Mini Project 6: Gemerative Network Models

Siddhant Prakash
1211092724
sprakas9@asu.edu

***Abstract*—Report for mini-project 6 to study generative network models including auto-encoders, generative adversarial networks and de-noising auto-encoders.**

1. Introduction

In this mini-project we are provided with the YANN [1] toolbox using which, we experiment with different generative neural networks model. Auto-encoders and Generative Adversarial Networks (GANs) are two such models. The mini project is divided in three parts. In the first part, we run the auto-encoder tutorial from the YANN tutorial pantry. We are provided with the MNIST [2] dataset and we run the base tutorial to generate the 28X28 images learning the weights through the encoder-decoder network.

The second part deals with Generative Adversarial Network. We run the base tutorial for Generative Adversarial Networks and try to generate the images as close as to the input dataset images. In the third and the last part, we create a denoising auto-encoder to which the input is an image from the given dataset, added with random noise. The goal of the auto-encoder is to denoise the image and output the images similar to the original dataset.

2. Auto-encoder Tutorial

In the auto-encoder tutorial, we are provided with two types of auto-encoder, viz. a simple auto-encoder with just the fully connected layers and a deep convolutional auto-encoder. Intially, we run both auto-encoders, but eventually we manipulate and test the structure of convolutional auto-encoders to see effect of different layers as well as code-words on the auto-encoder output. The network architecture for the base convolutional auto-encoder in the tutorial is given in Table 1.

The training was done on an Ubuntu 14.04 system with 1.7 GHz Intel Core i5 processor, 8GB 1600 MHz DDR3L SD-RAM and hosting Nvidia GeForce 840M GPU. The memory required to run the auto-encoder described in Table 1 on this system was approximately 700-800 MiB. It took an average of 30-35 minutes to train this auto-encoder for 20 epochs having 2 eras of 10 epochs each. The auto-encoder was trained using 'nesterov' momentum [4] with an 'rmsprop' optimizer [5] and the activation used was 'ReLU' for every layer.

(a) Input Image (b) Generated Image

Figure 1: Generating Image using Auto-encoder Architecture 1

2.1. Experiments

The base tutorial was run intially with early terminate condition set as 'True'. But due to the frequent early stops, we couldn't get a good learning of the weights due to it, so we set it to 'False' and ran the auto-encoder for 20 epochs having 2 eras of 10 epochs each. We also experimented with changing the architecture by increasing and removing the layers in the auto-encoder. The architectures we tested are listed in Table 2 and Table 3.

As expected, the memory utilization in Architecture II increased significantly with introduction of extra layers and peaked at a maximum of about 1350 MiB. The time taken to train the network was also significantly higher and averaged around 85-90 minutes as compared to 30 minutes of architecture I. With architecture III, the time and memory complexity was almost the same as that of Architecture I, but the difference in performance was much higher.

2.2. Results

The best results obtained was for architecture I, which was able to converge to a very good generated image of the input signal. We show the comparison of input image and the genrated image in Figure 1. Although, we explored deeper models with architecture II and III, we could not get a better representation than what we got for architecture I. We also experimented by changing the length of the codeword of the architecture. The filters learnt via different sized codewords, which is the number of neurons in encoder layer, is shown in Figure 3

The final error for all three architectures learnt with the same parameters for the same number of epochs (10,10),

Layer Name.	Input Shape	Receptive Field	No. of Feature Maps	Type of Neurons	Output Shape
Conv	28X28X1	5X5	20	Convolutional	24X24X20
	24X24X20	1X1			24X24X20
Flatten	24X24X20			Flatten	11520
Hidden-encoder	11520		1200	Fully Connected	1200
Encoder	1200		128	Fully Connected	128
Decoder	128		1200	Fully Connected	1200
Hidden-decoder	1200		11520	Fully Connected	11520
Unflatten	11520			Unflatten	20X24X24
Deconv	20X24X24	5X5	20	De-convolutional	1X28X28

TABLE 1: Auto-encoder Architecture I

Layer Name.	Input Shape	Receptive Field	No. of Feature Maps	Type of Neurons	Output Shape
Conv1	28X28X1	5X5	20	Convolutional	24X24X20
	24X24X20	1X1			24X24X20
Conv2	28X28X1	5X5	60	Convolutional	20X20X60
	20X20X60	1X1			20X20X60
Flatten	20X20X60			Flatten	24000
Hidden-encoder1	24000		1200	Fully Connected	1200
Hidden-encoder2	1200		256	Fully Connected	256
Encoder	256		128	Fully Connected	128
Decoder	128		256	Fully Connected	256
Hidden-decoder2	256		1200	Fully Connected	1200
Hidden-decoder1	1200		24000	Fully Connected	24000
Unflatten	24000			Unflatten	60X20X20
Deconv2	60X20X20	5X5	60	De-convolutional	20X24X24
Deconv1	20X24X24	5X5	20	De-convolutional	1X28X28

