CS671 - Deep Learning and Applications

Assignment 2 : Optimizers for Backpropagation and Autoencoders

ASSIGNMENT 2 REPORT

submitted by

Aditya Sarkar	Aaron Thomas Joseph	Srishti Ginjala
IIT Mandi	IIT Mandi	IIT Mandi
B19003	B19128	B19084

under the supervision of

Dr. Dileep AD

INDIAN INSTITUTE OF TECHNOLOGY, MANDI

March 2022

Abstract

In this programming assignment, we have improved our understanding of the optimizers for backpropagation algorithms and autoencoders. For the first part, we have trained the fully connected neural network (FCNN) using different optimizers, such as NAG, RMS Prop and Adam, for the backpropagation algorithm. To compare these algorithms, we relied on the number of epochs that it takes for convergence along with achieving a humble classification accuracy. We have experimented with both vanilla and stochastic gradient descent algorithms. For the other one, we have built an autoencoder to obtain the hidden representation and evaluate it for classification. Number of nodes in the hidden layer was increased to see its impact on the classification performance. We then added noise to the best autoencoder architecture (achieved after trying multiple values for the number of nodes in the hidden layer), and observed the difference in the quality of reconstructed images, by using the mean squared error.
This report is made in LaTeX

Contents

Chapter 1

Optimizers

1.1 Brief description of the dataset

We were given a subset of the MNIST digit dataset consisting of five classes [0,2,4,6,7]. The data consisted of three folders for training, test and validation.

1.2 Experiments with different Architectures

In this section, we present our results obtained using various architectures on all the given optimizers. We varied the number of nodes in each hidden layer and noted the training and validation accuracy. In the first column architecture the tuple (a,b,c) means 'a' no.of nodes in first hidden layer, 'b' no.of nodes in second hidden layer and 'c' no.of nodes in third hidden layer.

The **Training Accuracies** are presented below.

Architecture	SGD	VGD	NAG	RMSProp	Adam
(30,20,10)	0.9998	0.9654	0.9908	0.9937	0.9993
(15,30,10)	0.9999	0.9914	0.9949	0.9956	0.9991
(25,15,8)	0.9999	0.9906	0.9937	0.9837	0.9993
(18,20,12)	0.9999	0.9887	0.9937	0.9891	0.9992

The **Validation Accuracies** are presented below.

Architecture	SGD	VGD	NAG	RMSProp	Adam
(30,20,10)	0.9834	0.9607	0.9744	0.9789	0.9823
(15,30,10)	0.9815	0.9742	0.9768	0.9768	0.9791
(25,15,8)	0.9836	0.9712	0.9799	0.9707	0.9805
(18,20,12)	0.9818	0.9750	0.9794	0.9749	0.9799

The **number of epochs** taken for convergence are presented below.

Architecture	SGD	VGD	NAG	RMSProp	Adam
(30,20,10)	28	31625	320	305	25
(15,30,10)	26	30795	301	280	33
(25,15,8)	31	32871	300	282	27
(18,20,12)	43	31896	303	285	29

1.3 Observations

This section will discuss the observations from our experiments. It will also mention the inferences that one can take from the observations.

1.3.1 Stochastic Gradient Descent

While training the algorithm using SGD, the number of nodes in best architecture in first, second and third hidden layers to be 30,20 and 10 respectively. The learning rate was set to 0.001. The batch size would be 1 as it is stochastic gradient. The stopping criterion was taken to be the minimum error below which the training would stop is 1e-4 and the maximum number of epochs was set to 1e5. The number of epochs taken to converge was estimated using early stopping which was implemented by the inbuilt callbacks function in Keras module.

1.3.1.1 Results and Observations

In the below plot (fig 1.1), we can see that the loss (for both training and validation) is going down and is tending towards zero. This indicates that the model was able to learn the patterns in the digits. Comparing both the curves, we can see that is drop is more steep for training data than for the validation data. This is expected because the model is being trained on the former i.e. it is bound to learn its patterns. While for validation, model is only predicting and not learning anything. Coming to the accuracy plots, we can observe that the training accuracy is greater than the validation accuracy across the range of epochs. This is inline with loss curves. From the above plot, one can conclude that the learning was generalized.

Key points :

1. Number of epochs to converge : **28**
2. Time taken for convergence : **19min**
3. Training Accuracy = **0.9996**
4. Validation Accuracy = **0.9823**

The confusion matrix for the training, validation and test data is shown below.

The classification metrics are also presented below:

Metric	Training Data	Validation Data	Test Data
Accuracy	0.9996	0.9823	0.9808
Precision	0.9996	0.9824	0.9808
Recall	0.9996	0.9823	0.9808
F1-Score	0.9996	0.9823	0.9808

1.3.1.2 Inferences

One thing to note about Stochastic Gradient Descent is that it is considering only one datapoint to update the weights. This datapoint is trying to trying to manage the weights in such

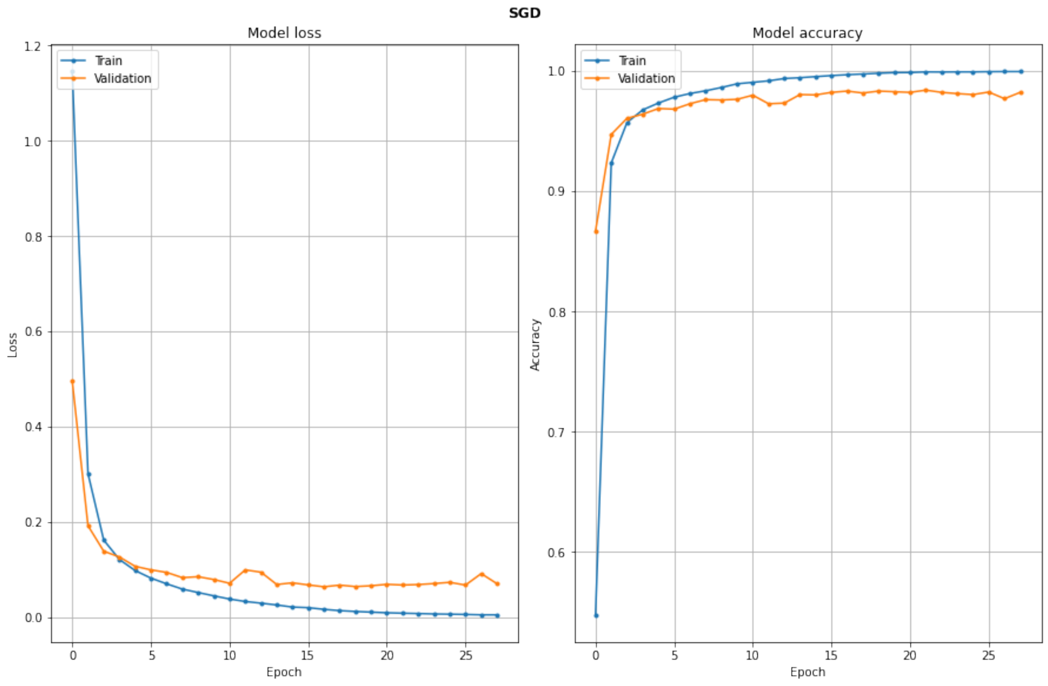

Fig. 1.1: Model loss and accuracy vs epochs

a way that it benefits itself. So in the beginning, one can observe that the convergence path is more noisier as compared to other algorithms. The reason is we are approximating the actual gradient of the data by considering only one point. It is similar to the statistical tests that we do where we consider only a subset of a population (also known as sample) and try to estimate the parameters rather that deriving the actual parameters. Estimating the actual gradient from just one datapoint will definitely lead to some kind of error, which produces noisier convergence.

Another thing that one can see is faster convergence. This is actually weird because if we still have to loop over all data points, then why it is converging in a less number of epochs? We will revisit this question after going through the vanilla gradient descent technique.

1.3.2 Vanilla Gradient Descent

While training the algorithm using VGD, the number of nodes in best architecture in first, second and third hidden layers were found to be 30,20 and 10 respectively. The learning rate was set to 0.001. The batch size would be length of training data as it is vanilla gradient.The

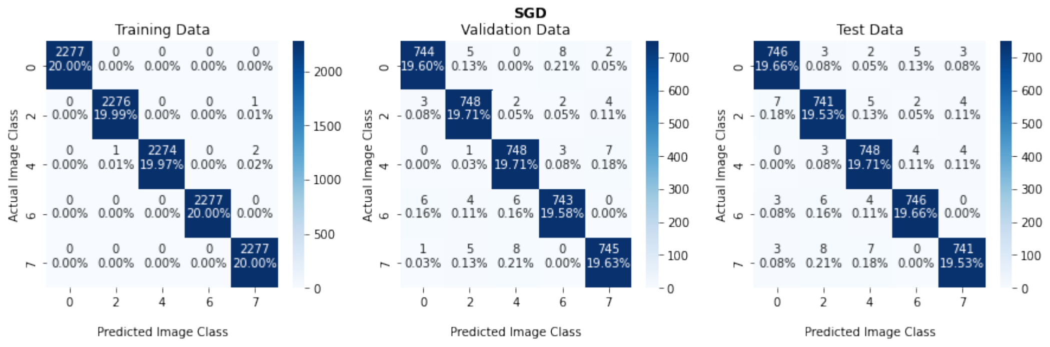

Fig. 1.2: Confusion Matrix for SGD

weights will be updated after all the training examples are passed once. The minimum error below which the training would stop is 1e-4 and the maximum number of epochs was set to 1e5. The number of epochs taken to converge was estimated using early stopping which was implemented by the inbuilt callbacks function in Keras module. This model converged in **31625** epochs and took around **90min** to run.

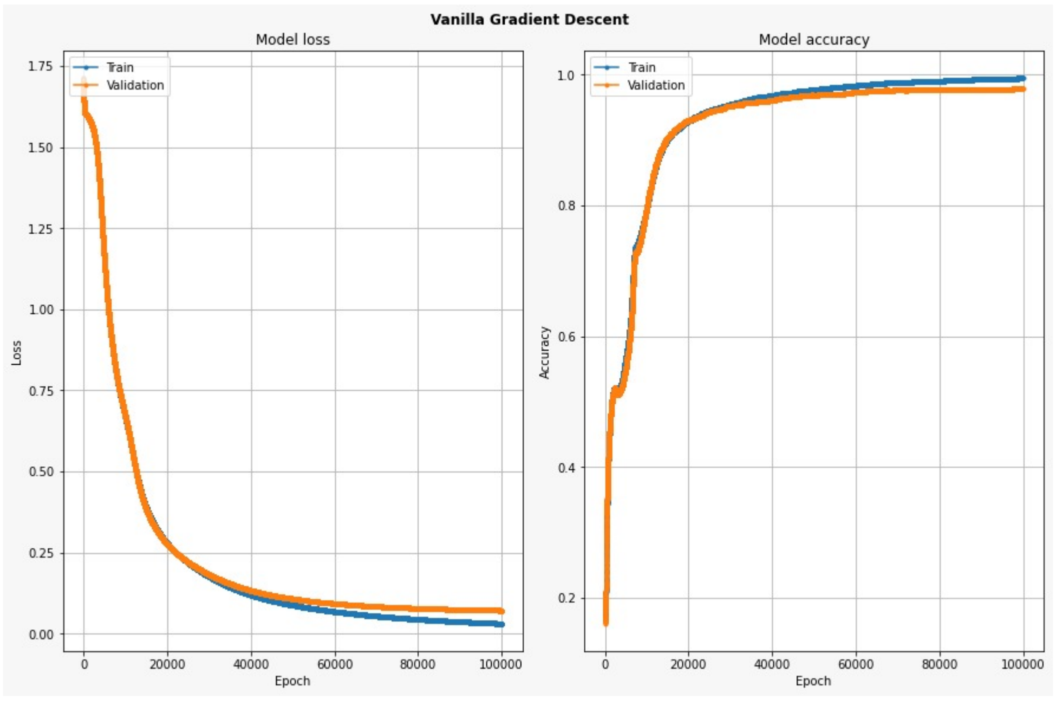

Fig. 1.3: Model loss and accuracy vs epochs

1.3.2.1 Results and Observations

In the below plot (fig 1.3), we can see that both the training and validation loss is going down in a similar fashion and is tending towards zero. This indicates that the model was able to learn the patterns in the digits. Coming to the accuracy plots, we can observe that even here the training and validation accuracy increase slowly across the range of epochs. This is inline with loss curves. From the above plot, one can conclude that the learning was generalized.

