2021 Fall STATSM231A/CSM276A: Pattern Recognition and Machine Learning

Yaxuan Zhu
Department of Statistics
University of California, Los Angeles
9401 Bolter Hall., Los Angeles, California 90095
`yaxuanzhu@g.ucla.edu`

Abstract

In this course, you will learn multiple algorithms in machine learning. In general, for each homework, you are required to write the corresponding codes. We will provide a structure of code and a detailed tutorial online. Your job is (1) to deploy and understand the code. (2) Run the code and output reasonable results. (3) Make ablation study and parameter comparison. (4) Write the report with algorithms, experiment results and conclusions. For detailed instructions, please check the specification of each homework.

Homework submission forms

You have to submit both code and reports. For the code, zip everything into a single zip file. For the report, use this latex template. When submitting, no need to submit the tex file. Submit one single pdf file only.

1 Coding part: GAN and VAE

1.1 Problems

In this homework, we are asked to train two configurations of a Generative Adversarial Network (GAN) and one version of Variational Auto Encoder (VAE) on the MNIST dataset so that we can use the models to generate novel images. Since the goal of the assignment is the generation of distinct pixel values, the problem can be thought of as a regression problem.

1.2 Introduction: GAN

GAN, which is an acronym for Generative Adversarial Network is a deep learning architectıre that can be used to learn the distribution of a dataset in order to create novel examples that appear to be part of the existing dataset. This architecture was introduced by Ian J. Goodfellow et. al. in the paper Generative Adversarial Networks in 2014.

The structure of a GAN is made up of two architectures with opposite goals. The first architecture is referred to as the Generator. The goal of the generator is to create a data sample that appears to belong to the training dataset from a random input. The other architecture is referred to as the Discriminator. The goal of the discriminator is to determine if a sample data originated from the actual training dataset or if it was generated by the generator. These two architectures are trained adversarially. The generator attempts to create realistic samples to fool the discriminator and the discriminator attempts to perfectly distinguish between generated and original samples.

The structure of a GAN can be illustrated as follows:

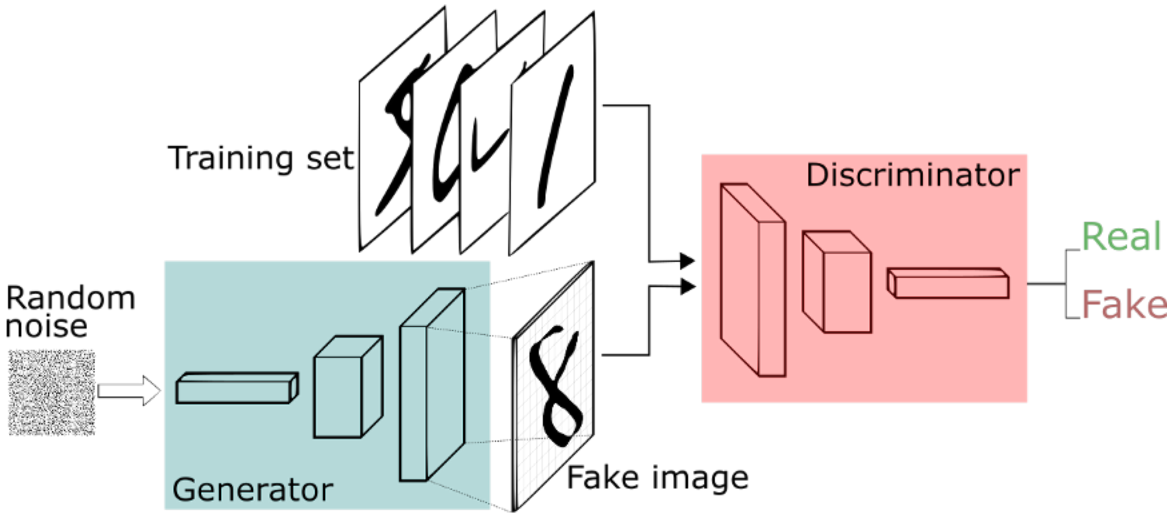

Figure 1: The illustration of the GAN structure

As can be seen in the GAN architecture, a random noise is given to the generator which produces a fake data sample. These samples are combined with data from the training set and given to the discriminator which is tasked with differentiating between generated and actual samples.

The mathematical representation of the objective function of a GAN is as follows:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\substack{\text{Discriminator output} \\ \text{for real data x}}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\substack{\text{Discriminator output for} \\ \text{generated fake data G(z)}}}) \right]$$

Here, the first expectation refers to the log probability of the discriminator predicting that the sampled data comes from the training set and the second expectation refers to the log probability of the discriminator predicting that the data generated by the generator is not from the training set. The best possible discriminator maximises this function while the best possible generator minimises this function. So, the updates of the discriminator are made with gradient ascent while the updates to the generator are made with gradient descent.

The training algorithm of a GAN can be expressed as follows:

Algorithm 1: The training algorithm of GAN

```
Result: Trained Generator and Discriminator
initialization;
for epochs do
    sample m noise samples: $z_1 ... z_m$;
    sample m training samples: $x_1 ... x_m$;
    Give the noise samples to the generator;
    Perform gradient ascent on the entire loss function to update the discriminator using both
      generated and actual data;
    Perform gradient descent on the second part of the loss function using the generated samples
      to update the generator;
end
```

After training, we ideally get a generator that can fool a good discriminator. In other words, the samples generated by the generator have a strong resemblance to the samples from the training set. In this case, we can generate novel samples by giving noise as input to the generator.

The structure of the generator and discriminator can vary based on the type of input data. For images, CNN architectures can be used and for tabular data feed forward connected layers can be utilised.

