UNIVERSITY COLLEGE CORK

CS6421: Deep Learning

FINAL PROJECT

MSc COMPUTING SCIENCE

BEJOY JOSE KACHAPPILLY

Submitted To: - Dr. Gregory Provan

By submitting this exam, I declare

(1) that all work of it is my own;
(2) that I did not seek whole or partial solutions for any part of my submission from others; and
(3) that I did not and will not discuss, exchange, share, or publish complete or partial solutions for this exam or any part of it.

REPORT

Autoencoding is a algorithm primarily used for data compression and decompression. It is a lossy compression technique which uses auto learning techniques to perform well on a specific type of input. The compression and decompression may result in a outputs which will be degraded compared to the original inputs. Two important practical applications of autoencoders today are denoising data, and reducing dimensionality for data visualization. Autoencoder, consists of three parts: An encoding function, a decoding function and a distance function to compute information loss.

PART 1

1. Basic Autoencoder

1.1 Basic Dense model 1

Part one focuses on implementing basic autoencoder using fully-connected (Dense Layer) architectures with TensorFlow and Keras. The basic model consists of a single hidden and output layer for both encoder and decoder. ReLU activation function was used in both the encoder and decoder. Following layer parameters were used for basic model.

Type	Encoder/Decoder (0,1)	Layers	Intermediate Dimensions	Learning Rate	Activation	Size
Dense	0	1	128	1e-2	ReLU	
Output	0	ReLU				128
Dense	1	1	128	1e-2	ReLU	
Output	1	ReLU				784
Loss	**3.82**					

Table 1

We ran the above initial model for 20 epochs and got following results.

```
Epoch 13/20. Loss: 3.902657985687256
Epoch 14/20. Loss: 3.824237823486328
Epoch 15/20. Loss: 3.8316173553466797
Epoch 16/20. Loss: 3.8453874588012695
Epoch 17/20. Loss: 3.836491346359253
Epoch 18/20. Loss: 3.8128395080566406
Epoch 19/20. Loss: 3.820082902908325
Epoch 20/20. Loss: 3.8294782638549805
```

Figure 1. a

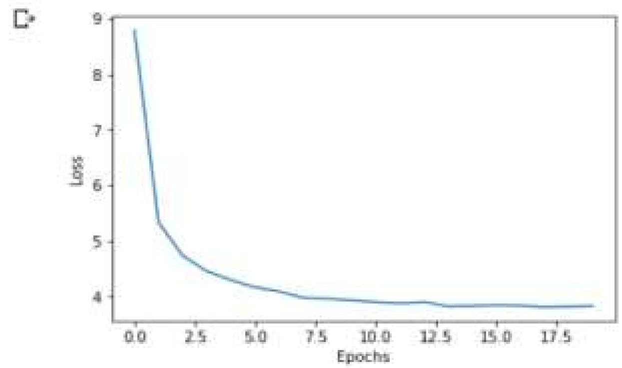

Figure 1.b

<u>Note</u>: - An error loss of **3.8294782638549805** was observed in this model.

Figure 1.c represents the Input and the corresponding output after using the basic model.

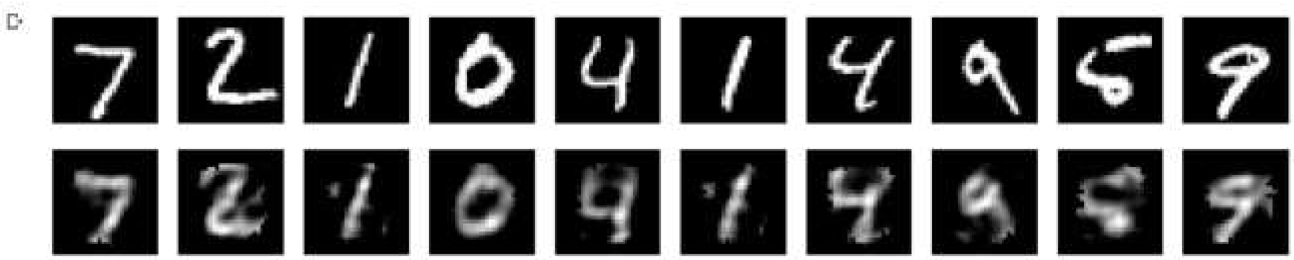

Figure 1. c

1.2 Basic Dense model 2

In this model we changed the activation function. We used **Sigmoid** instead of ReLU. We have even increased the inter dimensions and decreased learning rate for decreasing loss and to get better output. Note that this model trained better than the basic dense model

Type	Encoder/Decoder (0,1)	Layers	Intermediate Dimensions	Learning Rate	Activation	Size
Dense	0	1	512	1e-3	Sigmoid	
Output	0	ReLU				128
Dense	1	1	512	1e-3	Sigmoid	
Output	1	ReLU				784
Loss	**1.475**					

Table 2

Figure 2. A, figure 2. B and figure 2. C shows the corresponding epoch run, graph and output of this model. The error was considerably deceased to **1.475**.

```
Epoch 17/20. Loss: 1.7061666250228882
Epoch 18/20. Loss: 1.621521949768O664
Epoch 19/20. Loss: 1.5450788736343384
Epoch 20/20. Loss: 1.4755326509475708
```

