NANYANG TECHNOLOGICAL UNIVERSITY

CZ4042: Neural Networks

Project 2

Deep Convolutional Neural Networks and Autoencoders

Submitted by:

Khare Simran (U1423025D)
Marathe Ajinkya Avinash (U1522716K)

Contents

Part A: Deep Convolutional Neural Network

Introduction

Convolutional Neural Networks are deep, feed forward artificial neural networks used for analyzing visual imagery. The aim of this part of the assignment is to design a convolutional neural network to recognize the handwritten digits in the MNIST database. We have used python along with the theano library for this project. The MNIST database contains handwritten digit 28*28 pixel images for this assignment, we have used the first 12000 records for training and the first 2000 records for testing. We use different algorithms to train the neural networks and evaluate their performance.

Task 1: Design a two-layer convolutional neural network

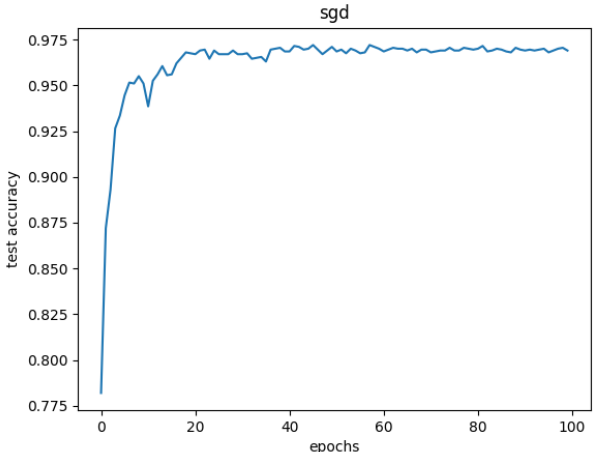
Specifications:

• An Input layer of 28x28 dimensions
• A convolution layer $C1$ of 15 feature maps and filters of window size 9x9. A max pooling layer $S1$with a pooling window of size 2x2.
• A convolution layer $C2$ of 20 feature maps and filters of window size 5x5. A max pooling layer $S2$with a pooling window of size 2x2.
• A fully connected layer $F3$ of size 100.
• A softmax layer $F4$ of size 10.

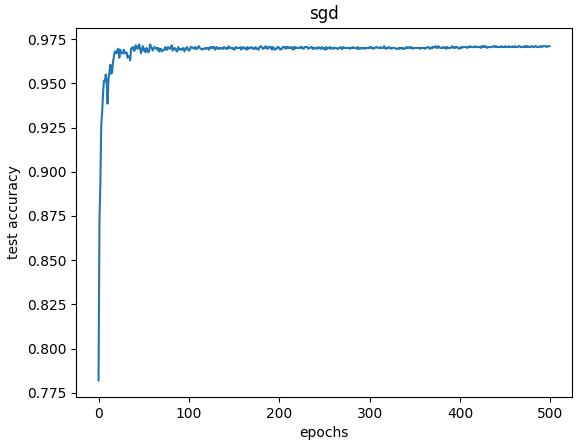
We trained the network using ReLu activation functions for neurons and mini batch gradient descent learning. We used the following parameters-batch size 128, learning rate α = 0.05 and decay parameter β = 10^−4.

Test Accuracy against the number of epochs:

The neural network was trained for 100 and 500 iterations. No significant change in the accuracy was observed, therefore the rest of the neural networks will be trained for 100 iterations.

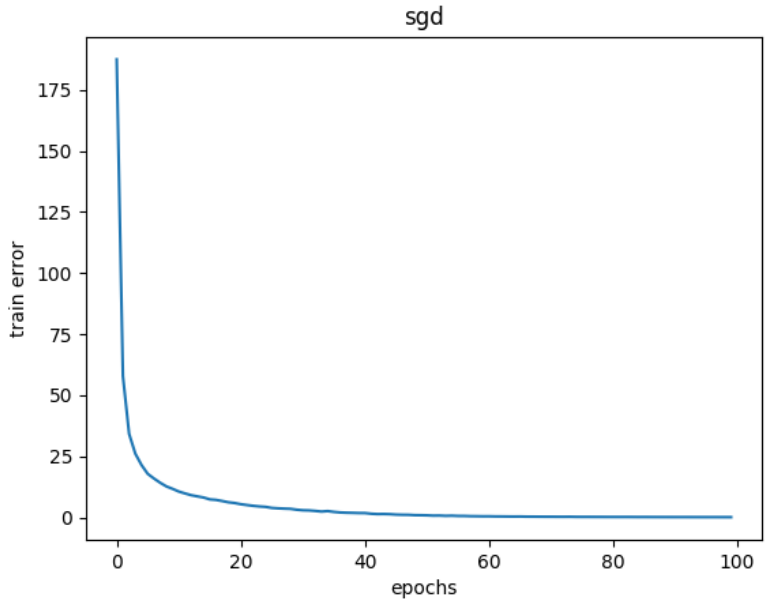


Against 100 iterations



Against 500 iterations

Training error against the number of epochs:



As we can see from the graphs, as the convolutional neural network was trained, the training cost decreased while the test accuracy increased against the number of iterations.

Feature Maps at the convolution and pooling layer:

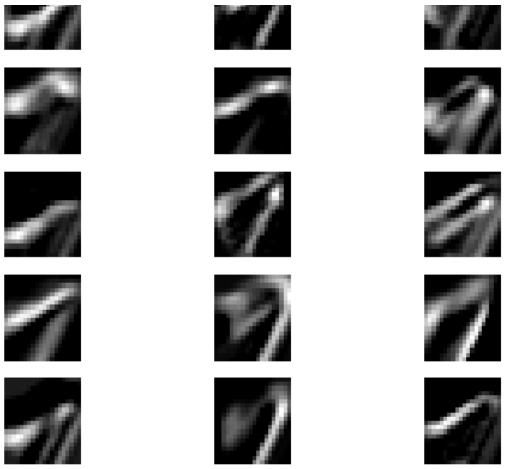
At the convolutional layer, each convolutional neuron processes data for its receptive fields. At the convolution layer feature maps are applied to the images and feature maps are produced. The feature map is the output of one filter applied to the previous layer. Each position results in an activation of a neuron and output is collected in the feature map. The pooling layer helps to reduce the dimensions of the feature maps.
The images given below show the 15 and 20 feature maps for the convolution C1 and convolution C2. It also shows the feature maps at Pooling layer P1 and pooling layer P2.

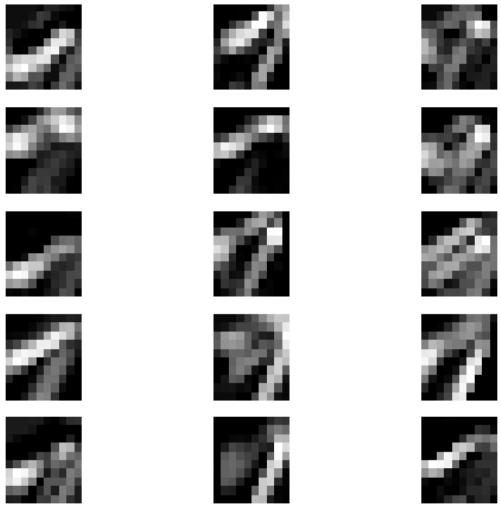
Feature Maps (Sample 2 in the Appendix A)

Sample 1:

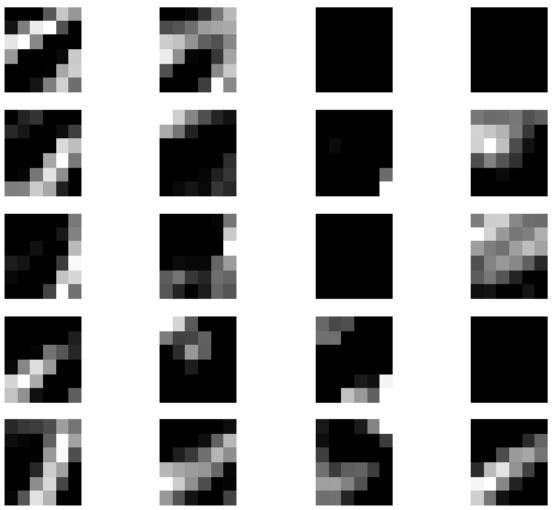
Input Image:

Convolution C1

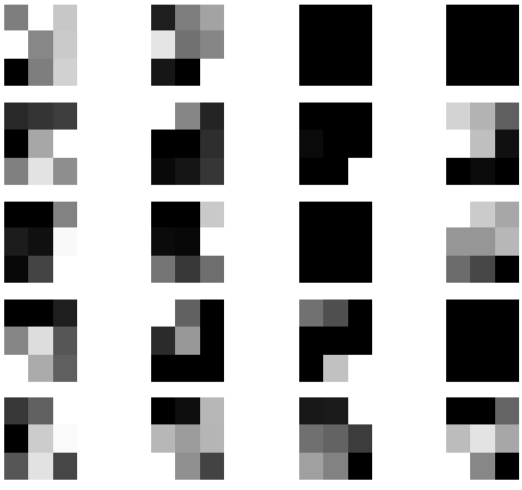




Pooling P1



Convolution C2



Pooling P2

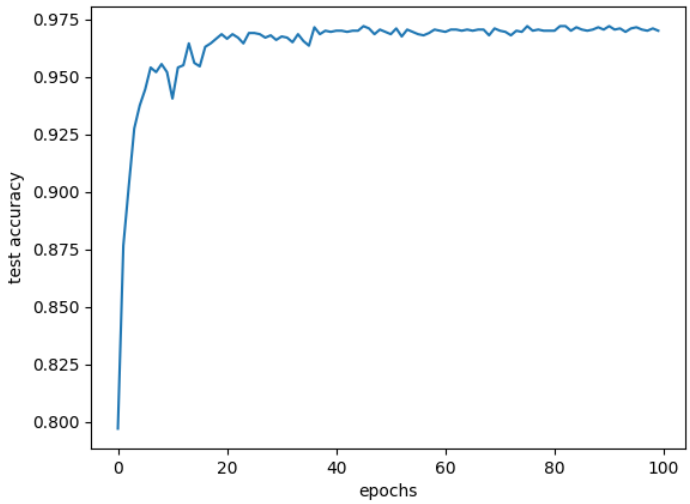
Task 2: Repeat task 1 with momentum term

Repeat part 1 by adding the momentum term to mini batch gradient descent learning with momentum parameter $\gamma = 0.1$

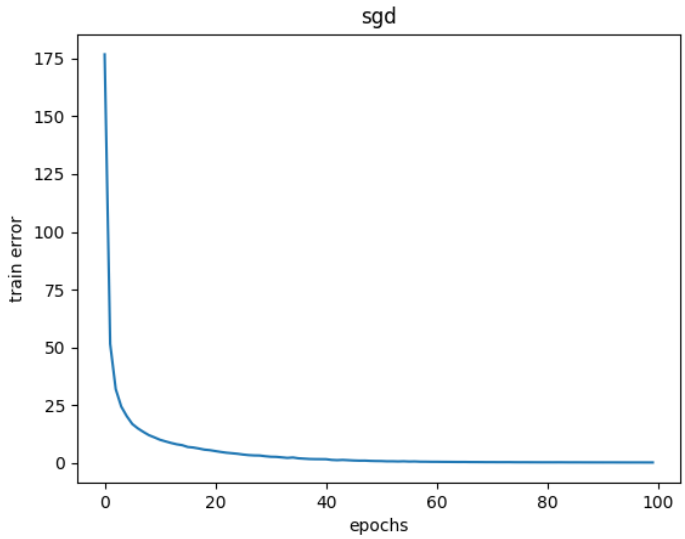
Stochastic Gradient with Momentum:

The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradient or noisy gradient. The momentum algorithm accumulates an exponentially decaying moving average of past gradient and continues to move in their direction.

Test Accuracy against the number of epochs:



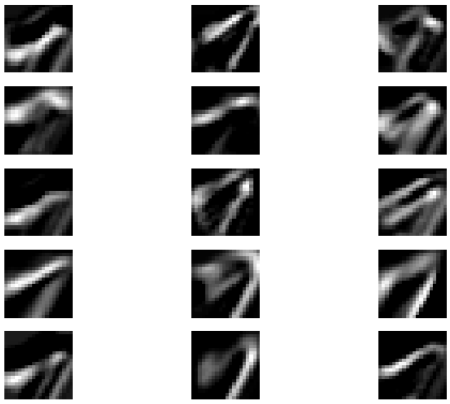
Training error against the number of epochs:



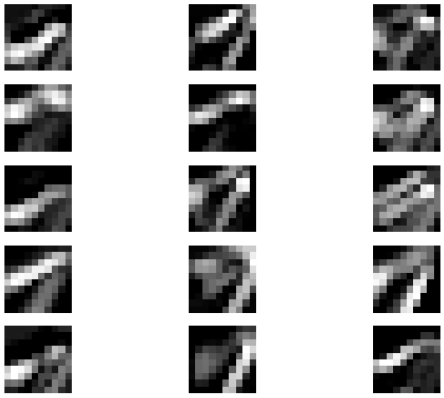
Feature Maps (Sample 2 in the Appendix A)

Sample 1:

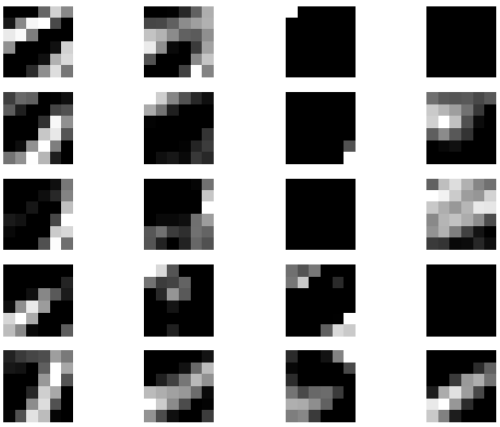
Input Image:



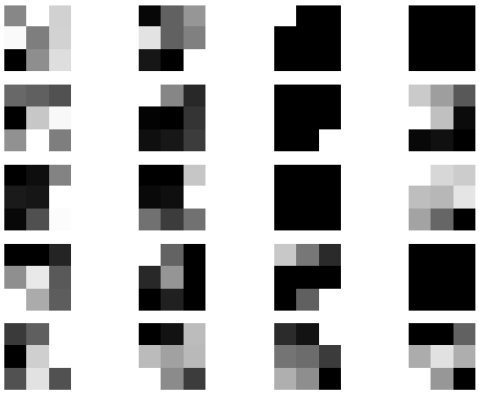
Convolution C1



Pooling P1



Convolution C2



Pooling P2

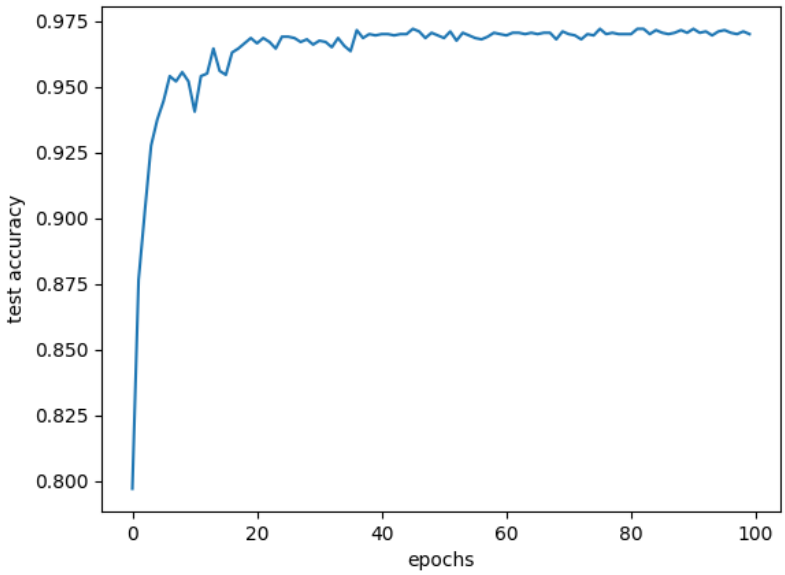
Task 3: Repeat task 1 with RMSProp Algorithm

Repeat part 1 by using RMSProp algorithm for learning. Use α = 0.001, β = 1e −4, ρ = 0.9, and ϵ = 10−6 for RMSProp.