TABLE 2: Auto-encoder Architecture II

Layer Name.	Input Shape	Receptive Field	No. of Feature Maps	Type of Neurons	Output Shape
Conv	28X28X1	5X5	20	Convolutional	24X24X20
	24X24X20	1X1			24X24X20
Flatten	24X24X20			Flatten	11520
Hidden-encoder1	11520		1200	Fully Connected	1200
Hidden-encoder2	1200		256	Fully Connected	256
Encoder	256		128	Fully Connected	128
Decoder	128		256	Fully Connected	256
Hidden-decoder2	256		1200	Fully Connected	1200
Hidden-decoder1	1200		11520	Fully Connected	11520
Unflatten	11520			Unflatten	20X24X24
Deconv	20X24X24	5X5	20	De-convolutional	1X28X28

TABLE 3: Auto-encoder Architecture III

Architecture	RMSE
I	**0.163**
II	0.308
III	0.3

TABLE 4: Training Error for Different Auto-encoder Architecture

can be seen in Table 4. The comparison of the final filters learnt by the three architectures are shown in Figure 2.

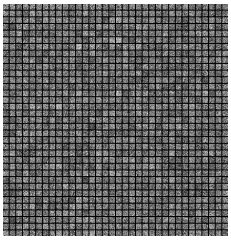
3. Generative Adversarial Networks Tutorial

The GAN tutorial on YANN consists of 4 different architectures for training the GAN model, viz. shallow GAN, Deep GAN, Deep De-Convolutional GAN (DC-GAN) and Deep De-Convolutional LS-GAN (DC-LSGAN). We are provided with the choice of 2 datasets, MNIST and CIFAR-10 [3] to train the models. Since, there was an issue with the deep convolutional GANs with MNIST dataset, we train the DC-GAN and DC-LSGAN with CIFAR-10 dataset.

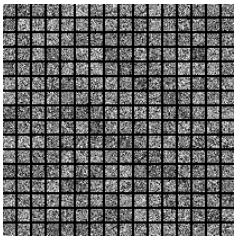
3.1. Experiments

The base architecture we used for generating images with DC-GANs is summarized in Figure 10. To disucss the 4 different type of GANs, we will briefly overview the architecture and the parameters. Shallow GAN has only fully-connected layers in both the discriminator and generator models. While the generator model has the architecture 784-800-500-10, the discriminator model has a shape of 32-784-800-1. The generator has a softmax classifier, which classifies the images geenrated and validates if its the correctly labelled image generated.

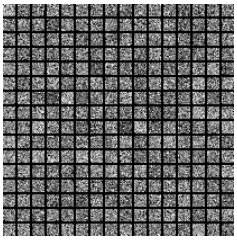
While, the Deep GAN model was similar to Shallow GAN model, in terms of having only fully connected layer,



(a) Filter learnt for Architecture 1

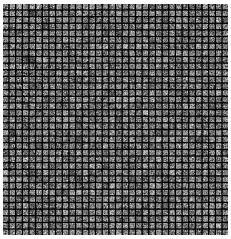


(b) Filter learnt for Architecture 2

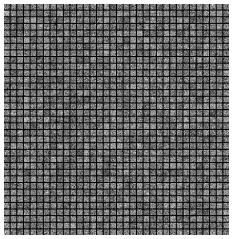


(c) Filter learnt for Architecture 3

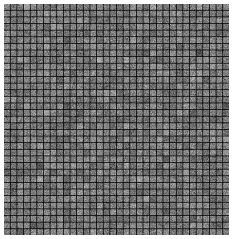
Figure 2: Comparing Filters learnt using Auto-encoder Architecture 1, 2 and 3 for 20 epochs and same hyper-parameters



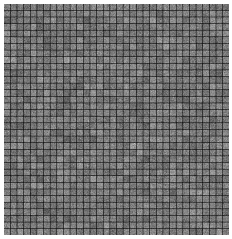
(a) Filter learnt with codeword size = 64



(b) Filter learnt with codeword size = 128



(c) Filter learnt with codeword size = 256



(d) Filter learnt with codeword size = 512

Figure 3: Comparing Filters learnt using Auto-encoder Architecture 1 with different Codeword lengths

the generator has an architecture of 784-240-240-1-10 while the discriminator has an architecture of 10-1200-1200-784-240-240-1. This model also uses a softmax classifier to label the classes of images generated. We train the networks using 'nesterov' momentum [4] with 'rmsprop' optimizer [5]. The activation function used is 'ReLU'.

