UNIVERSITY OF TEHRAN

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NEURAL NETWORK & DEEP LEARNING

MINI PROJECT#3

SIAVASH SHAMS

810197644

MOHAMMAD HEYDARI
810197494

UNDER SUPERVISION OF:

DR. AHMAD KALHOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

July. 2022

Contents

1 QUESTION #1: SGAN

In this part we are going to implement SGAN in the case of semi-supervised learning task using MNIST dataset.

We extend Generative Adversarial Networks (GANs) to the semi-supervised context by forcing the discriminator network to output class labels. We train a generative model G and a discriminator D on a dataset with inputs belonging to one of N classes. At training time, D is made to predict which of N+1 classes the input belongs to, where an extra class is added to correspond to the outputs of G. We show that this method can be used to create a more data-efficient classifier and that it allows for generating higher quality samples than a regular GAN.

1.1

SGAN Training Approach:

Below is the model for Semi-Supervised GAN:

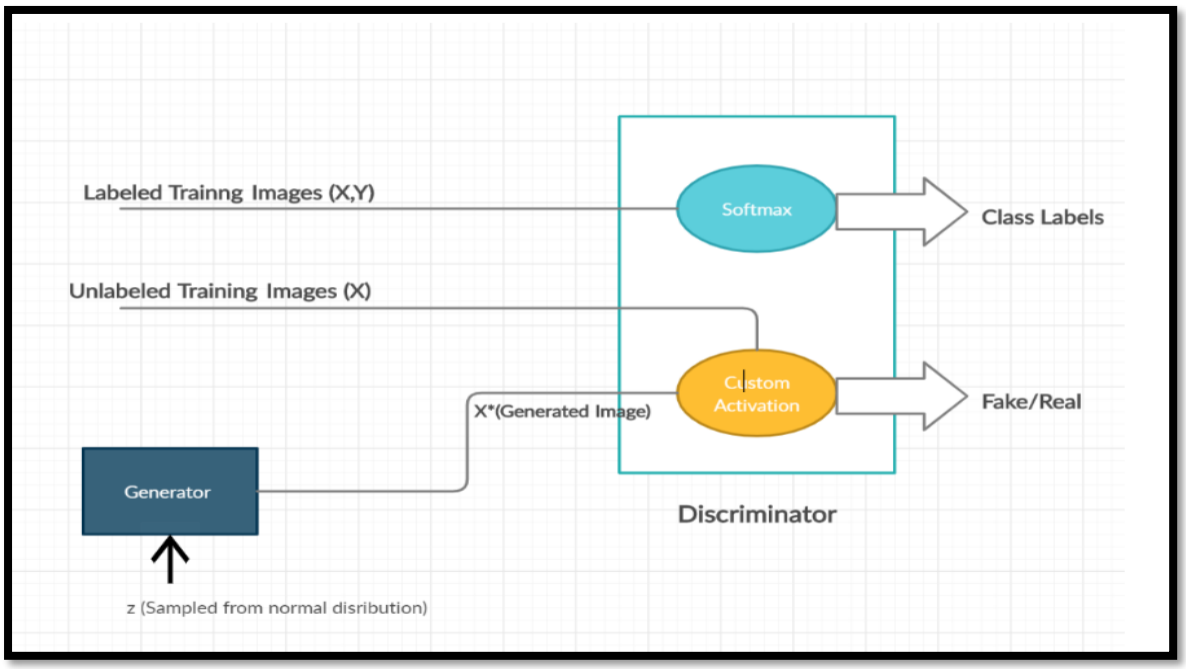

Figure1. Semi-Supervised GAN

Let's Understand the model:

The discriminator is passed through three types of images namely Labeled Training Images, Unlabeled Training Images, and Fake Images generated by Generator. Its job is not only to distinguish between Real/Fake Images but also to classify the Labeled Training Images into their correct classes.

The Discriminator has two different outputs:

- Softmax Activation for classifying labeled data into their correct classes i.e this is supervised discriminator.
- Custom Activation for classifying into real or fake. We'll see about the custom activation in implementation.

Training an SGAN is similar to training a GAN. We simply use higher granularity labels for the half of the minibatch that has been drawn from the data generating distribution. D/C is trained to minimize the negative log likelihood with respect to the given labels and G is trained to maximize it, as shown in Algorithm 1.

Algorithm 1 SGAN Training Algorithm

Input: I: number of total iterations
for $i = 1$ **to** I **do**
Draw m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
Draw m examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ from data generating distribution $p_d(x)$.
Perform gradient descent on the parameters of D w.r.t. the NLL of D/C's outputs on the combined minibatch of size $2m$.
Draw m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
Perform gradient descent on the parameters of G w.r.t. the NLL of D/C's outputs on the minibatch of size m.
end for

Figure2. SGAN Training Algorithm

1.2

In this part we are going to implement SGAN using MNIST dataset please note that in this section I have just used 100 labeled sample and rest of that chosen from un-labeled data.

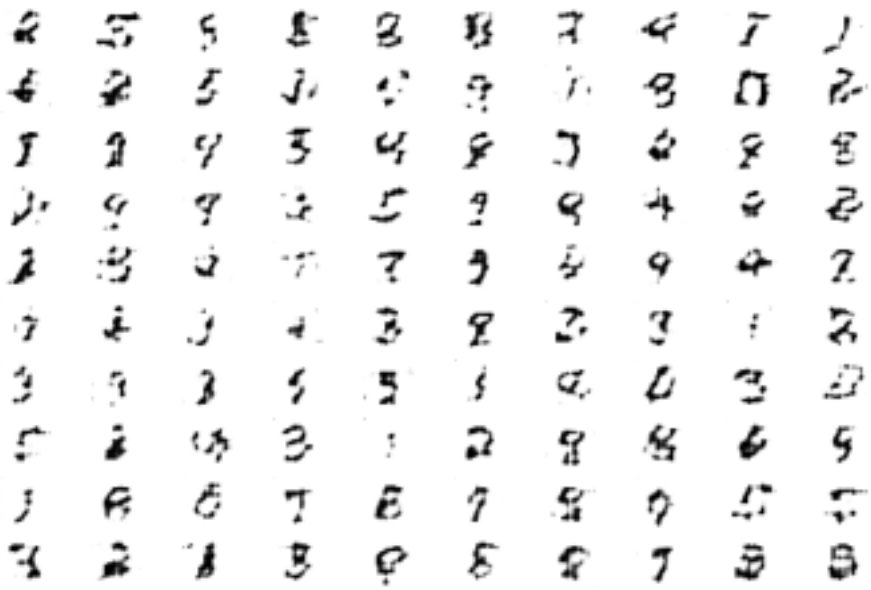

Figure3. Result of learning process(epoch#600)

Figure4. Result of learning process(epoch#1200)

Figure5. Result of learning process(epoch#3600)

Figure6. Result of learning process(epoch#12000)

Further we have reported the accuracy of the training and testing process on MNIST dataset:

```
Train Accuracy: 94.803%
Test Accuracy: 95.150%
```

Figure7. Test and Train accuracy

1.3

In this part we are going to implement classifier without using GAN.

further the results have been attached :

```
Classifier Accuracy: 80.205%
INFO:tensorflow:Assets written to: c_model_12000.h5_without_GAN/assets
>Saved: c_model_12000.h5_without_GAN
```

Figure8. Test and Train accuracy

Analyse:

As you can see the classifier accuracy has been decreased in comparison to the previous part.

Now let's have a look on below table to make sense about the train accuracy in both cases :

Our Cases:	Using GAN	Without GAN
Accuracy:	94.803%	80.205%

1.4

In this part we are going to implement the model using Variational Autoencoder (VAE).

Please note that in this section we have used VAE just instead of Generator not whole of GAN module, furthermore please note that in this case we declare loss function as summation of three separated terms which abbreviated below:

Total-Loss= GAN Loss + Reconstruction Loss + KL-divergence Loss

NOTE : in this section I have used the paper which referenced below:

https://arxiv.org/pdf/1512.09300.pdf

First of all, let's have a dive into the theoretical concept of VAEGAN:

In this method, a variational autoencoder (VAE) is combined with a Generative Adversarial Network (GAN) in order to learn a higher level image similarity metric instead of the traditional element-wise metric. The model is shown below:

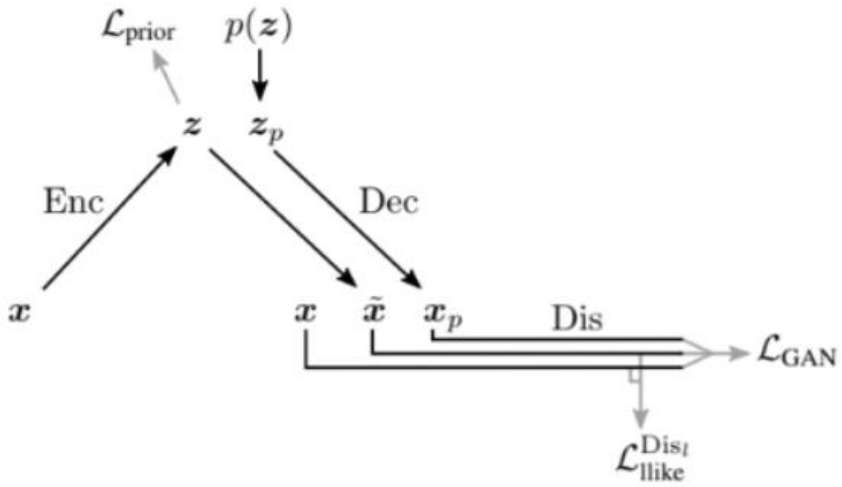

Figure9. VAE-GAN Network Overview

The encoder encodes the data sample into a latent representation while the decoder tries to reconstruct from the latent vector. This reconstruction is fed to the discriminator of the GAN in order to learn the higher-level sample similarity.