Figure 2. A

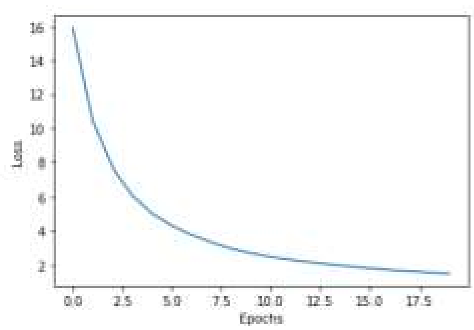

Figure 2. B

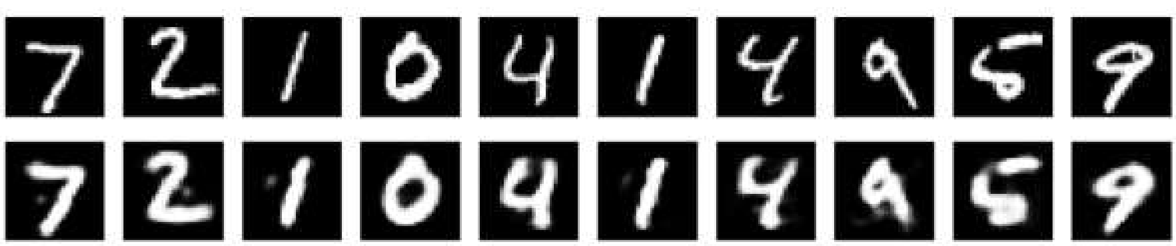

Figure 2. C

1.3 Improved Dense model 3

We were able to improve the above model by changing various parameters, like intermediate dimensions, learning rate, activation functions and changing the layers. We couldn't find significant improvement after increasing layers and activation functions but saw a significant decrease in loss and increase in image reconstruction using following parameters in Table 2.

Type	Encoder/Decoder (0,1)	Layers	Intermediate Dimensions	Learning Rate	Activation	Size
Dense	0	1	512	1e-3	ReLU	
Output	0	ReLU				128
Dense	1	1	512	1e-3	ReLU	
Output	1	ReLU				784
Loss	**0.981**					

Table 3

Figure 2. A, figure 2. B and figure 2. C shows the corresponding epoch run, graph and output of this model. The error was considerably deceased to **0.981**.

```
Epoch 13/20. Loss: 1.0444037914276123
Epoch 14/20. Loss: 1.0454233884811401
Epoch 15/20. Loss: 1.0444309711456299
Epoch 16/20. Loss: 1.0413607358932495
Epoch 17/20. Loss: 1.0026257038116455
Epoch 18/20. Loss: 1.0033574104309082
Epoch 19/20. Loss: 1.0264246463775635
Epoch 20/20. Loss: 0.9813662171363831
```

Figure 3. A

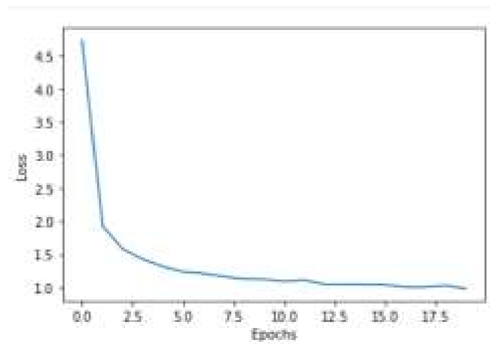

Figure 3. B

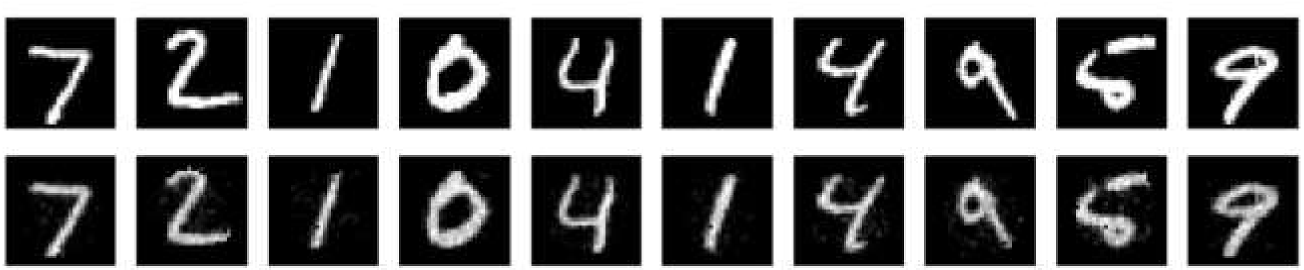

Figure 3. C

1.4 Basic CNN Model (Model 4)

One of the alternatives to improve image quality and decrease loss is by using a convolutional neural network architecture as the basis of the encoding and decoding. Autoencoders applied to images as convolutional autoencoders simply perform much better. The encoder will consist in a stack of Conv2D and MaxPooling2D layers, while the decoder will consist in a stack of Conv2D and UpSampling2D layers. The given basic CNN model uses following layers and parameters.

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernal	Activation	Size
Convolution	0	3	(16,8,8)	(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Convolution	1	3	(16,8,8)	(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	784
Loss	**0.2156**					

Table 4

Note there was a significant decrease in error loss when we switch from dense layer to CNN. The error in the basic dense model was **3.82,** where as CNN basic model with 3 layers gave a minimum loss of **0.21.** Though the output was not regenerated correctly. Tweaking the parameters and layers will give significant improvement in output.

```
Epoch 45/50
469/469 [==============================] - 3s 6ms/step - loss: 0.2187 - val_loss: 0.2188
Epoch 46/50
469/469 [==============================] - 3s 6ms/step - loss: 0.2181 - val_loss: 0.2182
Epoch 47/50
469/469 [==============================] - 3s 6ms/step - loss: 0.2174 - val_loss: 0.2175
Epoch 48/50
469/469 [==============================] - 3s 6ms/step - loss: 0.2168 - val_loss: 0.2169
Epoch 49/50
469/469 [==============================] - 3s 6ms/step - loss: 0.2162 - val_loss: 0.2163
Epoch 50/50
469/469 [==============================] - 3s 6ms/step - loss: 0.2156 - val_loss: 0.2157
```

