GAN vs VAE

Ariz Ahmad
University of Florida
Gainesville, FL
ariz.ahmad@ufl.edu

Abstract

Variational autoencoders and Generative Adversarial networks are popular generative models used for generating data. In this project I used the generic versions of these, and then made some changes to test variants of these on the MNIST dataset. Thereafter I have made an analysis of the results based on the architectural differences, and how these models compare with each other.

1 Introduction

Autoencoders learn dense representations of unsupervised data. These are called codings or latent representations of the input data. These have much lower dimensionality, and hence have a lot of applications like dimensionality reduction for visual purposes and feature detection. In this project, I have explored the generative aspect of autoencoders, generating new data which is very similar to the training data. The replicated data however, is not very accurate.

GANs, on the other hand, create much more accurate representation of the data, making them ubiquitous in the industry. These are used for other applications too, like photoshop and testing the strength of other models by generating contrived dataset to test them on.

Both GANs and Autoencoders are supervised models, which are used to generate data, but there are subtle differences in the two in terms of how they work.Autoencoders in essence, simply copy the input to the output, despite the difficulties it faces like constraint on the size of the network, or the introduction of noise. In other words, autoencoder learns the codings of the data under constraints.

GANs are composed of a generator and a discriminator neural network.The former generates data that further tries to fool the discriminator. These two networks compete against each other during the training process, and eventually obtain an accurate depiction of the input data.

2 Deep Generative Models Overview

Generative Models are unsupervised learning models for data distributions. We try to model the real distribution of the underlying data. Variational AutoEncoders and Generative Adversarial Networks are the two most ubiquitous used approaches of Generative Models. These try to generate images similar to the ones they are trained on. GAN tries to achieve this using a Generator and Discriminator while VAE maximizes lower limit of the log-likelihood of the data.

2.1 Variational AutoEncoder

Autoencoders was introduced by Diederik Kingma and Max Welling in 2013. Their unique feature is that they are probabilistic. Being a type of generative autoencoder, they generate images which are similar to the image in the training dataset. They are easier to train, because sampling from them is fast.

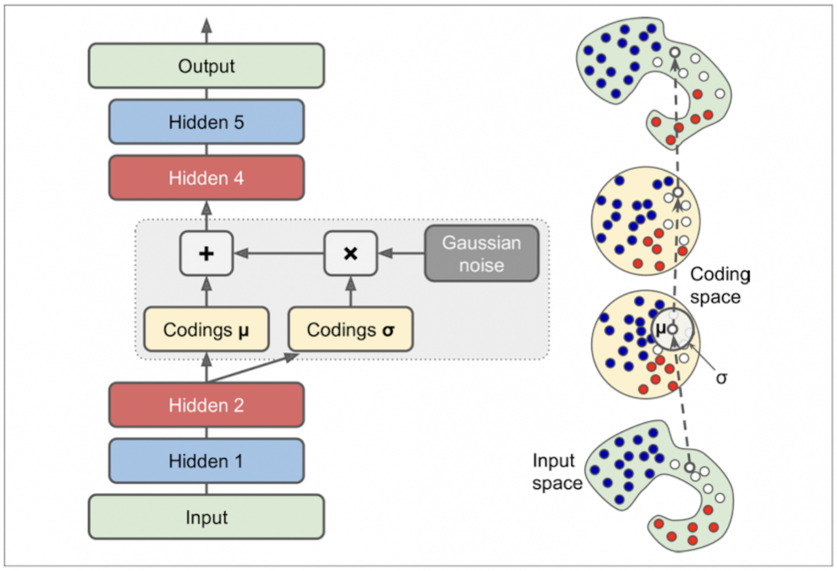


Figure 1: Variational autoencoder (left) and an instance going through it (right)

Figure 1 shows the structure of a variational autoencoder. There is an encoder followed by a decoder. To illustrate, this figure has only two layers for each of the encoder and decoder. The encoder then produces a coding of mean μ and standard deviation σ. After this, random sampling is done using a Gaussian distribution. Following this, decoder decodes the coding which has been sampled. The final output of this decoder is a training instance.

Despite the nature of the input distribution, variational autoencoder makes it appear as if the codes have been sampled from a Gaussian distribution. The cost function enables the code to be constrained to the latent coding space. An observation which can be made is that a random sampling from the trained Variational Autoencoder, and then decoding it, enables us to quickly generate a new instance.

The cost function consists of reconstruction loss and latent loss. Reconstruction loss, which may be cross-entropy, pushes the auto-encoder to generate the images same as the input. Latent loss enables the autoencoder to generate images as though they were generated from a simple Gaussian distribution. This is the KL divergence which is present between the actual distribution and the target distribution.

