model.
- *labelsTrain.mat*: The target data, 10x5005 matrix of one hot encoded values. Each column in the matrix represent the number of observations, which in this case is 5005. The row is a dummy variable representing a particular class, where all entries in a column are zero except for a single row. This single entry indicates the class label for the sample. There are 10 possible image labels, which is a number between 0-9. This data will be used for training a network model for classification task.
- *dataTest.mat*: 1x1000 cells, with 28x28 matrix on each cell. This data will be used to evaluate the model performance.
- *labelsTest.mat*: 10x1000 matrix. This data will be used to evaluate the network model for classification task.

2.2 Model Architectures and Learning Algorithms

2.2.1 1 Autoencoder

In part A, we implemented ANN model with 1 hidden layer. This layer is an autoencoder which made of 100 neurons. These neurons will then store the features of the input during the encoding process and tries to reconstruct the image back during the decoding process.

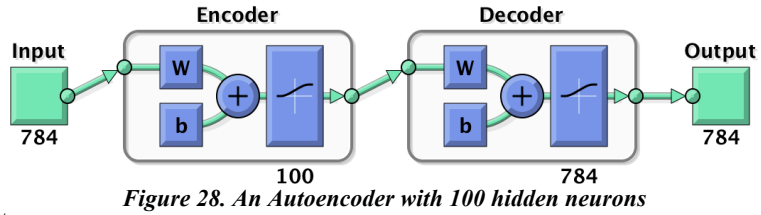


Figure 28. An Autoencoder with 100 hidden neurons

For the 1st experiment, we tested three hyperparameter variations for the AutoEncoder: 1) number of epochs, 2) sparsity regularization coefficient and sparsity proportion, and 3) transfer functions.

2.2.2 2 Autoencoders

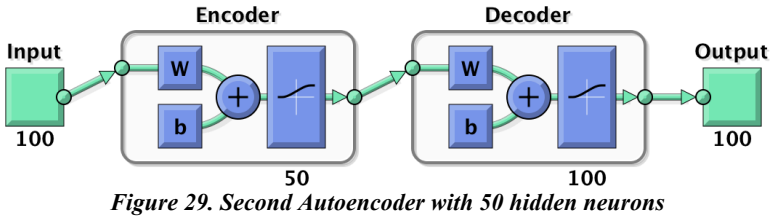


Figure 29. Second Autoencoder with 50 hidden neurons

In part B, we introduced a second autoencoder into the network model. The second autoencoder has 50 neurons and uses the outputs of the first Autoencoder in part 2.2.1 as the input features. As seen in the diagram above, because the first Autoencoder has 100 neurons, the second encoder will receive 100 input features, and reduce the dimensionality of the features into 50 features. Then, at the decoder it will try to reconstruct the input back into 100 features.

2.2.3 2 Autoencoders and 1 Softmax layer

In part C, we added a softmax layer as the last layer in the network model. In this architecture, there are 3 hidden layers: 2 hidden layer of autoencoders and the last layer for softmax.

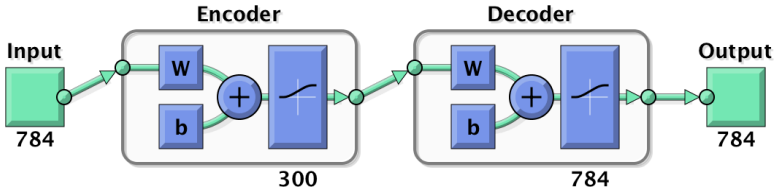


Figure 30. 1st Layer: AutoEncoder with 300 hidden neurons

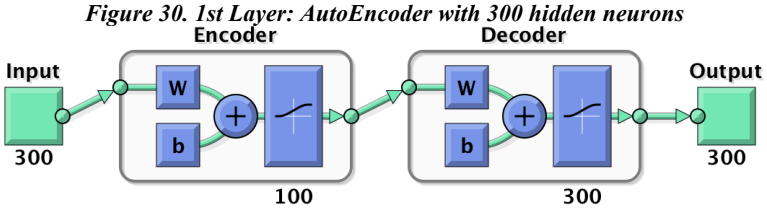


Figure 31. 2nd Layer: AutoEncoder with 100 hidden neurons

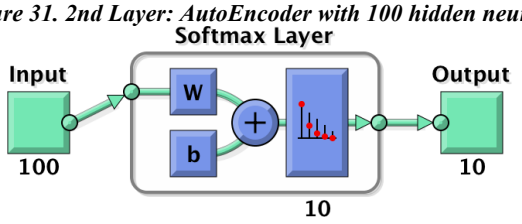


Figure 32. 3rd Layer: Softmax with 10 possible classes.

First, the image data will be encoded using the first autoencoder and the output is fed into the second autoencoder. The second autoencoder will further encode the encoded image and generate some outputs. Next, the softmax layer will receive the second AutoEncoder outputs. The softmax layer is used to solve multi-class labels classification problem, which in this experiment is given image data, classify which numeric label (0-9) is appropriate for that image data. Because there are 10 possible classes, the softmax layer will have 10 neurons at the output. Unlike autoencoders, a softmax layer is trained in a supervised using the labels from the training data.

For the third experiment, we tested different hyper parameters of the network model, such as: 1) number of neurons in the hidden layers, and 2) sparsity proportion and sparsity regularization coefficient of the autoencoders. In this part, we also did some experiment with deep network by stacking several autoencoders together with the softmax layer and compare its performance.

3 Implementation

3.1 Model with 1 Autoencoder Layer

The following parameters for training the Autoencoder are first set to the following values at initial.

- **EncoderTransferFunction and DecoderTransferFunction: logsig**
- **MaxEpochs: 1000**
- **SparsityProportion: 0.05**
- **SparsityRegularization: 1**

After each experiment, we observe the results and change the parameter of the next experiment to use the best parameters from previous observations. The best parameter in this case is the hyper parameters that generate the lowest MSE

1) Number of Epochs

First, we performed the first experiment to find the best model among different number of epochs. We used the following numbers for the maximum epoch size:

Different Epochs: [100 200 300 400 500 600 700 800 900 1000].

2) Sparsity Proportions and Sparsity Regularization Coefficient

Best Epoch: 1000 (from previous experiment)

We used 1000 as the max epoch size to experiment with different sparsity proportion and regularization coefficient.

Different Sparsity Proportions: [0.01 0.05 0.1 0.2 0.3].

Different Sparsity Regularization Coefficients: [1 4 7 10]

3) Transfer Functions

Best Epoch: 1000

Best Sparsity Regularization Coefficient: 10

Best Sparsity Proportion: 0.1

Based on the best parameters found in the previous experiment, we used them to test with different transfer functions.

Different Encoding Transfer Functions: [logsig satlin]

Different Decoding Transfer Functions: [logsig satlin purelin]

3.2 Model with 2 Autoencoder Layers

The learning parameters used to create the Autoencoders in the second experiment used the best parameters found in experiment 2.2.1. The two autoencoders in this experiment will have the same hyper parameters except for the number of neurons. These controlled variables of the Autoencoders include:

- **EncoderTransferFunction and DecoderTransferFunction: logsig**
- **MaxEpochs: 1000**
- **SparsityProportion: 0.1**
- **SparsityRegularization: 10**

Here, the first autoencoder receives input features and encode it into an intermediate representation. This output is fed as an input to the encoder of the second autoencoder to generate an encoded representation of the encoded image. Next, this output is decoded using the second autoencoder's decoder, and reconstructed back to the input features using the decoder of the first autoencoder. The reconstruction MSEs of using two autoencoders are then compared to the results obtained from using 1 autoencoder.

3.3 Model with 2 Autoencoder Layers and 1 Softmax Layer

The following parameters for the Autoencoder are first set to the following values at initial.

- **EncoderTransferFunction and DecoderTransferFunction: logsig**
- **MaxEpochs: 200**
- **SparsityProportion: 0.1**
- **SparsityRegularization: 10**

1) Number of Neurons in the Hidden Layers

Hidden Neurons of First Autoencoder = [400, 300, 200, 100]

Hidden Neurons of the Second Autoencoder = [200, 100, 50]

2) Sparsity Proprotion and Sparsity Regularization Coefficient

Best Hidden Neurons for the First Hidden Layer: 300

Best Hidden Neurons for the Second Hidden Layer: 100

sparsity_regs = [1, 4, 7, 10]

sparsity_props = [0.1, 0.2, 0.3]

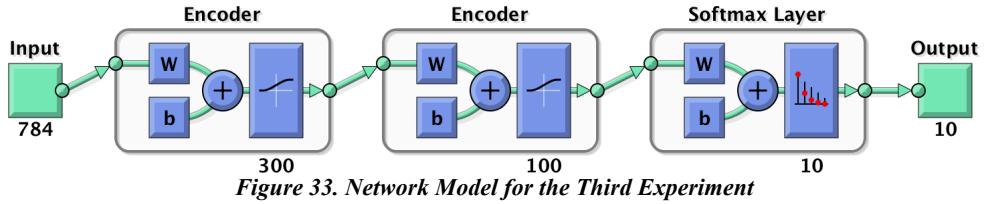


Figure 33. Network Model for the Third Experiment

In this experiment, the softmax layer receives the intermediate representation from the second autoencoder as the input features. This layer is trained using the training image labels and tested using the testing image data and testing image labels to evaluate the performance. The accuracy of each model is compared using confusion matrix.