1.3 Introduction: VAE

VAE, which is an acronym for Variational Auto Encoder, is a type of autoencoder that can be utilised to generate novel samples that appear to be from the training dataset. This architecture was introduced by Diederik P. Kingma and Max Welling in their paper titled "Auto-Encoding Variational Bayes" in 2014.

Much like the previously examined GAN architecture, the goal of the VAE is to be able to produce new samples that resemble previously encountered training data. However, instead of relying on two opposing architectures during training, VAEs accomplish their goal by mapping the input to a continuous latent space during training. After training, this latent space is sampled and fed to the decoder of the VAE to produce new data.

Before moving further into the VAE architecture, let us briefly remember the autoencoder structure.

1.3.1 The regular Autoencoder

An autoencoder is a network structure that is composed of two connected architectures: the encoder and the decoder.

The goal of the encoder is to take an input with a large dimension and compress the input to a latent space with a much smaller dimension. In a way, what the encoder does can be likened to a dimensionality reduction process. On the other hand, the goal of the decoder is to take the compressed representation given by the encoder as input and reconstruct the original input. This process can be illustrated as follows:

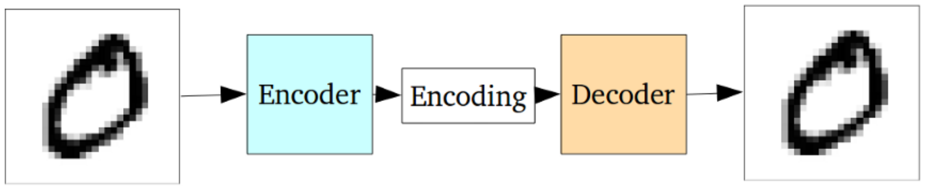

Figure 2: The illustration of the Autoencoder structure

Although this can be useful, the standard autoencoder is not really suitable for data generation. This is because the encoding space is not continuous, so a sample from this space when given as input to the decoder may give unrealistic results if the sample originated from a discontinuous region of the latent space.

1.3.2 Variational Auto Encoder

Unlike the standard autoencoders, the latent space of the VAE is continuous and suitable for sampling. To accomplish this, instead of compressing the input to a low dimension space, the encoder in a VAE outputs two values: a mean μ and a variation σ. When giving input to the decoder to reconstruct the original input, the normal distribution with these parameters is randomly sampled. The resulting sample is then given as input to the decoder. This process can be illustrated as follows:

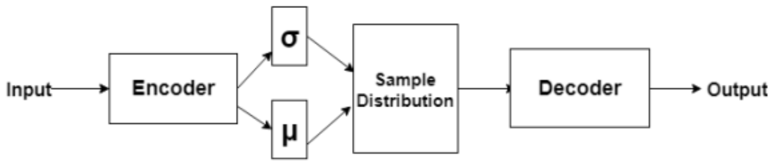

Figure 3: The illustration of the VAE architecture

Although this structure is very intuitive, there is a problem. Without any restraint, the encoder may assign drastically different μ and σ values to different classes. This may result in a separation of class distributions in the latent space, which can lead to the same problem as standard autoencoders. To overcome this issue, we include KL divergence in the training loss to ensure that each distribution represented by the μ and σ of the encoder is as close as possible to the standard normal distribution with 0 mean and 1 variance. In summation, the loss function of the VAE turns out to be the following:

$$Loss = -E_{z \sim q_\theta(z|x_i)}[log_{p\phi}(x_i|z)] + KL(q_\theta(z|x_i)||p(z)))$$

where x_i is a data sample, q_θ is the encoder, p_ϕ is the decoder and p(z) is the desired prior distribution of the latent space, which is generally the standard normal distribution.

In the loss function, the leftmost term can be thought as the reconstruction loss, which forces the decoder to perform an accurate reconstruction of the original image from the latent space. The rightmost term on the other hand forces the encoder to keep the latent space distribution for each data sample close to the standard normal distribution.

The training algorithm for the VAE can be thought as follows:

Algorithm 2: The training algorithm of VAE

```
Result: Trained VAE
initialization;
for epochs do
    get data samples;
    pass them to the encoder of VAE to generate $\mu$ and $\sigma$ for each sample;
    Randomly sample data from the normal distributions defined by $\mu$ and $\sigma$ ;
    Feed random samples to the decoder;
    Calculate reconstruction loss;
    Update encoder and decoder parameters with gradient descent ;
end
```

After training, to generate new samples, all we have to do is randomly sample from the standard normal distribution and feed these samples to the decoder of the VAE.

This concludes the discussion of the architectures of the different models. Now, let us move on to the different experiments.

1.4 Experiments

1.4.1 Linear GAN on MNIST Dataset

How to run To run the code for this section, use the linearGAN.ipynb notebook. To load the dataset, use section 5, to instantiate the generator, use section 8, to instantiate the discriminator, use section 10, to train the model use section 12, to generate the loss curves use section 16 and 17, to get the images from fixed noise, use section 18 and finally to get the latent space grid use section 19 and 21.

For this experiment, we are tasked with using training a GAN with linear layers on the MNIST dataset and then using this GAN to generate novel images that resemble the ones in the dataset. To accomplish this task, the network structure given in the Github link provided in the homework description was utilised.

The structure of the generator was as follows:
Generator Structure

```
Linear Layer (input dim = latent space dim, output dim = 256)-->
Leaky Relu (0.2) -->
Linear Layer(input dim = 256, output dim = 512) -->
Leaky Relu (0.2) -->
Linear Layer(input dim = 512, output dim = 1024) -->
Leaky Relu (0.2) -->
Linear Layer(input dim = 1024, output dim = 28*28) -->
```

```
Tanh   -->
Output (28*28)
```

Here, the latent space dimension was set to 2 as instructed. The reason that the output dimension is 28x28 is because the images in the MNIST dataset are of 28x28 size.