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log\left(\sigma_i^2\right) - \sigma_i^2 - \mu_i^2$$

Variational autoencoder Latent Loss

In the above equation, n is the dimensionality of the coding, and L is the latent loss. μ_i and σ_i are respectively the mean and standard distribution of the i-th data point. The encoder outputs vectors μ and σ. To speed up training, we can also make the output $\gamma = \log(\sigma^2)$.

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \gamma_i - \exp\left(\gamma_i\right) - \mu_i^2$$

Log of Variational autoencoder's latent loss

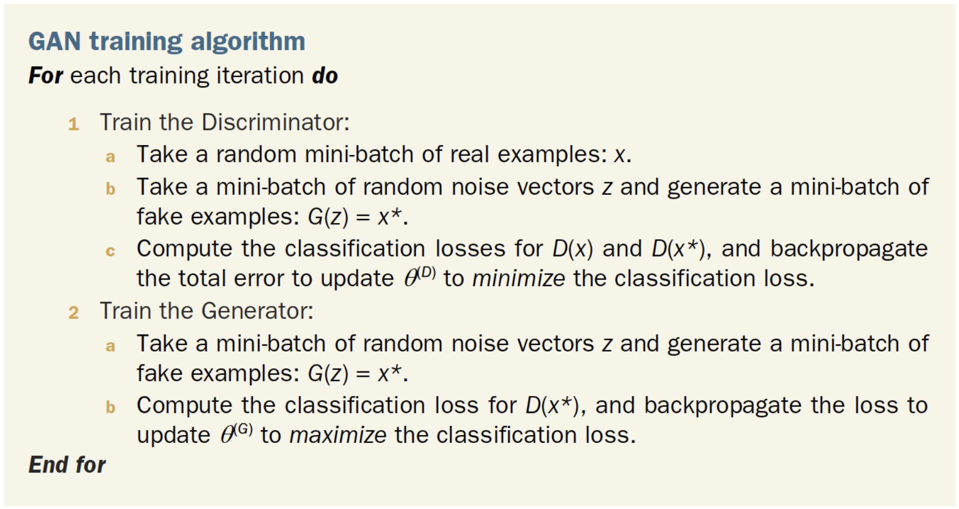
2.2 Generative Adversarial Network

Ian Goodfellow et al. proposed Generative Adversarial networks in their 2014 paper. It is based on the simple idea of making neural networks compete against each other, enabling them to consistently try to beat each other and thereby excel at performance. There are two neural networks components:

- Generator takes a random Gaussian distribution which can be considered to be a latent representation of the coding of the image generated. This can be compared to the functionality as that of the decoder in a variational autoencoder.
- Discriminator takes images from the generator and real input images and tries to guesses if the input image is real.

The generator and discriminator compete against each other during the training phase. The generator produces images which are as close to the real image as possible while the discriminator tries to distinguish between real and fake images. The training iteration has two phases:

- In phase 1, the discriminator is trained. The training set is sampled to obtain a batch of images and the generator produces an equal number of fake images. I then set labels to the real and fake images, 1 and 0 respectively. Thereafter the discriminator is trained with binary cross-entropy as the loss function. The weights of the discriminator are optimized using back-propagation.
- In phase 2, training of the generator is done. It produces a batch of fake images, while the discriminator tells whether the images are real or fake. No real images are used in this batch, causing the discriminator to think that all the images are real, as all of them are labelled 1 (wrongly). The discriminator weights are not changed in this phase, only the weights of the generator change because of backpropagation.



GAN training algorithm

```
For each training iteration do
    1 Train the Discriminator:
        a Take a random mini-batch of real examples: $x$.
        b Take a mini-batch of random noise vectors $z$ and generate a mini-batch of
          fake examples: $G(z) = x^*$.
        c Compute the classification losses for $D(x)$ and $D(x^*)$, and backpropagate
          the total error to update $\theta^{(D)}$ to minimize the classification loss.
    2 Train the Generator:
        a Take a mini-batch of random noise vectors $z$ and generate a mini-batch of
          fake examples: $G(z) = x^*$.
        b Compute the classification loss for $D(x^*)$, and backpropagate the loss to
          update $\theta^{(G)}$ to maximize the classification loss.
End for
```

Figure 2: GAN-Algorithm pseudocode

3 Tools

In this project, I implemented various Variational Autoencoders and Generative Adversarial Models. The code is present at

`https://github.com/ariz-ahmad/GAN/`

The code has been uploaded as a couple of ipython notebook files which can be run in Google Colaboratory.