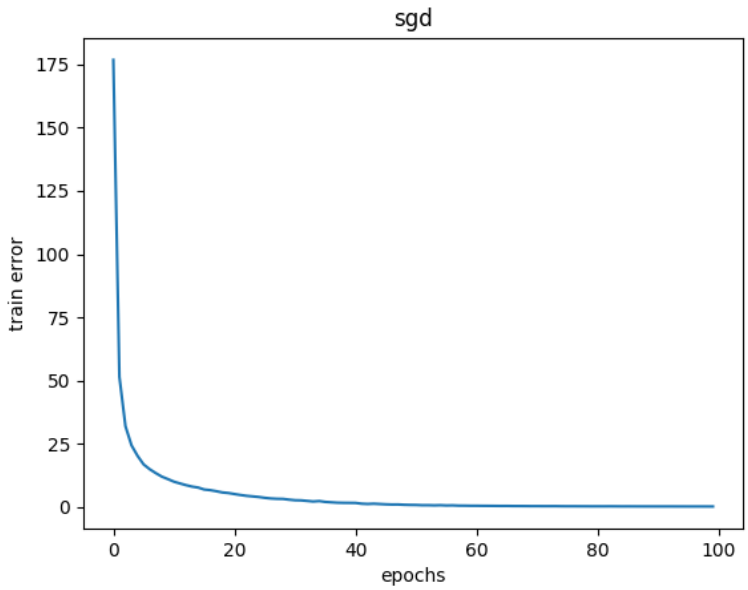
RMSProp Algorithm:

Adaptive learning rates with annealing usually works with convex cost functions. RMSProp uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region

Test Accuracy against the number of epochs:



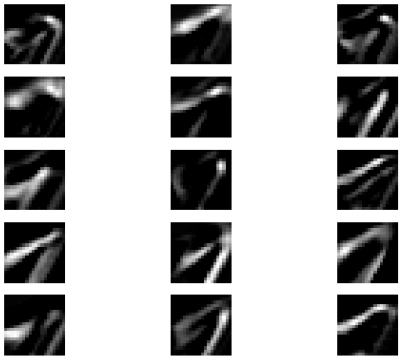
Training error against the number of epochs:



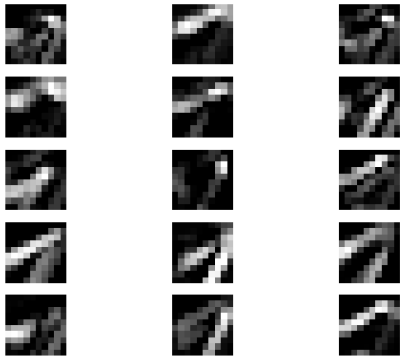
Feature Maps (Sample 2 in the Appendix A)

Sample 1:

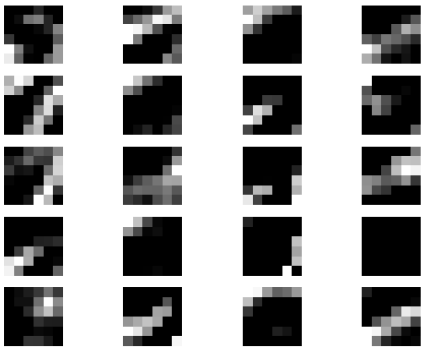
Input Image:



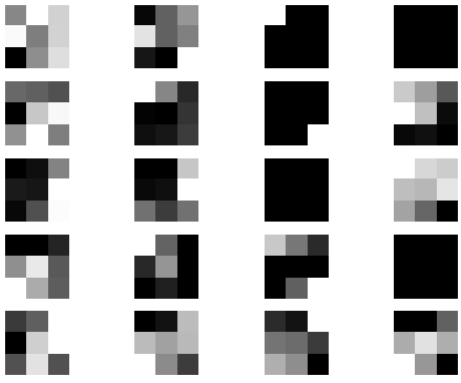
Convolution C1



Pooling P1



Convolution C2

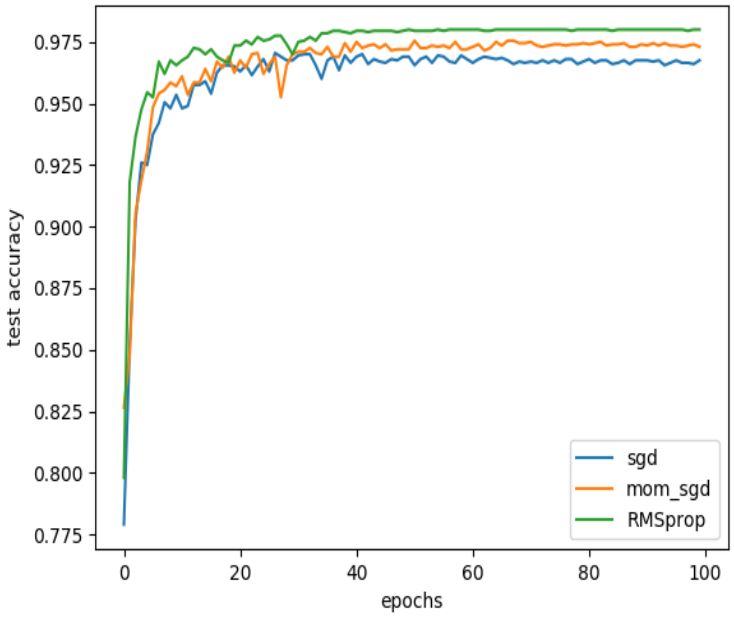


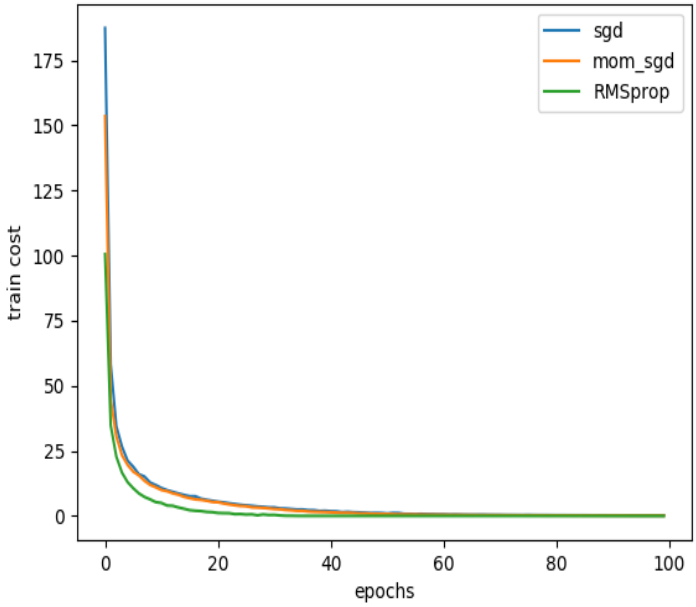
Pooling P2

Comparison of Algorithms

As we can see in the diagrams below, that CNN with the RMSProp algorithm converges faster than the normal stochastic gradient algorithm and stochastic gradient with momentum.
The training errors of RMSProp also seem to decrease at a higher rate than the other two.
I think that RMSProp gives the best performance.

Test Accuracy and Training Error against epochs:





Accuracy Comparison:

SGD- 0.969
SGD with momentum – 0.970
RMSProp – 0.978

Part B: Autoencoders

Introduction

The objective of the second part of the project was to build a stacked denoising autoencoder. The dataset we used was MNIST, which is a large database of handwritten digits. Given noisy or corrupted data points as input, our autoencoder could reconstruct the original uncorrupted data points as output, which is an essential feature of a ***denoising*** autoencoder. Moreover, the autoencoder we built was a ***deep stacked*** autoencoder with the output layer being a ***softmax layer*** that can be used for ***classification*** of MNIST images. Finally, a sparsity constraint was added to make the weights learnt by the autoencoder to be sparse. Furthermore, the momentum term for gradient descent learning as well as a penalty parameter were also introduced, and an analysis of the results was done.

Task 1: Designing a Stacked Denoising Autoencoder

The first task required us to design a stacked denoising autoencoder comprising three hidden layers with 900, 625 and 400 neurons in the first, second and third hidden layers respectively. Since the MNIST dataset consists of images of 28x28 pixels each, we had 784 dimensional inputs. An autoencoder attempts to copy its input to its output, hence the output was 784 dimensional as well.

The ***entire MNIST dataset was used.*** First, we corrupted the input samples using multiplicative noise, with the corruption level being 0.1. We trained the network for ***100 epochs*** and plotted the learning curves, weights learned as well as the activations for each hidden layer. Finally, we plotted the images reconstructed by the network.

1.1. Plotting the Learning Curve for each layer

We plotted the learning curves, i.e. the Cross Entropy versus Iterations graphs for the three hidden layers. Our observation was that after 100 epochs, the cross entropy was the lowest (approximately 55) for the first hidden layer with 900 neurons and highest (approximately 400) for the second hidden layer. This could be because the first layer with 900 neurons has a higher dimension than the input layer, hence we are effectively obtaining a higher dimensional representation of the input signals and passing that to the second hidden layer. Below are the graphs.