Training Accuracy = 0.9684

Validation Accuracy = 0.9625

The confusion matrix for the training, validation and test data is shown below.

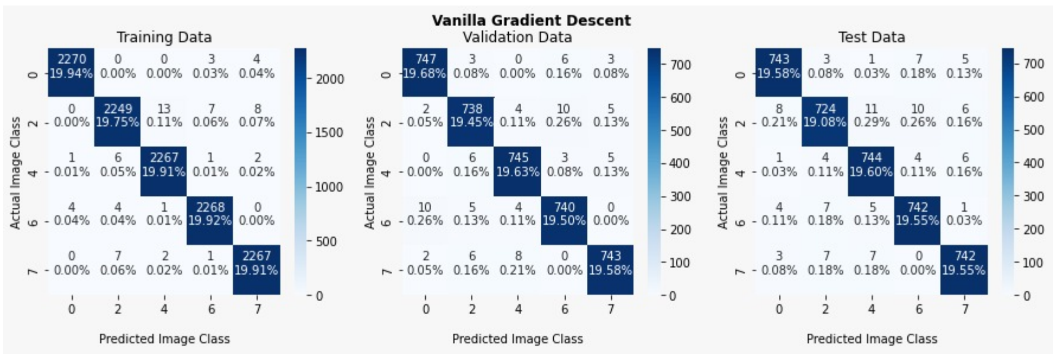

Fig. 1.4: Confusion Matrix for VGD

The classification metrics are also presented below:

Metric	Training Data	Validation Data	Test Data
Accuracy	0.9685	0.9626	0.957
Precision	0.9685	0.9626	0.9571
Recall	0.9685	0.9626	0.957
F1-Score	0.9684	0.9626	0.957

1.3.2.2 Inferences

We can notice that Vanilla Gradient Descent has taken far greater number of epochs (31000) than SGD (20) to converge. This is because VGD updates the weights after passing all the training examples whereas SGD updates weights after every training example which is much faster. This is also evident from the amount of time taken by both the algorithms to run. However, the gradient(loss) indicated by the SGD is not true loss because the weights may be biased towards individual data points. But, the loss in VGD is the actual loss as it represents all the training examples.

1.3.3 SGD with momentum (NAG)

While training the algorithm using Nesterov Accelerated Gradient Descent (NAG), the number of nodes in best architecture in first, second and third hidden layers were found to be 30,20 and 10 respectively. The learning rate was set to 0.001. The batch size was given as 32.The momentum parameter(alpha) was 0.9. The weights will be updated taking into account the history of the gradient along with the current gradient. The minimum error below which the training would stop is 1e-4 and the maximum number of epochs was set to 1e5. The number of epochs taken to converge was estimated using early stopping which was implemented by the inbuilt callbacks function in Keras module. NAG was implemented using the inbuilt optimisers for SGD function. This model converged in **320** epochs and took around **9min** to run.

1.3.3.1 Results and Observations

In the below plot (fig 1.5), we can see that the loss (for both training and validation) is going down and is tending towards zero. This indicates that the model was able to learn the patterns in the digits. Comparing both the curves, we can see that is drop is more steep for training data than for the validation data. This is expected because the model is being trained on the former i.e. it is bound to learn its patterns. While for validation, model is only predicting and not learning anything. Coming to the accuracy plots, we can observe that the training accuracy is greater than the validation accuracy towards the end. This is inline with loss curves. From the above plot, one can conclude that the learning was generalized.

Training Accuracy = 0.9961

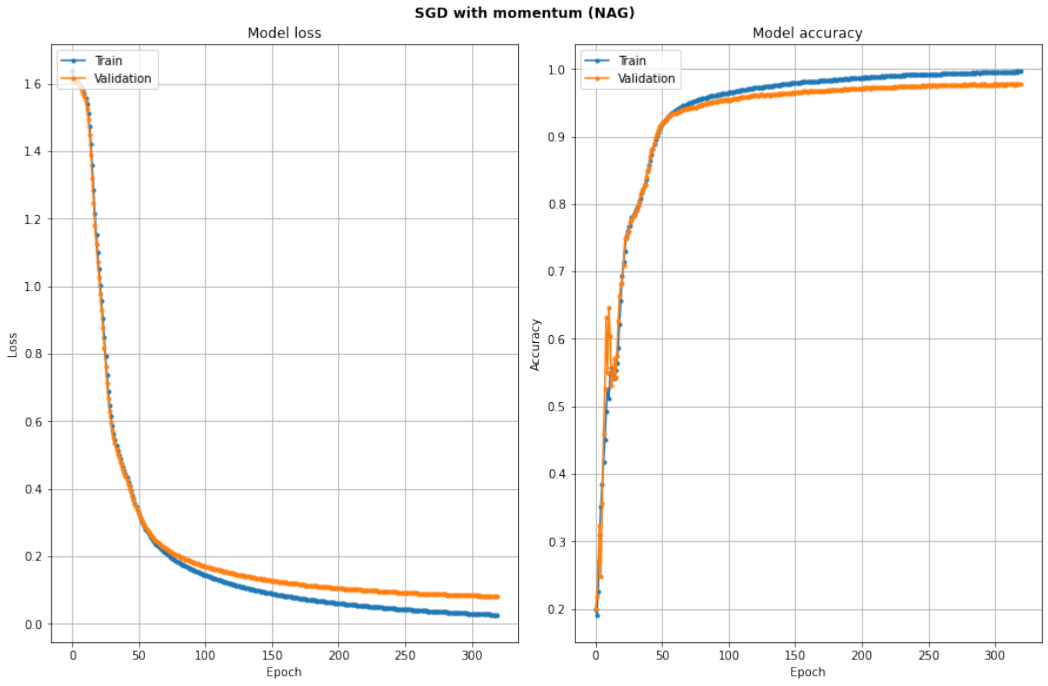

Fig. 1.5: Model loss and accuracy vs epochs

Validation Accuracy = 0.9770

The confusion matrix for the training, validation and test data is shown below.

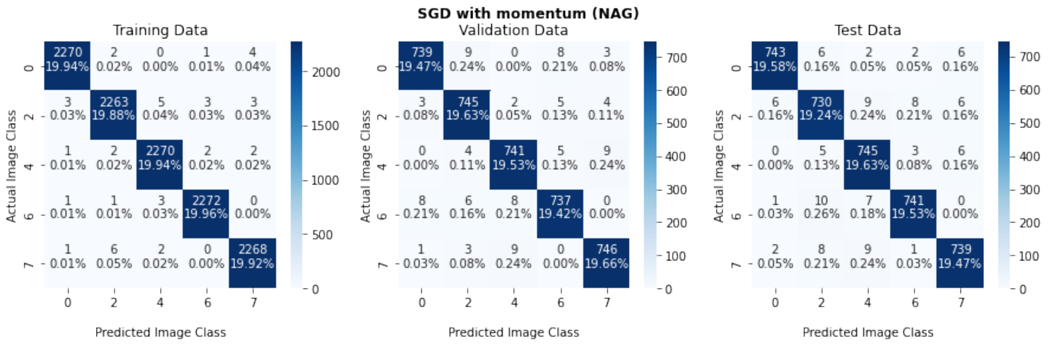

Fig. 1.6: Confusion Matrix for SGD with NAG

The classification metrics are also presented below:

1.3.3.2 Inferences

We can observe that NAG is quite faster than simple SGD though the batch size is larger as NAG took only 9min compared to SGD's 19 min. This is because of the momentum term in NAG which adds a part of the change in previous update to the current gradient. This

Metric	Training Data	Validation Data	Test Data
Accuracy	0.9963	0.9771	0.9744
Precision	0.9963	0.9771	0.9745
Recall	0.9963	0.9771	0.9744
F1-Score	0.9963	0.9771	0.9745

ensures that the minima is reached quickly. It's also quite evident that NAG is more efficient than VGD in terms of time and accuracy(marginal).

1.3.4 RMSProp

While training the algorithm using RMSProp, the number of nodes in best architecture in first, second and third hidden layers were found to be 30,20 and 10 respectively. The learning rate was set to 0.001. The batch size was given as 32.The momentum parameter was 0.9 and the discounting factor for the history/coming gradient is 0.99. The exponential moving average of squared gradient is not growing rapidly in this case. The minimum error below which the training would stop is 1e-4 and the maximum number of epochs was set to 1e5. The number of epochs taken to converge was estimated using early stopping which was implemented by the inbuilt callbacks function in Keras module. RMSProp was implemented using the inbuilt optimisers function. This model converged in **305** epochs and took around **2min** to run.

1.3.4.1 Results and Observations

In the below plot (fig 1.7), we can see that the loss (for both training and validation) is going down and is tending towards zero. This indicates that the model was able to learn the patterns in the digits. Comparing both the curves, we can see that is drop is more steep for training data than for the validation data. This is expected because the model is being trained on the former i.e. it is bound to learn its patterns. While for validation, model is only predicting and not learning anything. Coming to the accuracy plots, we can observe that the training accuracy is greater than the validation accuracy across the range of epochs.

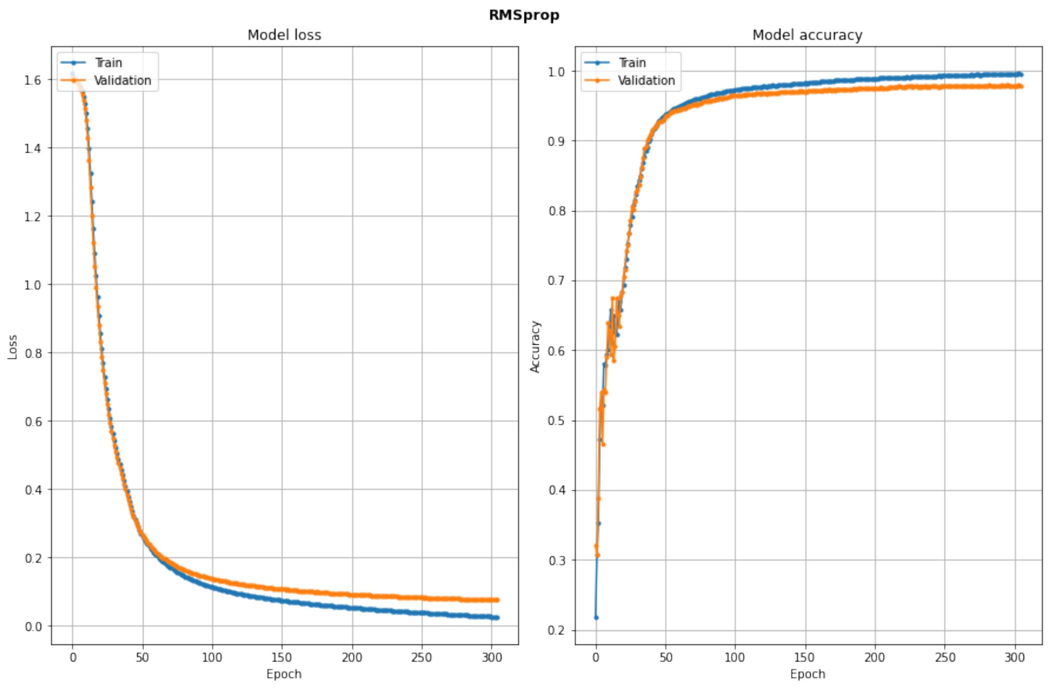

Fig. 1.7: Model loss and accuracy vs epochs

This is inline with loss curves. From the above plot, one can conclude that the learning was generalized.