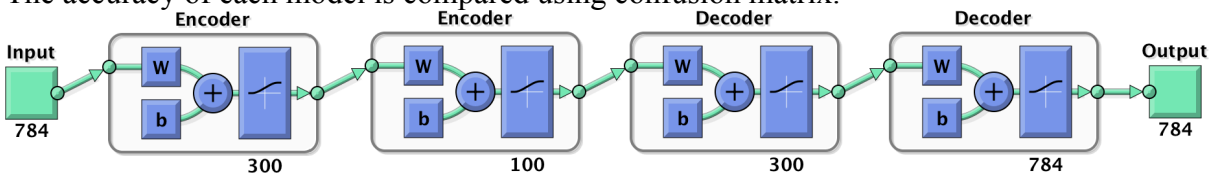


Figure 34. Autoencoders in the Third Experiment

Similar to part 3.2, in order to compare between the reconstruction errors of each models, the image needs to be decoded back to the original data by using decoders.

3) Stacked Neural Network

For the deep network experiment, the size of the hidden representation of the layer must match the input size of the next layer or network in the stack. The first input argument of the stacked network is the input argument of the first autoencoder. The output from the first encoder is the input of the second autoencoder in the stacked network. The output from the second encoder is the input argument to the final layer of in the stacked network, the softmax layer. We used the best parameters obtained in the previous experiment which were 300-100 hidden neurons in the first and second autoencoder.

4 Results and Analysis

In this section, the results of different hyper parameters will be discussed. The comparison method will Mean Squared Errors (MSE) between inputs and its reconstructed output. This MSE obtained from experimenting with various hyper parameters will be displayed as a diagram and a table in each experiment. For reconstruction results, we will have a table. The first row (or the first 2 rows for multiple autoencoders) will show the weight visualizations while the last row displays the reconstructed image. Note that for the reconstruction results, we will only compare and discuss the best and worst hyper parameters observed during the experiment. The best and worst hyper parameters of an experiment is defined as those parameters that generate the lowest and highest MSE errors respectively.
Below is the original test image data used to evaluate the models.

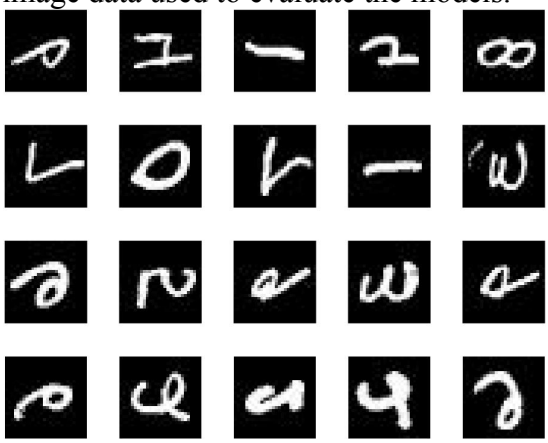


Figure 35. Original Test Image

4.1 Model with 1 Autoencoder layer

4.1.1 Number of Epochs

4.1.1.1 MSE

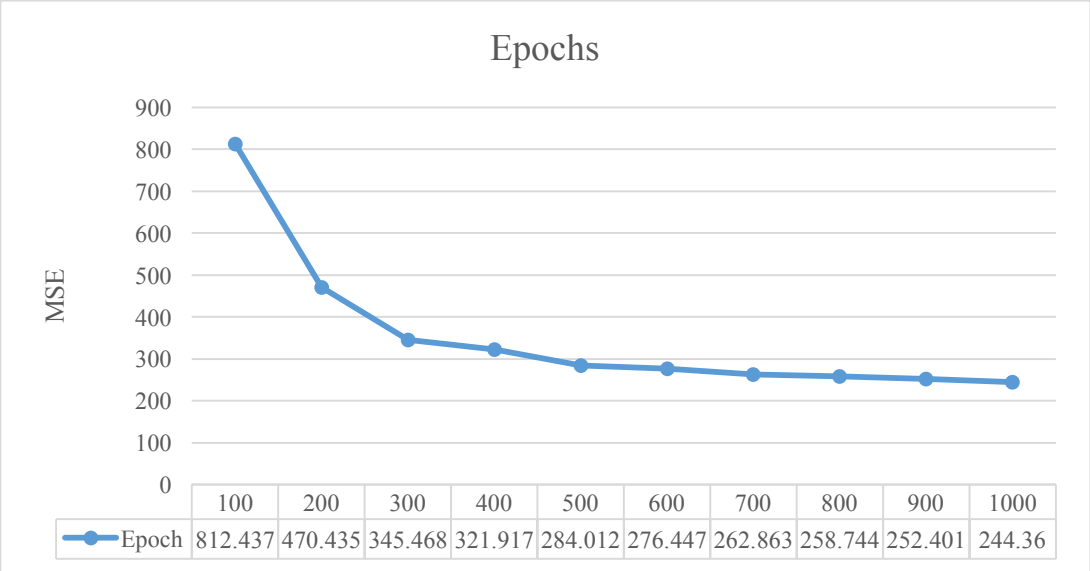


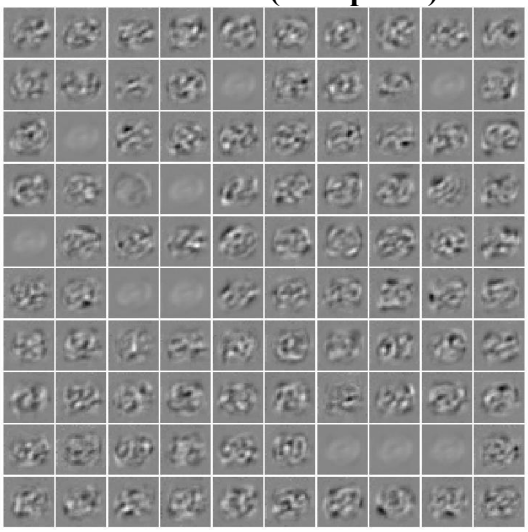
Figure 36. MSE Plot of Various Max Epoch Size

As seen in the diagram and the table above, we can see the inverse relation between MSE and epoch size. As we increase the number of epochs, the backpropagation will continue to optimize the network by minimising the MSE, hence the MSE will decrease as the number of epochs in increased. The highest epoch size, 1000 epochs, gives the best MSE of 244.36 while the lowest epochs, 100 gives the worst MSE of 812.437.