We also experimented with Deep De-convolutional GANs (DC-GANs) and DC-LSGANs. The base architecture of both the models are same with the difference beign that while DC-GANs uses log-likelihood as the error function, DC-LSGANs use 'Least Squared' error for discriminating between the generated signals. Although, both these implementation had issues, we still ran them with Cifar-10 data set as input and report the results obtained.

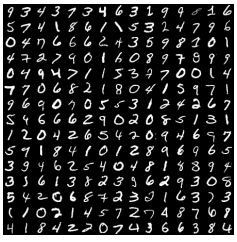
The training was done on an Ubuntu 14.04 system with 1.7 GHz Intel Core i5 processor, 8GB 1600 MHz DDR3L SD-RAM and hosting Nvidia GeForce 840M GPU. The training time for Shallow GAN was approximately 3 minutes and the training time for deep GAN was approximately 7 minutes.

3.2. Results

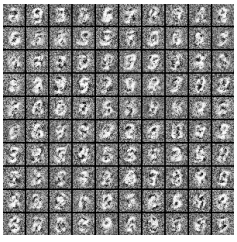
The final training accuracy, the discriminator and generator error obtained by the objective function of the softmax classifier for both Shallow and Deep GANs architecture after learning the model for 20 epochs is summarised in Table 5. The generated images for both Shallow GAN and Deep GAN models si shown in Figure 4 and 5 respectively. It must be noted that while Shallow GAN produces very noisy

	Shallow Model	Deep Model
Training Accuracy	99.738	97.714
Discriminator Sigmoid	0.843	0.032
Generator Sigmoid	0.141	0.989

TABLE 5: Training Error for Shallo and Deep GAN Model

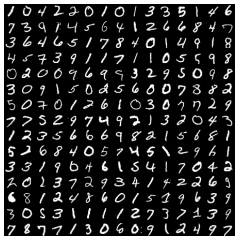


(a) Input Image



(b) Generated Image

Figure 4: Generating Image using Shallow GAN Architecture



(a) Input Image

(b) Generated Image

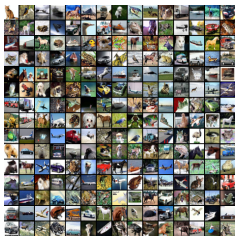
Figure 5: Generating Image using Deep GAN Architecture

	DC-GAN Model	DC-LSGAN Model
Training Accuracy	95.2075	97.714
Discriminator Sigmoid	0.194	0.032
Generator Sigmoid	0.714	0.989

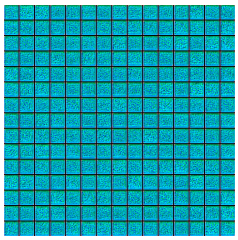
TABLE 6: Training Error for Shallow and Deep GAN Model

images, the images produced by Deep GAN looks pretty realistic but is actually almost the same image. The Deep GAN architecture model is actually one of the configuration in which we see mode collapse.

For DC-GANs and DC-LSGANs, the final training accuracy, the discriminator and generator error obtained by their respective objective function is summarised in Table 6. Also a comparison of the input image and the generated image for both DC-GANs and DC-LSGANs is shown in Figure 6 and 7 respectively. While, DC-LSGAN was able to train for the complete 20 epochs, we faced some error during training of DC-GAN and it had to be terminated at epoch 6. We show you the output we got at the last epoch of each model.

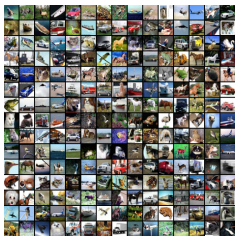


(a) Input Image

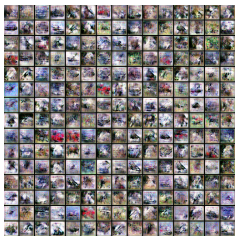


(b) Generated Image

Figure 6: Generating Image using DC-LSGAN Architecture



(a) Input Image



(b) Generated Image

Figure 7: Generating Image using DC-GAN Architecture

We show the various filters learnt by all the four GAN models in Figure 8. The general De-convolutional GAN architecture can be understood from the Figure 10. We also experimented with a few DC-GAN model with MNIST data set and discovered some configurations which demonstrated mode collapse. We would like to show the generated images along with the filters which learnt those in Figure 9.