The VAE and GAN is trained simultaneously using the loss function in Equation 1 which is composed of the prior regularization term from the encoder, the reconstruction error, and the style error from the GAN. However, this combined loss function is not applied on the entire network. Alg.1 shows the algorithm used in training the VAE/GAN network.

$$\mathcal{L} = \mathcal{L}_{\text{prior}} + \mathcal{L}_{\text{llike}}^{\text{Dis}_l} + \mathcal{L}_{\text{GAN}}$$

Algorithm 1 Training the VAE/GAN model

$\theta_{\text{Enc}}, \theta_{\text{Dec}}, \theta_{\text{Dis}} \leftarrow$ initialize network parameters
repeat
  $X \leftarrow$ random mini-batch from dataset
  $Z \leftarrow \text{Enc}(X)$
  $\mathcal{L}_{\text{prior}} \leftarrow D_{\text{KL}}(q(Z|X)\|p(Z))$
  $\tilde{X} \leftarrow \text{Dec}(Z)$
  $\mathcal{L}_{\text{llike}}^{\text{Dis}_l} \leftarrow -\mathbb{E}_{q(Z|X)}\left[p(\text{Dis}_l(X)|Z)\right]$
  $Z_p \leftarrow$ samples from prior $\mathcal{N}(0, I)$
  $X_p \leftarrow \text{Dec}(Z_p)$
  $\mathcal{L}_{\text{GAN}} \leftarrow \log(\text{Dis}(X)) + \log(1 - \text{Dis}(\tilde{X})) + \log(1 - \text{Dis}(X_p))$
  // Update parameters according to gradients
  $\theta_{\text{Enc}} \overset{+}{\leftarrow} -\nabla_{\theta_{\text{Enc}}}(\mathcal{L}_{\text{prior}} + \mathcal{L}_{\text{llike}}^{\text{Dis}_l})$
  $\theta_{\text{Dec}} \overset{+}{\leftarrow} -\nabla_{\theta_{\text{Dec}}}(\gamma\mathcal{L}_{\text{llike}}^{\text{Dis}_l} - \mathcal{L}_{\text{GAN}})$
  $\theta_{\text{Dis}} \overset{+}{\leftarrow} -\nabla_{\theta_{\text{Dis}}}\mathcal{L}_{\text{GAN}}$
until deadline

Figure10. Alg.1.Training Algorithm

The model trains the discriminator more often than the generator, which according to some papers, will yield better results.

After this supplementary description we are going to implement VAEGAN in python environment.

Results of implementation:

Accuracy:

```
Classifier Accuracy:% 85.33
```

Figure11. Train accuracy

Generated image:

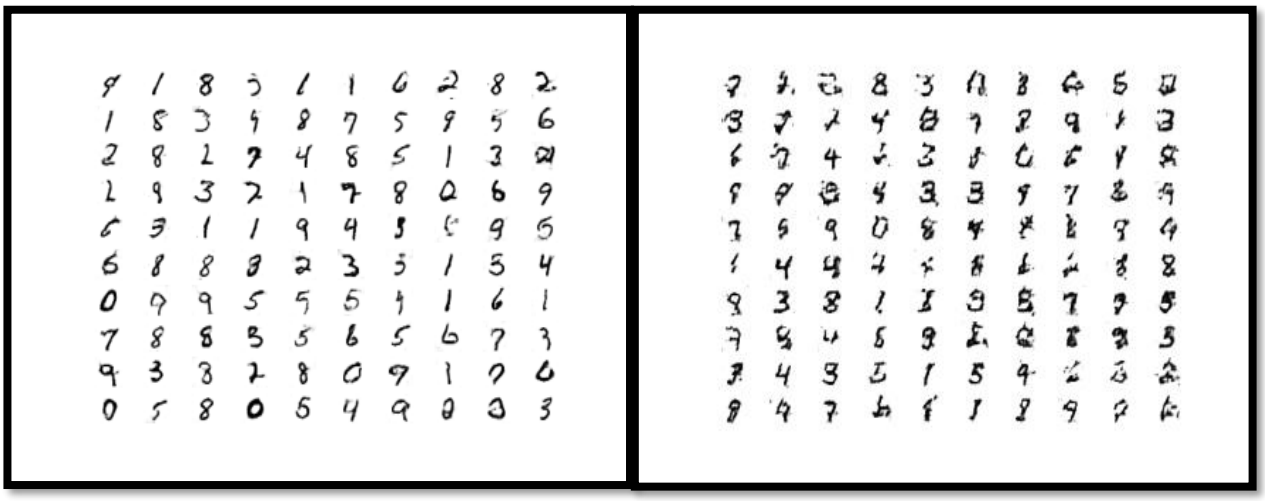

Figure12. Output samples from VAEGAN and GAN after #2400 MNIST epochs. GAN is on the left and VAEGAN is on the right.

Analyse:

Now let's have a look on below table to make sense about the train accuracy in all cases :

Our Cases:	Using GAN	Without GAN	Using VAE
Accuracy:	94.803%	80.205%	85.33%

As we can see when we are dealing with VAEGAN in same number of epochs we get a worser than results in comparison with normal GAN and accuracy has been decreased.

2 QUESTION #2: DCGAN

In this part we intend to implement a **DCGAN** which can be used to generate pictures from Abstract Art Gallery dataset.

2.1

HOW DCGAN WORK?

DCGAN, or Deep Convolutional GAN, is a generative adversarial network architecture. It uses convolutional and convolutional-transpose layers in the generator and discriminator, respectively. It was proposed by Radford et. al. in the paper Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Here the discriminator consists of strided convolution layers, batch normalization layers, and LeakyRelu as activation function. It takes a 3x64x64 input image. The generator consists of convolutional-transpose layers, batch normalization layers, and ReLU activations. The output will be a 3x64x64 RGB image.

It also uses a couple of guidelines, in particular:

- Replacing any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Using batchnorm in both the generator and the discriminator.
- Removing fully connected hidden layers for deeper architectures.
- Using ReLU activation in generator for all layers except for the output, which uses tanh.
- Using LeakyReLU activation in the discriminator for all layer.

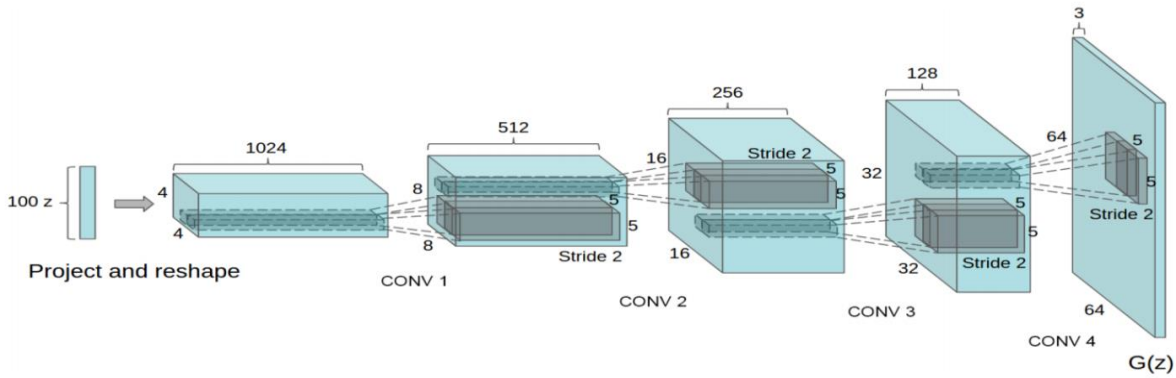

Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

This is the DCGAN generator presented in the LSUN scene modelling paper. This network takes in a 100x1 noise vector, denoted z, and maps it into the G(Z) output which is 64x64x3.

the first layer expands the random noise. The network goes from 100x1 to 1024x4x4! This layer is denoted ‘project and reshape’.

We see that following this layer, classical convolutional layers are applied which reshape the network with the (N+P — F)/S + 1 equation classically taught with convolutional layers. In the diagram above we can see that the N parameter, (Height/Width), goes from 4 to 8 to 16 to 32, it doesn’t appear that there is any padding, the kernel filter parameter F is 5x5, and the stride is 2. You may find this equation to be useful for designing your own convolutional layers for customized output sizes.

We see the network goes from:

100x1 → 1024x4x4 → 512x8x8 → 256x16x16 → 128x32x32 → 64x64x3.

2.2

In this section, we are going to implement DCGAN according to “Abstract Art Gallery” dataset:

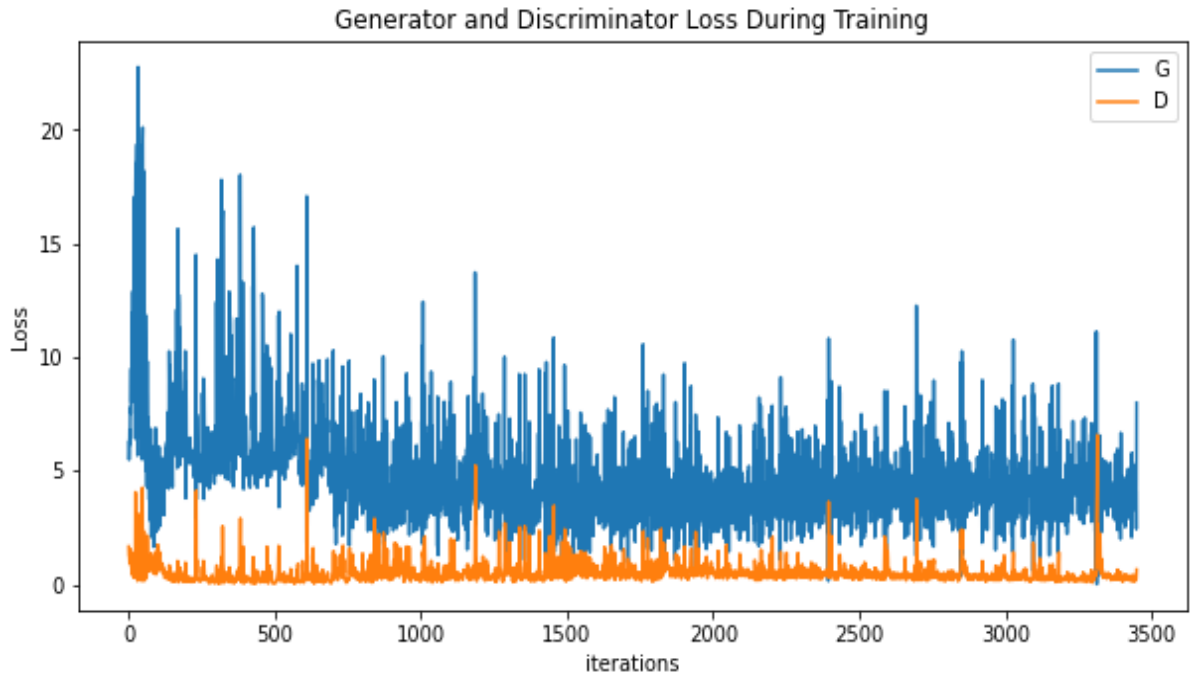

Figure13. Generator and discriminator loss per iteration

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
   ConvTranspose2d-1            [-1, 512, 7, 7]         819,200
       BatchNorm2d-2            [-1, 512, 7, 7]           1,024
              ReLU-3            [-1, 512, 7, 7]               0
   ConvTranspose2d-4          [-1, 256, 14, 14]       2,097,152
       BatchNorm2d-5          [-1, 256, 14, 14]             512
              ReLU-6          [-1, 256, 14, 14]               0
   ConvTranspose2d-7          [-1, 128, 28, 28]         524,288
       BatchNorm2d-8          [-1, 128, 28, 28]             256
              ReLU-9          [-1, 128, 28, 28]               0
  ConvTranspose2d-10           [-1, 64, 56, 56]         131,072
      BatchNorm2d-11           [-1, 64, 56, 56]             128
             ReLU-12           [-1, 64, 56, 56]               0
  ConvTranspose2d-13          [-1, 3, 112, 112]           3,072
             Tanh-14          [-1, 3, 112, 112]               0
================================================================
Total params: 3,576,704
Trainable params: 3,576,704
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 9.19
Params size (MB): 13.64
Estimated Total Size (MB): 22.84
----------------------------------------------------------------
```

Figure14. Generator architecture

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           3,072
         LeakyReLU-2           [-1, 64, 32, 32]               0
            Conv2d-3          [-1, 128, 16, 16]         131,072
       BatchNorm2d-4          [-1, 128, 16, 16]             256
         LeakyReLU-5          [-1, 128, 16, 16]               0
            Conv2d-6            [-1, 256, 8, 8]         524,288
       BatchNorm2d-7            [-1, 256, 8, 8]             512
         LeakyReLU-8            [-1, 256, 8, 8]               0
            Conv2d-9            [-1, 512, 4, 4]       2,097,152
      BatchNorm2d-10            [-1, 512, 4, 4]           1,024
        LeakyReLU-11            [-1, 512, 4, 4]               0
           Conv2d-12              [-1, 1, 1, 1]           8,192
          Sigmoid-13              [-1, 1, 1, 1]               0
================================================================
Total params: 2,765,568
Trainable params: 2,765,568
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 2.31
Params size (MB): 10.55
Estimated Total Size (MB): 12.91
----------------------------------------------------------------
```