Training Accuracy = 0.9958

Validation Accuracy = 0.9791

The confusion matrix for the training, validation and test data is shown below.

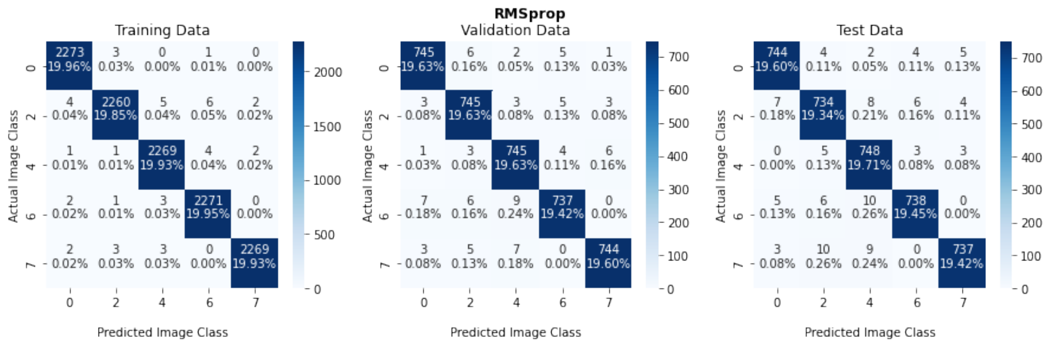

Fig. 1.8: Confusion Matrix for RMSProp

The classification metrics are also presented below:

Metric	Training Data	Validation Data	Test Data
Accuracy	0.9962	0.9792	0.9752
Precision	0.9962	0.9792	0.9753
Recall	0.9962	0.9792	0.9752
F1-Score	0.9962	0.9792	0.9752

1.3.4.2 Inferences:

We observe that there is no significant difference in performance between RMSProp and NAG. In RMSProp, we have different learning rates for different parameters. However, it is fast compared to both SGD and VGD.

1.3.5 Adaptive Moments (Adam)

While training the algorithm using RMSProp, the number of nodes in best architecture in first, second and third hidden layers were found to be 30,20 and 10 respectively. The learning rate was set to 0.001. Beta1=0.9, beta2=0.999 and epsilon=1e-08. The batch size was given as 32.The momentum parameter(alpha) was 0.9. The weights will be updated taking into account the history of the gradient along with the current gradient. The minimum error below which the training would stop is 1e-4 and the maximum number of epochs was set to 1e5. The number of epochs taken to converge was estimated using early stopping which was implemented by the inbuilt callbacks function in Keras module. Adam was implemented using the inbuilt Adam optimisers function. This model converged in **25** epochs and took around **2min** to run.

1.3.5.1 Results and Observations

In the below plot (fig 1.9), we can see that the loss (for both training and validation) is going down and is tending towards zero. This indicates that the model was able to learn the patterns in the digits. Comparing both the curves, we can see that is drop is more steep for training data than for the validation data. This is expected because the model is being trained on the former i.e. it is bound to learn its patterns. While for validation, model is only

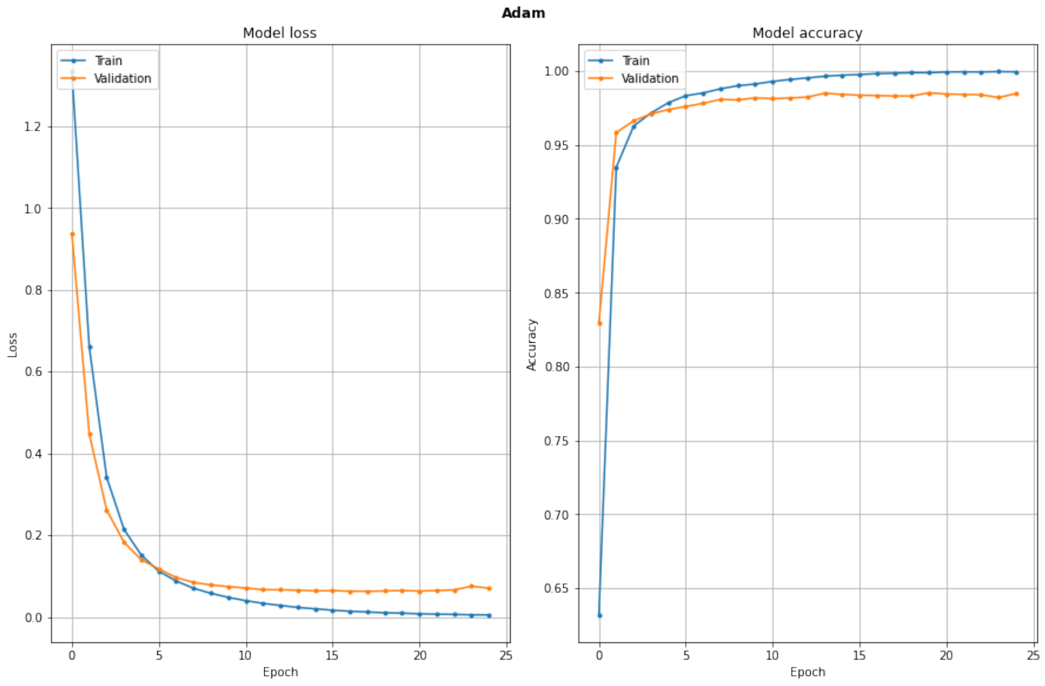

Fig. 1.9: Model loss and accuracy vs epochs

predicting and not learning anything. Coming to the accuracy plots, we can observe that the training accuracy is greater than the validation accuracy towards the end. This is inline with loss curves. From the above plot, one can conclude that the learning was generalized.

Training Accuracy = 0.9994

Validation Accuracy = 0.9847

The confusion matrix for the training, validation and test data is shown below.

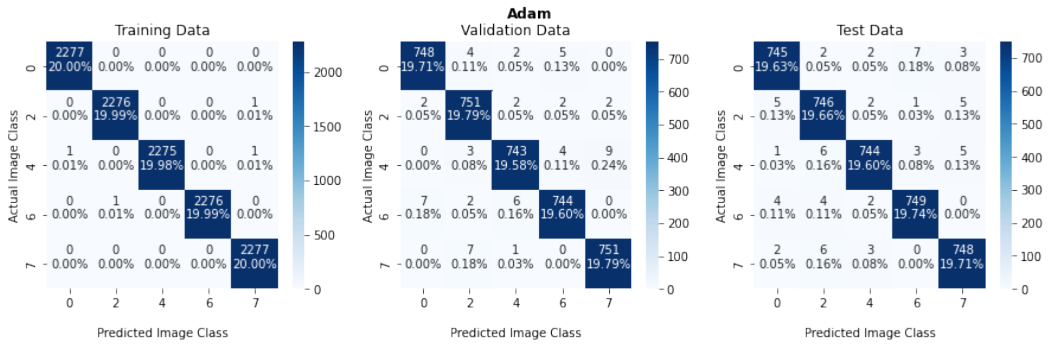

Fig. 1.10: Confusion Matrix for Adam

The classification metrics are also presented below:

Metric	Training Data	Validation Data	Test Data
Accuracy	0.9996	0.9847	0.9834
Precision	0.9996	0.9847	0.9834
Recall	0.9996	0.9847	0.9834
F1-Score	0.9996	0.9847	0.9834

1.3.5.2 Inferences

We can clearly notice that Adam optimizer takes least number of epochs and least time to converge to the solution. This can be accredited to the inclusion of both, moving exponential average of both the gradient and square of gradient. They accelerate the convergence while preventing the decay of weights in an effective manner which also shows empirically. But, the validation accuracy got compromised marginally(about 0.98).

In a nutshell, we can conclude that Adam optimizer converges in the least time(and also least epochs) without severely affecting the accuracy of the model.

Chapter 2

Autoencoders with one hidden layer

2.1 Brief description of the dataset

We were given a subset of the MNIST digit dataset consisting of five classes [0,2,4,6,7]. The data consisted of three folders for training, test and validation.

2.2 Observations

Autoencoders are simply a feedforward network which has a speciality of having hidden layers that has less number of nodes as compared to that of input layer and the number of nodes in input and output layers are same. This is also known as the auto association of the neural networks. Considering x_n as the input, autoencoder first encodes the it to a hidden layer h_n and then decodes it to the output (of same dimension as input) to x_n^p. This section will discuss the observations from our experiments. It will also mention the inferences that one can take from the observations.

2.2.1 Number of nodes in hidden layer

Basically, we have three cases in this depending on the number of input nodes (784 in our case). These three cases are -

1. When number of nodes in hidden layer is less than number of nodes in the input layer.

2. When number of nodes in hidden layer is more than number of nodes in the input layer.

3. When number of nodes in hidden layer is equal to number of nodes in the input layer.

Let's go into the details of each of the three cases.

2.2.1.1 Case 1: Less number of nodes

This type of encoder is known as under complete autoencoder, and widely used in dimensionality reduction and denoising. The bottleneck layer captures the important characteristics of the input. Features of the input data are usually correlated. Keeping the number of nodes lesser in the bottleneck layer eliminates all those features of input that are redundant, because of these correlations, and learn only the salient features of the data, assuming that the autoencoder is properly trained. The compressed representation can be further used for the classification purpose. Since in this, we are forcing the autoencoder to learn something, we will be expecting a high validation error as compared to below two cases.

2.2.1.2 Case 2: Equal number of nodes

This type of encoder is known as over complete autoencoder. If you force someone to do something, then it will always try to find an alternate way to escape your constraints that you are imposing. Hence this case will become a trivial case, because the autoencoder will simply try to learn weights in such a way that it copies the input to its hidden layer and then show it at the output layer as well. Please note that this is because of the inherent structure of the autoencoders, that is auto association. Keeping the size of hidden layer is not always bad, as it is beneficial while we perform classification or regression task, basically anything that relates to hetero-association neural networks. In this case of autoencoders only, it will not learn any hidden representation which can contain the salient features of data. As a result, we would expect lower validation and training error.

2.2.1.3 Case 3: More number of nodes

This type of encoder is also known as over complete autoencoder. This is again a trivial case where no learning of quality hidden representation will happen.

2.2.1.4 Experiments with different no.of nodes in hidden layer

We got the below table after experimenting with different number of nodes in the hidden layer and present the MSEs and classification accuracies for all the architectures. The results are inline with the theory mentioned in the above sections.