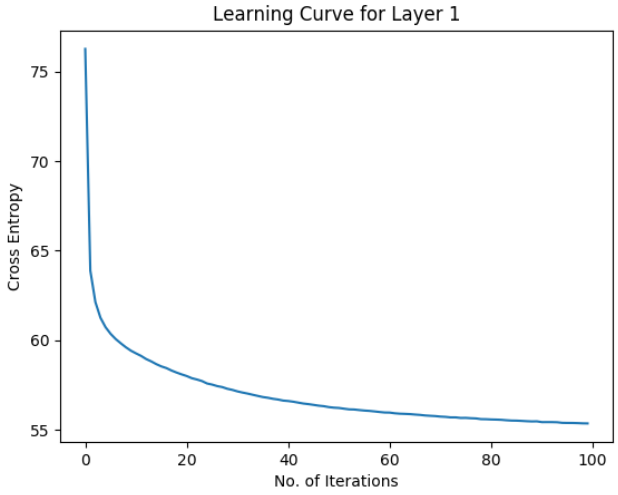


Figure 1: Learning Curve for Hidden Layer 1

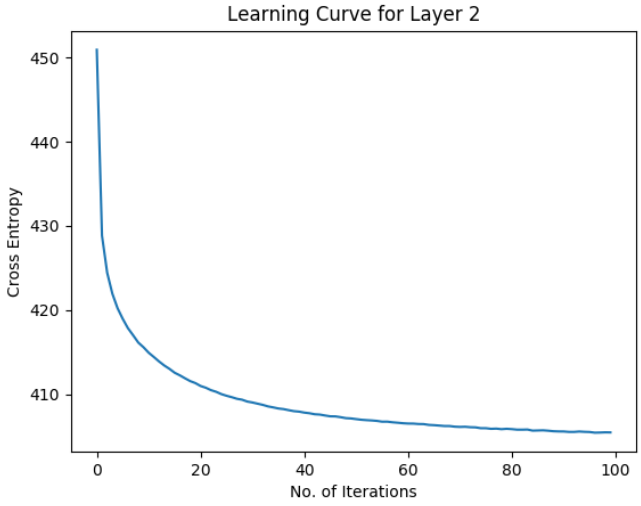


Figure 2: Learning Curve for Hidden Layer 2

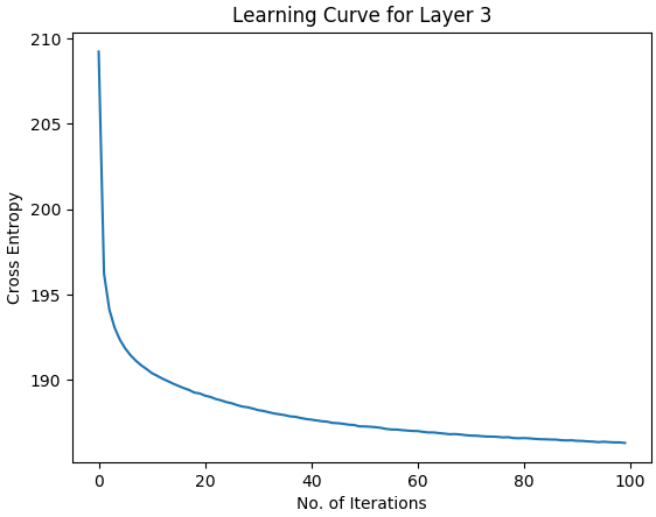


Figure 3: Learning Curve for Hidden Layer 3

1.2. Plotting the Weights learned at each layer

Below are 100 samples each of the weights learned by the three hidden layers. The dimensions of the weights for the first, second and third hidden layer are (784x900), (900x625) and (625x400) respectively. We notice that in general, the weights learned by the third hidden layer are relatively more regular and smaller, since there are more black pixels.

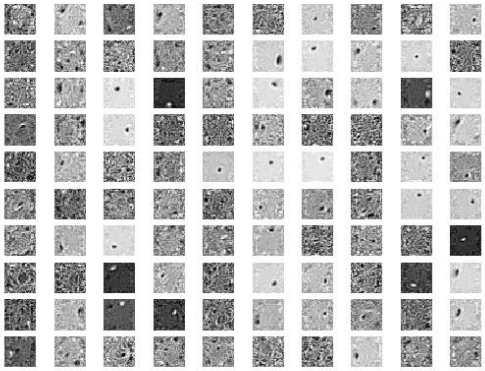


Figure 4: Weights learned by hidden layer 1

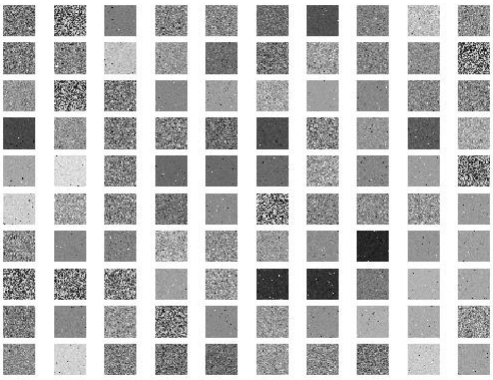


Figure 5: Weights learned by hidden layer 2

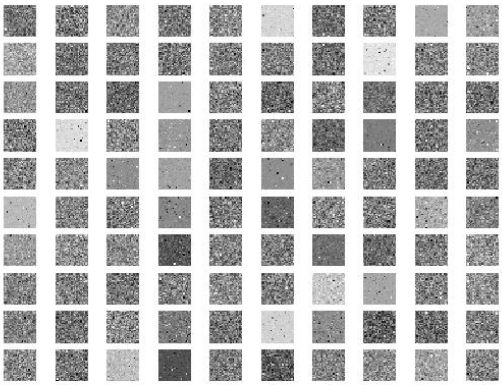


Figure 6: Weights learned by hidden layer 3

1.3. Plotting the Hidden Layer Activations

Next, we plotted the activations for each of the hidden layers. It can clearly be seen that the activations for the second hidden layer with 625 neurons are the most sparse, followed by the first hidden layer and then the third hidden layer. This observation is based on the fact that for the second hidden layer, the activation images plotted contain a lot of black pixels, which represent a lower intensity value.

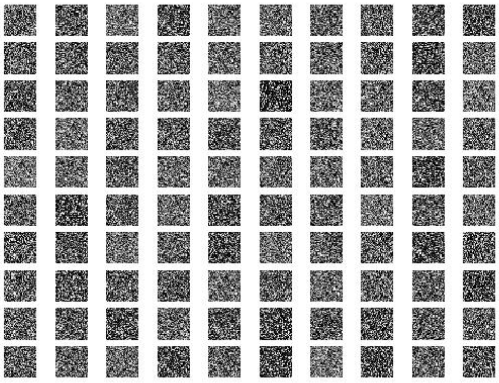


Figure 7: Hidden layer 1 activations

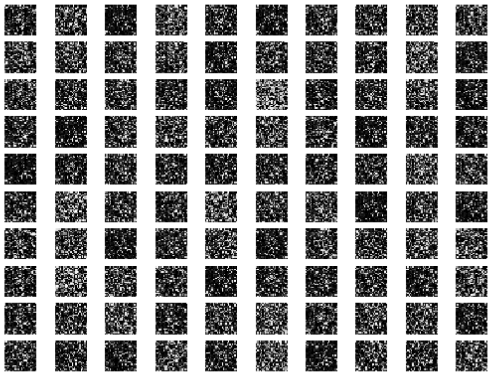


Figure 8: Hidden layer 2 activations

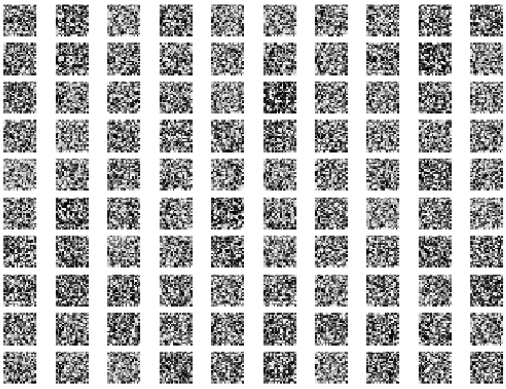


Figure 9: Hidden layer 3 activations

1.4. Plotting the Images Reconstructed by the Network and contrasting them with the Input Images

On performing testing, we conclude that our autoencoder is successful in reconstructing the original uncorrupted inputs from the given corrupted input data points. Below are the plots of 100 input samples and the corresponding output of the autoencoder.

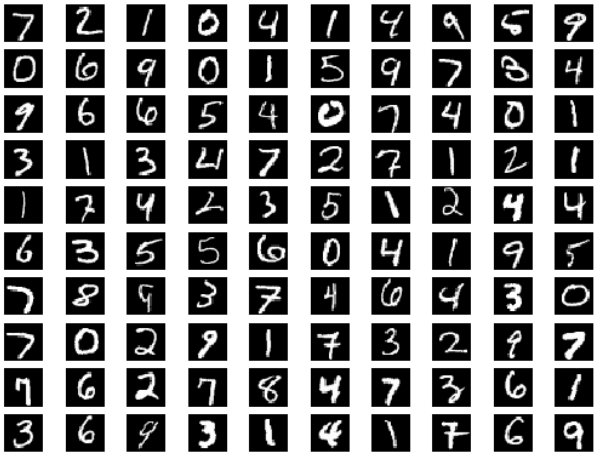


Figure 10: 100 Corrupted Input Samples

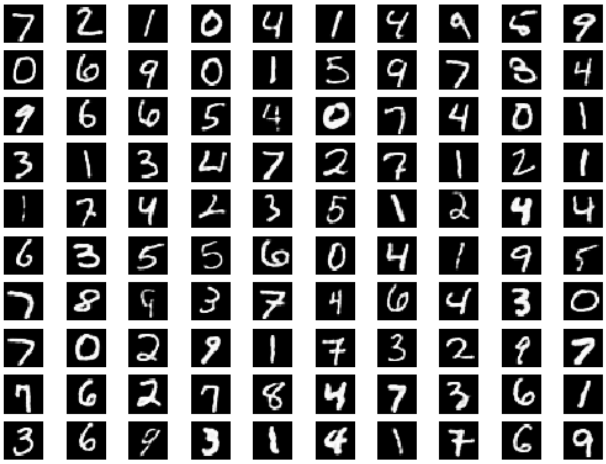


Figure 11: Reconstructed images produced by our autoencoder

Task 2: Training a five-layer feedforward network to recognize MNIST data

The second task required us to extend our autoencoder to build a five-layer feedforward neural network. To do this, we added a softmax layer as the output layer which would classify the inputs as one of the digits 0 to 9. Hence, we have 10 output neurons.