Figure15. Discriminator architecture

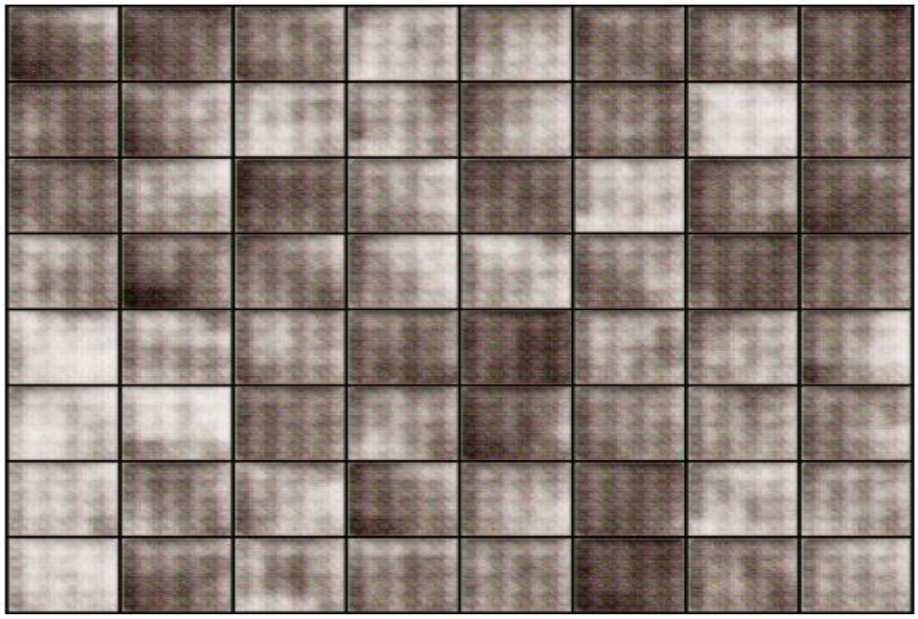

Figure16. Generated images after epoch10

Figure17. Generated images after epoch100

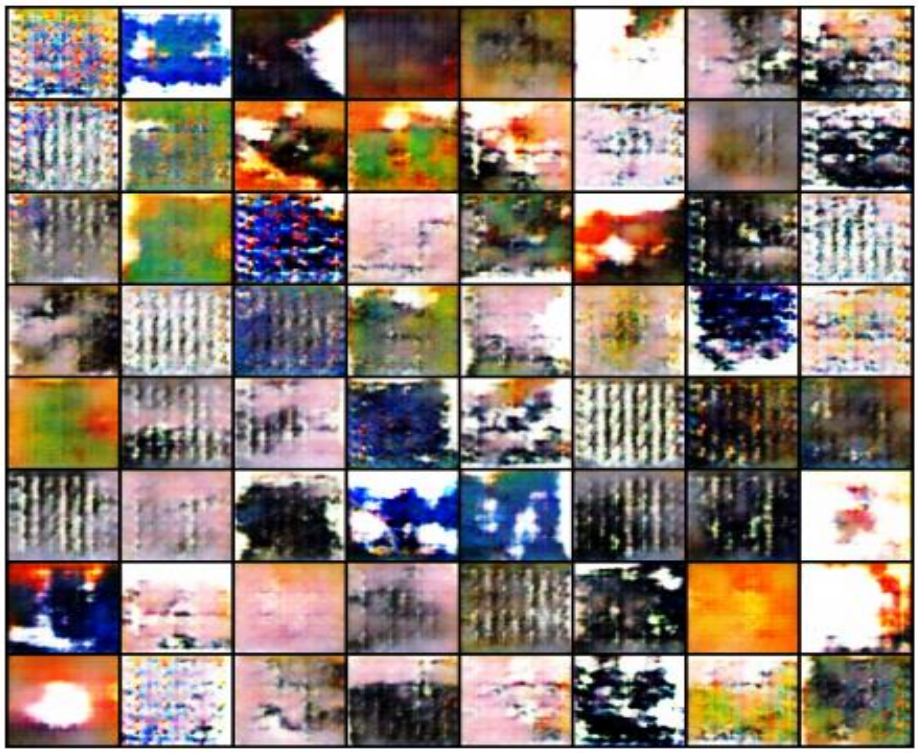

Figure18. Generated images after epoch150

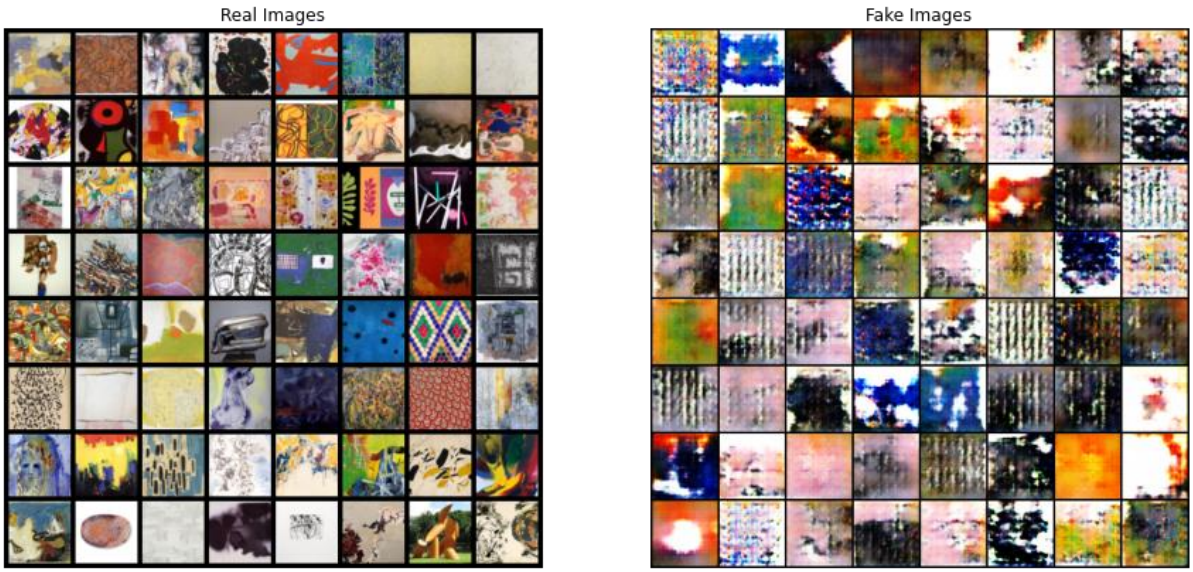

Figure19. Real images vs fake images

2.3

in this section we are going to change two of metrics and investigate whether the results tend to be better or not.

Many GAN models suffer the following major problems:

- **Non-convergence**: the model parameters oscillate, destabilize and never converge,
- **Mode collapse**: A mode collapse refers to a generator model that is only capable of generating one or a small subset of different outcomes, or modes.
- **Vanishing gradient**: minimizing the objective function of regular GAN suffers from vanishing gradients, which makes it hard to update the generator. LSGANs can relieve this problem because LSGANs penalize samples based on their distances to the decision boundary, which generates more gradients to update the generator.

Both mode collapses and non- convergence of GANs can be avoided with a gradient penalty scheme called DRAGAN

Now we are going to implement DRGAN, the architecture of the network is same as normal GAN showed in figure14

Loss function used in DRGAN is as follows:

$$L_D^{DRAGAN} = L_D^{GAN} + \lambda E[\left(|\nabla D(\alpha x - (1-\alpha x_p))| - 1\right)^2]$$