Table 2.1: Best Model table

Models' Accuracy table				
Nodes	MSE (Train)	MSE (Valid)	Acc (Test)	Epochs
32	0.01759	0.01756	95.8%	48
64	**0.01036**	**0.01037**	**96.68%**	**58**
128	0.00552	0.00560	96.89%	27
256	0.00266	0.00275	96.63%	18
512	0.00110	0.00114	96.73%	21
784	0.00080	0.00088	97.15%	24
1024	0.00107	0.00107	96.71%	21

2.2.1.5 Best Architecture keypoints

It is mentioned in the question that one has to select a model based on the validation error. But there will be a problem which is discussed in the very next secion. Hence we have considered the classification accuracy when the learnt latent representation is feeded into the FCNN. For selecting the best, we have picked up that model where there is an abrupt increase in the classification accuracy.
PS : We are using Google Colab GPU for running our models.
PS : Classification model parameters : learning rate=0.001, beta1=0.9, beta2=0.999, epsilon=$1e^{-}08$.

Keypoints of our best architecture is :

1. Number of nodes in input layer - 784

2. Number of nodes in hidden layer - 64
3. Number of nodes in output layer - 784
4. Optimizer - Adam
5. Time for convergence - 59 secs
6. Epochs for convergence - 58
7. Training loss - 0.01036
8. Validation loss - 0.01037
9. Test loss - 0.01038
10. Classification Accuracy - 96.63%

2.2.2 Reconstruction Error

We have defined the reconstruction error as the mean squared error between the actual images and the reconstructed images. The error will decrease as the number of epochs increase and as you increase the space in the hidden representation. Giving more space leads the autoencoders to start trivial copying of the inputs. Hence as mentioned above, we have considered only the classification accuracy to select our best model. The classification accuracy that we have achieved after our convergence criterion is reached is around 97% which shows that our autoencoder has learned well. It means that the hidden representation we have learnt is also good for performing any classification task, which we will discuss in the later section.

We have attached the confusion matrix for reference. You can see that the diagonal elements of the matrix are having high values, which means that FCNN is able to discriminate the images well from the hidden representation created by the autoencoder.

2.2.3 Reconstructed images

Reconstructed images look similar to that of the actual images. This is because our model is trained well.

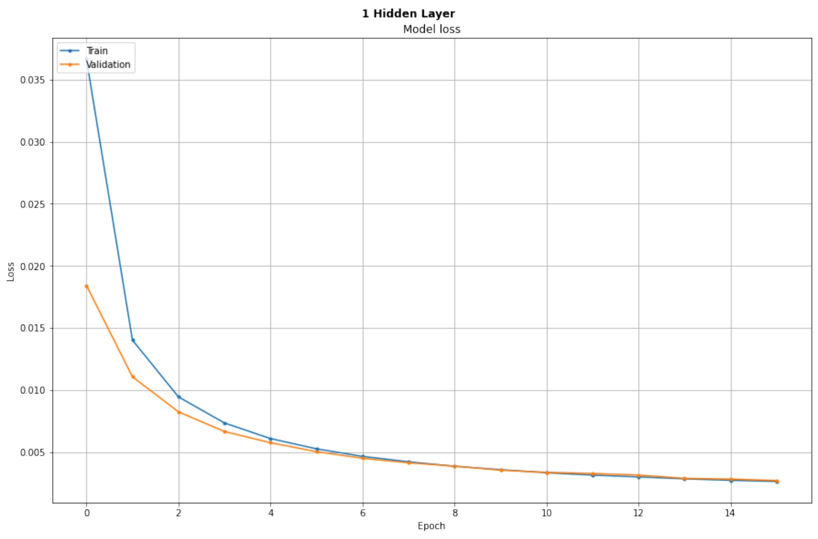

Fig. 2.1: Reconstruction Loss vs epochs

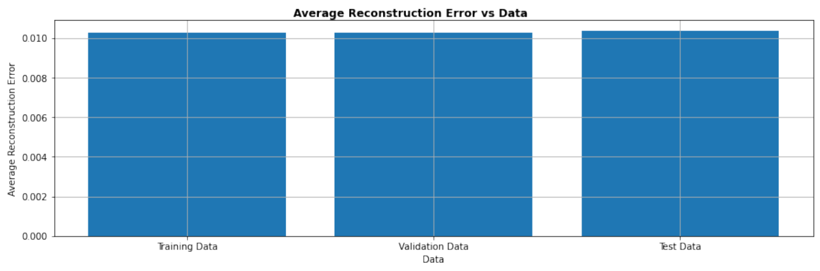

Fig. 2.2: Average reconstruction Loss

We have also looked in the reconstructed images for different number of nodes in the bottleneck layer. It will encompass the three cases described above as well.

2.2.3.1 Case 1: Less number of nodes

We will reduce the dimensionality first and then decode it. The reconstructed images are a little washed out. This is because we have not given much space to the hidden representation. If we slightly increase the number of nodes in the hidden representation, you can see the difference.

2.2.3.2 Case 2: More/Equal number of nodes

On increasing the space for representation in the hidden layer will given better reconstructed images. This is because of trivial copying of input. Hence the reconstructed images are much

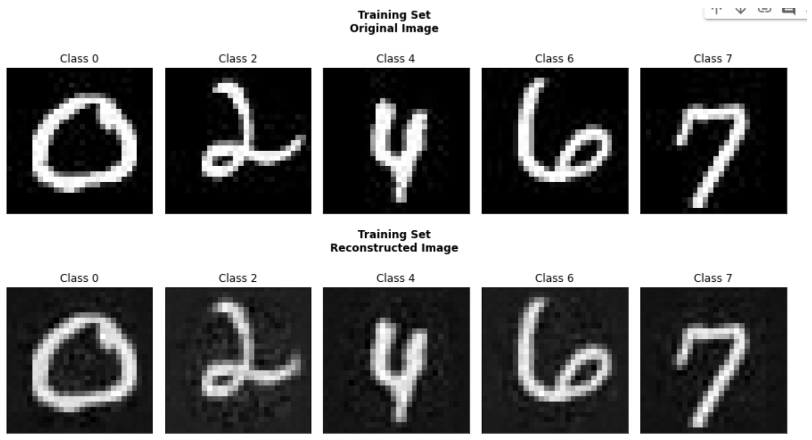

Fig. 2.3: Reconstruction Images for 64 hidden node

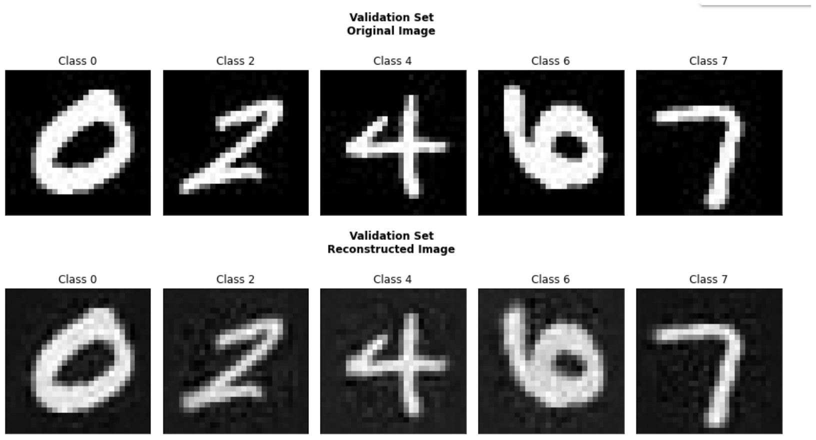

Fig. 2.4: Reconstruction Images for 64 hidden node

better than the previous case.

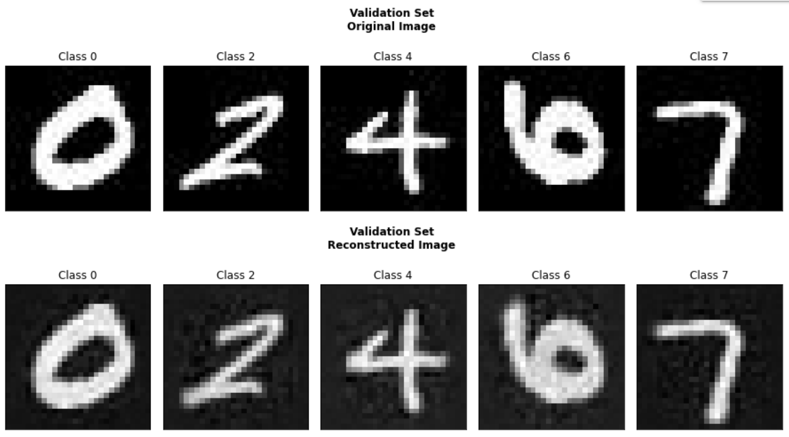

Fig. 2.5: Validation Images for 256 hidden node

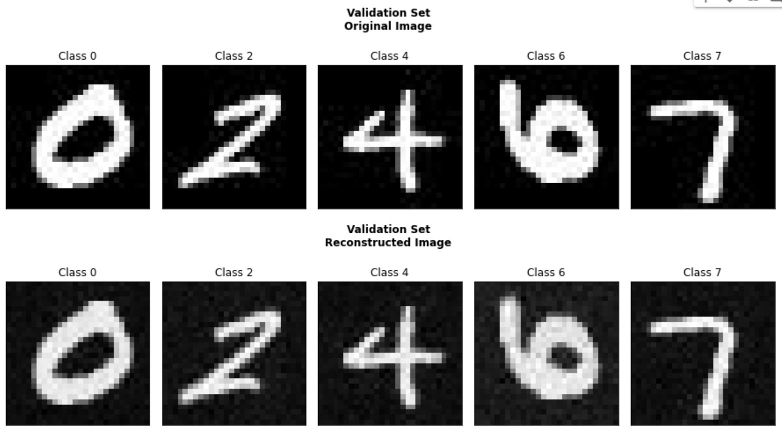

Fig. 2.6: Validation Images for 784 hidden node

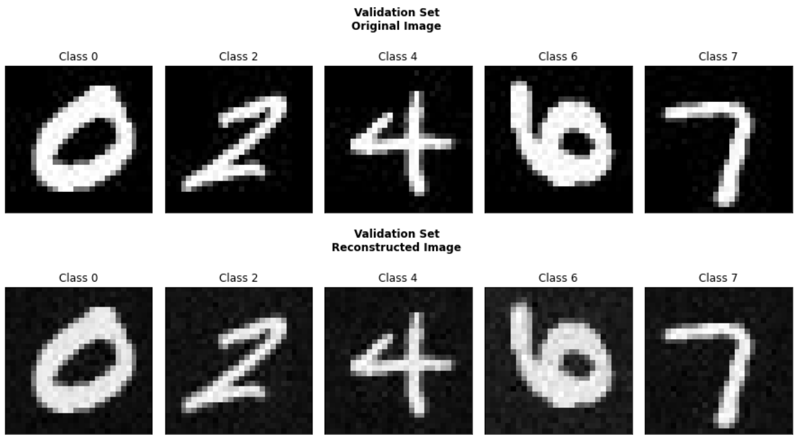

Fig. 2.7: Validation Images for 1024 hidden node

2.2.4 Classification

We sliced the decoder off the network and used the hidden representation as an input to the FCNN for classification purpose. We got a validation and test accuracy of around 97%, which is again high. While training the autoencoders, latent representation has actually learnt the salient features of data; those features that are not correlated or redundant. These salient features also contain the discriminative features which FCNN uses to classify the data. Thus, one can say that better the latent representation is, better the classification accuracy will be.