Figure 4. A

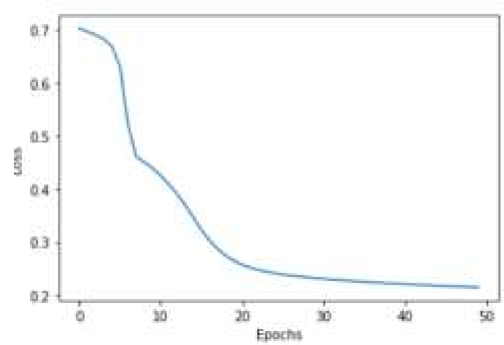

Figure 4. B

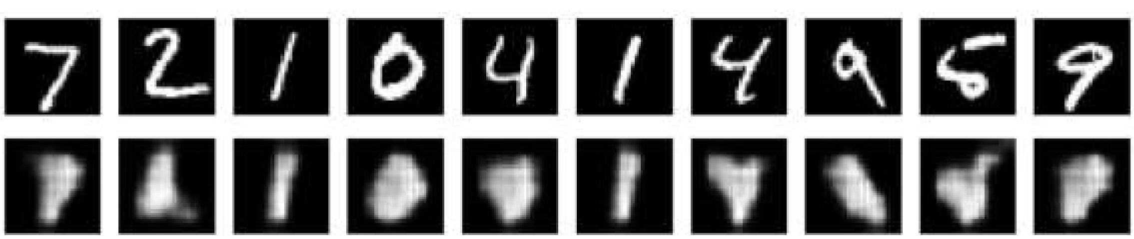

Figure 4. C

1.5 Complex CNN Model (Model 5 & 6)

The above model can be improved using various techniques like changing optimizer, loss function, Kernel Size, Filters, Epochs and adding new layers. Various permutations and combinations were worked upon and following results were found.

- Optimizer: - Tried using different optimizers ranging from Stochastic Gradient Descent, Adam, Adagrad, Adadelta, RMS Prop and Momentum. Adam and RMS gave the best results. The lowest loss captured was with Adam was 0.9 after 20 Iterations, which is represented in Model 1.
- Loss Function: - Both binary cross entropy and MSE models were tested. Though got better loss rate using MES, it's not suitable for image text classification, which was evident from blurry output. Finally, binary cross entropy was adapted shown in Model 1.
- Batch and Epoch: - No significant improvement was found by increasing Batch size though with increase in epoch there was a slight improvement in loss.
- Filter, Kernel Sizes and Layer: - Saw a increase in performance with changes in filter and kernel sizes. Though increased layer didn't prove much beneficial. It can be seen in upcoming models.
- Libraries: - By using 'keras.library' instead of 'tf.keras.library' showed a blizzard but significant improvement by half due to increase in training size from 469 to 60,000.

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernal	Activation	Size
Convolution	0	4	(64,32,32,32)	(3,3)(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Convolution	1	4	(64,32,32,32)	(3,3)(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	784
Loss	0.2445					
Changing Kernel Size						
Convolution	0	4	(64,32,32,32)	(3,3)(5,5)(5,5)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Convolution	1	4	(64,32,32,32)	(3,3)(5,5)(5,5)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	784
Loss	**0.2421**					

Table 5

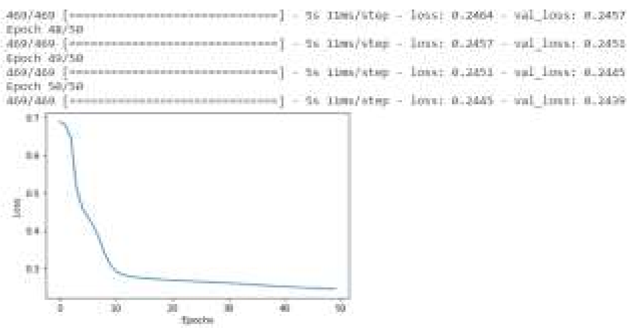

Figure 5. A (Kernel (3,3))

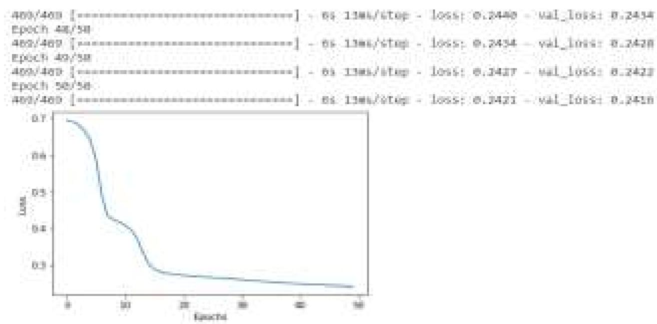

Figure 5. B (Kernel (5,5))

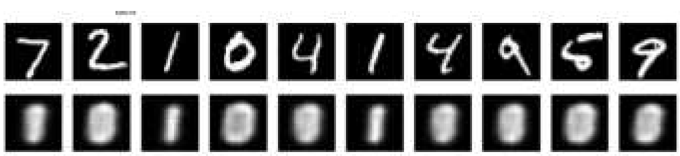

Figure 5. A (Kernel (3,3))

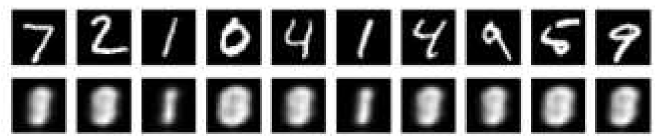

Figure 5. B (Kernel (5,5))

It can be noted from the above complex Models 2 that increasing the number of layers or the kernel size doesn't give much improvement to the model even though the filter size has been increased for improvement.