3.1 Dataset

I used the MNIST dataset for this study. The acronym stands for Modified National Institute of Standards and Technology and it is a database containing handwritten digits. It is a subset of the larger dataset available at National Institute of Standard and Technology. I divided into two datasets: the training dataset and test dataset of 60,000 and 10,000 respectively.

3.2 Programming language and Frameworks used

I used python programming language for this project. I used Keras which is an open-source neural-network library written in Python. I implemented the models on my local machine and then tested it on Google Colaboratory.

4 Analysis

4.1 Architecture

4.1.1 Variational AutoEncoder

To obtain results and make analysis, I first of all imported all the dependencies. Then I set all the hyperparameters and global variables. I flattened the image to obtain a vector of size 784, from the original images of 28 * 28 pixels. In addition to that, I have a single intermediate layer. This layer has 256 nodes; the number 256 being obtained after testing with values higher and lower compared to it. The number of epochs here is 5.

I declared each layer, by stating the previous input as a group of arguments following the first set of arguments. For instance, a layer h would take the vector x as input. On compilation, the start and end are indicated, which are x and [z_mean, z_log_var and z] respectively. In this case, z is my latent space, which is a normal defined by it's own mean and variance respectively.

Next step is to define the encoder. z_mean defines the mean of the latent space, h is the input to the encoder. h is the intermediate layer. z_log_var defines the log variance of the latent space. encoder variable finally defines the encoder as a Keras model.

The next step is to sample from the latent space in order for this data point to be fed to the deocder. It should be taken to account that z_mean and z_log_var are both connected to the intermediate layer h with a dense 2-node layer. These are the mean and variance of the normal distribution.

Thus, this is the process we learn mean(μ) and variance(σ). This process allows the training and efficient sampling of images efficient. For generating images, we obtain images based on the parameters of mean and standard deviation we obtain and then feed the obtained image to the decoder.

Next step is to write the decoder. I have first written the layers as variables, to make it easier to re-use these values for the generation step. The encoder and decoder are subsequently bound to a single VAE module.

The overall loss if formed using the binary cross-entropy and KL divergence added together. KL divergence is used to measure the difference between the different distributions. The model is defined such that is starts at x and ends at x_decoded_mean. RMSprop is used to compile the model. Backpropagation was used to navigate the parameter space. The model was trained using the standard train-test split and normalization. I normalized then reshaped the training and test data to a flat 784 (28*28) vector. Then I applied the fit function to shuffle and get an unordered dataset. Validation data was also used for monitoring the progress while training.

4.1.2 Generative Adversarial Networks Architecture

I adopted some of the boilerplate code from the open source Github repository of GAN implementation by Erik Linder-Norén (https://github.com/eriklindernoren/Keras-GAN). Firstly, I imported all the libraries and packages to run the model on the MNIST dataset. I imported the dataset directly from keras.datasets. The images are grayscale, hence only a single channel. I specified 28 * 28 pixels to be the input dimension, which s same as the MNIST dataset. z_dim is used to set the noise vector z. I have set it to 100, which is passed as input to the Generator.

For implementing the Generator, I used a neural network with one single hidden layer which uses a Leaky ReLU as activation function. this hidden layer takes z as input and returns a 28 * 28 * 1 image as output. The tanh activation function is used at the output layer. I observed that tanh produced crisper images compared to sigmoid or other activation functions.

A 28 * 28 * 1 image is fed to the Discriminator, and it returns a probability of the image being real. I used a two-layer neural network for the Discriminator with 128 hidden units. I used Leaky RELU as the activation function for the hidden layers of the Discriminator.

Finally, I combine the Generator and Discriminator models. I keep the parameters to the Discriminator fixed. The discriminator is therefore not trained and only the generator is trained. I train the discriminator later independently as a compiled model. I use binary cross-entropy for calculation of loss function. I used Adam optimization for optimization, as is often the go-to optimizer for training GAN implementations.

For the purpose of training the GAN, I use random mini-batch of MNIST real images. I used the noise vectors z to obtain a min-batch of fake images. These images are used for training of the Discriminator network, keeping the parameters of the generator constant. In the next step, I trained the generator using a mini-batch of fake images while keeping the parameters of the Discriminator fixed. One-hot encoding has been used, 1 for real images and 0 for fake images. I sample from the standard normal distribution to get z. The real labels are assigned to real images and the fake ones to fake images by the Discriminator. Discriminator assigns the label of real for the fake images generated by the Generator. Images in the training set are rescaled from -1 to 1. Also, the fake images end up in the range from -1 to 1. All the inputs of the Discriminator are also rescaled in this way.