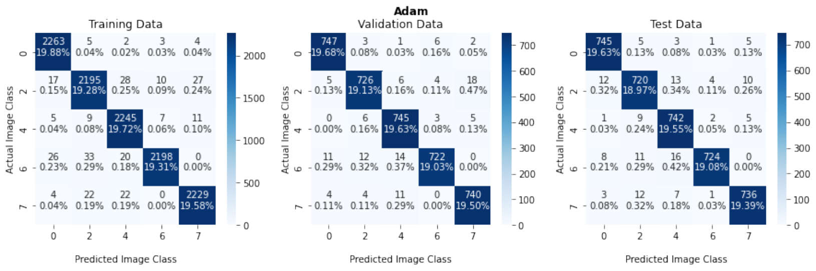

Fig. 2.8: Classification Confusion Matrix

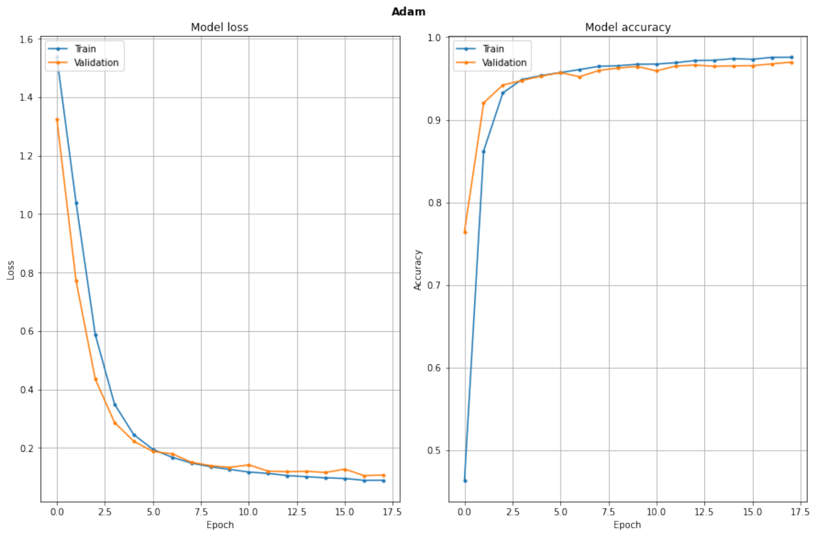

Fig. 2.9: Loss vs epochs for best model

2.2.5 Weight visualization

Since the inputs to model is normalized, those particular inputs which will fire the neurons is given by -

$$x_n = \frac{w_j}{||w_j||}$$

In the question, we were asked to plot the weights for the best model. But we have plotted for three cases that are mentioned above. For the model which has bottleneck layer, having the less number of nodes, We have plotted these for the weights from input layer to compressed layer for the three following cases. As the number of nodes in the bottleneck layer increases, it is hard to recognize the presence of a digit from the plotted weights. It becomes more noisier as shown in the figures below. This is because the model is only learning those weights that helps it copy images from the input. It is not interested to learn any kind of representation containing salient features.

2.2.5.1 Case 1: Less number of nodes

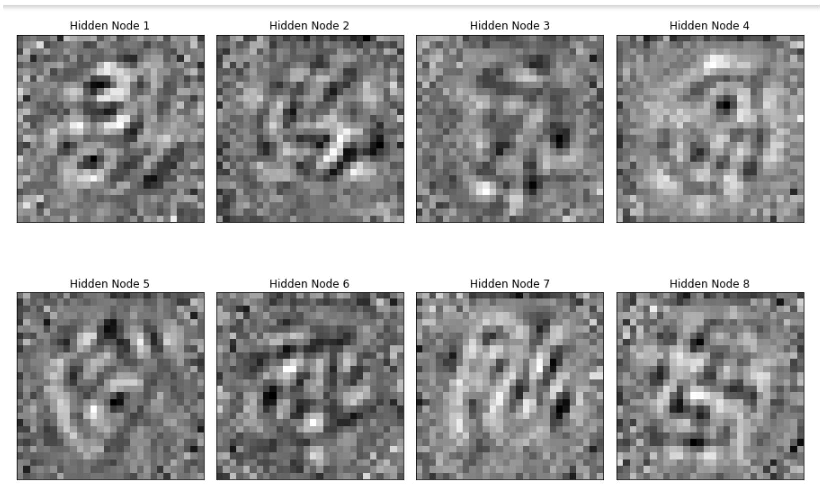

Fig. 2.10: Weight visualization for 64 hidden node

2.2.5.2 Case 2: Equal number of nodes

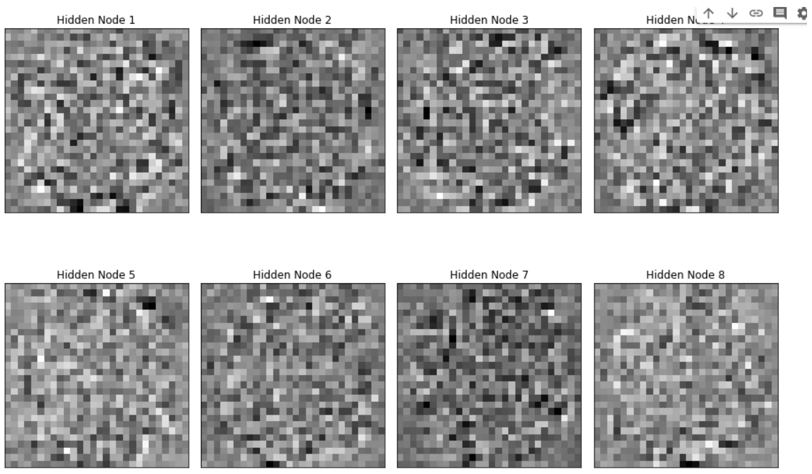

Fig. 2.11: Weight visualization for 784 hidden node

2.2.5.3 Case 3: More number of nodes

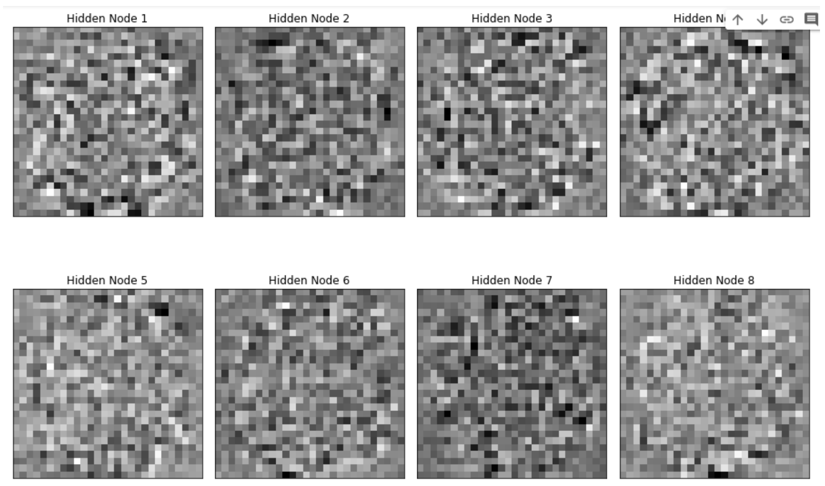

Fig. 2.12: Weight visualization for 1024 hidden node

Chapter 3

Autoencoders with three hidden layers

3.1 Brief description of the dataset

We were given a subset of the MNIST digit dataset consisting of five classes [0,2,4,6,7]. The data consisted of three folders for training, test and validation.

3.2 Observations

This section will explore the autoencoders that have three hidden layers. Most of the results are similar to the previous one, so we will wrap it quickly.

3.2.1 Number of hidden layers

In autoencoders, we have an odd number of hidden layers. In the previous chapter, we were having 1 and here, we have 3. Autoencoders we are using here are called stacked autoencoders as the layers are stacked after one another. Having a large number of hidden layers is a good thing keeping in mind that they are odd in number. However, too many hidden layers will lead to overfitting or problems like vanishing gradients. Vanishing gradients will make the initial layers useless as there will be very less update during backpropagation. One needs to be careful while deciding the number of hidden layers.

3.2.2 Number of nodes in bottleneck layer

Basically, we have three cases in this depending on the number of input nodes (784 in our case). In this we are fixing the first and the third layer to be 256. These three cases are -

1. When number of nodes in hidden layer is less than number of nodes in the first and third layers.
2. When number of nodes in hidden layer is more than number of nodes in the first and third layers.
3. When number of nodes in hidden layer is equal to number of nodes in the first and third layers.

Let's go into the details of each of the three cases.

3.2.2.1 Case 1: Less number of nodes

This type of encoder is known as under complete autoencoder, and widely used in dimensionality reduction and denoising. The bottleneck layer captures the the salient features of the data, assuming that the autoencoder is properly trained. The salient features are uncorrelated and compact. Since in this, we are forcing the autoencoder to learn something, we will be expecting a high validation error as compared to below two cases.

3.2.2.2 Case 2: Equal number of nodes

This type of encoder is known as over complete autoencoder. The autoencoder will simply try to learn weights in such a way that it copies the input to its bottleneck layer and then show it at the output layer as well. It will not learn any hidden representation which can contain the salient features of data. As a result, we would expect lower validation and training error.

3.2.2.3 Case 3: More number of nodes

This type of encoder is also known as over complete autoencoder. This is again a trivial case where no learning of quality latent representation will happen.

3.2.2.4 Best Architecture keypoints

It is mentioned in the question that one has to select a model based on the validation error. But there will be a problem which is discussed in the very next secion. Hence we have considered the classification accuracy when the learnt latent representation is feeded into the FCNN. For selecting the best, we have picked up that model where there is an abrupt increase in the classification accuracy.

PS : We are using Google Colab GPU for running our models.

PS : Classification model parameters : learning rate=0.001, beta1=0.9, beta2=0.999, epsilon=$1e^{-8}$.

Keypoints of our best architecture is :

1. Number of nodes in input layer - 784
2. Number of nodes in hidden layer - 64
3. Number of nodes in output layer - 784
4. Optimizer - Adam
5. Time for convergence - 62 secs
6. Epochs for convergence (during classification) - 30
7. Training loss - 0.00977
8. Validation loss - 0.00997
9. Test loss - 0.01301

3.2.2.5 Effect of number of nodes in bottleneck layer

We have also done experiments by changing the number of nodes in each hidden layer and present the classification accuracies for all the architectures. Since the number of nodes in the third hidden layer is same as that of first hidden layer we have written the number of nodes in only the first and second hidden layers(i.e. nodes1 and nodes2 respectively).

PS : The number of epochs mentioned is during running the classification model.

Table 3.1: Best Model table

Models' Accuracy table				
Nodes	MSE (Train)	MSE (Valid)	Acc (Test)	Epochs
32	0.0126	0.0129	95.68%	53
64	**0.00977**	**0.00997**	**96.52%**	**30**
128	0.00735	0.00744	97.5%	28
256	0.00651	0.00672	96.34%	21
512	0.00612	0.00631	97.44%	19

3.2.3 Reconstruction Error

We have defined the reconstruction error as the mean squared error between the actual images and the reconstructed images. The error will decrease as the number of epochs increase and as you increase the space in the hidden representation. Giving more space leads the autoencoders to start trivial copying of the inputs. Hence as mentioned above, **we have considered only the classification accuracy to select our best model.** The classification accuracy we have achieved after our convergence criterion is reached is around 97% which shows that our autoencoder has learned well.

The loss of test data is slightly higher than validation and training. It is because model has seen validation and training dataset but not test dataset. Test dataset is completely unseen but achieving such a high classification accuracy clearly indicates that our model is generalized.