1.6 Complex CNN Model (Model 7 & 8)

Significant improvement was achieved differing the filter size and trying out different permutations.

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernal	Activation	Size	Training set
Convolution	0	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		469
Pooling	0			**(2,2)**			
Convolution	1	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		
Pooling	0			**(2,2)**			
Output		1	(1)	**(3,3)**	Sigmoid	784	
Loss	0.1423						
Using 'Keras library' Instead of 'TF (Keras library)'							
Convolution	0	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		60000
Pooling	0			**(2,2)**			
Convolution	1	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		
Pooling	0			**(2,2)**			
Output		1	(1)	**(3,3)**	Sigmoid	784	
Loss	**0.0696**						

Table 6

```
x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same')(encoded)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu')(x)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = tf.keras.models.Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Figure 6. A

```
x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(128, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Figure 6. B

```
Epoch 95/100
469/469 [==============================] - 33s 70ms/step - loss: 0.1445 - val_loss: 0.1430
Epoch 96/100
469/469 [==============================] - 33s 70ms/step - loss: 0.1440 - val_loss: 0.1425
Epoch 97/100
469/469 [==============================] - 33s 70ms/step - loss: 0.1436 - val_loss: 0.1421
Epoch 98/100
469/469 [==============================] - 33s 69ms/step - loss: 0.1432 - val_loss: 0.1417
Epoch 99/100
469/469 [==============================] - 33s 70ms/step - loss: 0.1427 - val_loss: 0.1412
Epoch 100/100
469/469 [==============================] - 33s 70ms/step - loss: 0.1423 - val_loss: 0.1408
```

Figure 6. C

```
Epoch 97/100
60000/60000 [==============================] - 9s 158us/step - loss: 0.0697 - val_loss: 0.0696
Epoch 98/100
60000/60000 [==============================] - 10s 159us/step - loss: 0.0697 - val_loss: 0.0695
Epoch 99/100
60000/60000 [==============================] - 9s 157us/step - loss: 0.0696 - val_loss: 0.0698
Epoch 100/100
60000/60000 [==============================] - 9s 158us/step - loss: 0.0696 - val_loss: 0.0688
```

Figure 6. D

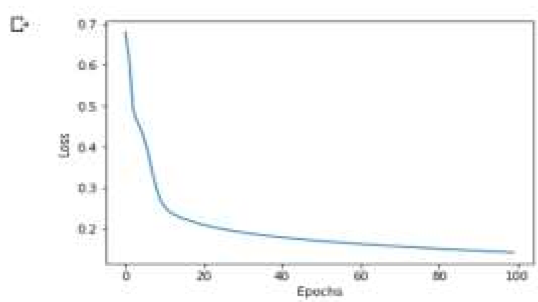

Figure 6. E

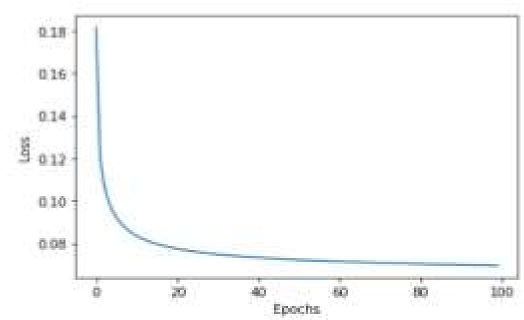

Figure 6. f

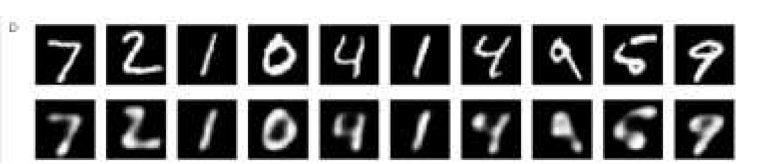

Figure 6. G

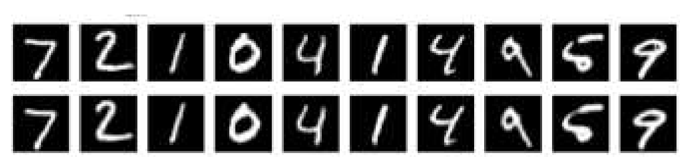

Figure 6. H

The above Table 6 and figures show the best two models using CNN architecture by trying out different permutations and combinations. The two models just differ by the datasets used one with 469 and other with 60000. This change in dataset is due to the direct use of 'keras.library' instead of 'tf.keras.library'. We finally get a loss of **0.0696.**

PART 2

2. Denoising

In this session we check the above model's performance with noisy dataset. For getting noisy dataset, we will generate synthetic noisy digits, we just apply a gaussian noise matrix and clip the images between 0 and 1. We added the noise to both testing and training dataset as follows. We use these datasets to train the model. Our aim is to find models performance in a noisy real time environment. Following shows the performance of dense layer autoencoder and convolutional autoencoder to work on an image denoising problem.

2.1 Basic Dense model 1 & 2

We first check the performance of noisy data on dense models. We had reshaped the model and the noisy datasets to make it compatible, below are the performance of these models.