$$L_G^{DRAGAN} = L_G^{GAN}$$

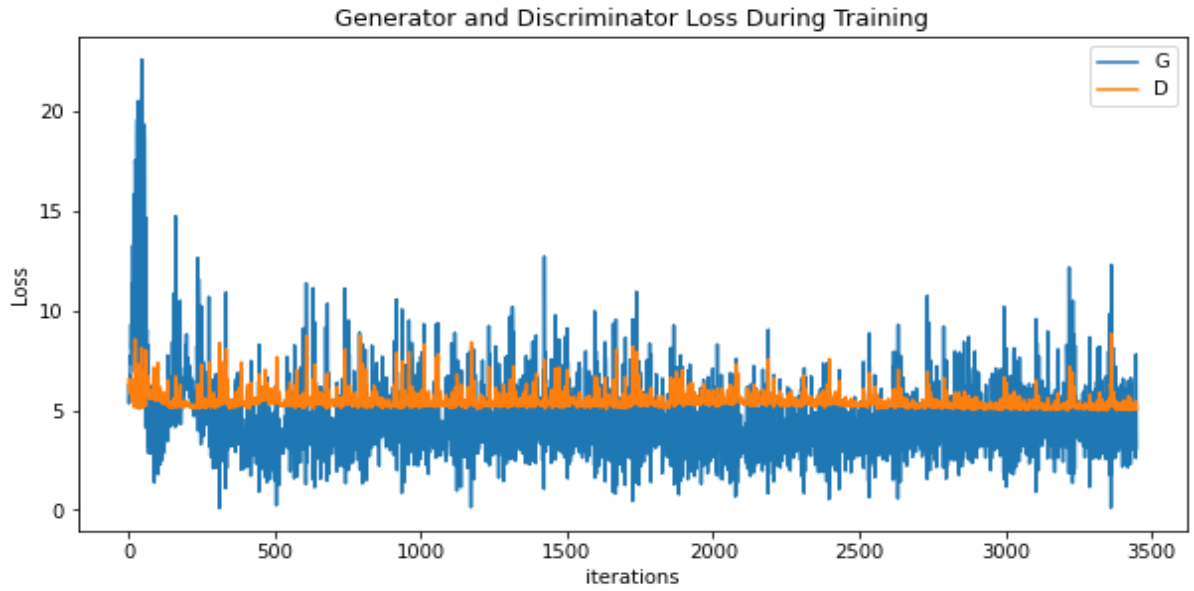

Figure20. Generated and discriminator loss per iteration

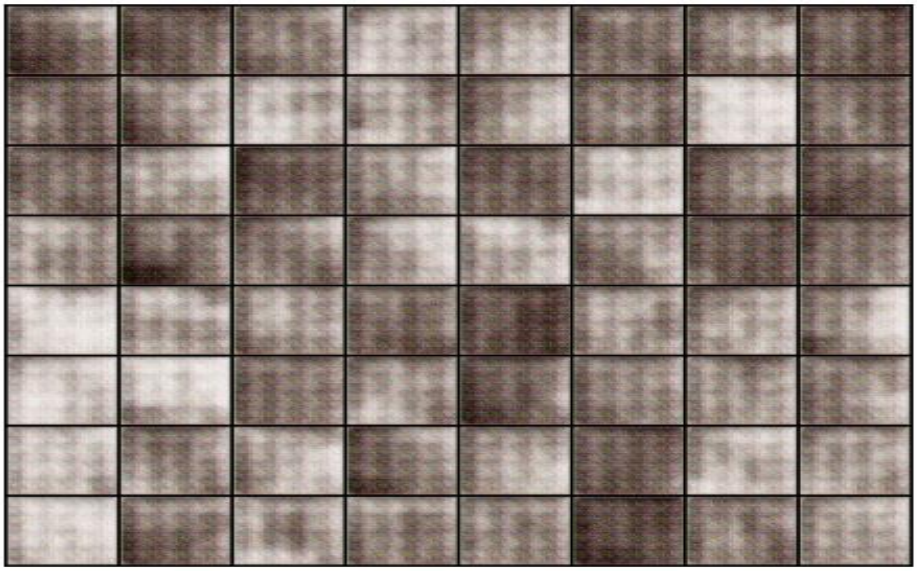

Figure21. Generated images after epoch10

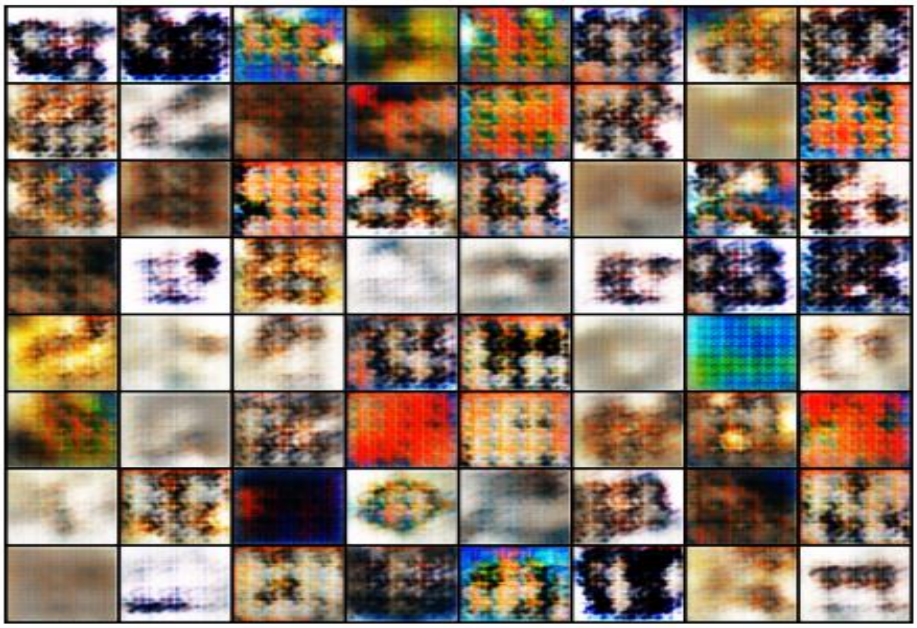

Figure22. Generated images after epoch100

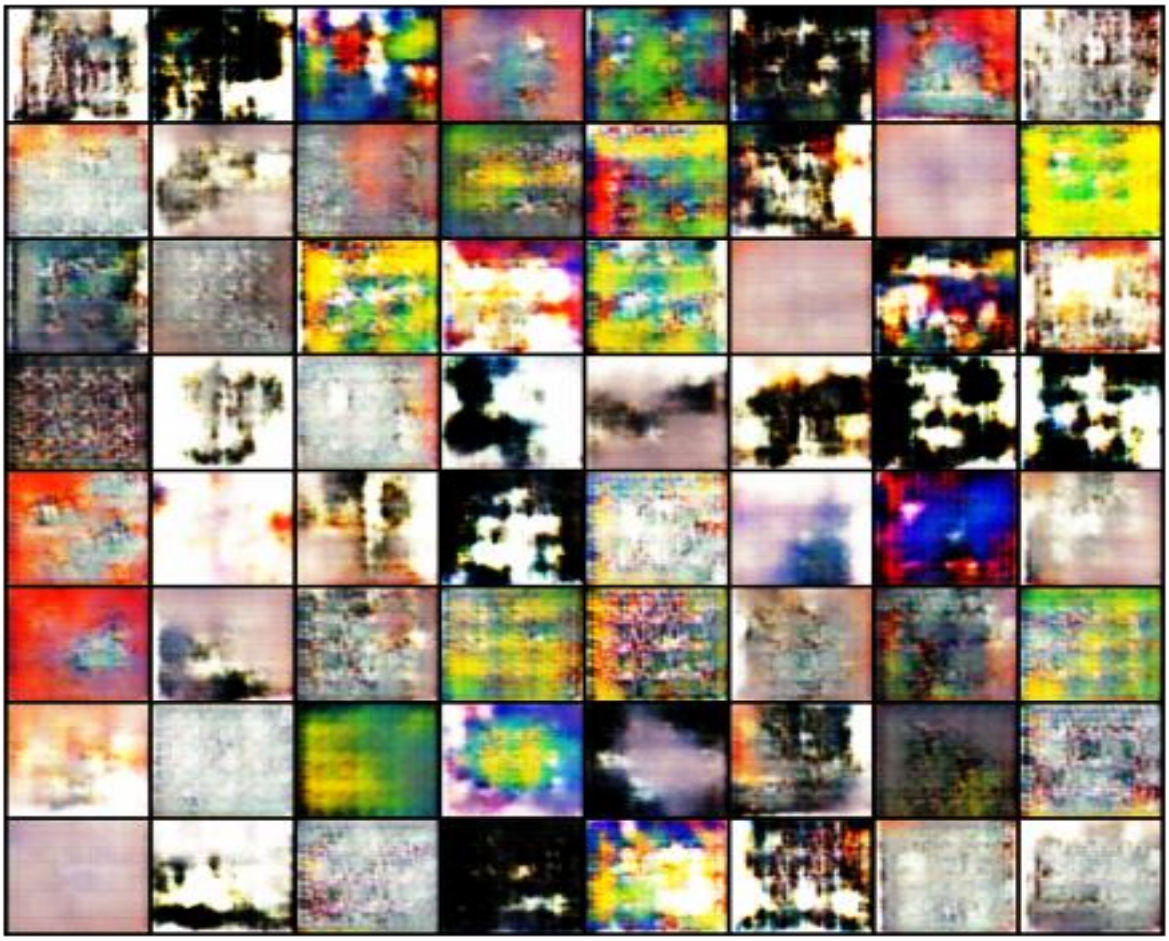

Figure23. Generated images after epoch150

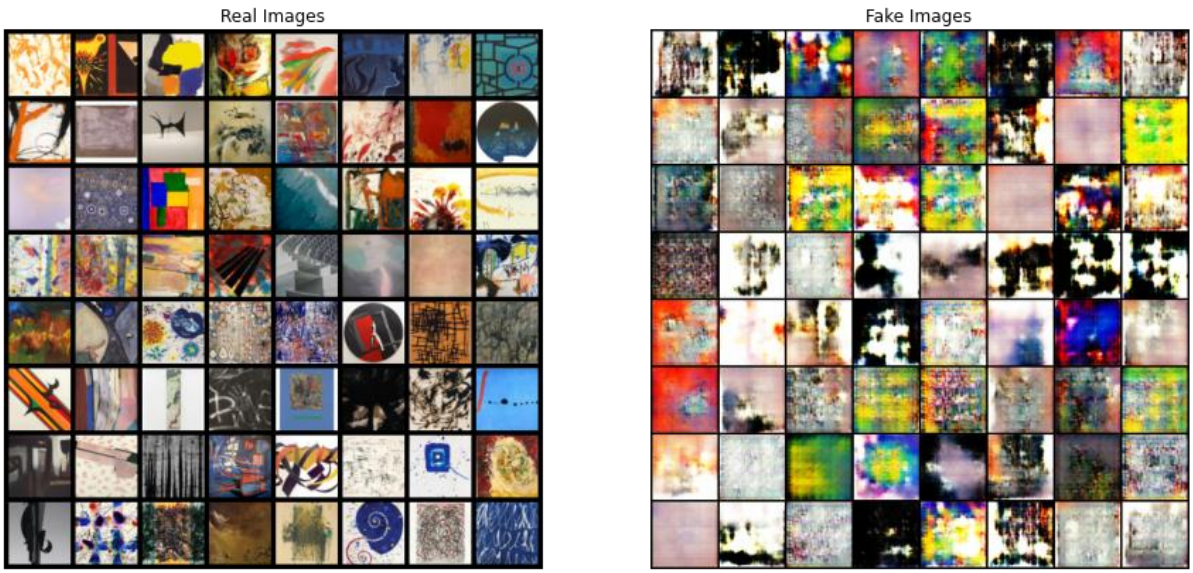

Figure24. Real images vs fake images

Reference: Kodali, N., Abernethy, J., Hays, J. and Kira, Z., 2017. On convergence and stability of gans. *arXiv preprint arXiv:1705.07215*.