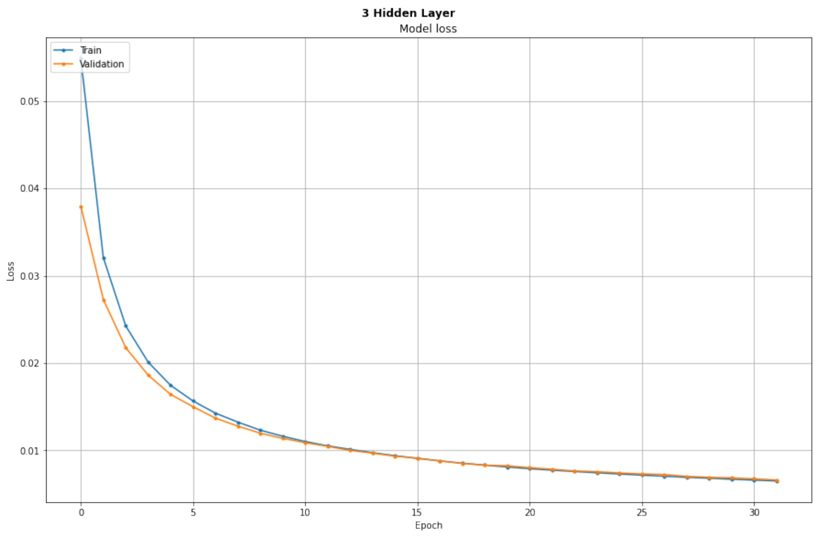

Fig. 3.1: Reconstruction Loss vs Epochs for 64 hidden node

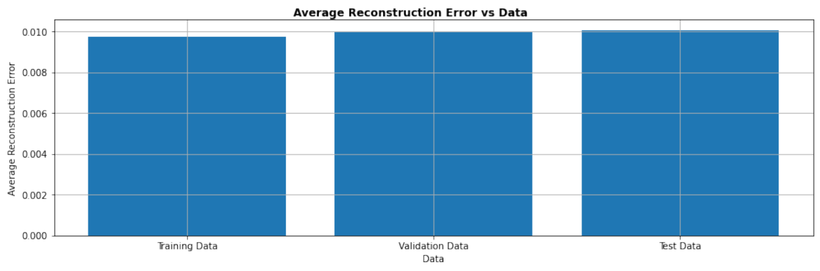

Fig. 3.2: Average Reconstruction Loss for 64 hidden node

3.2.4 Reconstructed images

Reconstructed images look similar to that of the actual images. This is because our model is trained well.

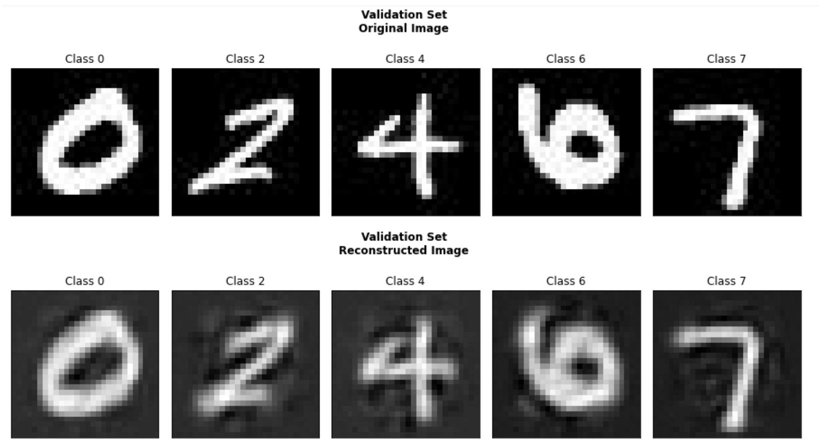

Fig. 3.3: Reconstructed images for best model

We have also looked in the reconstructed images for different number of nodes in the bottleneck layer. It will encompass the three cases described above as well.

3.2.4.1 Case 1: Less number of nodes

We will reduce the dimensionality first and then decode it. The reconstructed images are a little washed out. This is because we have not given much space to the hidden representation. If we slightly increase the number of nodes in the hidden representation, you can see the difference.

3.2.4.2 Case 2: More/Equal number of nodes

On increasing the space for representation in the hidden layer will given better reconstructed images. This is because of trivial copying of input.

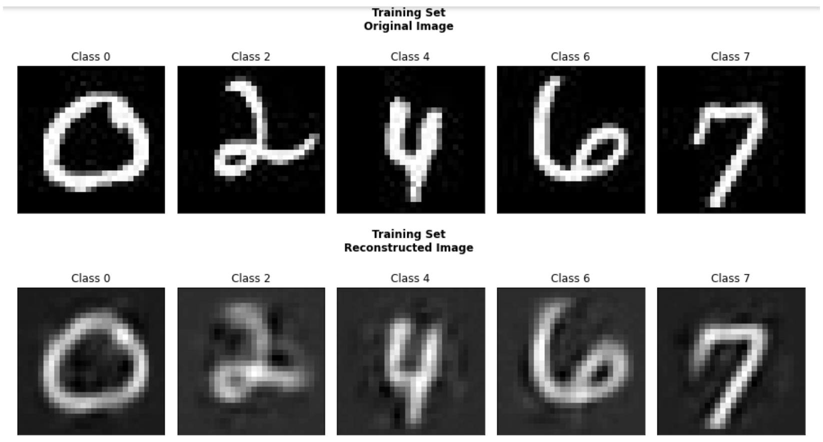

Fig. 3.4: Reconstructed images for 64 hidden node

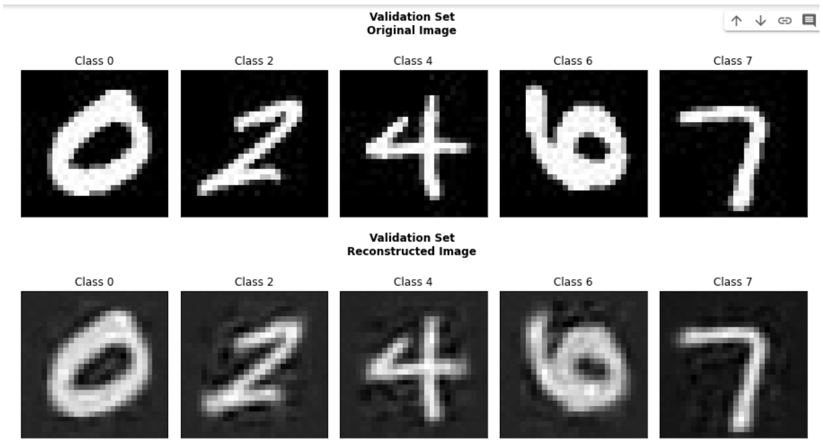

Fig. 3.5: Reconstructed images for 256 hidden node

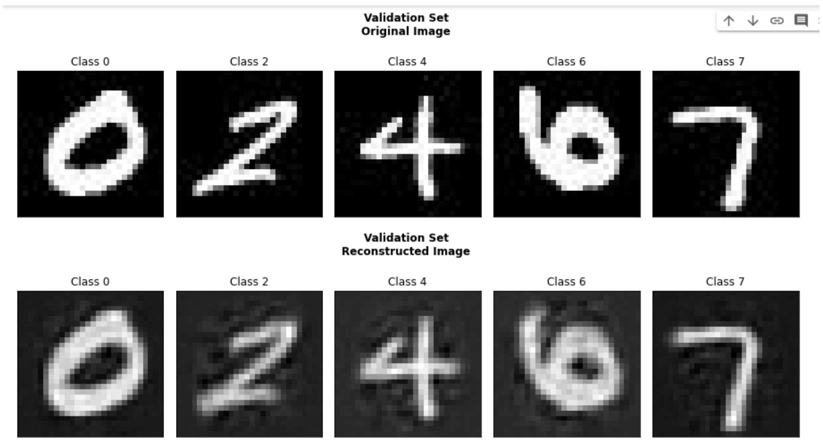

Fig. 3.6: Reconstructed images for 512 hidden node

3.2.5 Classification

We sliced the decoder off the network and used the hidden representation as an input to the FCNN for classification purpose. We got a validation and test accuracy of around 97%, which is again high. While training the autoencoders, latent representation has actually learnt the salient features of data; those features that are not correlated or redundant. These salient features also contain the discriminative features which FCNN uses to classify the data. Thus, one can say that better the latent representation is, better the classification accuracy will be.

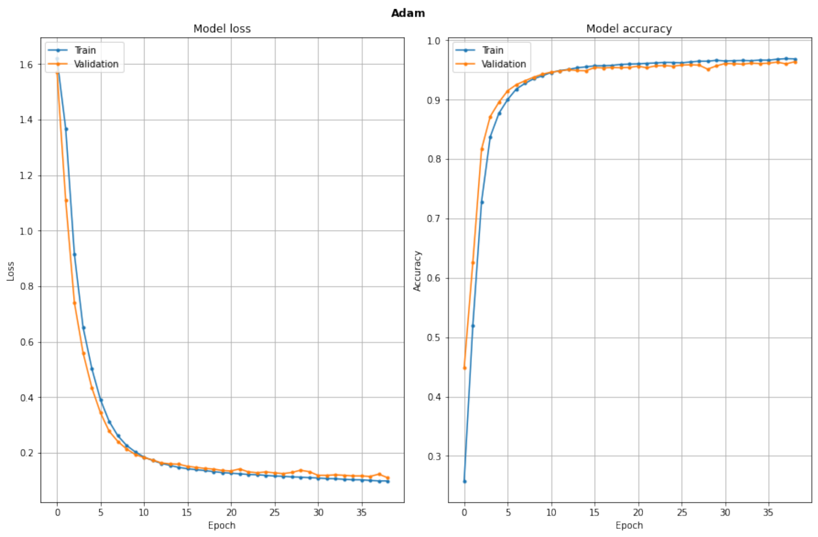

Fig. 3.7: Losses vs Epochs for 64 hidden node

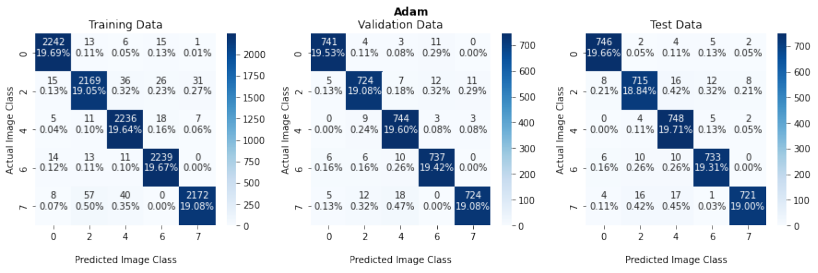

Fig. 3.8: Confusion matrix for 64 hidden node

Chapter 4

Denoising Autoencoders

4.1 Brief description of the dataset

We were given a subset of the MNIST digit dataset consisting of five classes [0,2,4,6,7]. The data consisted of three folders for training, test and validation.

4.2 Observations

This section will explore the denoising autoencoders that have one and three hidden layers. In the previous sections, we have always tried to keep the bottleneck layer small to force our autoencoder to learn an intelligent representation of the data. The same training can be done by adding noise to the input images. We will keep a probability p which will help us randomly pick up pixels on which we are supposed to add noise, at each epoch. After adding noise, we can try to extract the original noise free image. In this case, we are restricting the autoencoder to simply copy the input to its output. This type of autoencoder is called a denoising autoencoder. As mentioned in the question, we have used the best one-hidden layer autoencoder architecture from Question 2.a.i. The keypoints of the model are mentioned in Chapter 2. The number of nodes is 64.