The structure of the discriminator was as follows:
Discriminator Structure

```
Flatten -->
Linear Layer(input dim = 28*28, output dim = 1024) -->
Leaky Relu (0.2) -->
Dropout(0.3) -->
Linear Layer(input dim =1024, output dim = 512) -->
Leaky Relu (0.2) -->
Dropout(0.3) -->
Linear Layer(input dim =512, output dim = 256) -->
Leaky Relu (0.2) -->
Dropout(0.3) -->
Linear Layer(input dim = 256, output dim = 1) -->
Sigmoid  -->
Output
```

Here, the input is first flattened. This is because when the input to the discriminator is from the training set, it is of size 28x28. The output of the discriminator is simply a probability where a value greater than 0.5 indicates that the discriminator thinks the input is from the training set. If it is less than 0.5 the discriminator thinks it is from the generator.

After defining the network structures, the model was trained for 20 epochs with a learning rate of 0.0005. These values are higher from that of the tutorial because it was observed that the model did not converge with lower values.

The change in the loss functions as iterations progress can be visualised as below:

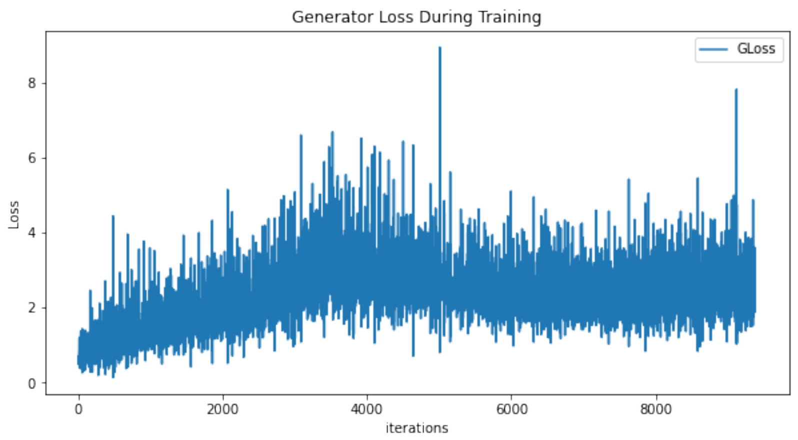

Figure 4: The generator loss during training of linear GAN

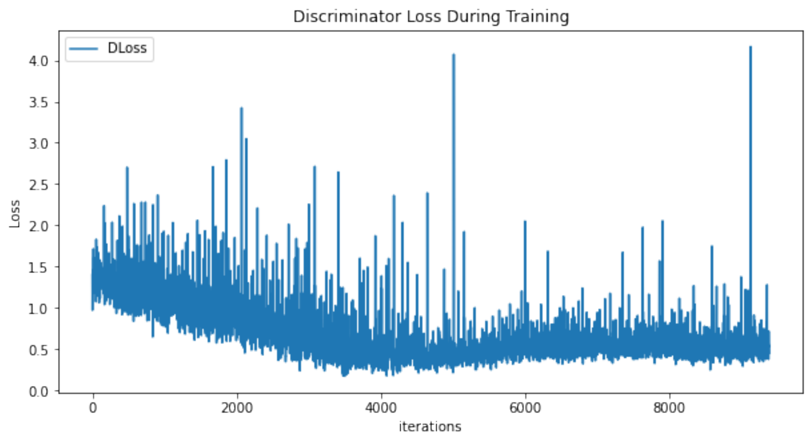

Figure 5: The discriminator loss during training of linear GAN

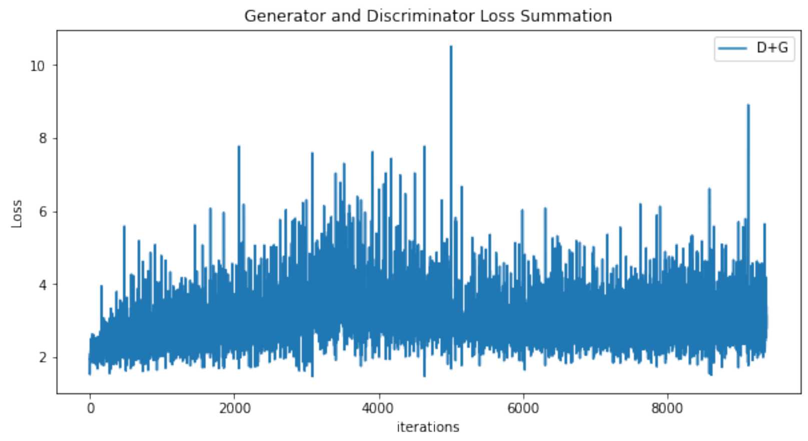

Figure 6: The total loss during training of linear GAN

As can be seen from the images, the loss of the discriminator is high at first, however, this does not imply that the initial generator is good. It simply means that the discriminator can not differentiate between fake and real images. As the performance of the discriminator improves, the loss of the generator increases, which forces it to generate better images to fool the discriminator. After 20 epochs, they settle at a middle ground. From the total loss graph, it can be seen that there is not much change throught the iterations. This is expected since as the loss of the generator and discriminator are opposing each other, i.e. when one goes down the other goes up.

Let us see the progression of the generator through the epochs. To do so, we sampled 25 samples of length 2 from a standard normal distribution before training. We then fed these samples to the generator after each epoch. To save space on the report, the generator output at every 2nd epoch will be displayed.