Now for solving vanishing gradient problem we will implement LSGAN, the discriminator in this part does not have sigmoid as activation function for last layer, because the loss is calculated with l2 norm. the loss function for generator and discriminator is as follows.

$$\min_D V_{\text{LSGAN}}(D) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}\left[(D(\boldsymbol{x})-b)^2\right] + \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[(D(G(\boldsymbol{z}))-a)^2\right]$$

$$\min_G V_{\text{LSGAN}}(G) = \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[(D(G(\boldsymbol{z}))-c)^2\right],$$

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
   ConvTranspose2d-1            [-1, 512, 7, 7]         819,200
       BatchNorm2d-2            [-1, 512, 7, 7]           1,024
              ReLU-3            [-1, 512, 7, 7]               0
   ConvTranspose2d-4          [-1, 256, 14, 14]       2,097,152
       BatchNorm2d-5          [-1, 256, 14, 14]             512
              ReLU-6          [-1, 256, 14, 14]               0
   ConvTranspose2d-7          [-1, 128, 28, 28]         524,288
       BatchNorm2d-8          [-1, 128, 28, 28]             256
              ReLU-9          [-1, 128, 28, 28]               0
  ConvTranspose2d-10           [-1, 64, 56, 56]         131,072
      BatchNorm2d-11           [-1, 64, 56, 56]             128
             ReLU-12           [-1, 64, 56, 56]               0
  ConvTranspose2d-13          [-1, 3, 112, 112]           3,072
             Tanh-14          [-1, 3, 112, 112]               0
================================================================
Total params: 3,576,704
Trainable params: 3,576,704
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 9.19
Params size (MB): 13.64
Estimated Total Size (MB): 22.84
----------------------------------------------------------------
```

Figure25. Generator architecture

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           3,072
         LeakyReLU-2           [-1, 64, 32, 32]               0
            Conv2d-3          [-1, 128, 16, 16]         131,072
       BatchNorm2d-4          [-1, 128, 16, 16]             256
         LeakyReLU-5          [-1, 128, 16, 16]               0
            Conv2d-6            [-1, 256, 8, 8]         524,288
       BatchNorm2d-7            [-1, 256, 8, 8]             512
         LeakyReLU-8            [-1, 256, 8, 8]               0
            Conv2d-9            [-1, 512, 4, 4]       2,097,152
      BatchNorm2d-10            [-1, 512, 4, 4]           1,024
        LeakyReLU-11            [-1, 512, 4, 4]               0
           Conv2d-12              [-1, 1, 1, 1]           8,192
================================================================
Total params: 2,765,568
Trainable params: 2,765,568
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 2.31
Params size (MB): 10.55
Estimated Total Size (MB): 12.91
----------------------------------------------------------------
```

Figure26. Discriminator architecture

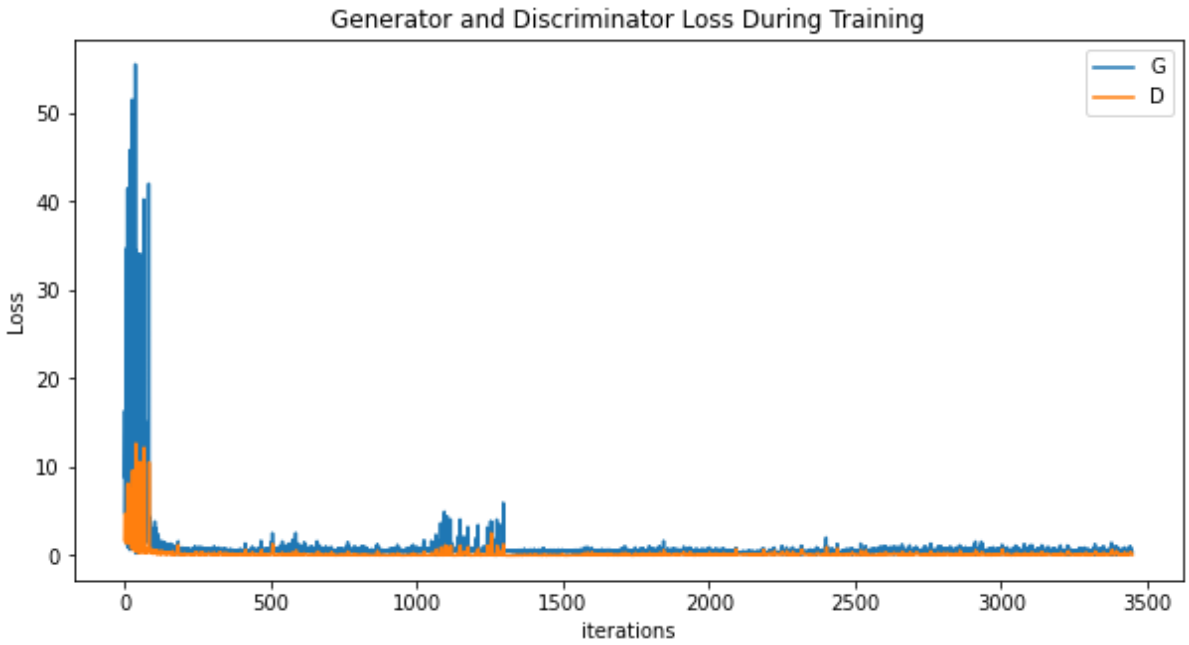

Figure27. Generated and discriminator L2 loss per iteration

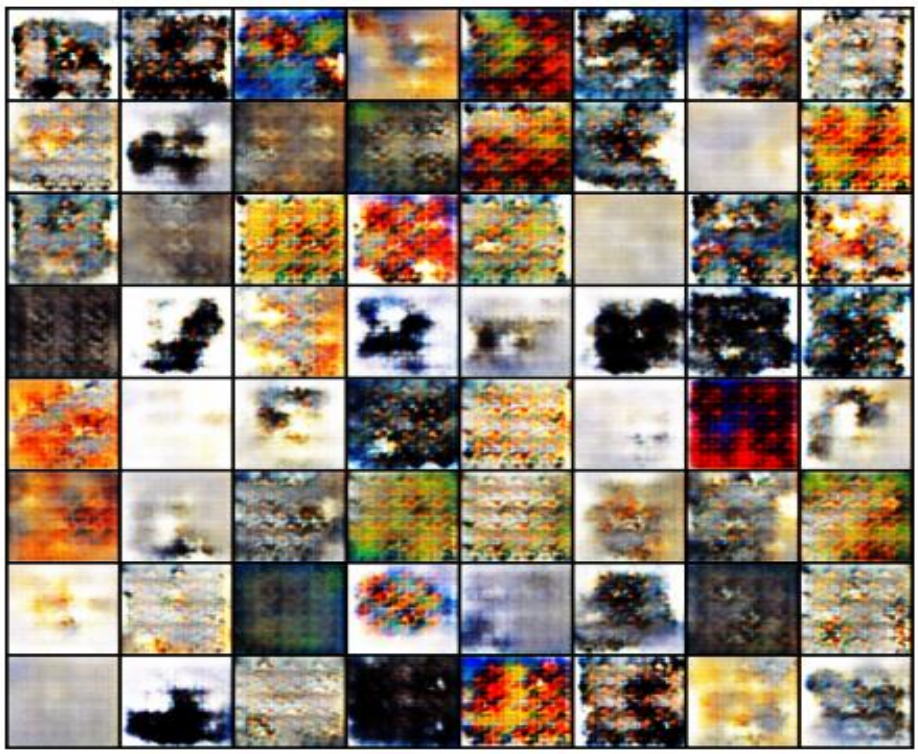

Figure28. Generated images after epoch150

Reference: https://machinelearningmastery.com/least-squares-generative-adversarial-network/

As we can see from the results, the DRGAN network is capable of solving mode collapse problem, and the image generated by DRGAN has more variety than the one generated with normal GAN.

But since our model didn't suffer from vanishing gradient problem, the LSGAN didn't improve the results very much and it still has less variety than DRGAN method because of mode collapses.

3 QUESTION3: VQ-VAE

3.1

The difference between VQ-VAE & VAE:

The fundamental difference between a VAE and a VQ-VAE is that VAE learns a continuous latent representation, whereas VQ-VAE learns a discrete latent representation. So far we have seen how continuous vector spaces can be used to represent the latents in an autoencoder. But latents do not necessarily need to be continuous vectors, it really just needs to be some numerical representation for the data. And one such, potentially desirable, alternative to a vector space is a discrete representation.

Advantages of using VQ-VAE:

VQ-VAE is a type of variational autoencoder that uses vector quantisation to obtain a discrete latent representation. It differs from VAEs in two key ways: the encoder network outputs discrete, rather than continuous, codes; and the prior is learnt rather than static. In order to learn a discrete latent representation, ideas from vector quantisation (VQ) are incorporated. **Using the VQ method allows the model to circumvent issues of posterior collapse** - where the latents are ignored when they are paired with a powerful autoregressive decoder - typically observed in the VAE framework. Pairing these representations with an autoregressive prior, the model can generate high quality images, videos, and speech as well as doing high quality speaker conversion and unsupervised learning of phonemes.

3.2

VQ-VAE:

The VQ-VAE (“Vector Quantized-Variational Autoencoder”; van den Oord, et al. 2017) model learns a discrete latent variable by the encoder, since discrete representations may be a more natural fit for problems like language, speech, reasoning, etc.

Vector quantisation (VQ) is a method to map K-dimensional vectors into a finite set of "code" vectors. The process is very much similar to KNN algorithm. The optimal centroid code vector that a sample should be mapped to is the one with minimum Euclidean distance.

Let e $\in R^{K\times D}$, i=1,…,K be the latent embedding space (also known as "codebook") in VQ-VAE, where K is the number of latent variable categories and D is the embedding size. An individual embedding vector is $e_i \in R^D$,i=1,…,K.

The encoder output $E\{x\} = z_e$ goes through a nearest-neighbor lookup to match to one of K-embedding vectors and then this matched code vector becomes the input for the decoder D(.):

$$\mathbf{z}_q(\mathbf{x}) = \text{Quantize}(E(\mathbf{x})) = \mathbf{e}_k \text{ where } k = \arg\min_i \|E(\mathbf{x}) - \mathbf{e}_i\|_2$$

Note that the discrete latent variables can have different shapes in different applications; for example, 1D for speech, 2D for image and 3D for video.