4. De-noising Auto-encoders

For De-noising encoder, we were supposed to use the merge layer provided by the YANN toolbox to introduce noise in a given image and then use the de-noising auto-encoder to train the network to de-noise the image.

We created the random layer, which is another layer provide by the YANN toolbox and used it to create two different noise distribution. The first being a normal distribution and the second being a gaussian distribution. We merged the input layer with the random layer to provide the input to the network and fed to the autoencoder to train the network for denoising.

The training was done on an Ubuntu 16.04 server with 3.5 GHz Intel E5-1620 processor, 16GB 1200 MHzDDR3L SD-RAM and hosting Nvidia K620 GPU. All training and testing, memory and time is reported on GPU.

4.1. Experiments

We use the same architecture of the auto-encoder as described in Architecture 1. We trained the auto-encoder network with the normal distribution noise added with an early terminate flag set as True. We again used the same network architecture to train the Gaussian noise added images, and this time, we switched off the early terminate flag.

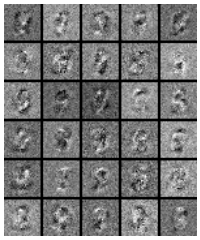
Instead of the architecture, we focused on varying the codeword length of the auto-encoders and we tested the auto-encoders for a codeword length of [64, 128, 256, 512 and 1024].

4.2. Results

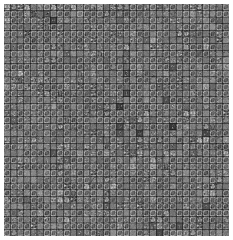
Codeword Length	RMSE
64	0.3
128	0.3
256	0.3
512	0.3

TABLE 7: Training error for Normal Distribution noise added

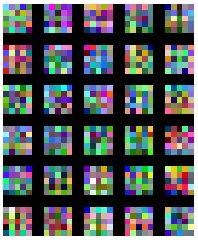
The final training error for normal noise added image is summarised in Table 7 while the final training error for Gaussian noise added image is summarissed in Table 8. We notice that most of the auto-encoders are not able to learn a good enough stable representation within 20 epochs except for Gaussian noise with a codeword of 512. We show the generated de-noised learnt along with the input noisy image in Figure 11. We do notice that some of the models learn



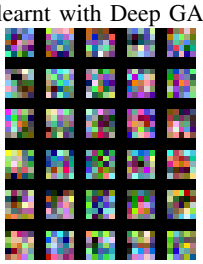
(a) Filter learnt with Shallow GAN Model



(b) Filter learnt with Deep GAN Model

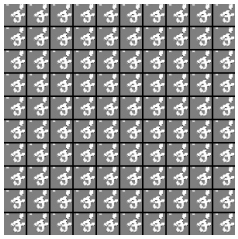


(c) Filter learnt with DC-LSGAN Model

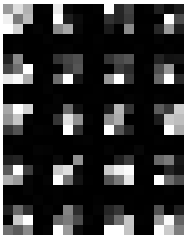


(d) Filter learnt with DC-GAN Model

Figure 8: Comparing Filters learnt using GANs Model with Shallow, Deep, DC-LSGAN and DC-GAN respectively