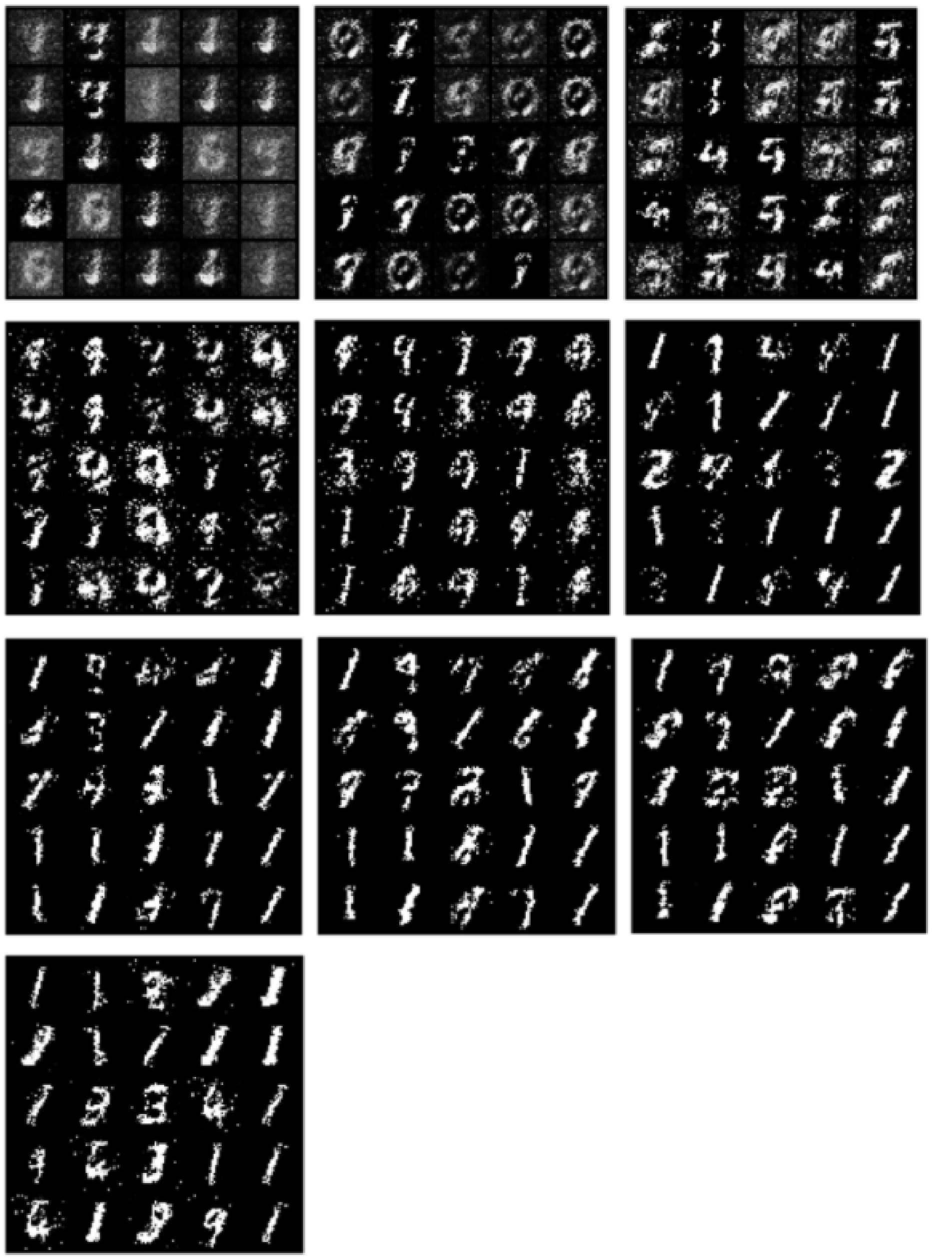

Figure 7: The generated images from linear GAN through epochs

As can be seen, during the initial epochs, the generator outputs are very noisy. As the generator trains, it starts producing more crisp images. In the final epoch, we can differentiate between some numbers such as 1, 3, 4 , 8 and 9. It can also be observed that there is still some noise in the generated images at the last epoch, implying that further training is required to produce better results.

Now, let us see which areas of the latent space correspond to which digits generated by the generator. To do so, the standard normal distribution was divided to 10 sections. This implied that there were 100 sections in the latent space as the size of the latent vector is 2. These spaces were given as input to the generator to generate new samples from each section. The result can be visualised as follows:

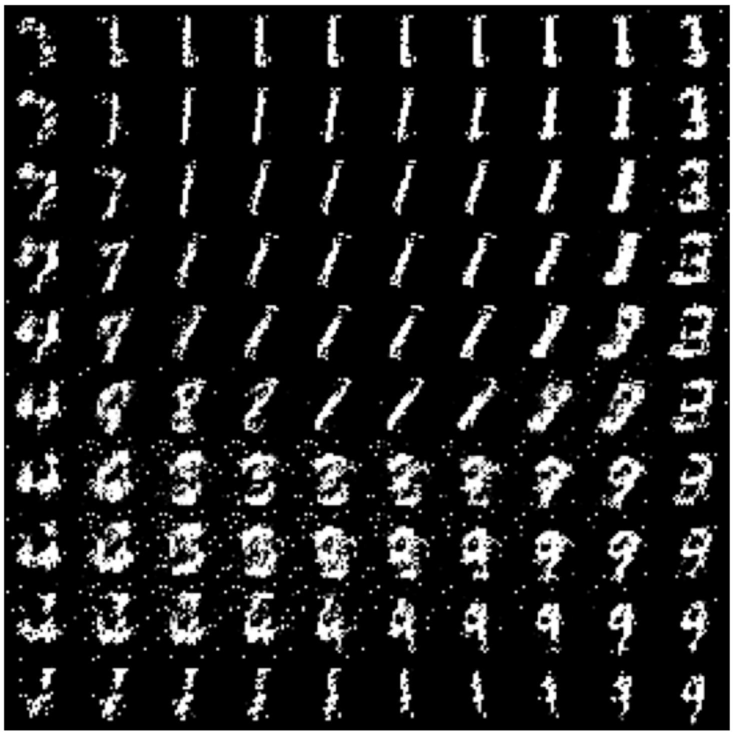

Figure 8: The generated images from linear GAN for sections of the latent space

As can be seen the sections are shared between digits 7,4,8,9,1 and 3.