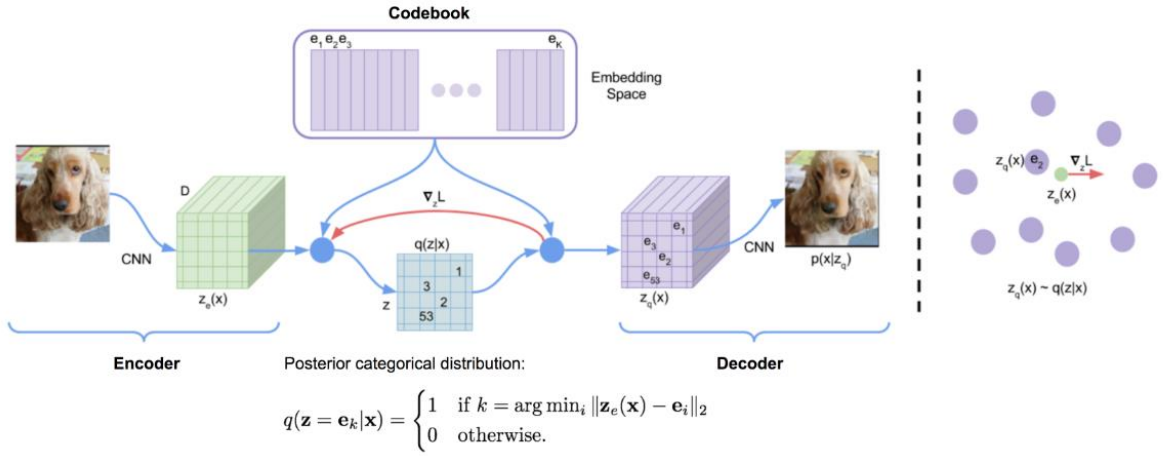

Figure29. The architecture of VQ-VAE

Because argmin() is non-differentiable on a discrete space, the gradients from decoder input is copied to the encoder output. Other than reconstruction loss, VQ-VAE also optimizes:

- VQ loss: The L2 error between the embedding space and the encoder outputs.
- *Commitment loss*: A measure to encourage the encoder output to stay close to the embedding space and to prevent it from fluctuating too frequently from one code vector to another.

$$L = \underbrace{\|\mathbf{x} - D(\mathbf{e}_k)\|_2^2}_{\text{reconstruction loss}} + \underbrace{\|\text{sg}[E(\mathbf{x})] - \mathbf{e}_k\|_2^2}_{\text{VQ loss}} + \underbrace{\beta\|E(\mathbf{x}) - \text{sg}[\mathbf{e}_k]\|_2^2}_{\text{commitment loss}}$$

The embedding vectors in the codebook is updated through EMA (exponential moving average). Given a code vector e_i , say we have n_i encoder output vectors, $\{z_{i,j}\}_{j=1}^{n_i}$ that are quantized to e_i :

$$N_i^{(t)} = \gamma N_i^{(t-1)} + (1-\gamma) n_i^{(t)} \quad \mathbf{m}_i^{(t)} = \gamma \mathbf{m}_i^{(t-1)} + (1-\gamma) \sum_{j=1}^{n_i^{(t)}} \mathbf{z}_{i,j}^{(t)} \quad \mathbf{e}_i^{(t)} = \mathbf{m}_i^{(t)} / N_i^{(t)}$$

Where (t) refers to batch sequence in time. N_i and m_i are accumulated vector count and volume, respectively.

3.3

In this section we intend to implement the VQ-VAE algorithm on both MNIST & CIFAR10 dataset.

1) Result of training on CIFAR10:

Figure30. Reconstructions result (Generated results)

Figure31. Original results (From CIFAR10 dataset)

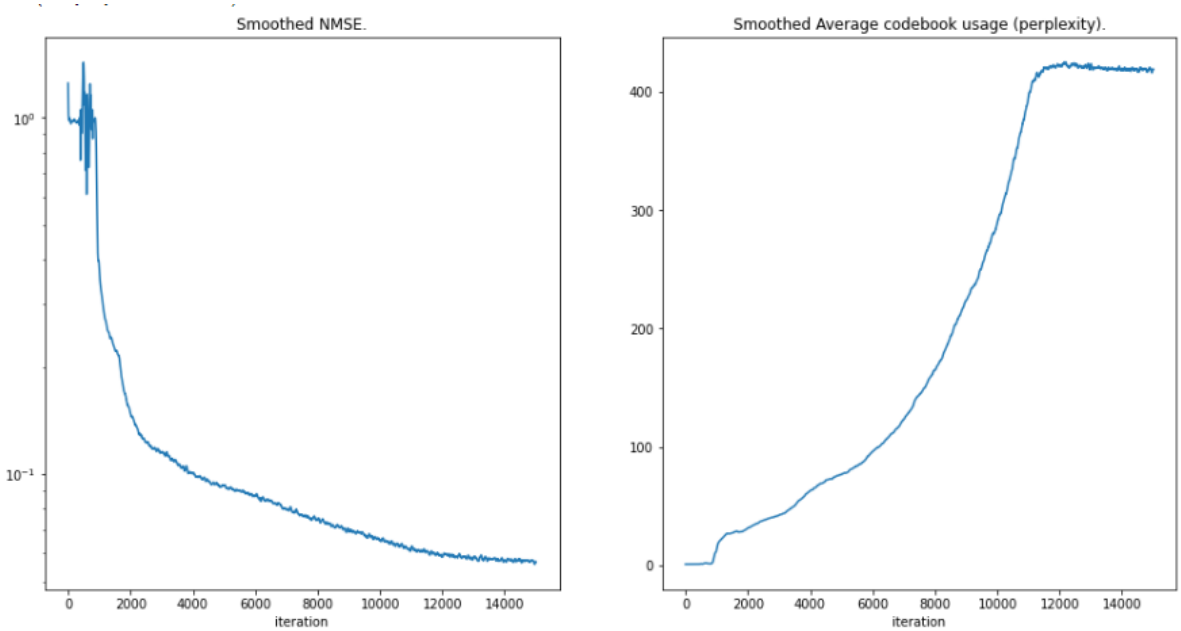

Figure32. Train perplexity & train loss during epochs (CIFAR10)

2) Result of training on MNIST:

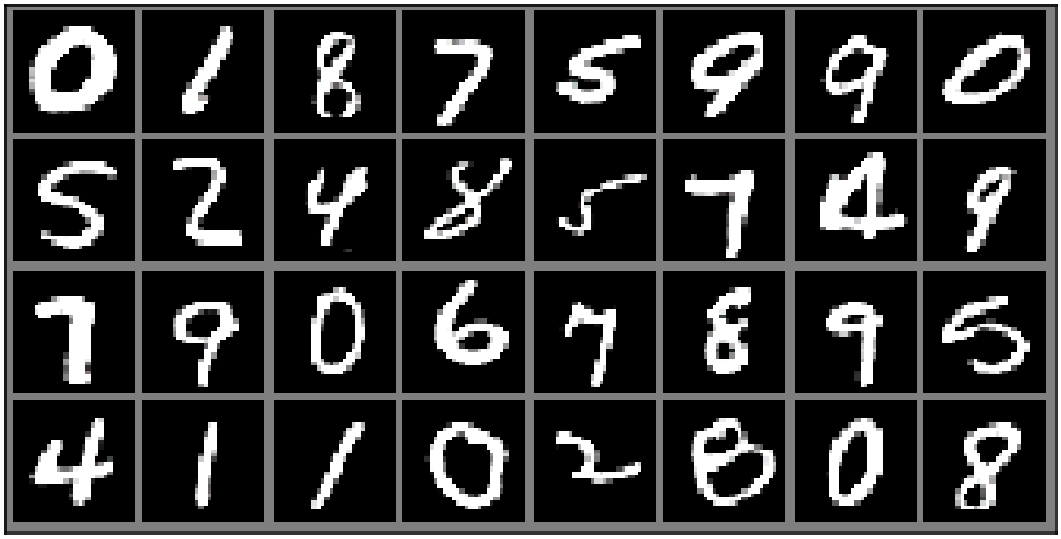

Figure33. Reconstructions result (Generated results)

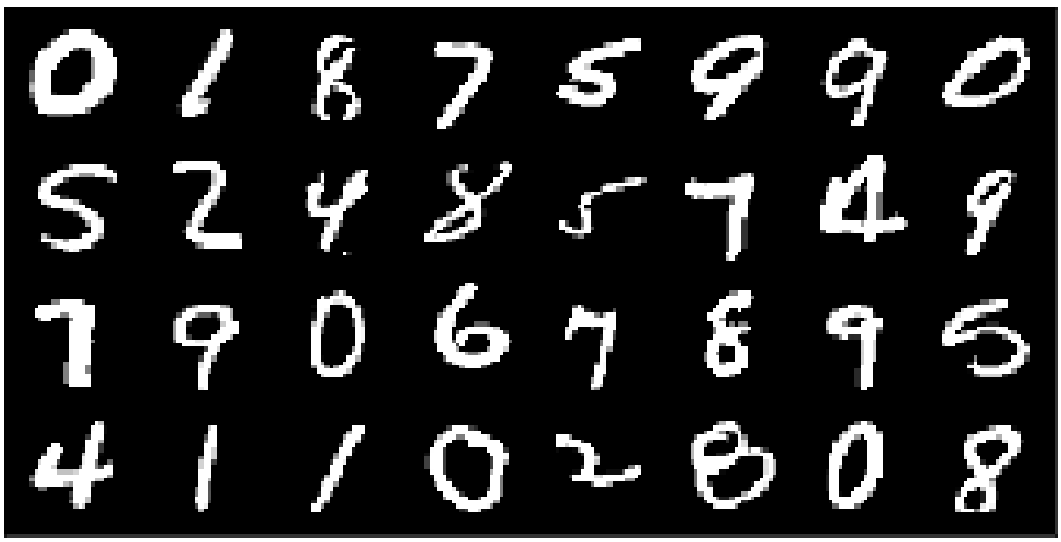

Figure34. Original results (From MNIST dataset)

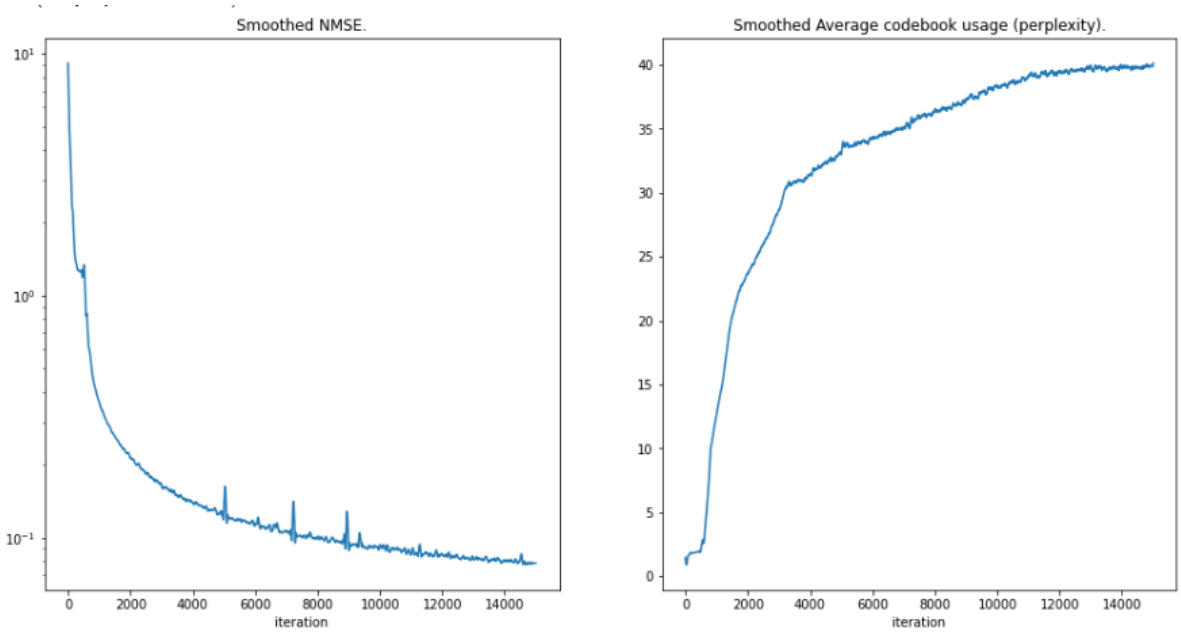

Figure35. Train perplexity & train loss during epochs (MNIST)

3.4

In this section we are going to compare the result of training in two cases using **MSELoss** or **NLLLoss:**

First I have attached the results and after that I have **compared the results** of both cases.