Type	Encoder/Decoder (0,1)	Layers	Intermediate Dimensions	Activation	Size
	Basic Dense Model 2				
Dense	0	1	128	ReLU	
Output	0	ReLU			128
Dense	1	1	128	ReLU	
Output	1	Sigmoid			784
Loss	**0.1497**				
	Multilayer Dense Model 2				
Dense	1	3	(128,64,32)	ReLU	
Dense	1	3	(128,64,32)	ReLU	
Output	1	Sigmoid			784
Loss	**0.1573**				

Table 7

```
Epoch 97/100
235/235 [==============================] - 1s 3ms/step - loss: 0.1496 - val_loss: 0.1612
Epoch 98/100
235/235 [==============================] - 1s 4ms/step - loss: 0.1497 - val_loss: 0.1614
Epoch 99/100
235/235 [==============================] - 1s 3ms/step - loss: 0.1497 - val_loss: 0.1616
Epoch 100/100
235/235 [==============================] - 1s 3ms/step - loss: 0.1497 - val_loss: 0.1618
```

Figure 7. A

```
Epoch 96/100
235/235 [==============================] - 1s 4ms/step - loss: 0.1578 - val_loss: 0.1737
Epoch 97/100
235/235 [==============================] - 1s 4ms/step - loss: 0.1575 - val_loss: 0.1736
Epoch 98/100
235/235 [==============================] - 1s 4ms/step - loss: 0.1574 - val_loss: 0.1738
Epoch 99/100
235/235 [==============================] - 1s 4ms/step - loss: 0.1573 - val_loss: 0.1744
Epoch 100/100
235/235 [==============================] - 1s 4ms/step - loss: 0.1573 - val_loss: 0.1735
```

Figure 7. B

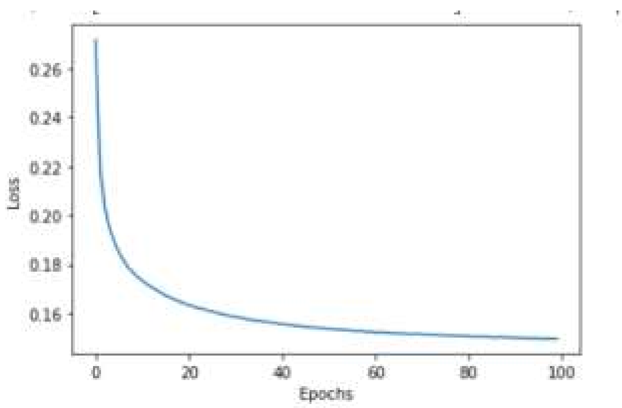

Figure 7. C

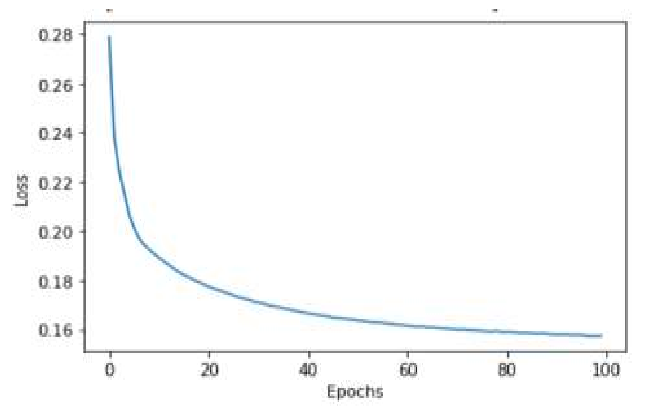

Figure 7. C

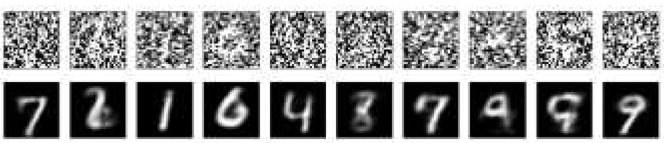

Figure 7. D

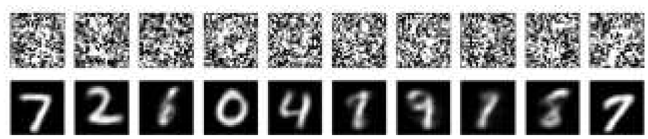

Figure 7. E

It can be seen from the above to models that increasing the layers, kernels and filter sizes doesn't improve the model to greater extent, when dealing with a noisy input.

2.2 CNN model 3

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernal	Activation	Size
Convolution	0	3	(16,8,8)	(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Convolution	1	3	(16,8,8)	(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	784
Loss	**0.2643**					

Table 8

```
Epoch 97/100
469/469 [==============================] - 3s 5ms/step - loss: 0.2645 - val_loss: 0.2644
Epoch 98/100
469/469 [==============================] - 3s 5ms/step - loss: 0.2645 - val_loss: 0.2643
Epoch 99/100
469/469 [==============================] - 3s 5ms/step - loss: 0.2644 - val_loss: 0.2642
Epoch 100/100
469/469 [==============================] - 3s 5ms/step - loss: 0.2643 - val_loss: 0.2641
```

Figure 8. A

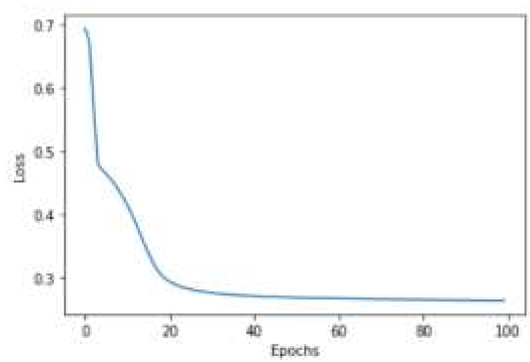

Figure 8. B

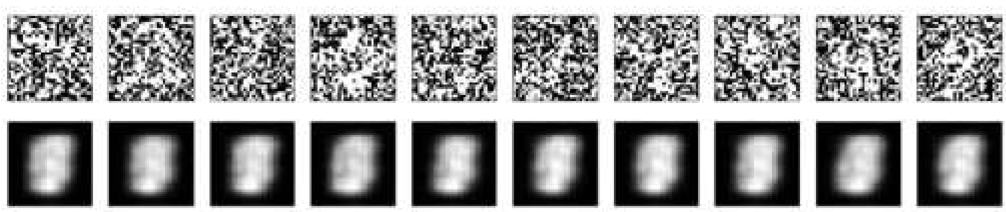

Figure 8. C

The above complex model performed similarly with normal and noisy dataset but performed worse compared to basic and multimodal dense layer network. We got a overall loss of **0.246** while training noisy induced dataset whereas **0.2152** with normal dataset. Further changes in parameter were required in both the model to improve the output.