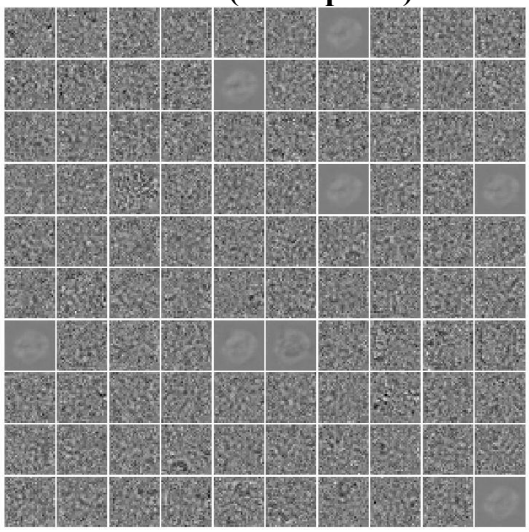
Best Epoch	Lowest MSE
1000	244.36

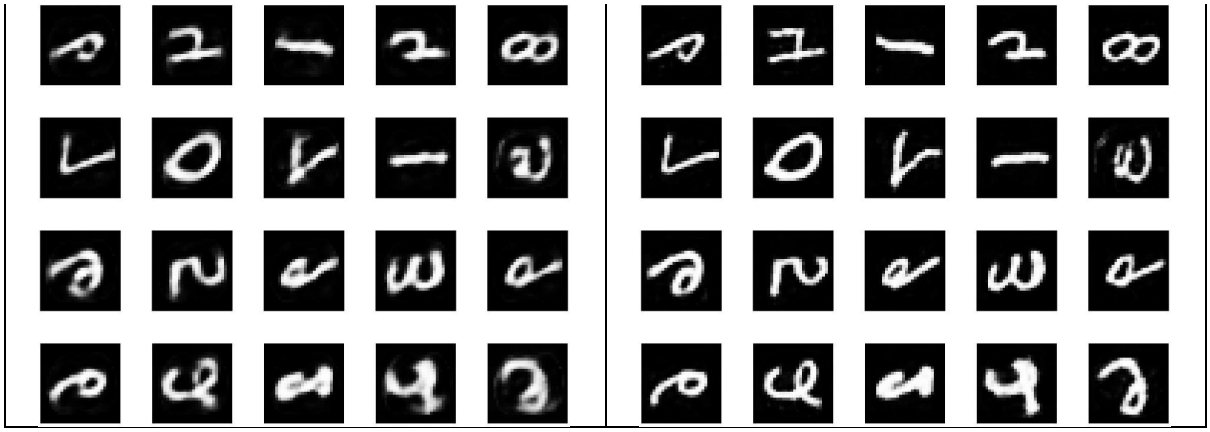
4.1.1.2 Reconstruction

Worst MSE Error (100 epochs) ≈ 812



Best MSE Error (1000 epochs) ≈ 244.36

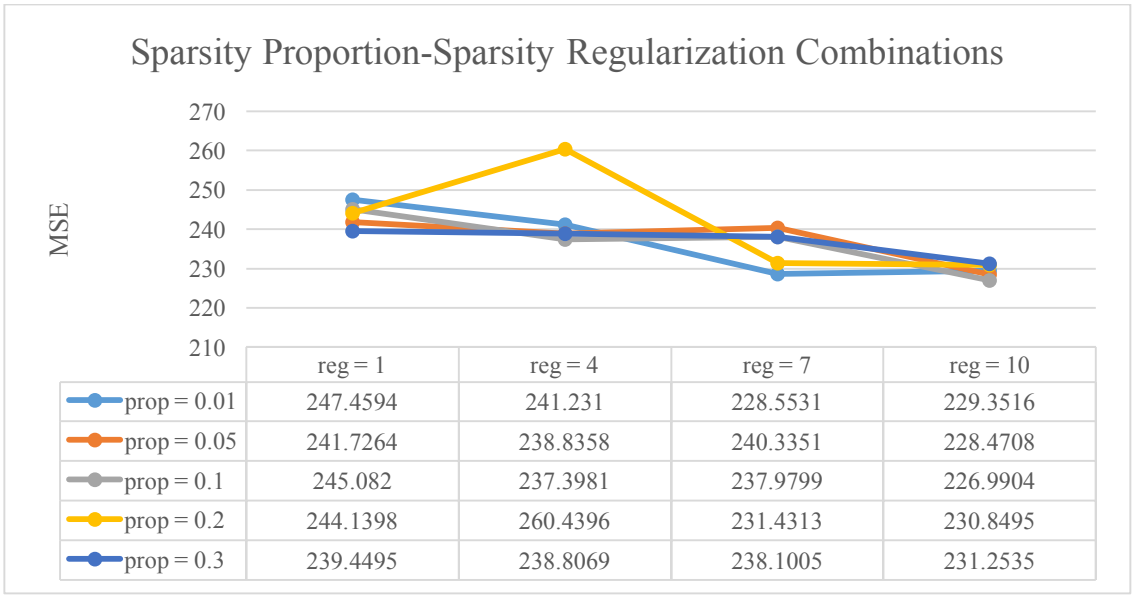




As visualized above, model with 1000 epochs gives a reconstructed image that looks closer to the original image compared to the 100 epochs. The reason is because the model with 1000 epochs can learn more information about the input features because of the cost function minimization in backpropagation algorithm, hence this model stores better weights/features that best represent the image features than other models with lower epochs.

4.1.2 Sparsity Proportions and Sparsity Regularization Coefficients

4.1.2.1 MSE



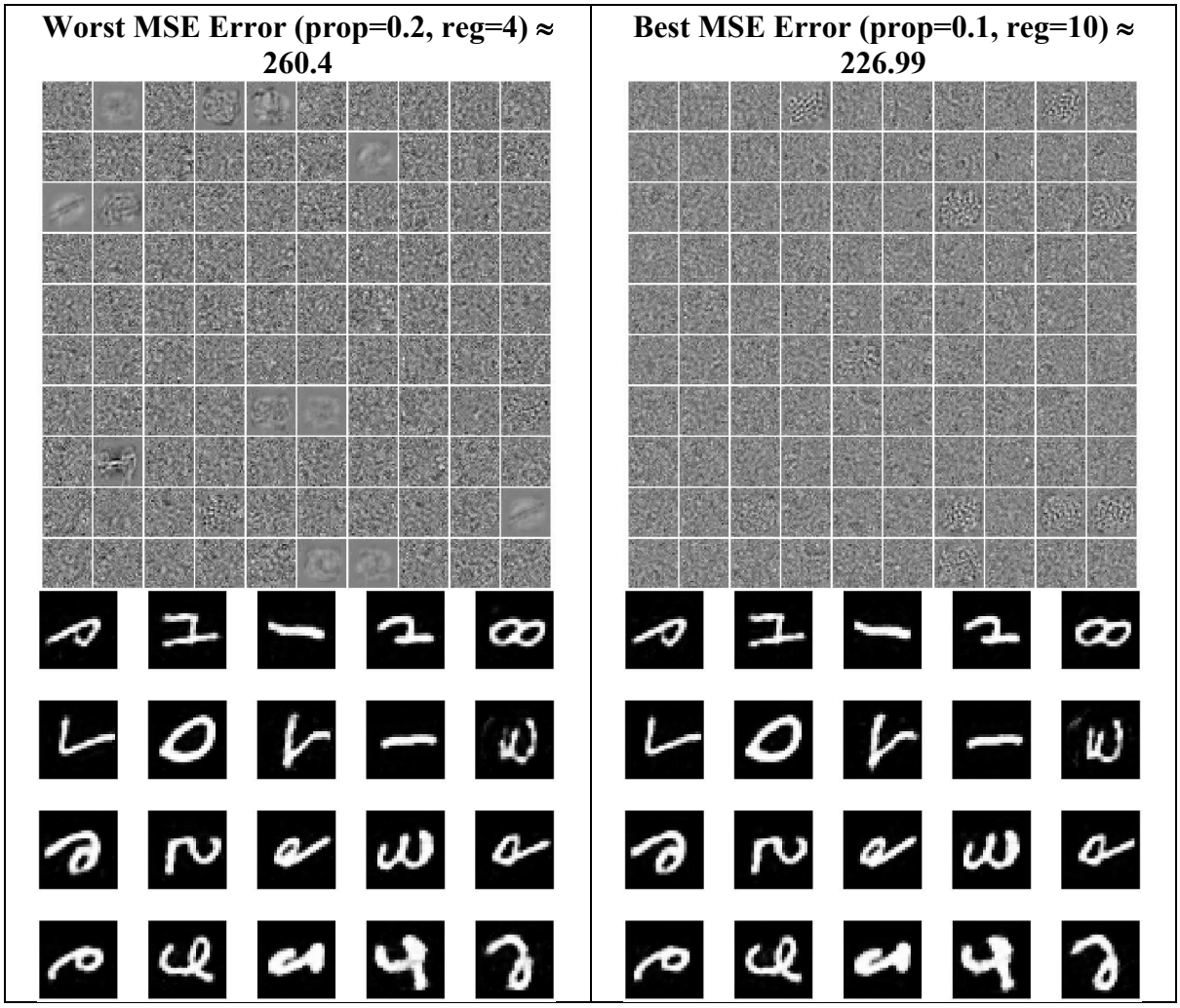
	reg = 1	reg = 4	reg = 7	reg = 10
prop = 0.01	247.4594	241.231	228.5531	229.3516
prop = 0.05	241.7264	238.8358	240.3351	228.4708
prop = 0.1	245.082	237.3981	237.9799	226.9904
prop = 0.2	244.1398	260.4396	231.4313	230.8495
prop = 0.3	239.4495	238.8069	238.1005	231.2535

Figure 37. MSE Plot of Various Sparsity Proportion and Sparsity Regularization Pairs

From the plot and the table, we can see that there seems to be no correlation between sparsity and MSE, nor sparsity proportion and sparsity regularization, this is shown by the random and imbalanced fluctuation of the MSE when the sparsity parameters are varied. However, it can be seen that as we vary these parameters, the range between the lowest MSE = 226.9904 and highest MSE = 260.4396 is relatively small, only ~33.4 . This means that for the network model with 50 hidden neurons, varying sparsity proportion into 0.01-0.3 and sparsity regularization into 1-10 does not really matter because these models generate MSE that fall within similar range. We suspect this is because the size of the autoencoder is small (50 neurons only). It is not a huge layer, so sparsity in this case is not significant. Observing from the MSE results, the sparsity proportion of 0.1 and sparsity regularization of 10 generates the lowest MSE.

Best Sparsity Proportion	Best Sparsity Regularization	Lowest MSE
0.1	10	226.9904

4.1.2.2 Reconstruction



From the visual observations, it can be seen that both reconstructed images look similar because their MSE difference is relatively small.