1.4.2 DCGAN on MNIST Dataset

How to run To run the code for this section, use the DCGAN.ipynb notebook. To load the dataset, use section 5, to instantiate the generator, use section 8, to instantiate the discriminator, use section 10, to train the model use section 12, to generate the loss curves use section 16 and 17, to get the images from fixed noise, use section 18.

For this experiment, we are tasked with training a GAN with convolutional layers on the MNIST dataset and then using this GAN to generate novel images that resemble the original ones. To accomplish this task, the network structure given in the Github link provided in the homework description was utilised.

The structure of the generator was as follows:
Generator Structure

```
ConvTranspose2d( latent dim, ngf * 8, 4, 1, 0)-->
BatchNorm2d(ngf * 8)-->
ReLU()-->
ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1)-->
BatchNorm2d(ngf * 4)-->
ReLU(True)-->
ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1)-->
BatchNorm2d(ngf * 2)-->
ReLU(True)-->
ConvTranspose2d( ngf * 2, ngf, 4, 2, 1)-->
BatchNorm2d(ngf)-->
ReLU(True)-->
ConvTranspose2d( ngf, nc, 4, 2, 1)-->
Tanh()
```

Here, the latent space dimension was set to 100, which is the same as the tutorial. ngf refers to the size of the feature maps in the generator, which was set to be 128 and nc is the number of channels in the training images. Since the MNIST dataset is composed of black and white images, this value was set to 1.

The structure of the discriminator was as follows:
Discriminator Structure

```
Conv2d(nc, ndf, 4, 2, 1)-->
LeakyReLU(0.2)-->
Conv2d(ndf, ndf * 2, 4, 2, 1)-->
BatchNorm2d(ndf * 2)-->
LeakyReLU(0.2)-->
Conv2d(ndf * 2, ndf * 4, 4, 2, 1)-->
BatchNorm2d(ndf * 4)-->
LeakyReLU(0.2)-->
Conv2d(ndf * 4, ndf * 8, 4, 2, 1)-->
BatchNorm2d(ndf * 8)-->
LeakyReLU(0.2)-->
Conv2d(ndf * 8, 1, 4, 1, 0)-->
Sigmoid()
```

Here, the ndf refers to the size of the feature maps in the discriminator, which was set to be 128. As can be seen, the size of the output of discriminator is 1, which represents the probability of the image coming from the training set.

For training, the images in MNIST were resized to be 64x64 as done in the given Gihub link. The DCGAN model was trained for 10 epochs with a learning rate of 0.0002.

The change in the loss functions as iterations progress can be visualised as below:

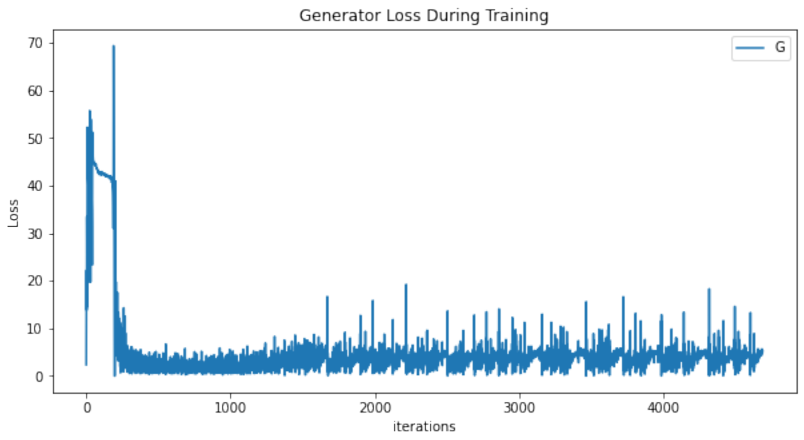

Figure 9: The generator loss during training of DCGAN

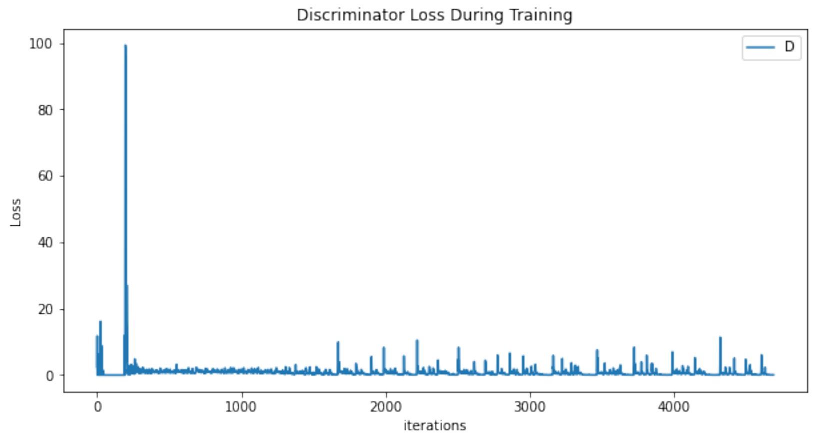

Figure 10: The discriminator loss during training of DCGAN

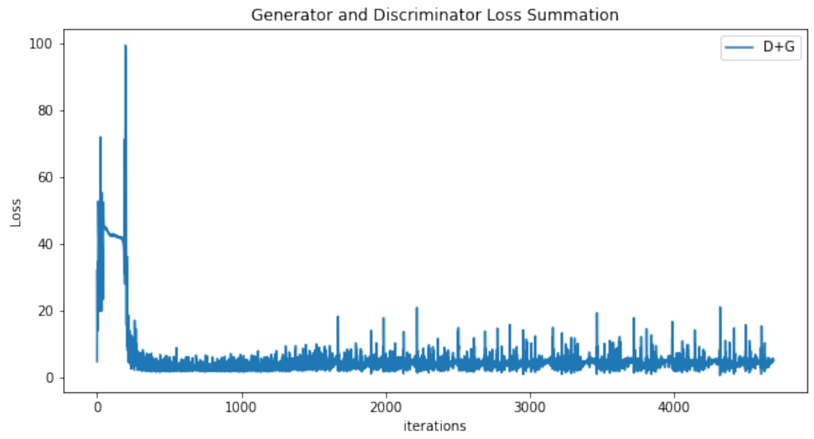

Figure 11: The total loss during training of DCGAN

As can be seen from the images, the loss of the discriminator is during the first iterations, which correspond to a high loss for the generator. However, after some iterations, there is a dramatic decrease in the generator loss, which causes a sudden increase in the discriminator loss. Then, we can observe that both losses are low, indicating that we managed to obtain a good generator and discriminator. This result can also be obtained by examining the total loss graph. Here, the total loss starts high, as we have a bad generator and discriminator. However, after some iterations, the loss stabilises, indicating that the generator and the discriminator have reached an equilibrium.

Let us now see the progression of the generator through the epochs. To do so, we sampled 25 samples of length 100 from a standard normal distribution before training and fed these samples to the generator after each epoch. The results for 10 epoch are as follows:

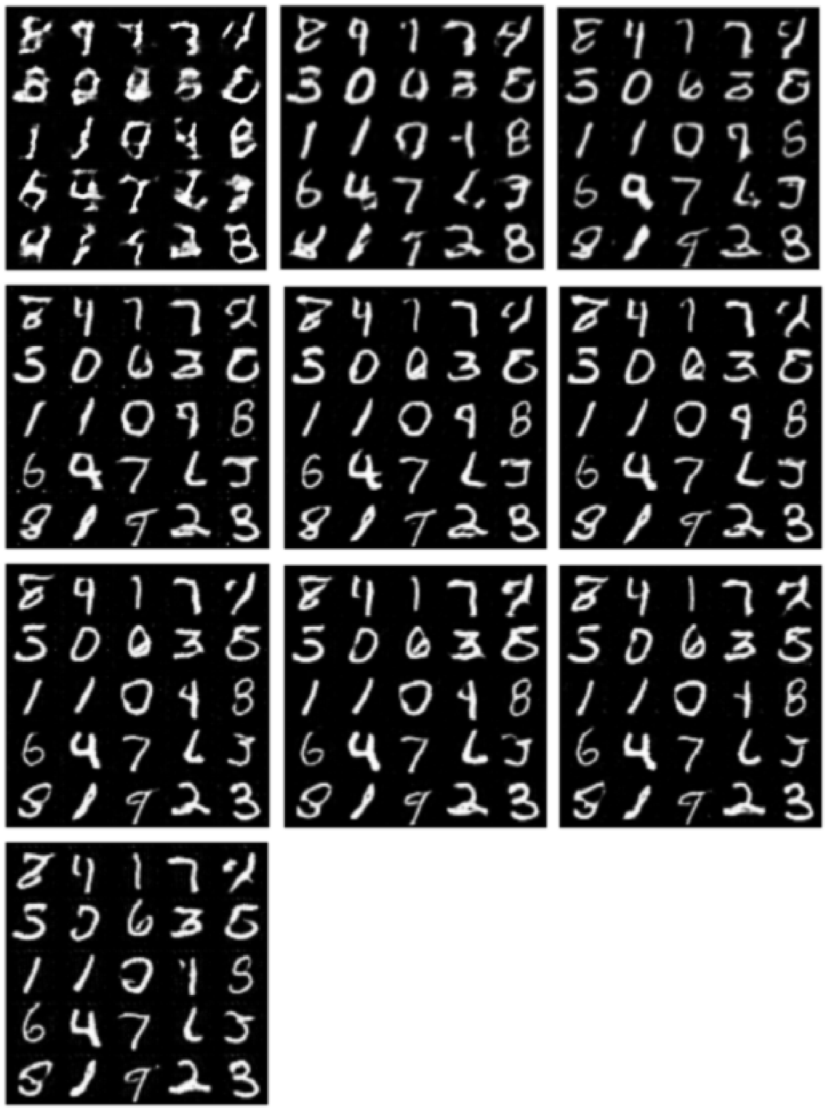

Figure 12: The generated images from DCGAN through epochs

As can be seen, the images generated in the first epoch are a bit blurry. However, they are still far clearer than that of the linear GAN. As the epochs progress, the blur effect on the numbers decrease and we can discern some of the numbers. By the last epoch, most of the generated images are clear and it is easy to distinguish between digits.

1.4.3 VAE on MNIST Dataset

How to run To run the code for this section, use the VAE.ipynb notebook. To load the dataset, use section 5, to define the VAE structure, use section 6, to train the model use section 10, to generate the loss curve use section 14, to get the images from fixed noise, use section 15 and to see the reconstruction images use section 16.

For this experiment, we are tasked with training a VAE with on the MNIST dataset and then using this VAE to generate novel images that resemble the original ones. To accomplish this task, the network structure given in the Github link provided in the homework description was utilised.