2.3 Complex CNN Model (Model 4 & 5)

Significant improvement was achieved differing the filter size and trying out different permutations and combinations. The Complex CNN performed slightly better compared to dense network when inter epoch cycle was increased to 60000. Otherwise the with inter epoch cycle rate 469 dense layer can be seen performing better that the basic and complex CNN. This is evident from the below figure and tables.

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernel	Activation	Size	Training set
Convolution	0	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		469
Pooling	0			**(2,2)**			
Convolution	1	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		
Pooling	0			**(2,2)**			
Output		1	(1)	**(3,3)**	Sigmoid	784	
Loss	**0.2530**						
Using 'Keras library' Instead of 'TF (Keras library)'							
Convolution	0	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		60000
Pooling	0			**(2,2)**			
Convolution	1	3	(128,64,32)	(3,3)(3,3)(3,3)	ReLU		
Pooling	0			**(2,2)**			
Output		1	(1)	**(3,3)**	Sigmoid	784	
Loss	**0.1410**						

Table 9

```
Epoch 97/100
469/469 [==============================] - 6s 13ms/step - loss: 0.2537 - val_loss: 0.2534
Epoch 98/100
469/469 [==============================] - 6s 13ms/step - loss: 0.2535 - val_loss: 0.2532
Epoch 99/100
469/469 [==============================] - 6s 13ms/step - loss: 0.2532 - val_loss: 0.2530
Epoch 100/100
469/469 [==============================] - 6s 13ms/step - loss: 0.2530 - val_loss: 0.2526
```

Figure 9. A

```
Epoch 97/100
60000/60000 [==============================] - 7s 111us/step - loss: 0.1413 - val_loss: 0.1519
Epoch 98/100
60000/60000 [==============================] - 7s 111us/step - loss: 0.1413 - val_loss: 0.1515
Epoch 99/100
60000/60000 [==============================] - 7s 111us/step - loss: 0.1412 - val_loss: 0.1531
Epoch 100/100
60000/60000 [==============================] - 7s 111us/step - loss: 0.1410 - val_loss: 0.1519
```

Figure 9. B

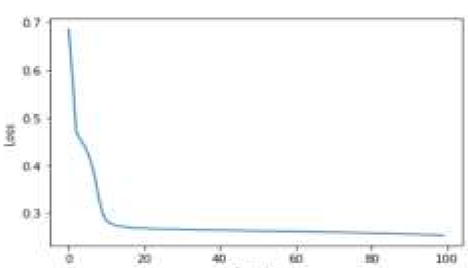

Figure 9. C

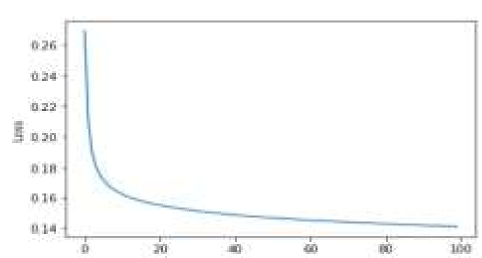

Figure 9. D

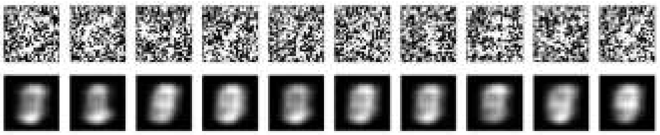

Figure 9. E

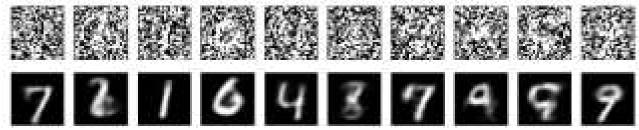

Figure 9. F

The above figures represent the incurred lose during training, loss to epoch ratio and the Input vs output figures for the two above mentioned model.

PART 3

3. Text Reconstruction Application

Our aim in this part is to create new models for reconstruction of text images. We have 3 sets of data train, test and test cleaned. We use train and clean test dataset to train our network and predict it with noisy test dataset. We use multiple models and try to minimize loss and reconstruct text. The loss function used for model comparison is **Root Mean Square Error**.

3.1 Basic Dense-Model 1 (Text Reconstruction)

First we use our basic dense layer network to reconstruct the noised text images. The minimum loss incurred using this model is **0.34**. Following figures 10.A, 10.B, 10.C shows the loss incurred, loss to epoch graph and output for basic dense model (model 1).