(a) Image generated in mode collapse using DC-GAN Architecture



(b) Filters learnt in mode collapse using DC-GAN Architecture

Figure 9: Mode Collapse using DC-GAN Architecture

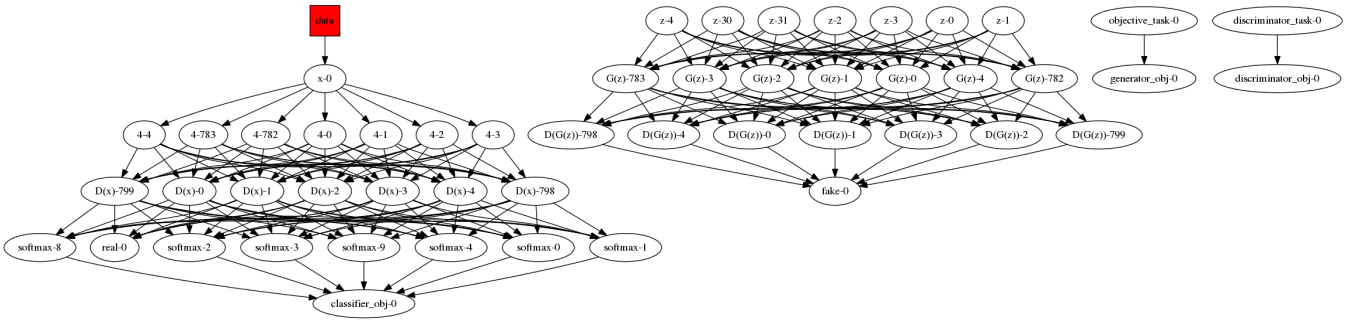


Figure 10: Deep Deconvolutional Generative Adversarial Network Architecture

Codeword Length	RMSE
64	0.3
128	0.3
256	0.3
512	0.22
1024	0.3

TABLE 8: Training error for Gaussian Distribution noise added

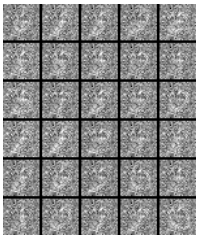
the representation at an early stage, but breaks down as the epochs increases.

The various filter learnt for de-noising auto-encoder along with the noisy input and the generated images are shown in Figure 12, 13, 14 and 15. The filter learnt for Gaussian noise added with varying codewords are shown in Figure 16, 17, 18, 19 and 20.

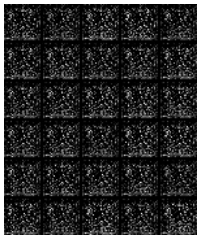
(a) Input Noise Image

(b) Generated Image

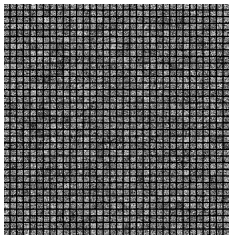
Figure 11: Generated Denoised image using Auto-encoder Architecture 1 with Codeword of 512



(a) Input Noisy Image

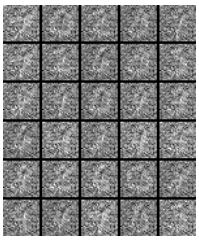


(b) Generated De-Noised Image

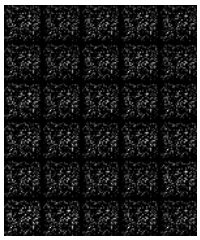


(c) Filter Learnt for the given architecture

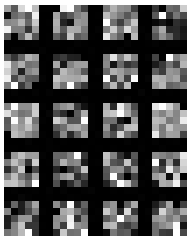
Figure 12: De-noising Auto-encoder results for Normal Noise added to image with Codeword length 64



(a) Input Noisy Image

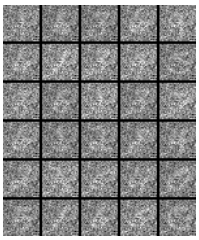


(b) Generated De-Noised Image

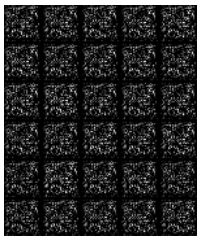


(c) Filter Learnt for the given architecture

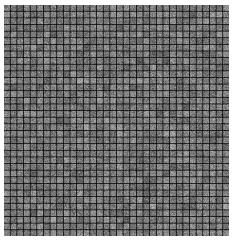
Figure 13: De-noising Auto-encoder results for Normal Noise added to image with Codeword length 128



(a) Input Noisy Image



(b) Generated De-Noised Image



(c) Filter Learnt for the given architecture

Figure 14: De-noising Auto-encoder results for Normal Noise added to image with Codeword length 256

References

[1] Yet Another Neural Network [YANN] Toolbox. https://github.com/ragavvenkatesan/yann

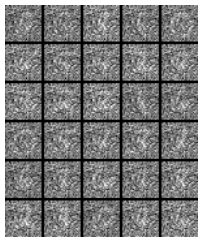
[2] LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database of handwritten digits." (1998).

[3] Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton. "The CIFAR-10 dataset." 2013-11-14]. http://www. cs. toronto. edu/ kriz/cifar. html (2014).