Below are the Categorical Cross Entropy versus No. of Iterations and Test Accuracy versus No. of Iterations plots respectively. We observe that the cross entropy reaches a very small value close to 0 at about 100 epochs, and a highest test accuracy of approximately 98% can be achieved.

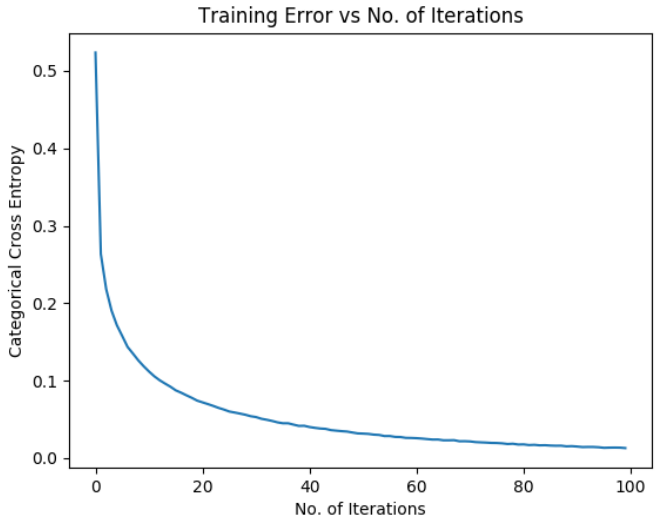


Figure 12: Categorical Cross Entropy versus Iterations graph for the Classifier

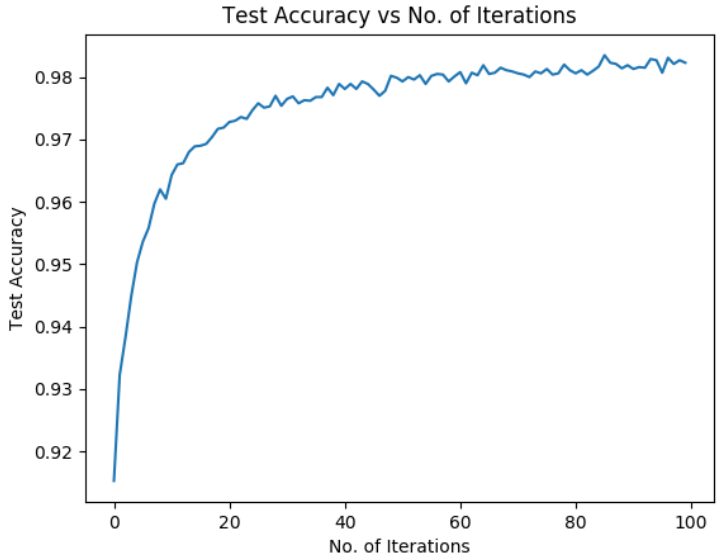


Figure 13: Test Accuracy versus Iterations graph for the Classifier

Task 3: Repeating Tasks 1 and 2 with the introduction of the Momentum Parameter, Penalty Parameter and Sparsity Parameter

For the final task, we introduced the momentum parameter (0.1), sparsity parameter (0.05) and the penalty parameter (0.5). Our objective here is to build a regularized and sparse autoencoder, that can always learn something useful about the data distribution, and has more properties besides the ability to copy its input to its output.

We plotted the learning curves for the hidden layers 1, 2 and 3 again and got the following results. We can observe that in contrast to the initial autoencoder, in which we did not use the aforementioned parameters, the learning curves for this autoencoder are more or less similar. The cross entropy values for the second hidden layer in the initial autoencoder were extremely high and only reduced to about 400 at 100 epochs, but in this case it reaches around 120, suggesting that are autoencoder is now more regularized.

3.1 Plotting the Learning Curve for each layer

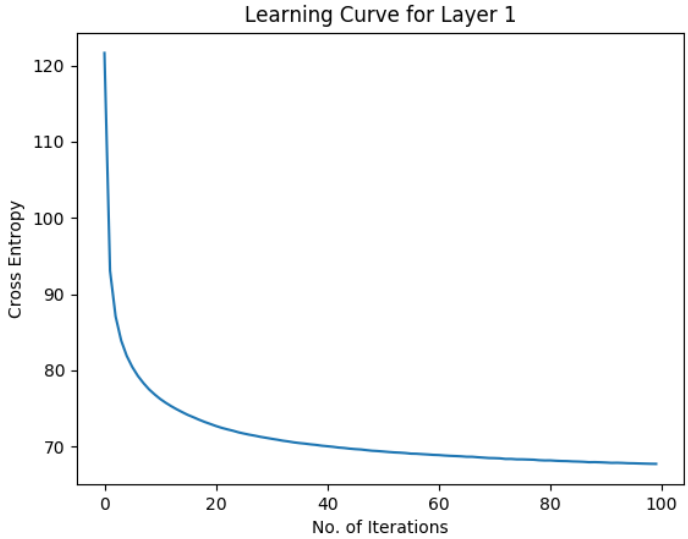


Figure 14: Learning Curve for Hidden Layer 1

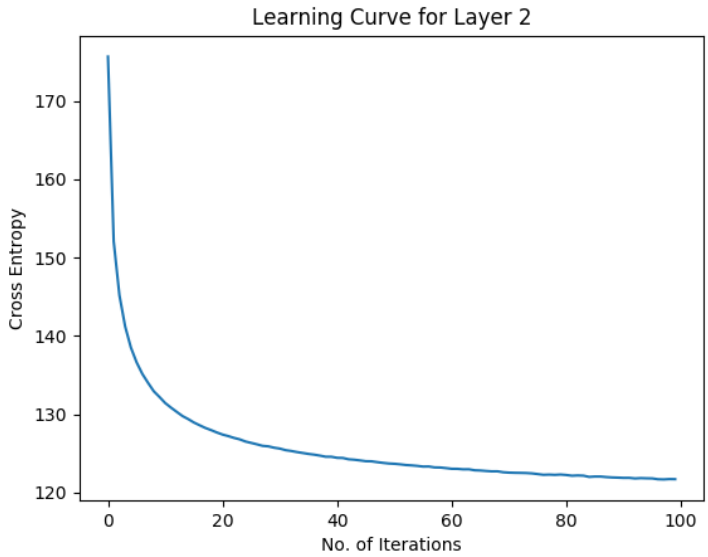


Figure 15: Learning Curve for Hidden Layer 2

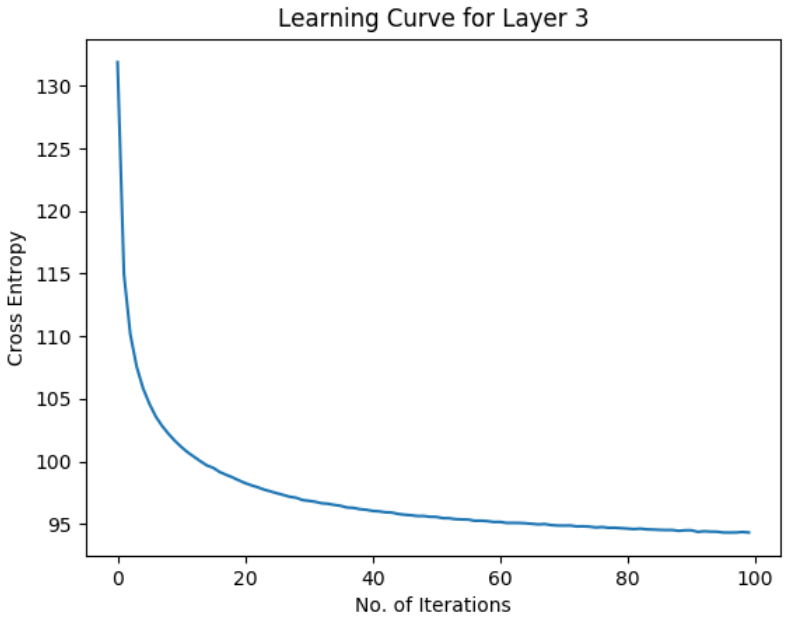


Figure 16: Learning Curve for Hidden Layer 3

3.2 Plotting the Weights learnt by each hidden layer

Below are the plots of the weights learnt by each hidden layer. We do not observe a very significant difference from the initial autoencoder, but we did notice that after the introduction of the given parameters, the time taken for training the network was reduced significantly.