Type	Encoder/Decoder (0,1)	Layers	Intermediate Dimensions	Learning Rate	Activation	Size
Dense	0	1	128	1e-2	ReLU	
Output	0	ReLU				128
Dense	1	1	128	1e-2	ReLU	
Output	1	ReLU				226800
Loss (RMSE)	0.3415					

Table 10

Figure 10. A

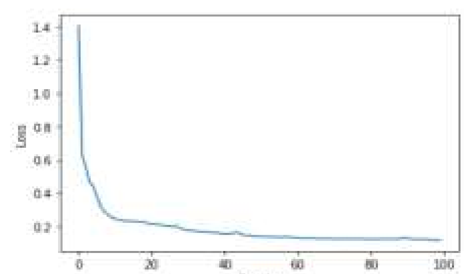

Figure 10. B

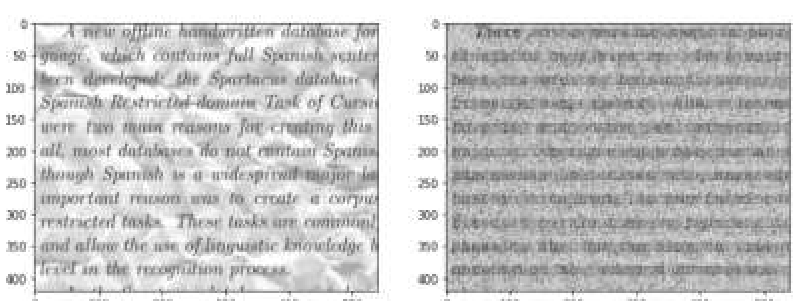

Figure 10. C

3.2 Basic CNN-Model 2 (Text Reconstruction)

The basic CNN model does do a decent job in removing the noise. This network uses 3 convolution layers followed by 3 pooling layers for each encoder and decoder. By comparing the loss after 100 epochs we can conclude that basic dense layer performed better than basic CNN. Following figures 11.A, 11.B, 11.C shows the loss incurred, loss to epoch graph and output for basic CNN model (model 2).

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernal	Activation	Size
Convolution	0	3	(16,8,8)	(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Convolution	1	3	(16,8,8)	(3,3)(3,3)(3,3)	ReLU	
Pooling	0			**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	226800
Loss (RMSE)	**0.2149**					

Table 11

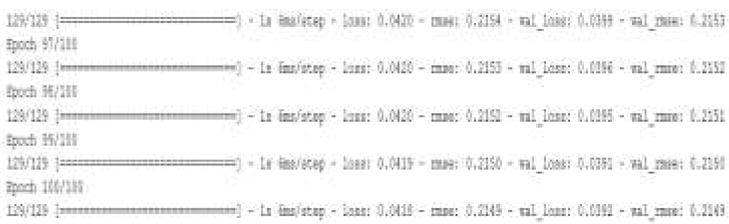

Figure 11. A

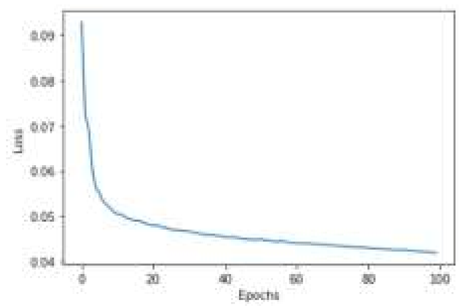

Figure 11. B

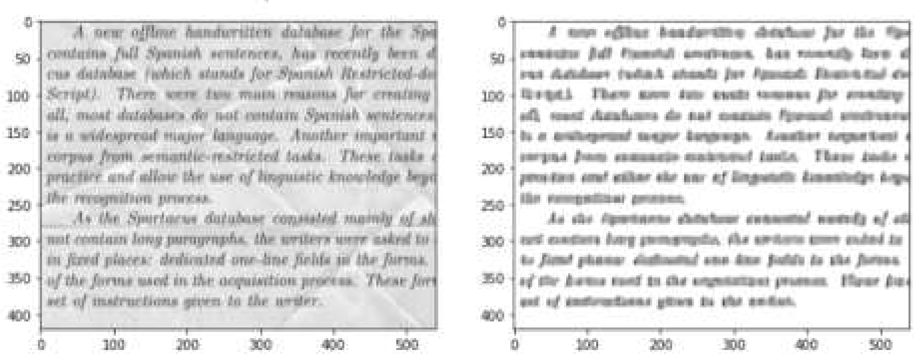

Figure 11. C

3.2 CNN Complex-Model 3 & 4 (Text Reconstruction)

We were able to almost fully remove noise and recreate text using our complex CNN models. Both 3- and 4- CNN architecture was able to generate good results. We decreased the up sampling and down sampling layers to minimize the losses in image. The best model recorded **0.0367** as the final loss after 100 epochs, whereas model 3 could only minimize the loss to **0.0485.** We used Root Mean Square Error to compute losses in both the models. Model 3 was relative faster than model 4 due to the presence of low filter intermediate layers.

Type	Encoder/Decoder (0,1)	Layers	Filters	Kernel	Activation	Size
Convolution	0	2	(128, 8)	(3,3)(3,3)	ReLU	
Pooling	0	1		**(2,2)**		
Convolution	1	2	(8,128)	(3,3)(3,3)	ReLU	
Pooling	0	1		**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	226800
Loss (RMSE)	**0.0485**					
Convolution	0	2	(128,32)	(3,3)(3,3)	ReLU	
Pooling	0	1		**(2,2)**		
Convolution	1		(32,128)	(3,3)	ReLU	
Pooling	1	1		**(2,2)**		
Output		1	(1)	**(3,3)**	Sigmoid	226800
Loss (RMSE)	**0.0367**					

Table 12

<u>Complex Model 3</u>

Following figures 12.A, 12.B, 12.C shows the loss incurred, loss to epoch graph and output for complex model 3.

```
Epoch 97/100
17/17 [==============================] - 4s 263ms/step - loss: 0.0023 - root_mean_squared_error: 0.0483
Epoch 98/100
17/17 [==============================] - 4s 262ms/step - loss: 0.0024 - root_mean_squared_error: 0.0492
Epoch 99/100
17/17 [==============================] - 4s 262ms/step - loss: 0.0025 - root_mean_squared_error: 0.0493
Epoch 100/100
17/17 [==============================] - 4s 261ms/step - loss: 0.0023 - root_mean_squared_error: 0.0485
```