4.1.3 Transfer Functions

4.1.3.1 MSE

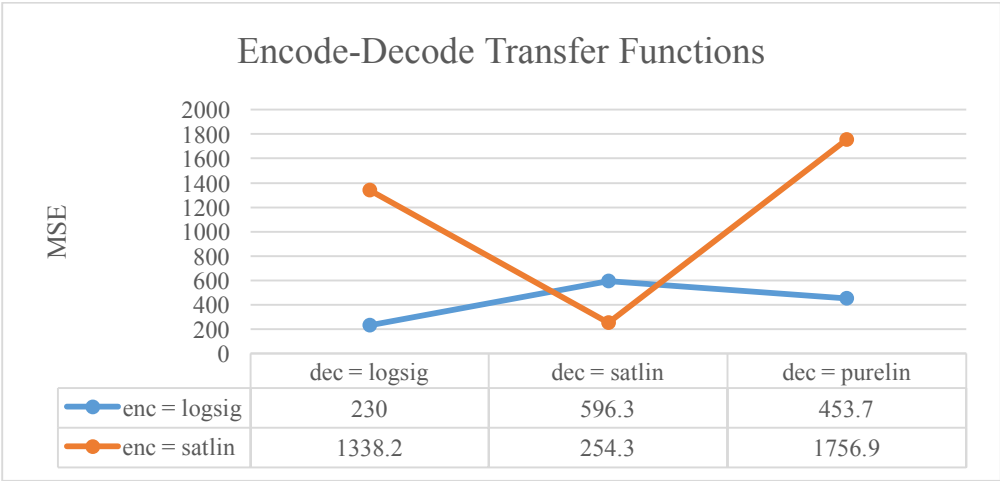
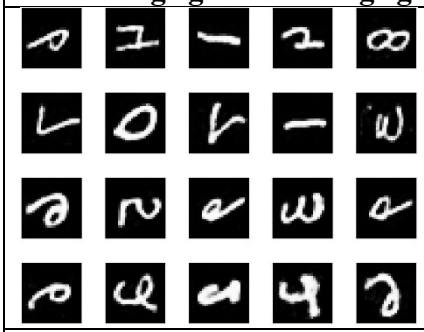
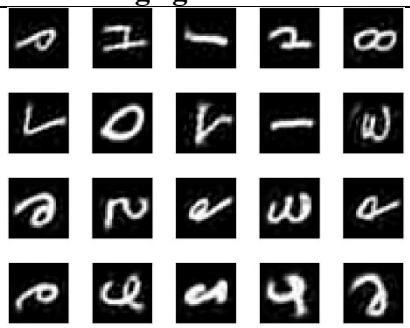
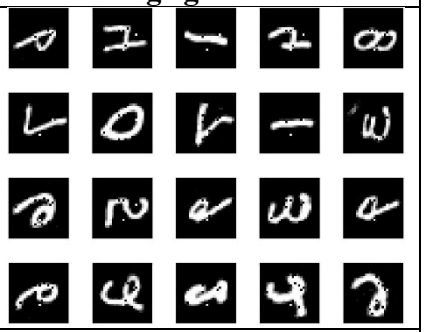
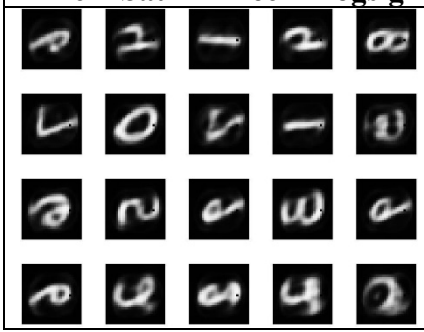
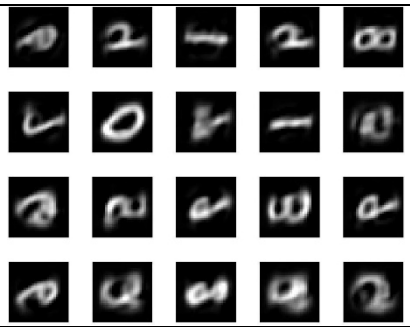
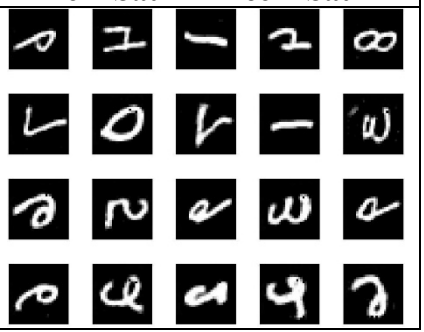


Figure 38. MSE Plot of Various Encode Decode Transfer Function Pairs

From the table, we can see how selecting transfer functions really affect the Autoencoder performance. This can be observed because the gap between the maximum MSE (~1756.9) and minimum MSE (230) = 1756.9-230 = 1526.9, is quite large. Based on the

observations, both logsig-logsig and satlin-satlin produces better results compared to other transfer function combinations. The reason is because, logsig and satlin are able to generalise better and the output always falls within the [0 1] range compared to the simplistic purelin model, therefore it is able to reconstruct the image better and fits better to the complex data.

Enc = Logsig – Dec = Logsig	Enc = Logsig - Dec = Purelin	Enc = Logsig - Dec = Satlin
		
Enc = Satlin - Dec = Logsig	**Enc = Satlin - Dec = Purelin**	**Enc = Satlin - Dec = Satlin**
		

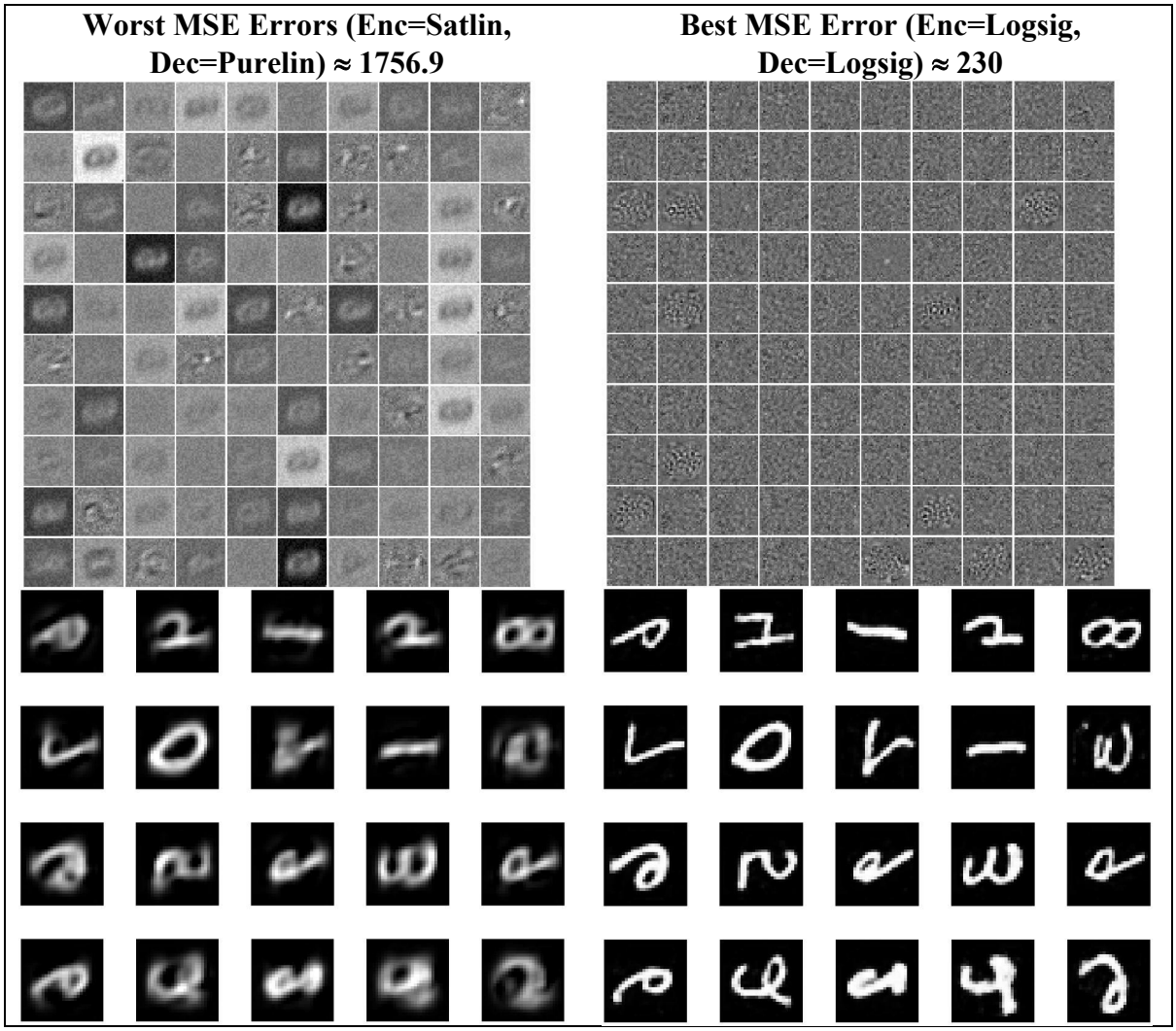
Comparing the reconstructed images and its mse from the table, it can be seen that generally, the Autoencoder learns the input features better if we use logsig transfer function as the encoder, espescially using logsig-logsig at both the encoder and decoder layer. From the table, using purelin in the decoder will generate worst results, because this transfer function is too simple to learn from the complex nature of the image data. Therefore, using non simple linear transfer function, such as logsig and satlin generalise better as a hidden representation function approximator than linear transfer functions. Aside from using logsigs, using satlin in both encoder and decoder also generates a good reconstructed results (mse error of ~253), this is because satlin has similar plot as sigmoid function.