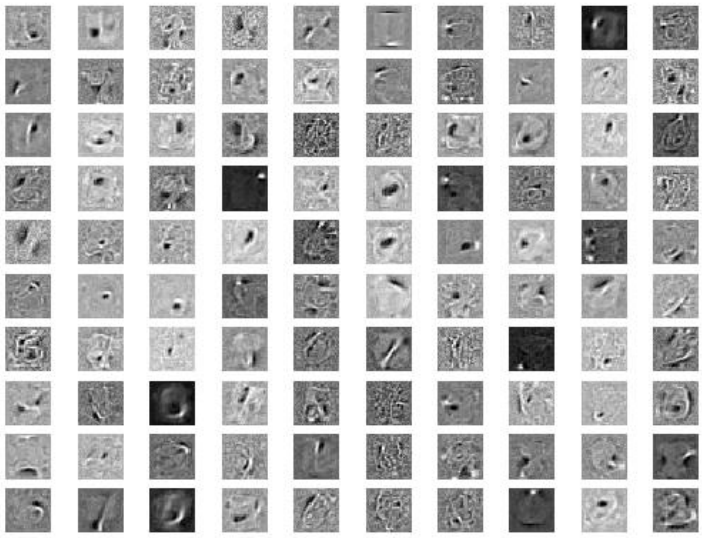


Figure 17: Weights learned by hidden layer 1

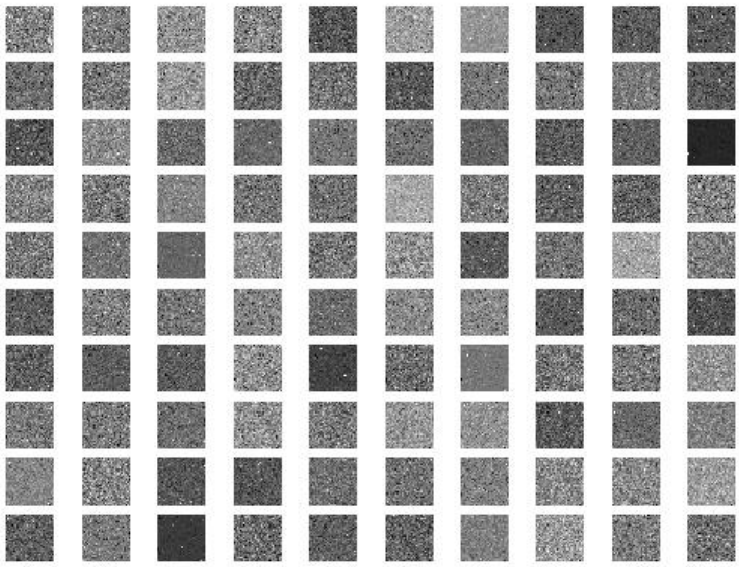


Figure 18: Weights learned by hidden layer 2

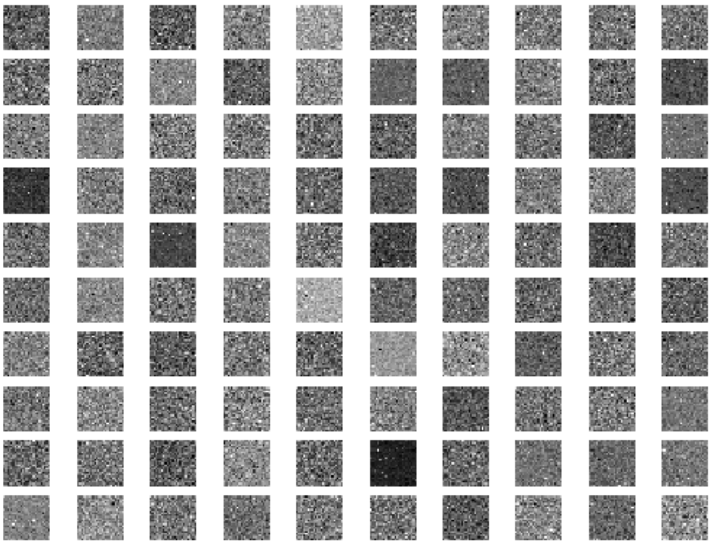


Figure 19: Weights learned by hidden layer 3

3.3 Plotting the activations for each hidden layer

Below are the plots of the activations of each hidden layer. As expected, we observe that the activations are much more sparse after the introduction of the sparsity constraint. Hence, we have ensured that the hidden layer neurons are inactive most of the time, and their activations are maintained at 0.05 on average.

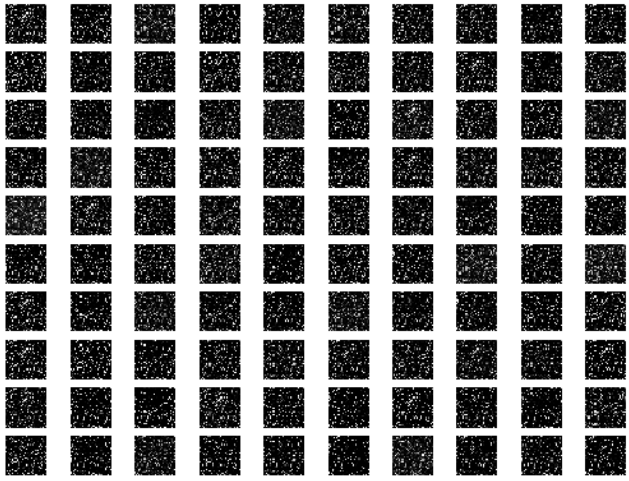


Figure 20: Hidden layer 1 activations

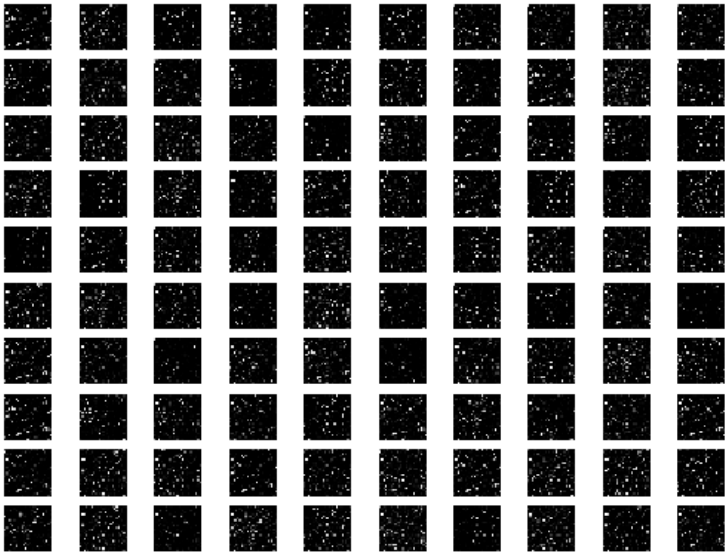


Figure 21: Hidden layer 2 activations

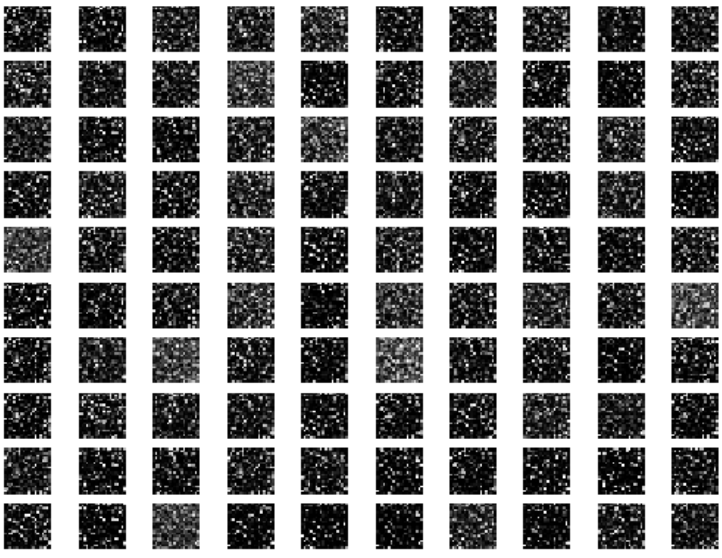


Figure 22: Hidden layer 3 activations

3.4 Plotting the images reconstructed by the network

The autoencoder is again successful in reconstructing the uncorrupted inputs from their corrupted versions.