The structure of the encoder of the VAE was as follows:
Encoder Structure

```
Linear(imgSize, 512)-->
ReLU()-->
Linear(512,256)-->
ReLU()--> a
a --> Linear(256,zDim) --> mu
a --> Linear(256,zDim) --> log(sigma)
```

Here, the input image is flattened and given to two linear layers. Since the images are 28x28, imgSize is 28*28=784. The output of the second linear layer is fed into another linear layer to generate the μ vector. The same output is then given to another linear layer to generate the $log(\sigma)$ vector. The dimension of these vectors were set to 2 as done in the Githıb link, so the zDim value was 2.

The structure of the decoder was as follows:
Decoder Structure

```
Linear(z_dim, 256)-->
ReLU()-->
Linear(256,512)-->
ReLU()-->
Linear(512,imgSize)-->
Sigmoid()
```

Here, the decoder is simply the reverse of the encoder. In the end, the decoder generates an imgSize or 28x28 size vector, which is then reshaped to a 28 by 28 matrix to get the output images.

Training of the VAE was conducted for 10 epochs with a learning rate of 0.001.

The change in the loss function through training iterations can be seen below:

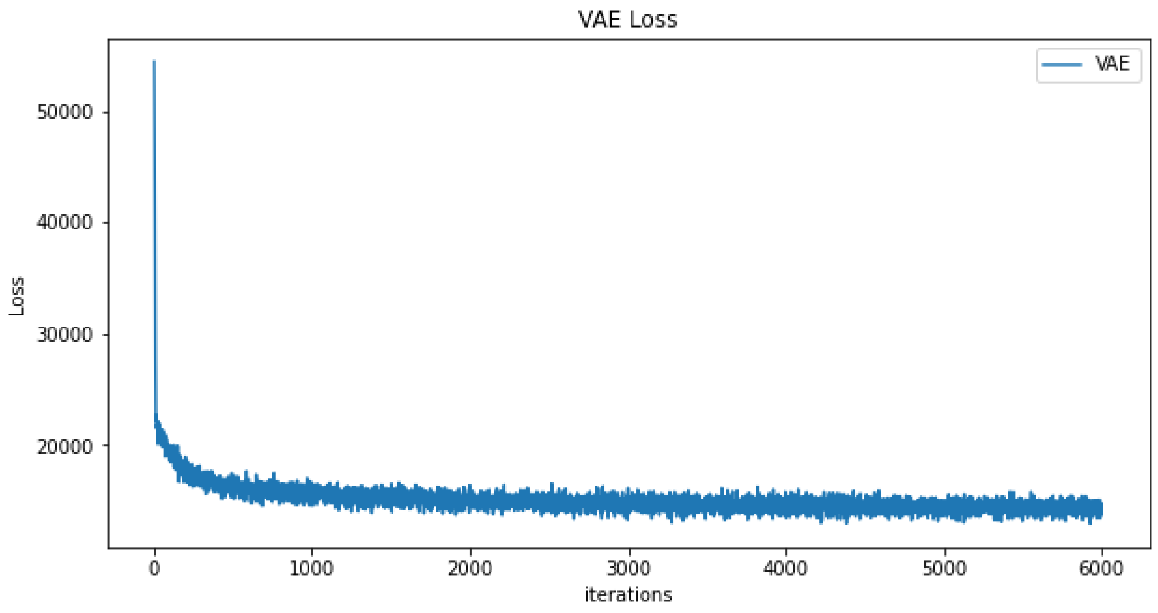

Figure 13: The change in VAE loss through training iterations

As can be seen from the graph, there is a high loss during the first iterations. However, the loss quickly diminishes as the model converges. This indicates that the decoder is becoming better at reconstructing images by using the random samples while the encoder is getting better at mapping an image to the latent space distribution.

Let us now see the progression of the generated images through the epochs. To do so, we sampled 25 samples of length 2 from a standard normal distribution before training and fed these samples to the decoder of the VAE after each epoch. The results for 10 epochs are as follows:

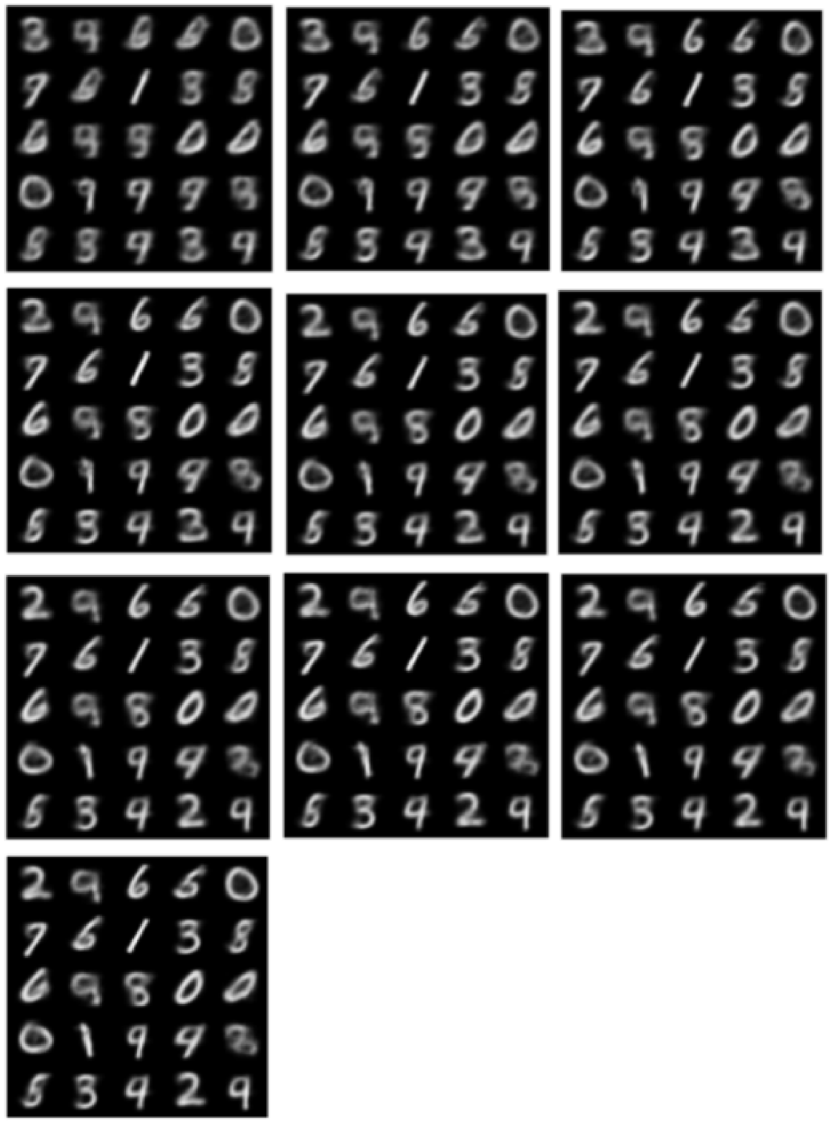

Figure 14: The generated images by the VAE through epochs

As can be seen, the images generated from the sampled noise in the first epochs are blurry, which makes it difficult to discern which number an image corresponds to for some images. As the epochs progress, the generated images become sharper and easier to understand, indicating that the decoder is becoming more capable of generating images from the latent space.

Finally, as VAE is an autoencoder, we can see how the reconstruction of the images changes through the epochs. To do so, 25 sample images were randomly picked from the training set. These images were passed through the encoder and the decoder of the VAE after each epoch. The original images were as follows:

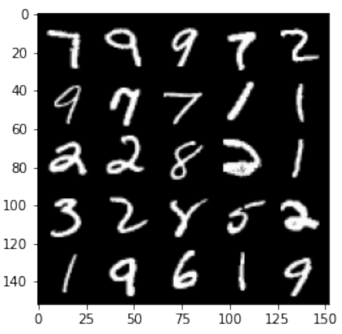

Figure 15: The images given for reconstruction

Now, let us see the change in reconstruction:

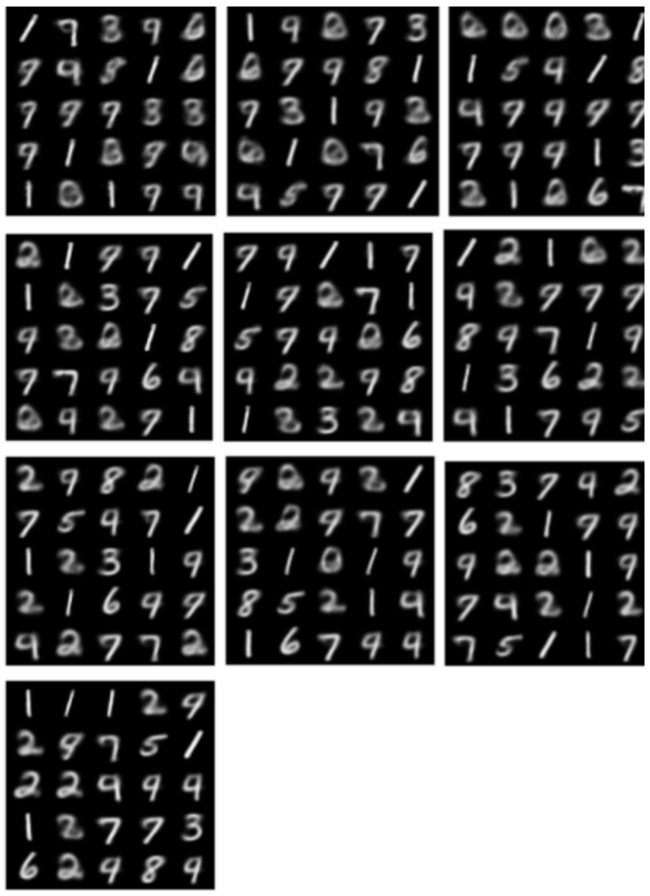

Figure 16: The recontructed images by the VAE through epochs

As can be seen, the initial reconstructions are blurry and not resemblant of the target images. As the epochs progress, some of the reconstructions converge to the target images. However, some of the images are still of the wrong class. Furthermore, even the generated images that resemble the given targets are not as sharp as the target images, so there is still some error present. This means that although the VAE is learning, it has to be run for more epochs to get better results.

1.5 Conclusion

In conclusion, we learned about the GAN and VAE architectures and saw how to train these architectures to be able to generate novel data samples.

Training the DCGAN took longer than the other models. Also, the linear GAN sometimes got stuck during training with 0 discriminator error and high generator error. Figuring out the required changes to train the linear GAN also took some time. However, overall no major issues were encountered.

Reference

Mohr, Felix. “Teaching a Variational Autoencoder (VAE) to Draw Mnist Characters.” Medium, Towards Data Science, 9 Nov. 2017, https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776.

Shafkat, Irhum. “Intuitively Understanding Variational Autoencoders.” Medium, Towards Data Science, 1 Oct. 2021, https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf.

Gandhi, Rohith. “Generative Adversarial Networks - Explained.” Medium, Towards Data Science, 14 May 2018, https://towardsdatascience.com/generative-adversarial-networks-explained-34472718707a.

Goodfellow, Ian, et al. "Generative adversarial networks." Communications of the ACM 63.11 (2020): 139-144.

Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." arXiv preprint arXiv:1312.6114 (2013).