Best Encoder Transfer Function	Best Decoder Transfer Function	Lowest MSE
Logsig	Logsig	230

If we look closely to the reconstructed images produced by satlin, we can see that there are some noises in the reconstructed image (the little dots). This is because, satlin will maps output to 0 if x is less than 0 and to 1 if x is > 1, therefore the output signals will change very fast to either black or white, unlike the smooth transition with logsig signals. With the same reason, using satlin for the encoder and logsig for the decoder will lower the model performance because there will be information loss when decoding the result from logsig to satlin as some outputs is set to 0 and 1 by the satlin function. This case does not happen if we are using logsig as the encoder and satlin as the decoder. This is because the input signals from logsig will always be decimal values between [0 1] which is within satlin allowable range, hence the satlin function is able to produce a comparable result.

Although logsig-logsig transfer functions win in this experiment, using satlin-satlin can beneficial to sparse network. Sparsity arises when the features ≤ 0. The more these units exist in a layer, the sparser the resulting representation become. On the other hand, logsig function always generates non-zero values resulting in dense representations. Therefore, in the case of sparse representations, using satlin-satlin will be beneficial.

4.1.3.2 Reconstruction Results



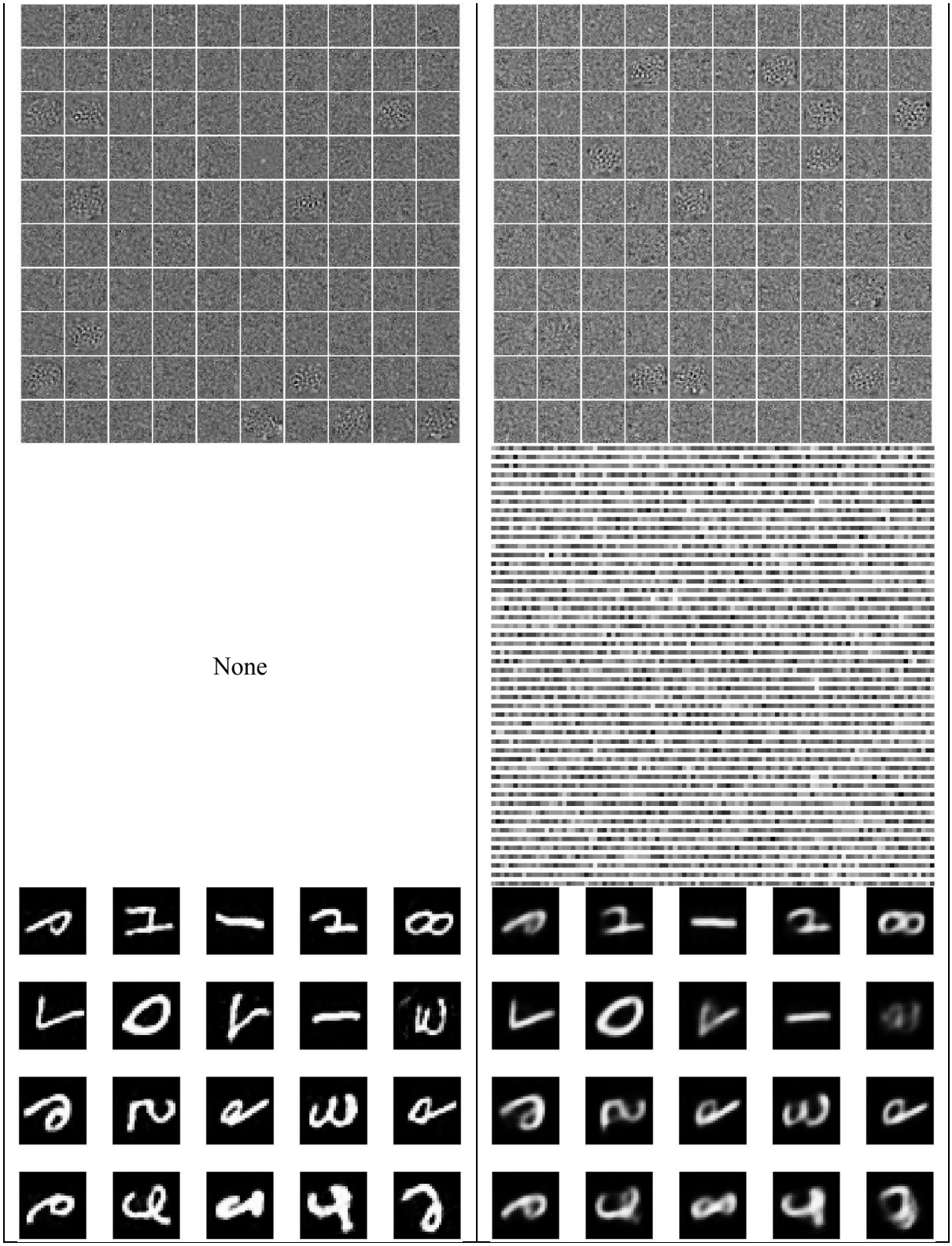
Based on the observation above, it can be seen that there is information loss in using satlin-purelin functions, shown clearly by the very low quality of the reconstructed image if compared to the high quality of the reconstructed image from using logsig-logsig transfer functions.

4.2 Model with 2 Autoencoder Layers

For the second part, we use the best parameters from part 4.1 to train 2 autoencoder layers and to create a network with 2 autoencoders. The MSE that we received from this network is **1723.5982**, which is very high compared to the lowest MSE found in part 4.1 (~227). This is because, with the addition of the second Autoencoder, the encoded image is further encoded again, hence there will be more dimensionality reduction. This will allow the network model to learn more about the important features of the input images, however it will make it harder to reconstruct the input image back because of the double compressions. So there will be trade-offs between the saved features in the autoencoders and the reconstructed image quality.

4.2.1 Sample Features and Reconstructed Image

1 Autoencoder (MSE ≈ 227)	2 Autoencoders (MSE ≈ 1723.6)



The features of network model with 2 Autoencoders (the second column of the table) is observed. From the table, the first row represents the extracted features of the first hidden layer, while the second row represents the extracted features of the second hidden layer. For the first layer, the dimension of the features is 10x10 which denotes the number of the hidden neurons in the first hidden layer, 100 neurons. For the second features image, the dimension of the image is 100x50 pixels which represent the 100 neurons from the first hidden layer and

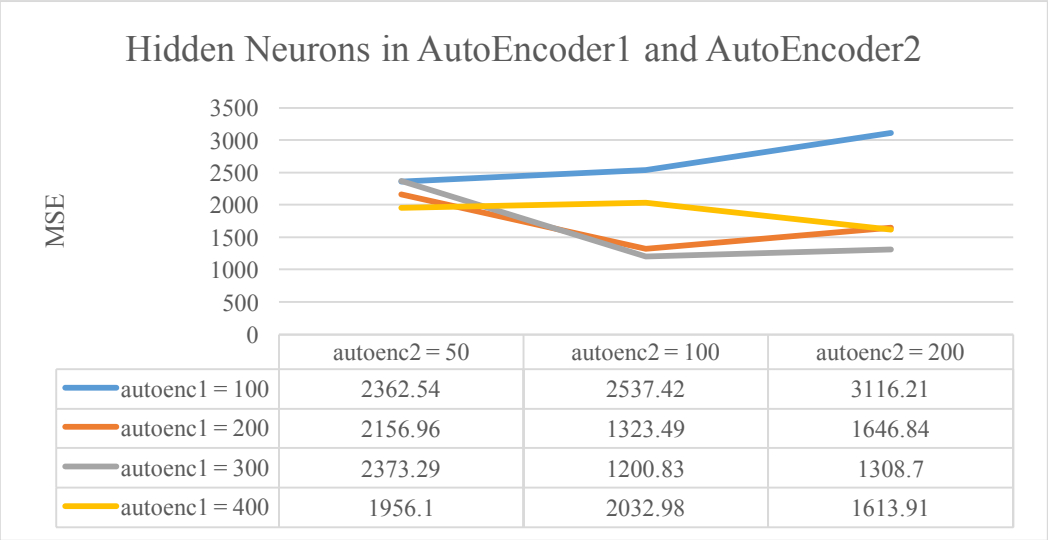
the 50 neurons in the second hidden layer. The black and white color of the weights represent the magnitude of the weights connected to the input features.