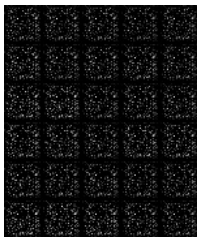
[4] Nesterov, Yurii. "A method of solving a convex programming problem with convergence rate O (1/k2)." Soviet Mathematics Doklady. Vol. 27. No. 2. 1983.

[5] Tieleman, Tijmen, and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural networks for machine learning 4.2 (2012).

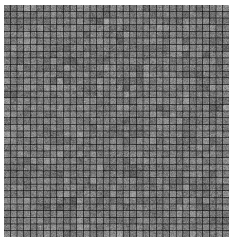
[6] Ioffe, Sergey, and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015).



(a) Input Noisy Image

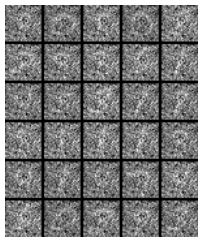


(b) Generated De-Noised Image

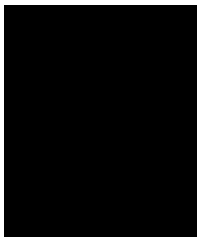


(c) Filter Learnt for the given architecture

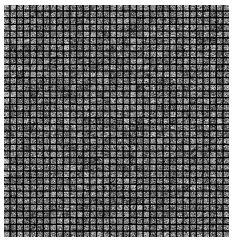
Figure 15: De-noising Auto-encoder results for Normal Noise added to image with Codeword length 512



(a) Input Noisy Image

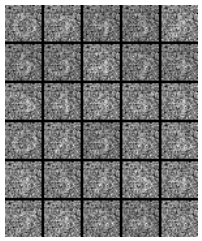


(b) Generated De-Noised Image

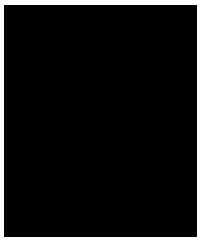


(c) Filter Learnt for the given architecture

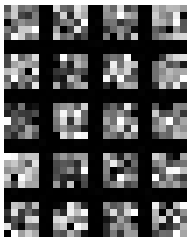
Figure 16: De-noising Auto-encoder results for Gaussian Noise added to image with Codeword length 64



(a) Input Noisy Image

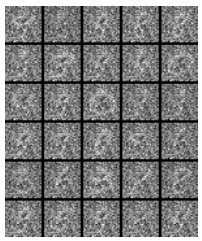


(b) Generated De-Noised Image

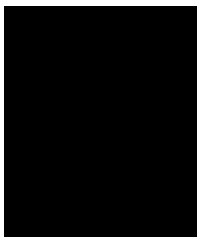


(c) Filter Learnt for the given architecture

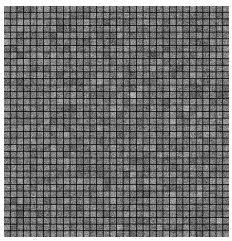
Figure 17: De-noising Auto-encoder results for Gaussian Noise added to image with Codeword length 128



(a) Input Noisy Image

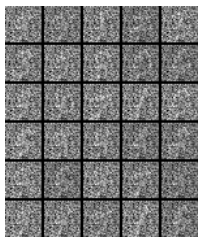


(b) Generated De-Noised Image



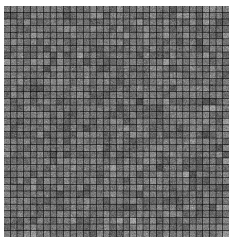
(c) Filter Learnt for the given architecture

Figure 18: De-noising Auto-encoder results for Gaussian Noise added to image with Codeword length 256



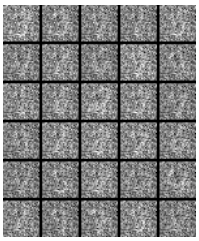
(a) Input Noisy Image

(b) Generated De-Noised Image

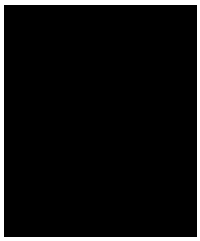


(c) Filter Learnt for the given architecture

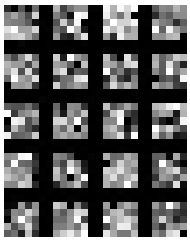
Figure 19: De-noising Auto-encoder results for Gaussian Noise added to image with Codeword length 512



(a) Input Noisy Image



(b) Generated De-Noised Image



(c) Filter Learnt for the given architecture

Figure 20: De-noising Auto-encoder results for Gaussian Noise added to image with Codeword length 1024