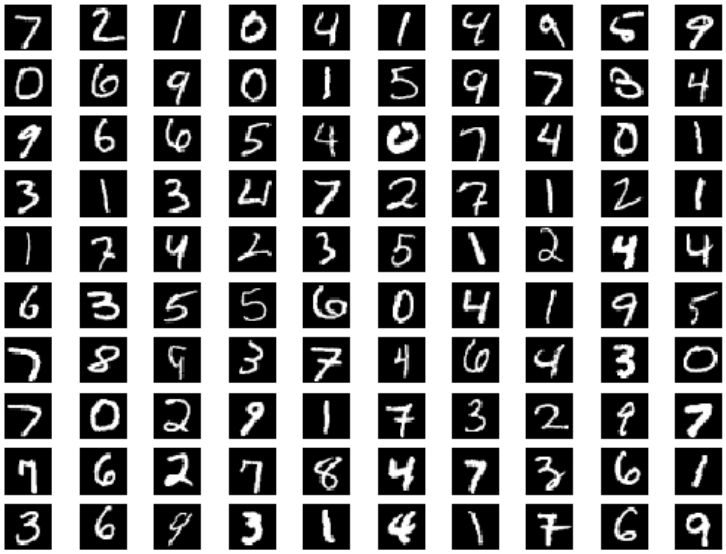


Figure 23: 100 Corrupted Input Samples

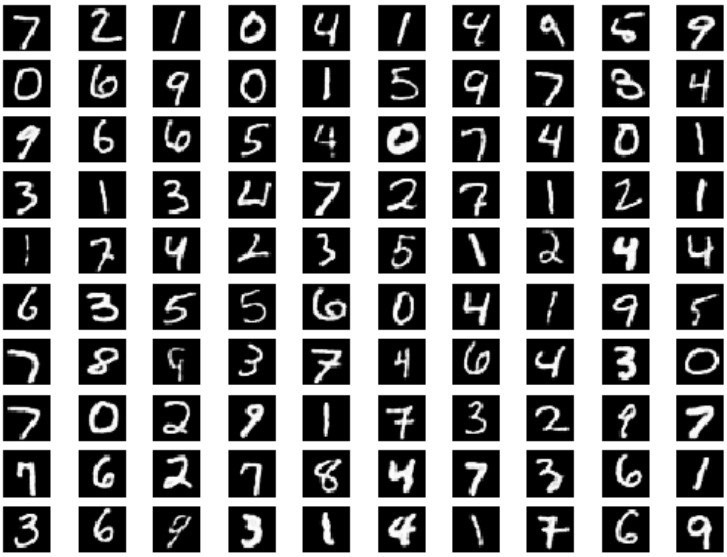


Figure 24: Reconstructed images produced by our autoencoder

3.5 Plotting the training error as well as test accuracy

Below are the plots for the Categorical Cross Entropy versus the Iterations as well as the Test Accuracy versus Iterations. We observe that even though the Cross Entropy is generally higher than our initial autoencoder built in task 1, we still get a reasonably low value for the cross entropy at 100 epochs.

Moreover, it can be observed that the test accuracy curve for our new regularized autoencoder is much smoother than before, however a relatively smaller test accuracy of 92% is achieved.

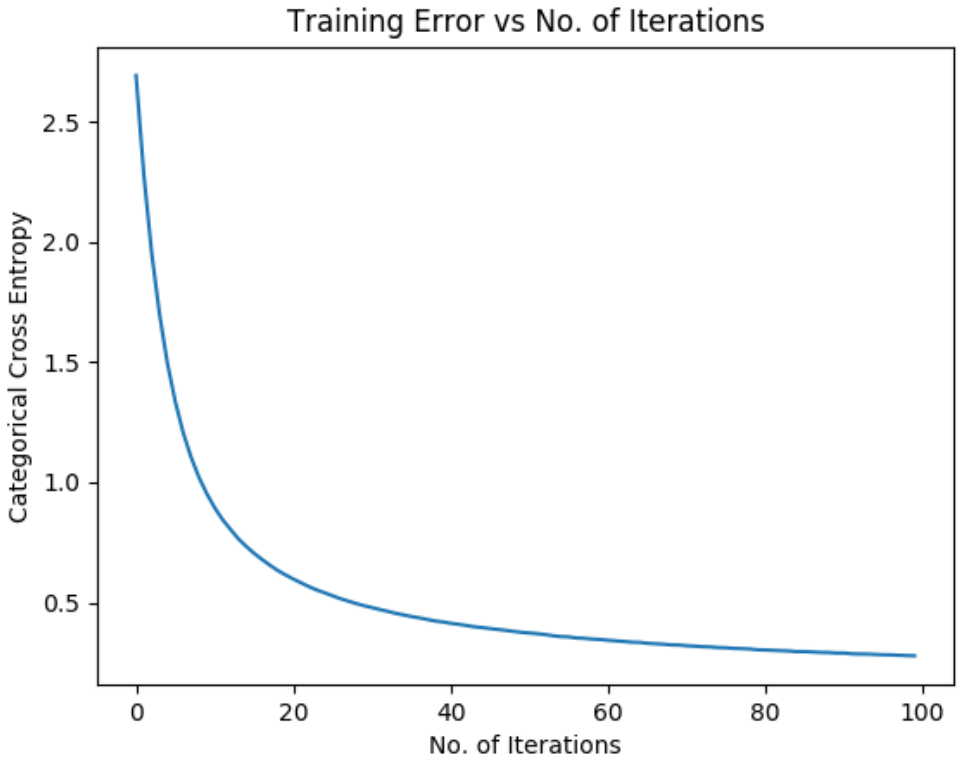


Figure 25: Categorical Cross Entropy versus Iterations graph for the Classifier

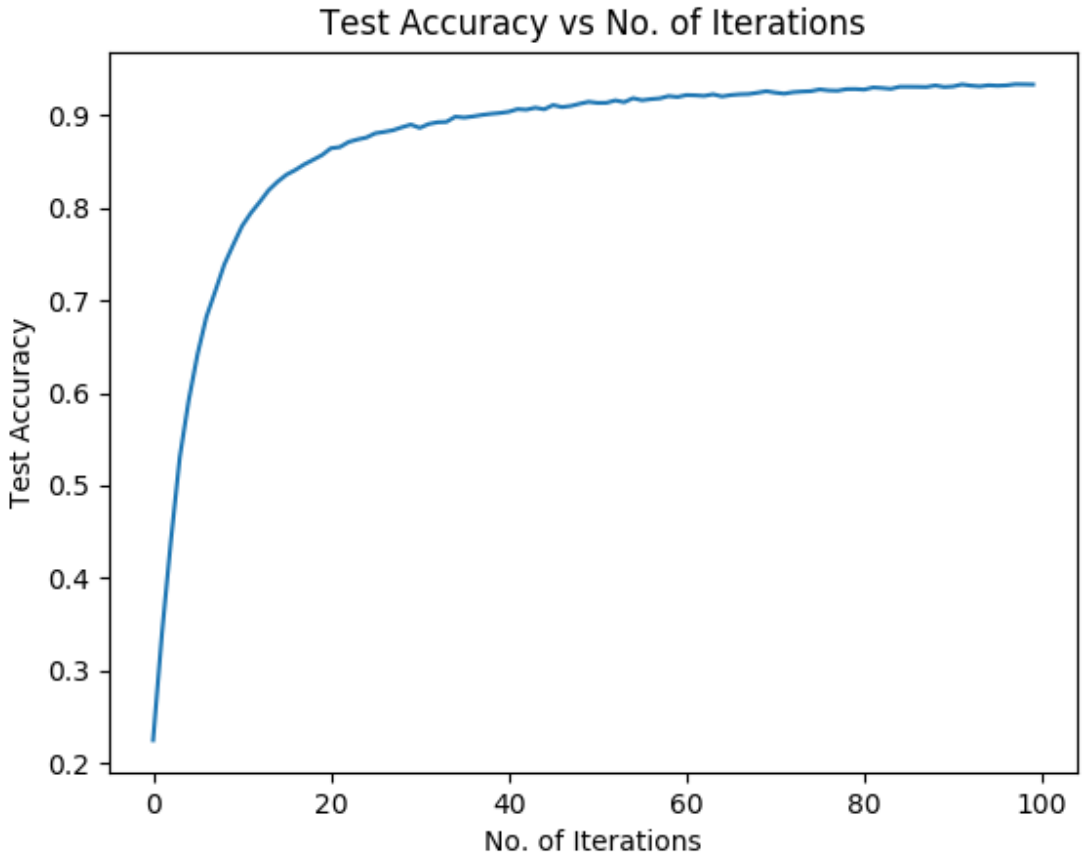


Figure 26: Test Accuracy versus Iterations graph for the Classifier

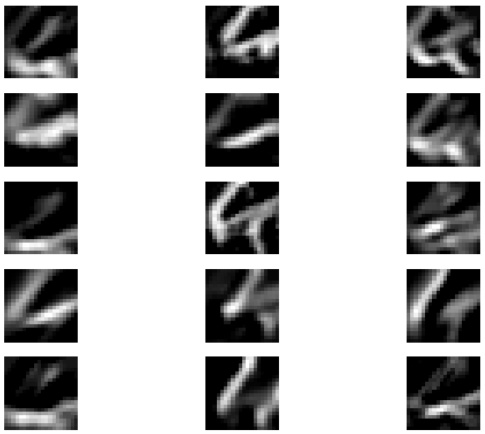
Conclusion

We conclude that even though a non-regularized autoencoder might sometimes give us better results, a regularized spare autoencoder can learn much more properties from the input data and provide more regularized and uniform results. This is because rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.