4.2.1 Reconstructed Error

As mentioned in earlier sections, to select our best model, we didn't consider the reconstruction loss. Instead we focused on the classification accuracy, that we have achieved after passing the hidden representation into the FCNN. Our model has achieved a classification accuracy of 98%, which the hidden representation is learnt well. Hence the reconstructed images should look similar to that of the actual images. We can also observe the same thing as shown below.

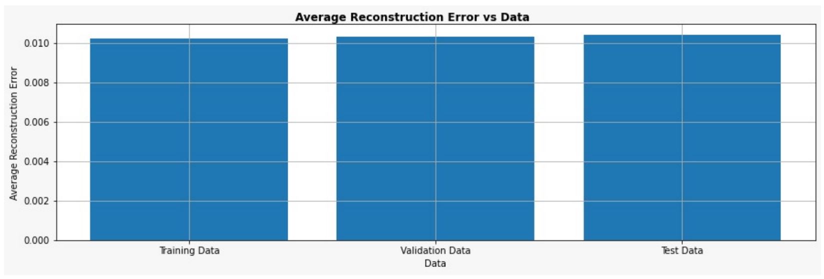

Fig. 4.1: Reconstruction loss for 20% noise

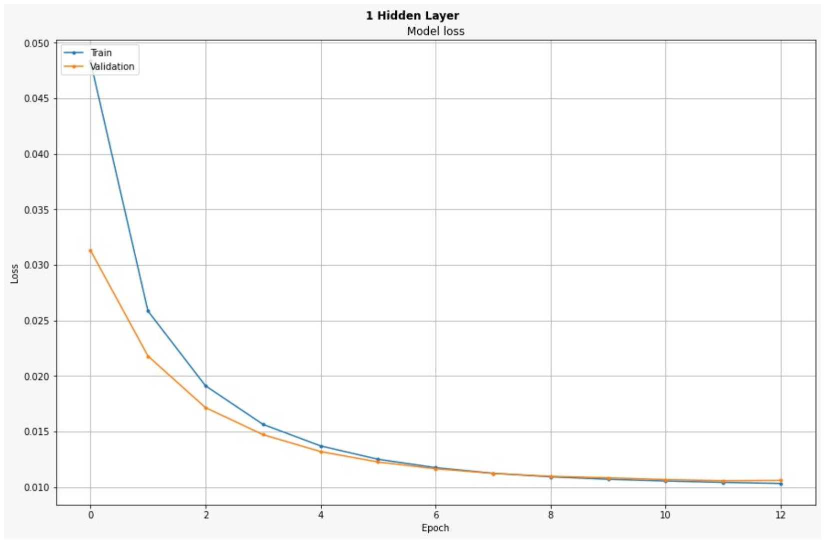

Fig. 4.2: Losses vs Epochs for 20% noise

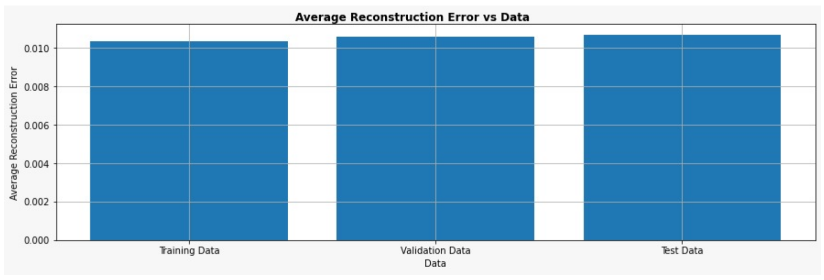

Fig. 4.3: Reconstruction loss for 40% noise

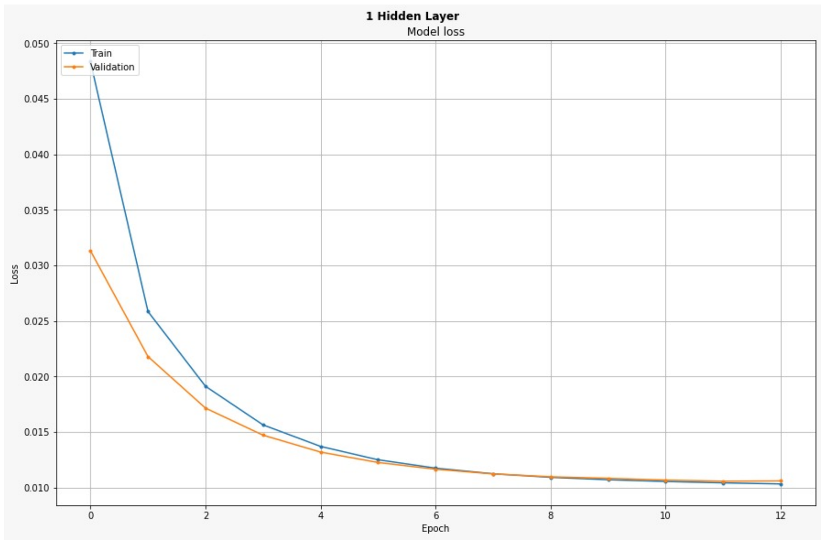

Fig. 4.4: Losses vs Epochs for 40% noise

4.2.2 Reconstructed images

Reconstructed images look similar to that of the actual images. This is because our model is trained well. We have also looked in the reconstructed images for different amount of noise that is being added to the images. It will encompass the two cases described above as well.

4.2.2.1 Case 1: 20% noise

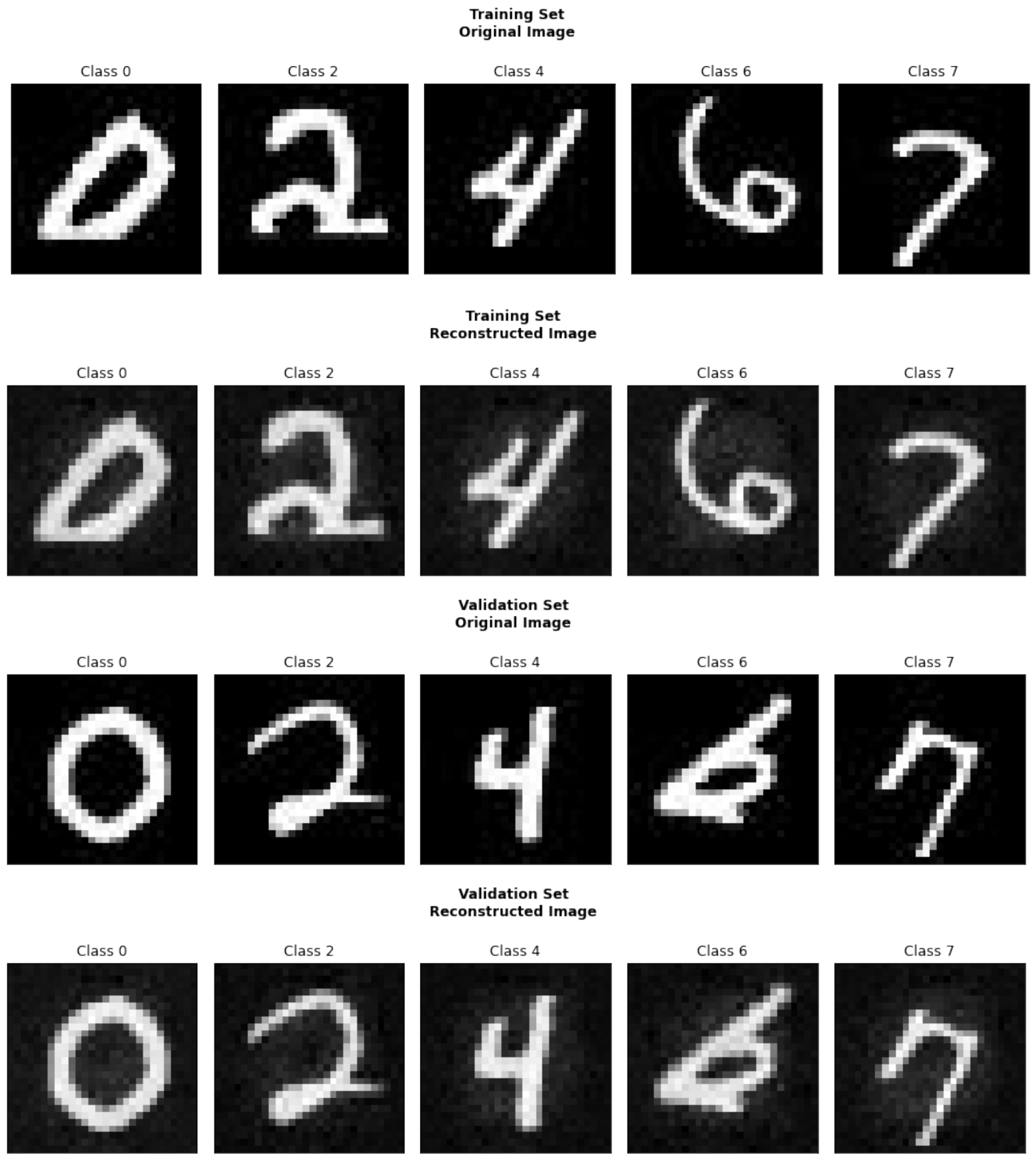

Fig. 4.5: Reconstructed images for inputs having 20% noise. PS : the input images plotted here are not corrupted