Because when using two autoencoders the image is compressed twice and decoded twice, the image will have some loss hence the quality of the image is much worse than using single autoencoder. However, note that quality of the image (mse magnitude) only reflects the autoencoder's ability to reconstruct images similar to the input image. We will see in part 4.3 how MSE / reconstructed image quality does not reflect the autoencoder's ability in predicting classification labels.

4.3 Model with 2 Autoencoder layers and 1 Softmax Layer

4.3.1 Different Hidden Layers

4.3.1.1 MSE



	autoenc2 = 50	autoenc2 = 100	autoenc2 = 200
autoenc1 = 100	2362.54	2537.42	3116.21
autoenc1 = 200	2156.96	1323.49	1646.84
autoenc1 = 300	2373.29	1200.83	1308.7
autoenc1 = 400	1956.1	2032.98	1613.91

Figure 39. MSE Plot for Various Hidden Neurons of 2 Autoencoders

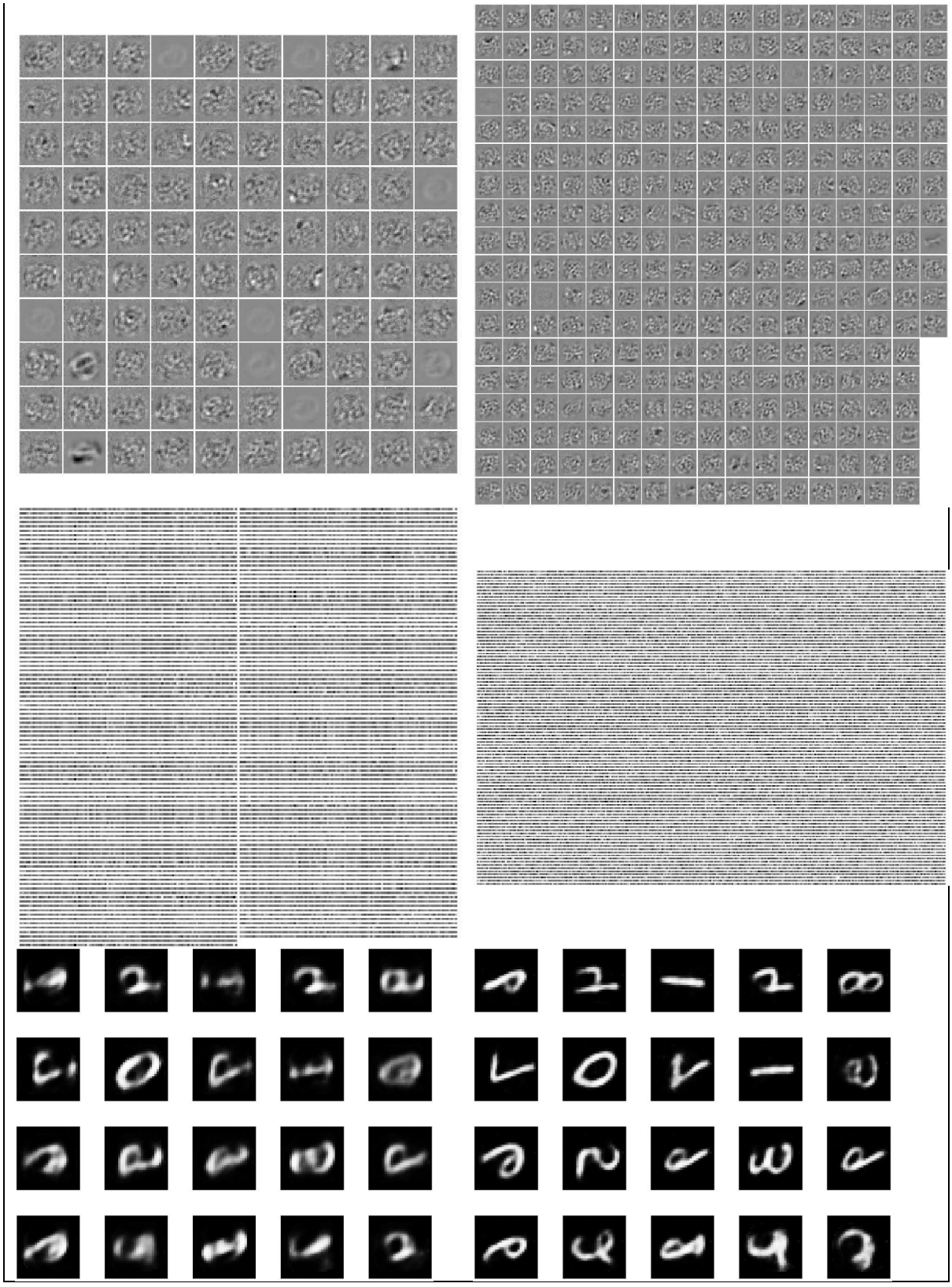
Similar to the previous experiment, because we are using 2 second Autoencoders, the image will be encoded and decoded twice, hence the MSE is quite high ranging from ~1200-3100.

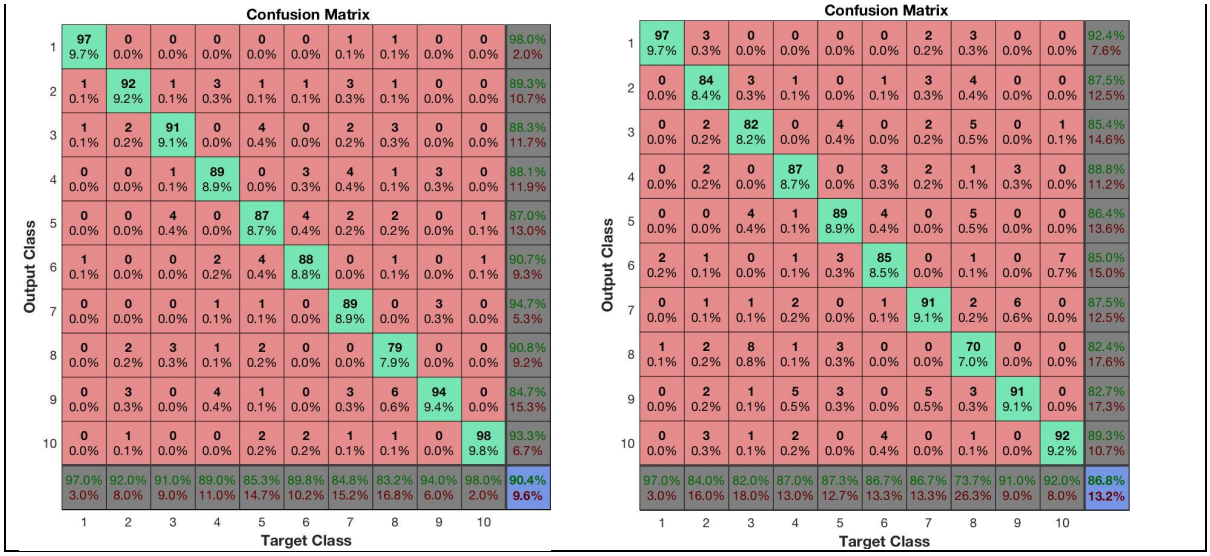
From the plot, it can be seen that there seems to be no correlation between the number of hidden neurons and the MSE (a lot of fluctuations), hence we must experiment with the number of hidden neurons manually in both autoencoders to get the best network model that best fits the input features. In this experiment we found that using 300 neurons in the first Autoencoder and 100 neurons in the second Autoencoder produce the lowest MSE.

Best Hidden Neuron Size (hiddenSize1)	**Best Hidden Neuron Size (hiddenSize2)**	**Lowest MSE**
300	100	1200.83

4.3.1.2 Reconstructed Image and Confusion Matrix

Worst MSE Error (100-200)	**Best MSE Error (300-100).**
MSE ≈ 3116.21, Accuracy = 90.4%	**MSE ≈ 1200.83, Accuracy = 86.8%**





From this comparison, it can be seen that lower MSE generates reconstructed image that is blurry and quite different from the original image. However, surprisingly, the softmax layer connected to these autoencoders are able to predict the class label with a higher accuracy (90.4%) compared to the autoencoders that has the best MSE (86.8%).

Because of this finding, we tried to plot the relationship between MSE and accuracy of the network with various number of hidden neurons.