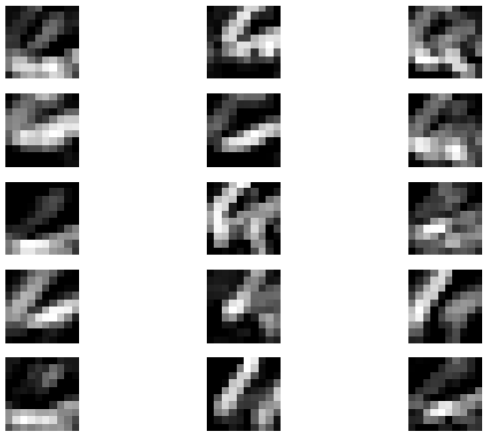
Appendix A:

Task 1: SGD

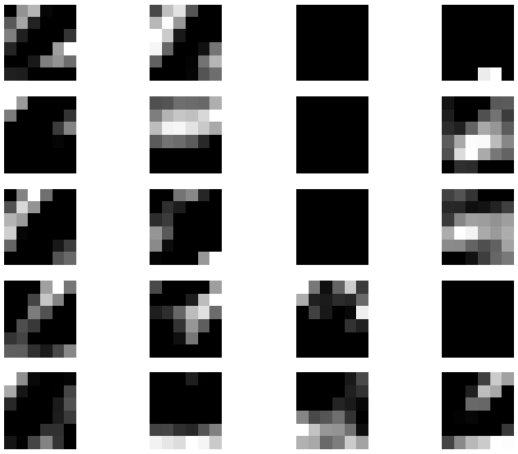
Sample 2:



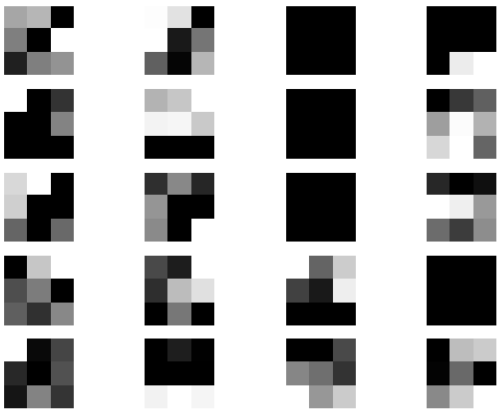
Convolution C1



Pooling P1



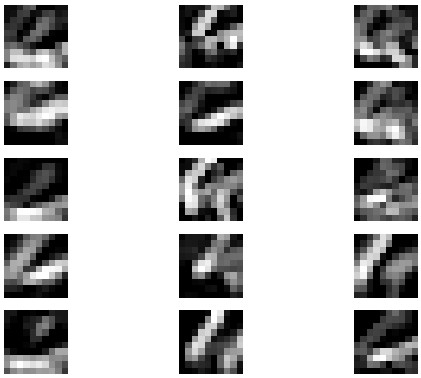
Convolution C2



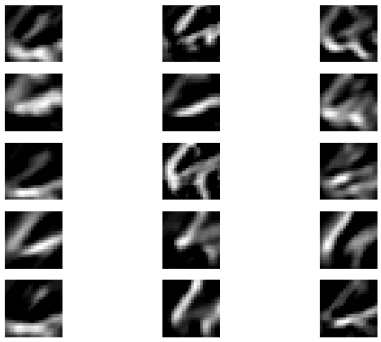
Pooling P2

Task 2: SGD with momentum

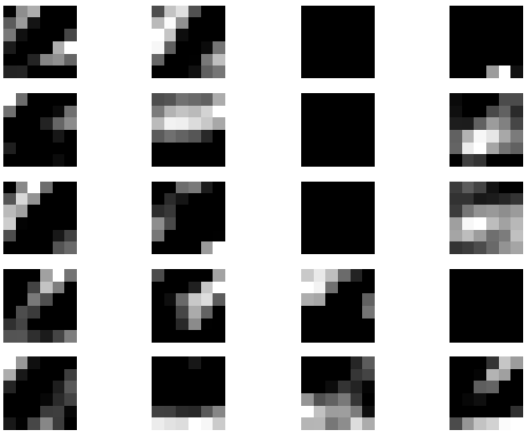
Sample 2:



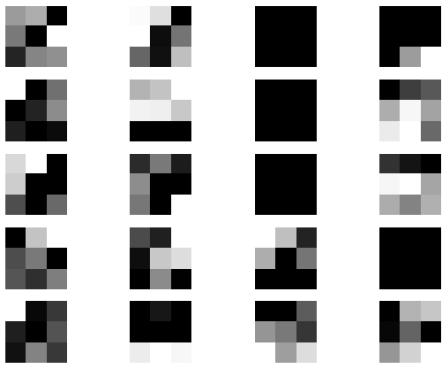
Convolution C1



Pooling P1



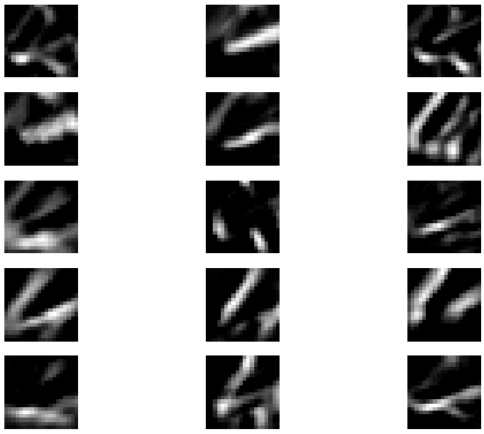
Convolution C2



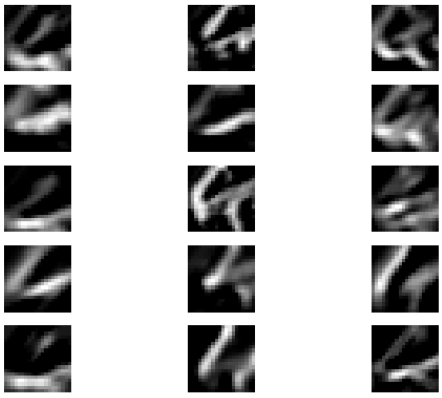
Pooling P2

Task 3: RMSProp

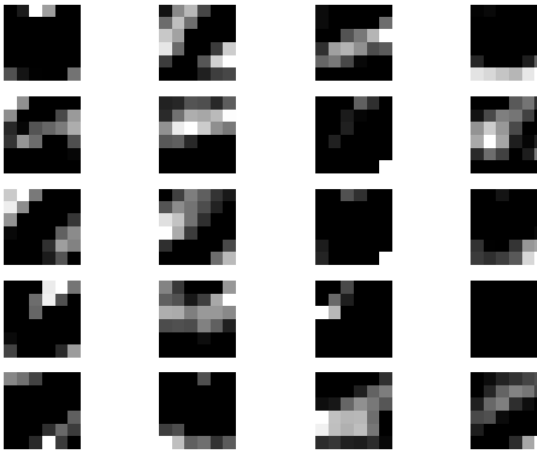
Sample 2:



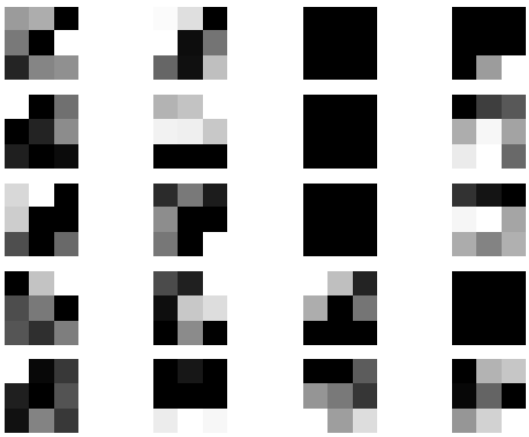
Convolution C1



Pooling P1



Convolution C2



Pooling P2