1) MSE-LOSS:

Result of training on CIFAR10:

Figure36. Reconstructions result (Generated results)

Figure37. Original results (From CIFAR10 dataset)

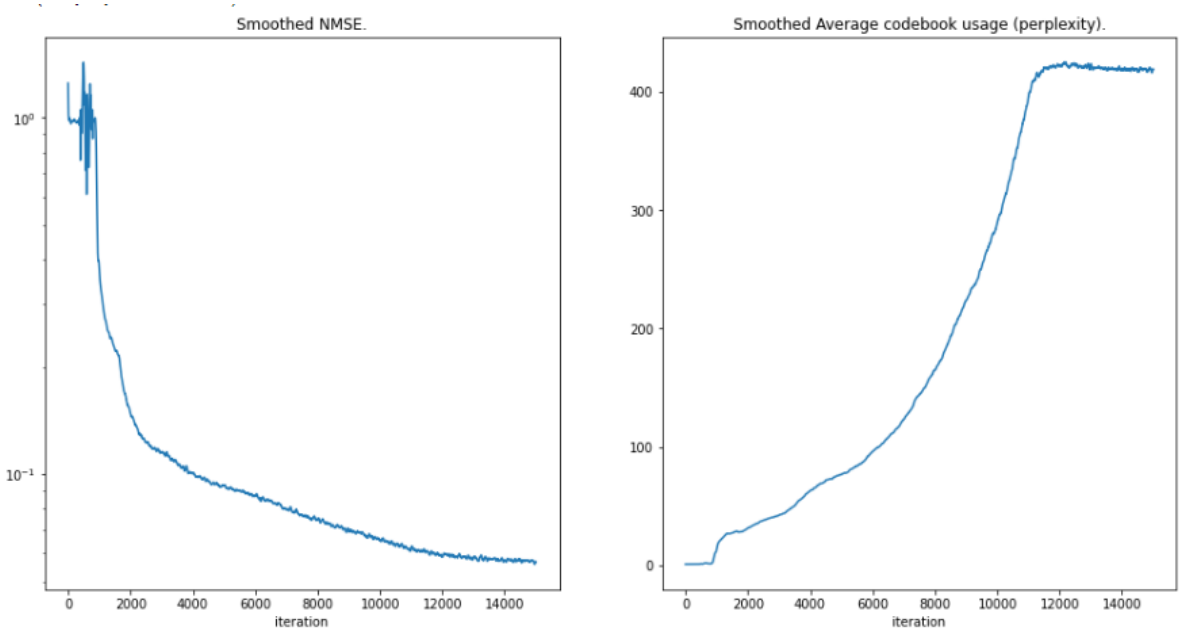

Figure38. Train perplexity & train loss during epochs (CIFAR10)

Result of training on MNIST:

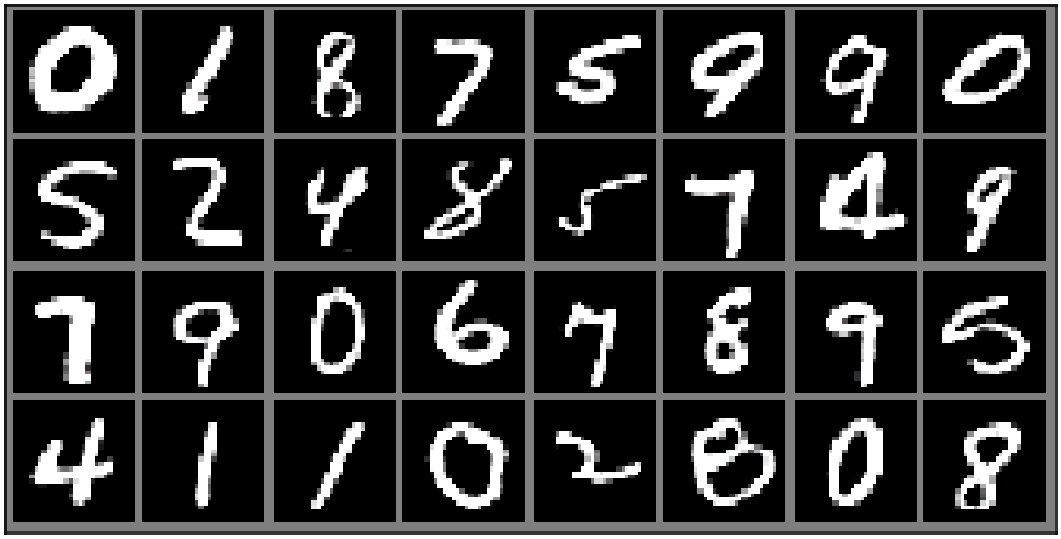

Figure39. Reconstructions result (Generated results)

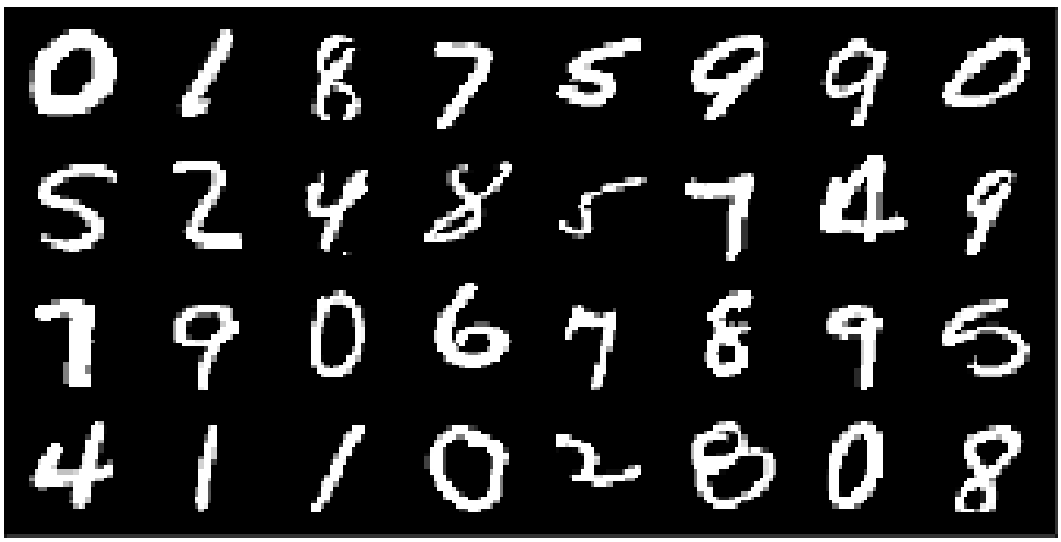

Figure40. Original results (From MNIST dataset)

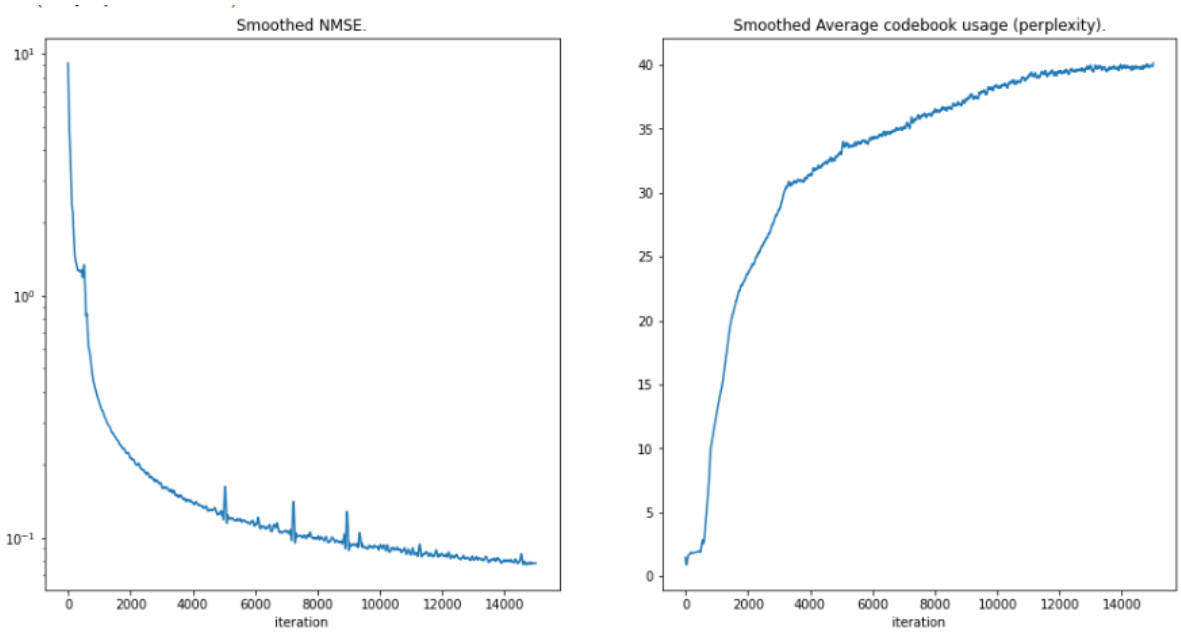

Figure41. Train perplexity & train loss during epochs (MNIST)

2) NLL-LOSS:

Result of training on CIFAR10:

Figure42. Reconstructions result (Generated results)

Figure43. Original results (From CIFAR10 dataset)