4.2.2.2 Case 2: 40% noise

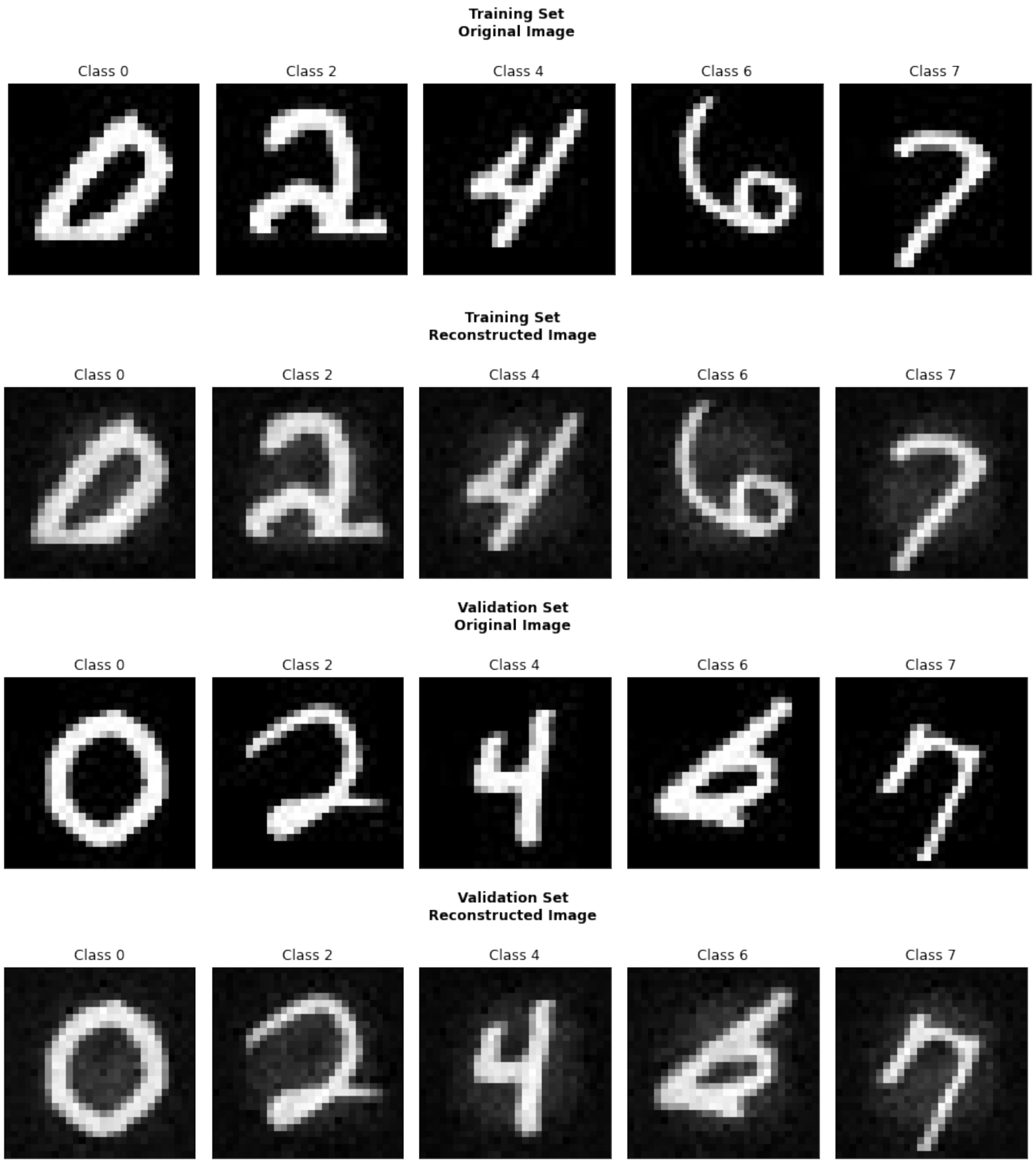

Fig. 4.6: Reconstructed images for inputs having 40% noise. PS : the input images plotted here are not corrupted

4.2.3 Classification

We sliced the decoder off the network and used the hidden representation as an input to the FCNN for classification purpose. We got a validation and test accuracy of 96.65%, which is again high. while training the autoencoders, hidden representation has actually learnt the salient features of data; those features that are not correlated or redundant. These salient features also contain the discriminative features which FCNN uses to classify the data. Thus, one can say that better the hidden representation is, better the classification accuracy will be.

We have also looked in the classification accuracies for different amount of noise that is being added to the images. It will encompass the two cases described above as well. We used the compressed images as input to the 3 hidden layer architecture(30,20,10).

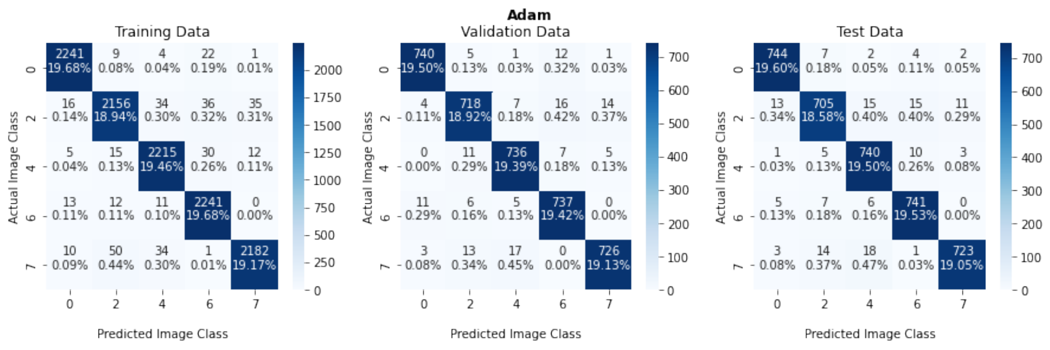

Fig. 4.7: Confusion matrix for 20% noise

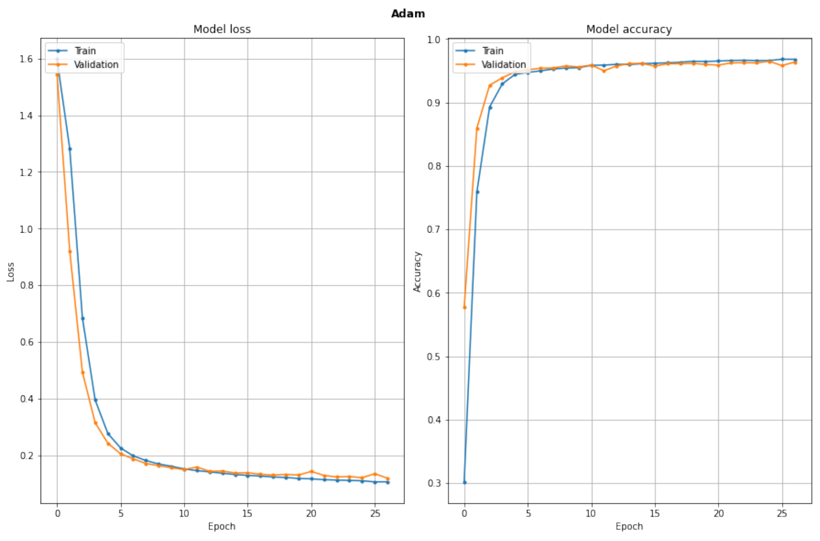

Fig. 4.8: Losses vs epochs for 20% noise

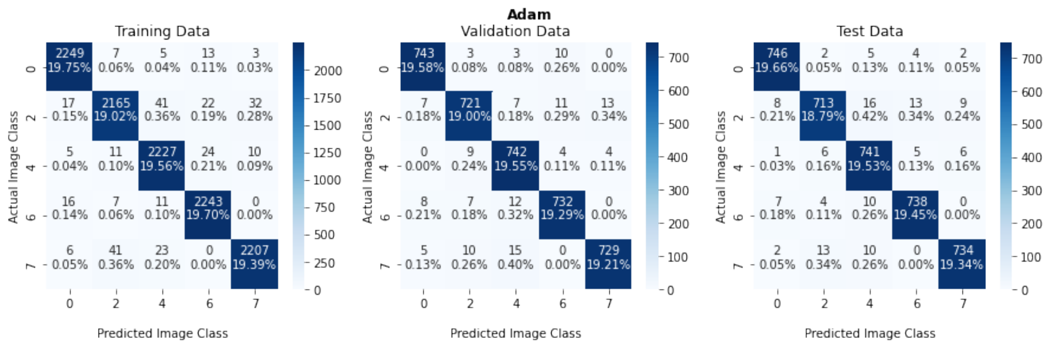

Fig. 4.9: Confusion matrix for 40% noise

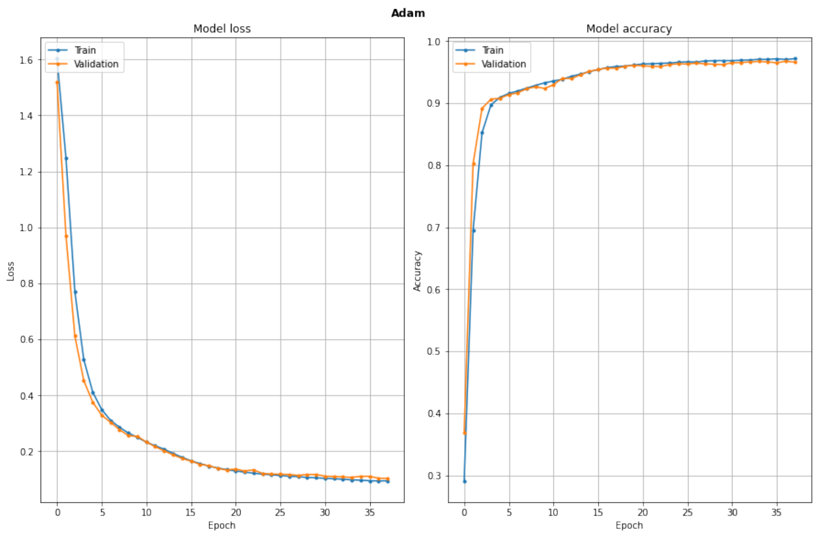

Fig. 4.10: Losses vs epochs for 40% noise

Case 1: 20% noise

Training Accuracy = 0.980.

Validation Accuracy = 0.969

Case 2: 40% noise

Training Accuracy = 0.980

Validation Accuracy = 0.971

4.2.4 Weight visualization

Since the inputs to model is normalized, those particular inputs which will fire the neurons is given by -

$$x_n = \frac{w_j}{||w_j||}$$

We have plotted these for the weights from input layer to compressed layer. As the amount of noise in the input images increases, the weights become more noisy and it is hard to recognise any presence of digits. This is because the output images have no noise while input images have. Since the weights are multiplied to input to get output, the weights tend to pick up noise to remove it from input.

4.2.4.1 Case 1: 20% noise

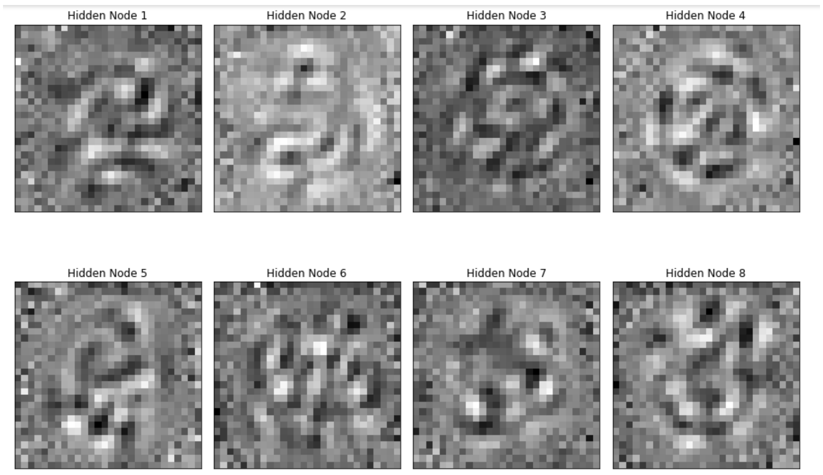

Fig. 4.11: Weight visualization for 20% noise

4.2.4.2 Case 2: 40% noise

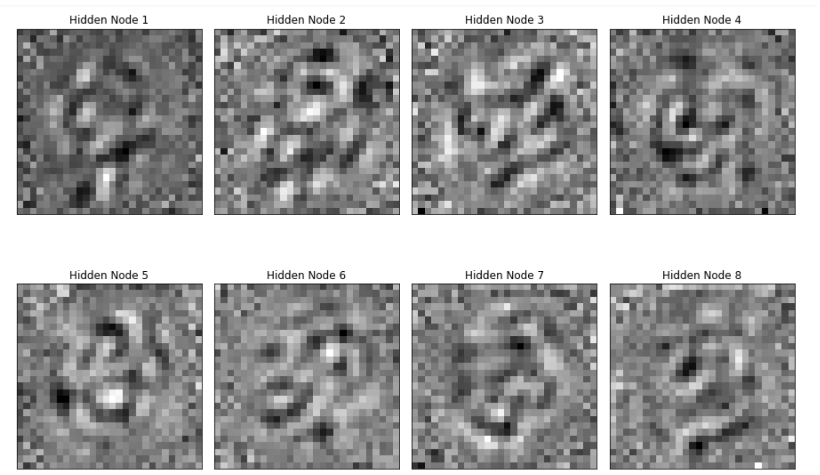

Fig. 4.12: Weight visualization for 40% noise

References