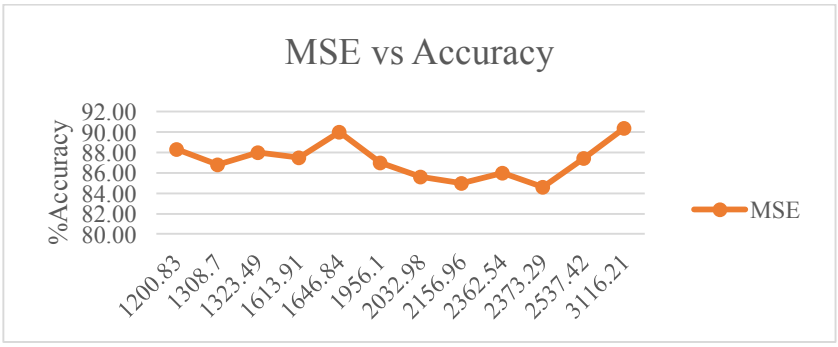
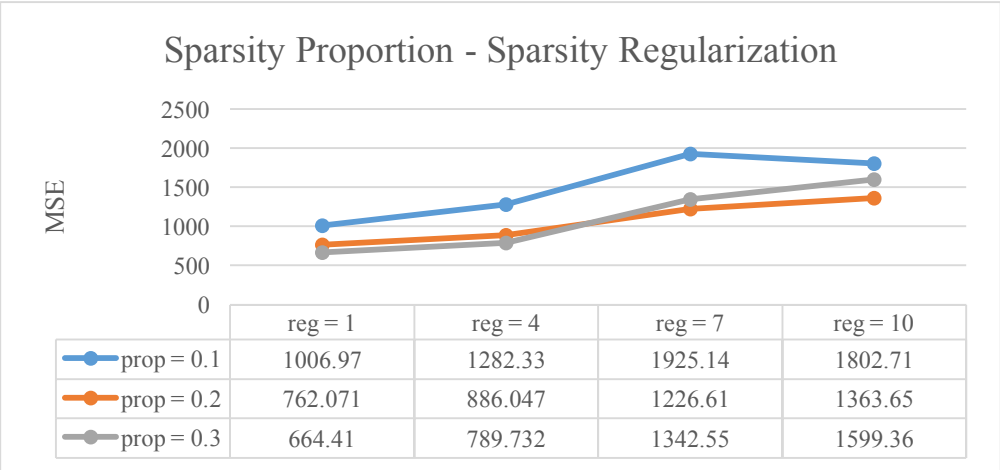


Figure 40. Plot of MSE against Accuracy for 2 Autoencoders and Softmax Layer with Different Hidden Neurons

From the plot, it is shown that there seems to be no correlation between MSE and accuracy. Therefore, the MSE of the Autoencoders cannot be used to reflect the model accuracy in predicting multiclassification labels. So again, we must experiment with the number of neurons manually in order to find the best network with the best predicting capabilities.

4.3.2 Different Sparsity Regularization and Proportion

4.3.2.1 MSE



	reg = 1	reg = 4	reg = 7	reg = 10
prop = 0.1	1006.97	1282.33	1925.14	1802.71
prop = 0.2	762.071	886.047	1226.61	1363.65
prop = 0.3	664.41	789.732	1342.55	1599.36

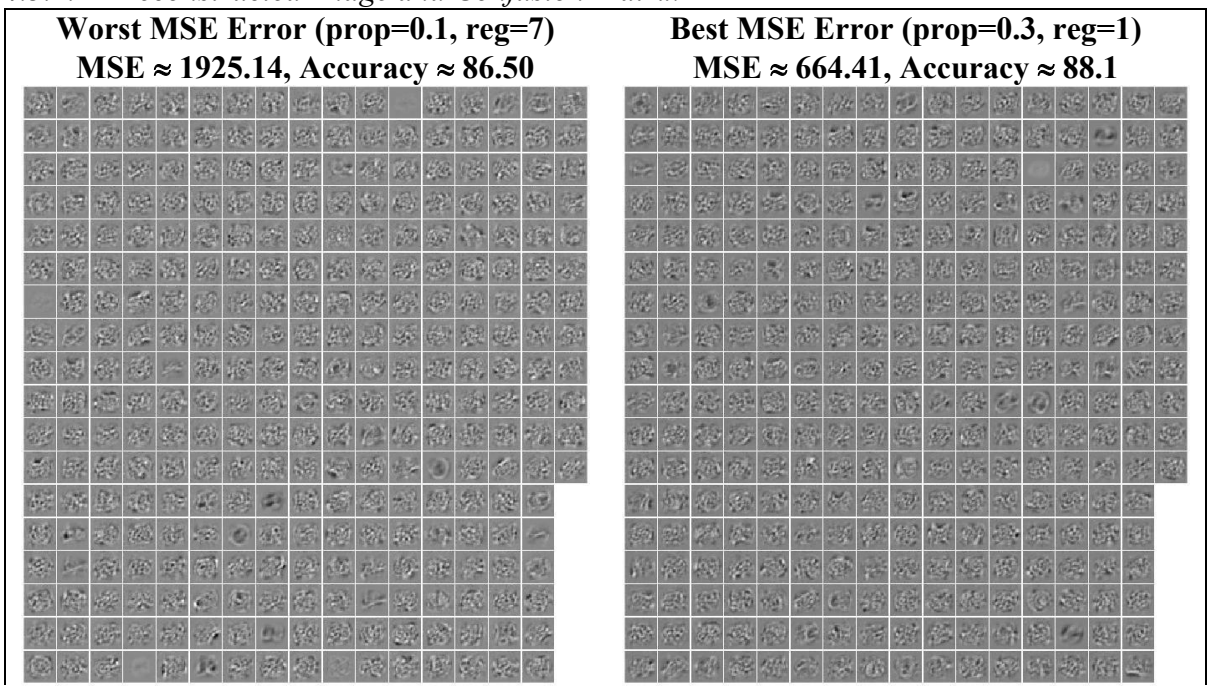
Figure 41. MSE Plot for Various Sparsity Parameters of 2 Autoencoders

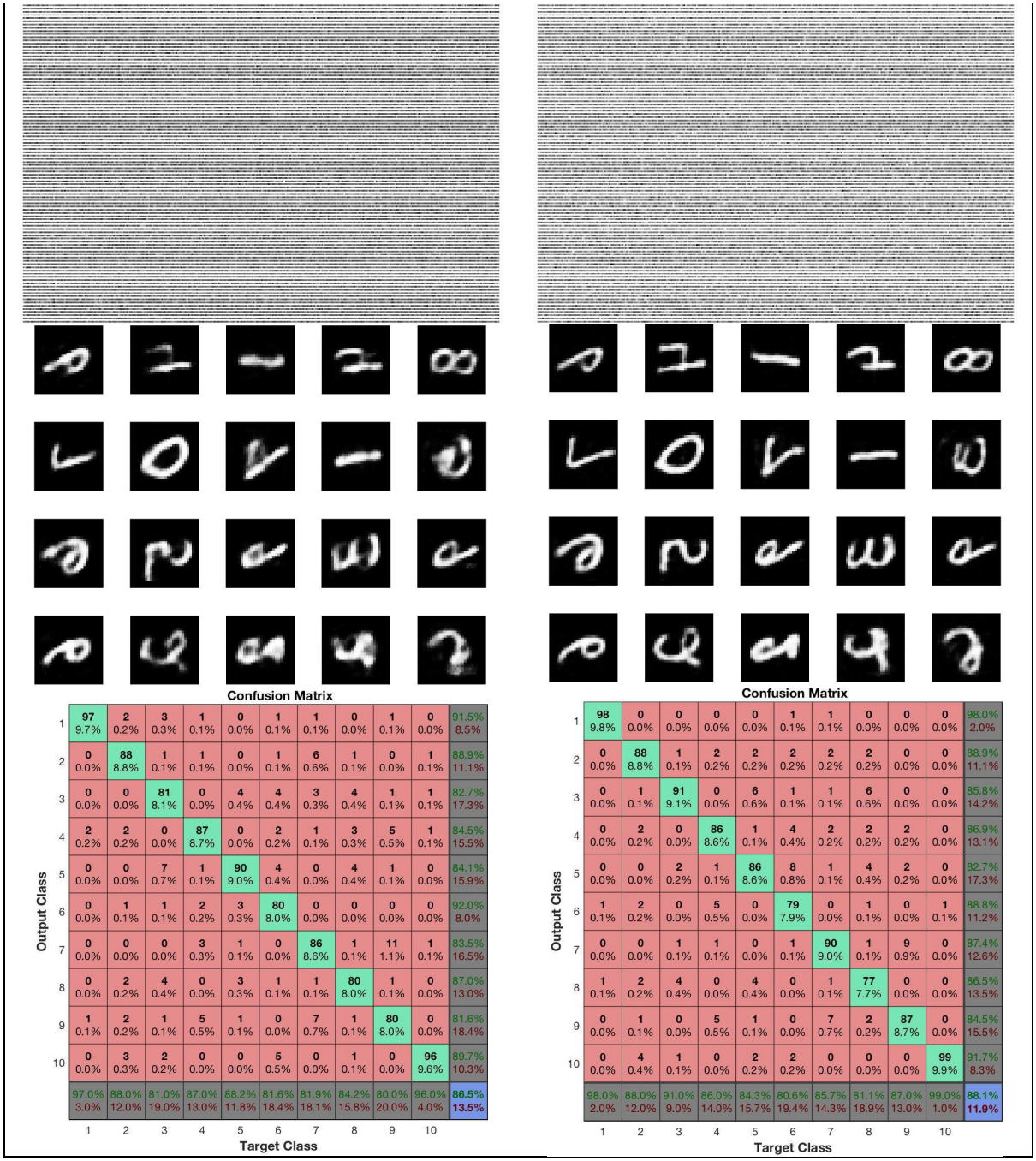
From these observations, it seems like the MSE errors generally increase when we increased the sparsity regularization. I suspect this is because increasing the sparsity of the network will reduce the number of important features learned by the Autoencoders. From the table, increasing the sparsity proportion seems like it decreases the MSE at the beginning, however we can see it is actually fluctuating when the regularization is larger than 7. These changes are quite random therefore, we have to experiment with different parameters manually to get the best network model.