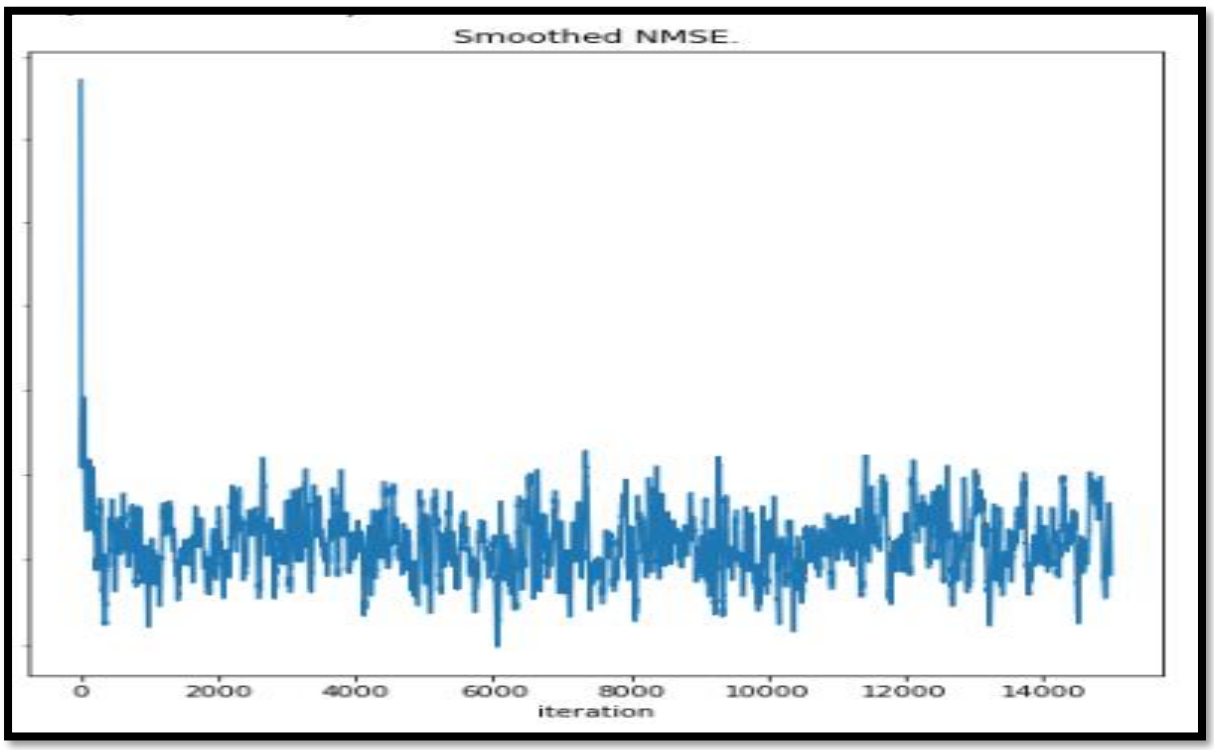

Figure44. Train perplexity & train loss during epochs (CIFAR10)

Result of training on MNIST:

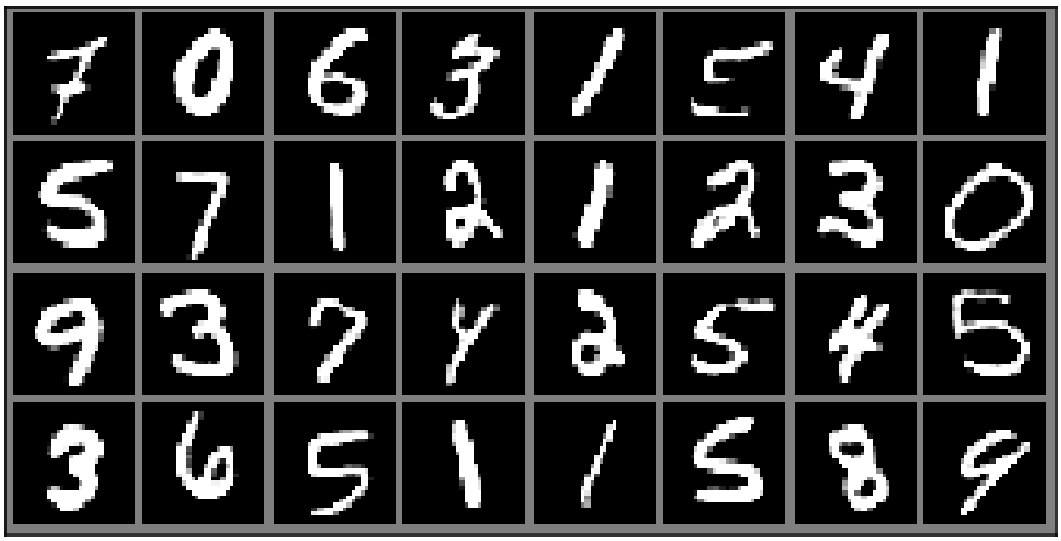

Figure45. Reconstructions result (Generated results)

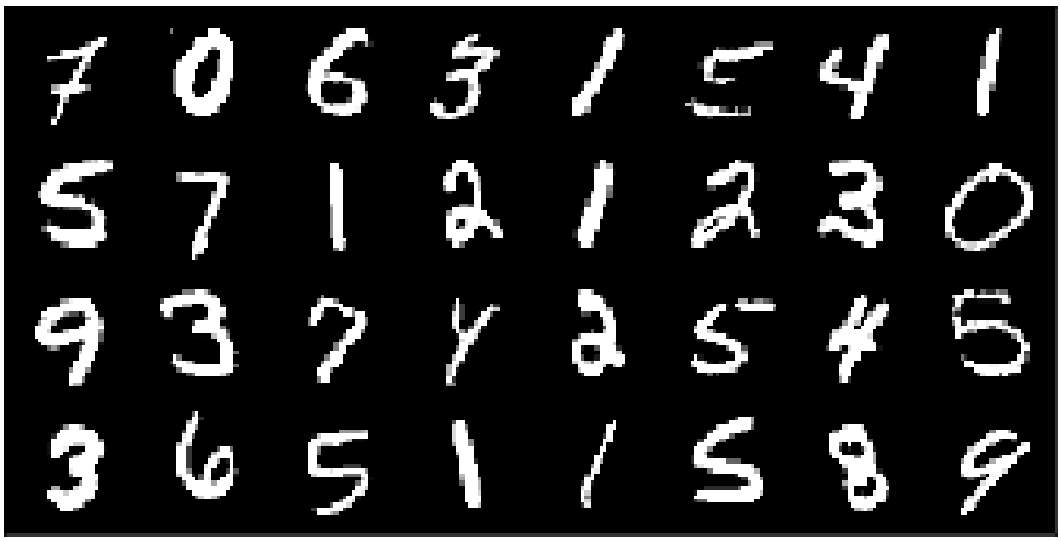

Figure46. Original results (From MNIST dataset)

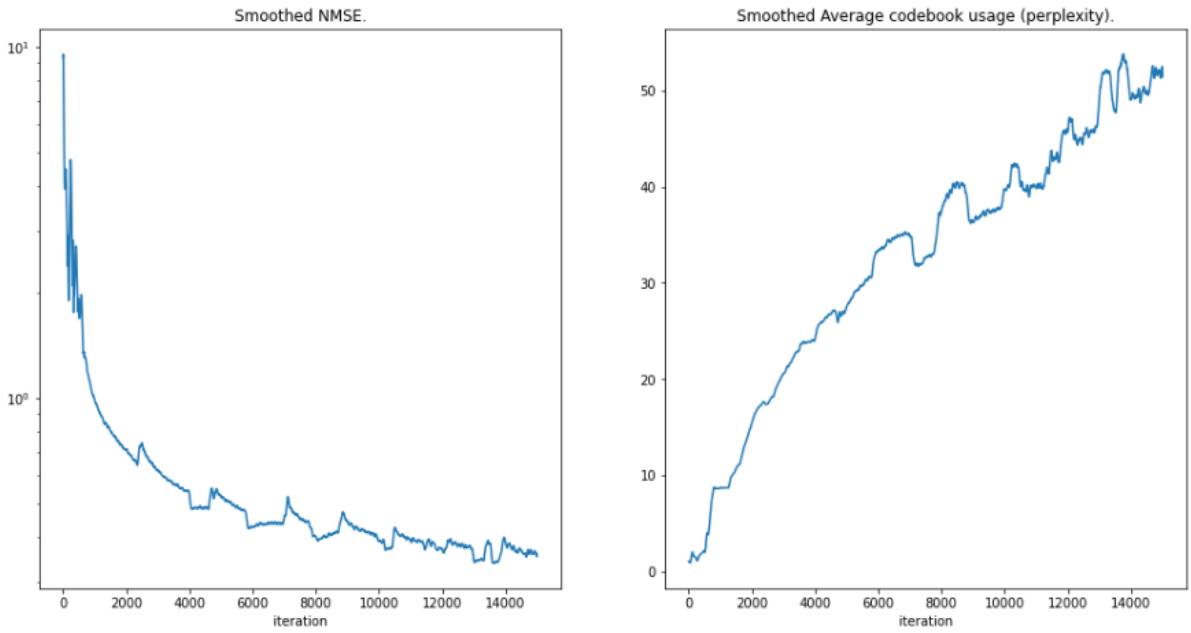

Figure47. Train perplexity & train loss during epochs (MNIST)

Comparing The Results:

According to the question descriptions we have implemented the algorithm for both NLL-Loss and MSE-Loss in the same number of epochs.

And as we can see the results of both of them are pretty good because VQ-VAE uses a strong approach in generating high-quality pictures and we expect to get sets of pictures closet to the original one.

But let's express some notes in case of NLL-Loss and Advantages:

finally found the answer in this paper "https://ieeexplore.ieee.org/document/374138" , also explained and referenced in a blogpost.

It mentions clearly in the **paper that NLL performs better then MSE** because the loss function becomes:

$$NLL = \sum log(\sigma^2(x_i))/2 + (\mu(x_i) - y_i)^2/2\sigma^2(x_i)$$

Now if I assume σ to be constant then my loss function becomes equivalent to MSE * const. which shows that the NLL acts better in comparison to MSE loss.

However, in current network σ is a variable, and hence the network gives higher weight to datapoints with lower variance**. Resulting in improved learning in current case.**

But the paper mentioned that if datasets are not large enough then NLL results in over fitting, which can again be explained on the same basis.

4 REFERENCES:

[1] https://lilianweng.github.io/posts/2018-08-12-vae/

[2] https://keras.io/examples/generative/vq_vae/

[3] VQ-VAE paper: https://arxiv.org/pdf/1711.00937.pdf

[4] OpenAI DALL-E blog: https://openai.com/blog/dall-e/

[5] OpenAI Jukebox blog: https://openai.com/blog/jukebox/

[6] Comp Three Inc. Autoencoders blog: https://www.compthree.com/blog/autoencoder/

[7] Jaan Altosaar VAE blog: https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

[8] https://github.com/deepmind/sonnet/blob/v1/sonnet/python/modules/nets/vqvae.py

[9] https://github.com/JovianML/opendatasets/blob/master/README.md#kaggle-credentials

[10] https://towardsdatascience.com/implementation-of-semi-supervised-generative-adversarial-networks-in-keras-195a1b2c3ea6

[11] https://machinelearningmastery.com/semi-supervised-generative-adversarial-network/

[12] https://github.com/bjlkeng/sandbox

[13] https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html#torch.nn.NLLLoss

[14] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html#torch.nn.MSELoss

[15] Semi-Supervised Learning with Generative Adversarial Networks (arxiv.org)