Figure 12. A

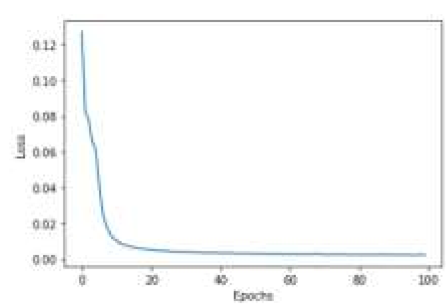

Figure 12. B

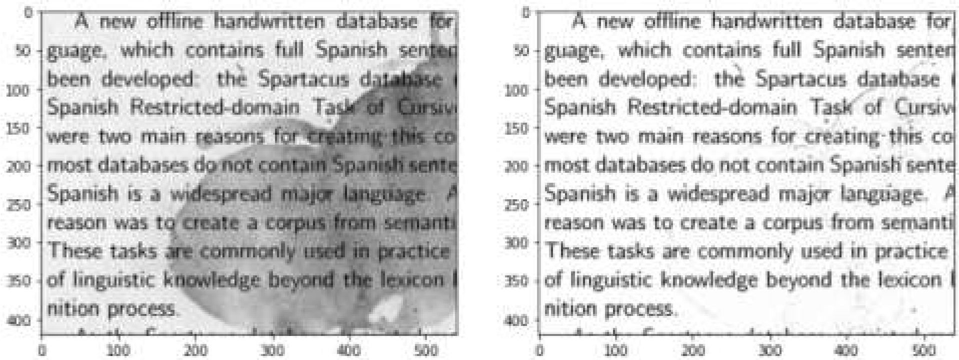

Figure 12. C

Complex Model 4

Following figures 12.E, 12.F, 12.G shows the loss incurred, loss to epoch graph and output for complex model 4.

```
17/17 [==============================] - 6s 332ms/step - loss: 0.0014 - root_mean_squared_error: 0.0366
Epoch 97/100
17/17 [==============================] - 6s 333ms/step - loss: 0.0014 - root_mean_squared_error: 0.0364
Epoch 98/100
17/17 [==============================] - 6s 332ms/step - loss: 0.0014 - root_mean_squared_error: 0.0369
Epoch 99/100
17/17 [==============================] - 6s 332ms/step - loss: 0.0013 - root_mean_squared_error: 0.0360
Epoch 100/100
17/17 [==============================] - 6s 331ms/step - loss: 0.0013 - root_mean_squared_error: 0.0367
```

Figure 12. E

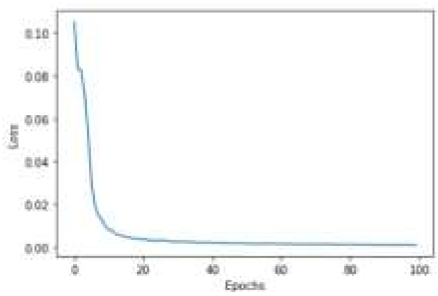

Figure 12. F

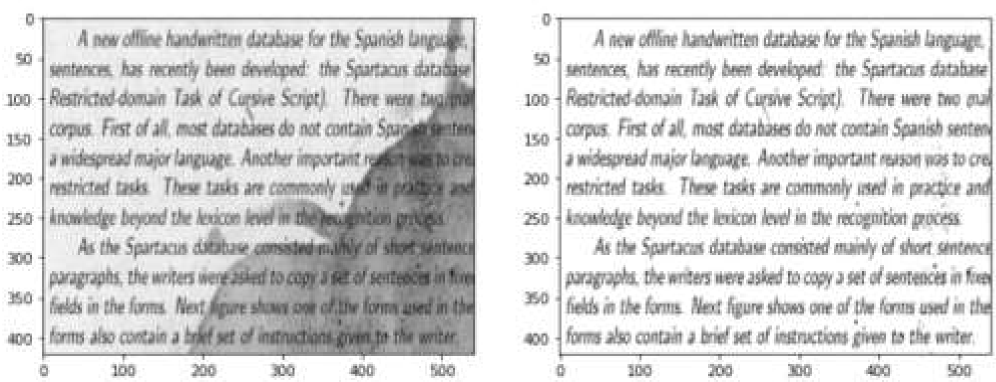

Figure 12. G

Its is evident from the above model that after increasing the filter size and by designing a complex CNN model with a smaller number of down sampling and up sampling layers yield significant better results. Adding too many convolution layers won't help minimizing loss, moreover it may worsen the performance. Increasing the filter size also increases the model taring time due to use of more parameters while training. Lower filter sized layers fasten the network considerably.

4 Models Comparison

4.1 Basic Autoencoder

Type	Loss
Basic Dense (Model 1)	3.82
Dense (Model 2)	1.475
Dense (Model 3)	0.981
CNN Basic (Model 4)	0.2156
CNN Complex (Model 5)	0.2445
CNN Complex (Model 6)	0.2421
CNN Complex (Model 7)	0.1423
CNN Complex (Model 8)	0.0696

Table 13. A

4.2 Denoising Models

Type	Loss
Basic Dense (Model 1)	0.1497
Dense (Model 2)	0.1573
CNN Basic (Model 3)	0.2643
CNN Complex (Model 4)	0.2530
CNN Complex (Model 5)	0.1410

Table 13. B

4.3 Text Reconstruction

Type	Loss (RMSE)
Basic Dense (Model 1)	0.3415
CNN Basic (Model 3)	0.2149
CNN Complex (Model 4)	0.0485
CNN Complex (Model 5)	0.0367

Table 13. C

Conclusion: -

All the above model was improved using various techniques like changing optimizer, loss function, Kernel Size, Filters, Epochs and adding new layers. Overall CNN's performed better than dense layer architecture in most of the parts. In most cases simpler the network, faster they trained.