Observing from the MSE results, the sparsity proportion of 0.3 and sparsity regularization of 1 generates the lowest MSE.

Best Sparsity Proportion	Best Sparsity Regularization	Lowest MSE
0.3	1	664.4104

4.3.2.2 Reconstructed Image and Confusion Matrix





Again, similar to the previous experiment, the network model with the worst MSE have a reconstructed image that is far from the original images. However, this time the accuracy of the worse MSE has a lower accuracy. This relation between MSE and accuracy can be visualized with the graph below. Similar results, because the accuracy fluctuates as the MSE value increases, we have to test each network model to find the model that generate the highest accuracy.

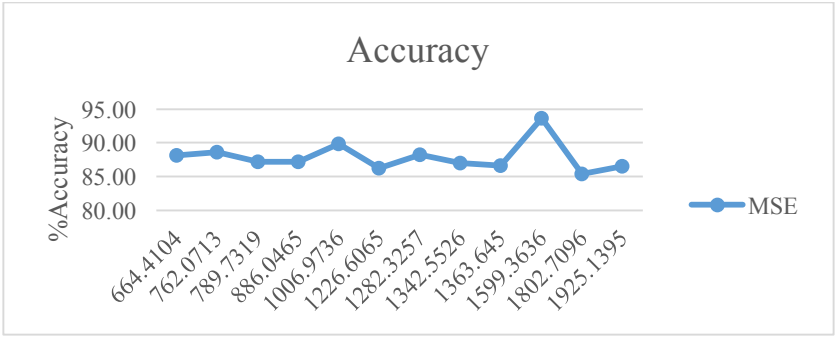


Figure 42. Plot of MSE against Accuracy for 2 Autoencoders anda Softmax Layer with Different Sparsity Parameters

The highest accuracy, 93.6% is achieved by the network model with sparsity regularization of 10 and sparsity proportion of 0.3.

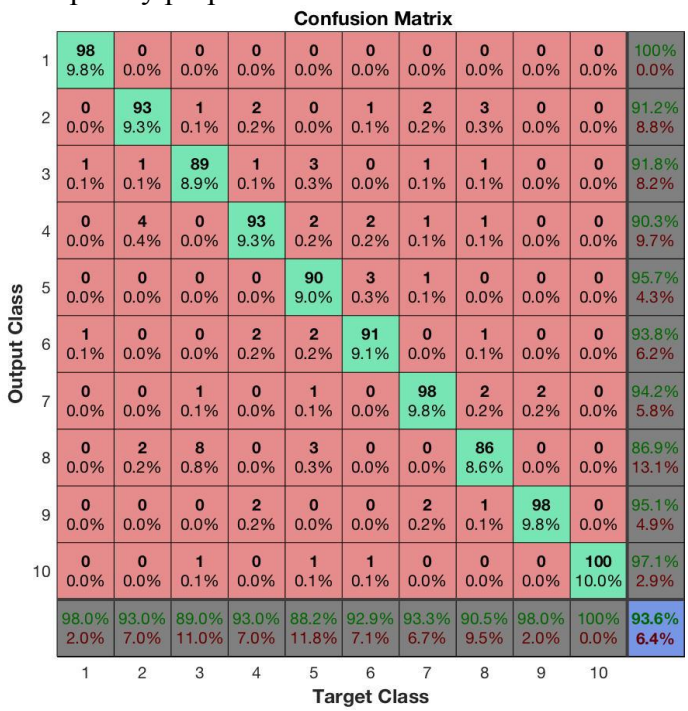
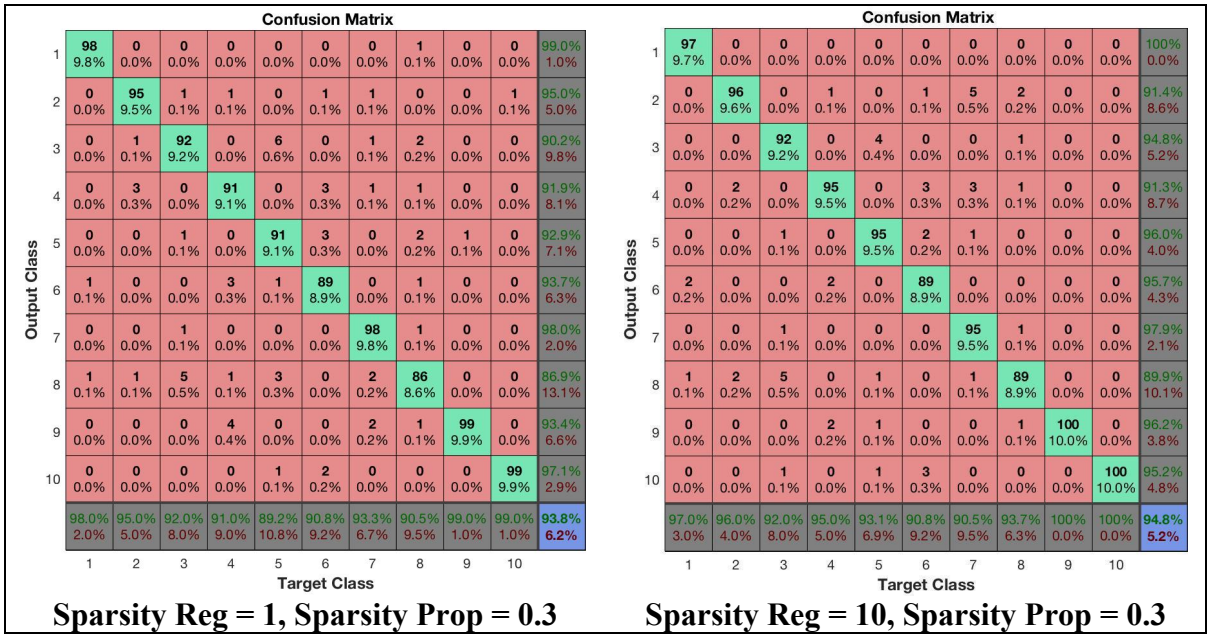


Figure 43. Best Confusion Matrix with Sparsity Regularization = 10 and Sparsity Proportion = 0.3

From the confusion matrix obtained in experiment 4.3.1 and 4.3.2, generally the network is able to distinguish images of 0 and 1, with almost 100% accuracy.

4.3.3 Stacked Network



Sparsity Reg = 1, Sparsity Prop = 0.3 **Sparsity Reg = 10, Sparsity Prop = 0.3**

We found that deep neural network using stacked autoencoders yields a better results compared to simply connecting softmax layer. The accuracy that we receive is always greater than 90%. We believe this is because with the deep neural network, the whole network will be trained together hence it has a better parameter tuning compared to performing training on each layer separately with the previous experiments.

5 Conclusion

We must experiment with a lot of hyper parameters to generate the best results, such as the sparsity constraint parameters, sparsity proportion and sparsity regularization; number of hidden neurons; and transfer functions. All in all, increasing the epochs will allow the autoencoders to learn the features better. Choosing the appropriate transfer functions are also important in order for the autoencoder to learn the features that best represent the input images. From our experiment, we get that logsig-logsig and satlin-satlin transfer functions in the encoder and decoder layer outperforms the combination of other transfer functions. For classification problem, using deepnet is recommended because it has a better performance than training each layer separately.

DISCUSSIONS AND CHALLENGES

Ideally, we should try all combinations of all the parameters while trying to determine the best model to use for training on the training data. However, due to the high computational power required for such a task and the time constraints for this project, we first experimented with individual or sets of two parameters while keeping the other parameters constant, and tried to determine the best parameter values from each experiment. We then again experimented with these parameters in the same way, but we set the constant parameters to the values obtained from the first time. The plots shown in this report are the ones we obtained on the second time, using which we again try to find the optimum values of the parameters. Although we might not be able to get the best model, we hope that this process would help us to get closer to the optimum model. It is much more feasible to try all possible models with all possible combinations of parameters in a distributed environment, where each model can be run separately on different machines, which will help to reduce the time taken to train all the models.