Finally, I set the number of iterations and batch size for training. I tried several variation =s for number of iterations using trial and error. I tried batch sizes of powers of 2: 16, 32, 64, 128, 256 and 512. Larger number of iterations takes a significant toll on the process of convergence. I used the training loss to set the number of iterations where the loss plateau.

4.2 Problems Encountered

4.2.1 Variational Autoencoders

VAE did not scale up as well as GAN. This may be due to the underlying fact that . Hence GANs are more complex and can learn different scenarios of the model rather than depending on the Gaussian distribution (which might not be the actual distribution).

VAE makes use of the latent space z. The Gaussian distribution is used to build the way VAE visualizes the data. Also, since the Gaussian distribution is centered near the mean, VAE tends to opt for the middle-ground.

4.2.2 Generative Adversarial Network

The constant competition between the Generator and Discriminator means that Nash equilibrium can be reached. The authors of the paper on GAN showed that only a single Nash equilibrium can be reached by the system but it is not guaranteed that the system would ever reach it.

Mode collapse is another big problem. At times the GAN specializes in a particular class of images, decreasing its diversity. The generator becomes good at producing image of a particular class, and the discriminator reinforces this behavior by producing more images of that particular class. The generator gradually fails to produce image of any other class. This might happen for another class after a while. In essence, the GAN cycles across selective classes, never being truly diversified.

At times it may happen that because of the competition between generator and discriminator, the parameters become unstable, and end up oscillating despite the training process beginning properly. This makes GANs very sensitive to hyperparameters and I had to spend a lot of time fine-tuning them.

5 Experimental Results

5.1 VAE

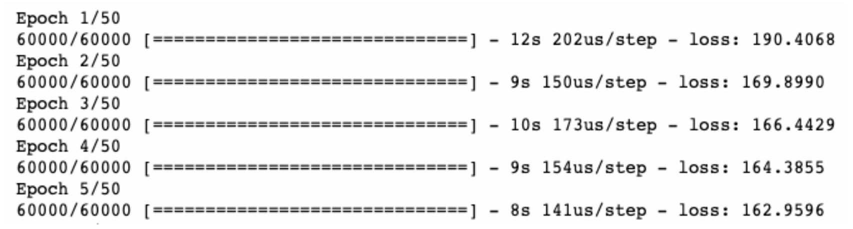
5.1.1 Definition

They key parameters of our model are the following:

- batch size = 100
- original dimension = 784
- latent dimension = 2
- intermediate dimension = 256
- epochs = 50
- epsilon std = 1.0

5.1.2 Training

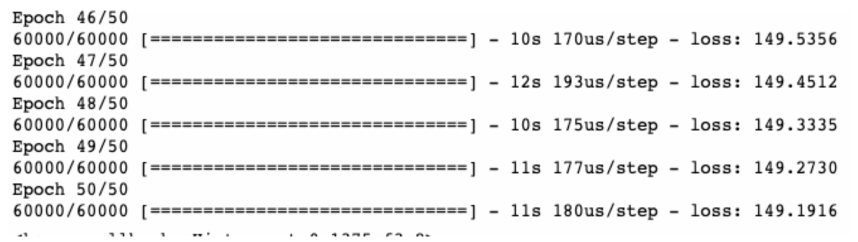
Following are the first five of the fifty training epochs.



```
Epoch 1/50
60000/60000 [==============================] - 12s 202us/step - loss: 190.4068
Epoch 2/50
60000/60000 [==============================] - 9s 150us/step - loss: 169.8990
Epoch 3/50
60000/60000 [==============================] - 10s 173us/step - loss: 166.4429
Epoch 4/50
60000/60000 [==============================] - 9s 154us/step - loss: 164.3855
Epoch 5/50
60000/60000 [==============================] - 8s 141us/step - loss: 162.9596
```

Figure 3: First 5 of 50 epochs

Following are the last five of the fifty training epochs.



```
Epoch 46/50
60000/60000 [==============================] - 10s 170us/step - loss: 149.5356
Epoch 47/50
60000/60000 [==============================] - 12s 193us/step - loss: 149.4512
Epoch 48/50
60000/60000 [==============================] - 10s 175us/step - loss: 149.3335
Epoch 49/50
60000/60000 [==============================] - 11s 177us/step - loss: 149.2730
Epoch 50/50
60000/60000 [==============================] - 11s 180us/step - loss: 149.1916
```

Figure 4: Last 5 of 50 epochs

5.1.3 Plot of digit classes in the latent space

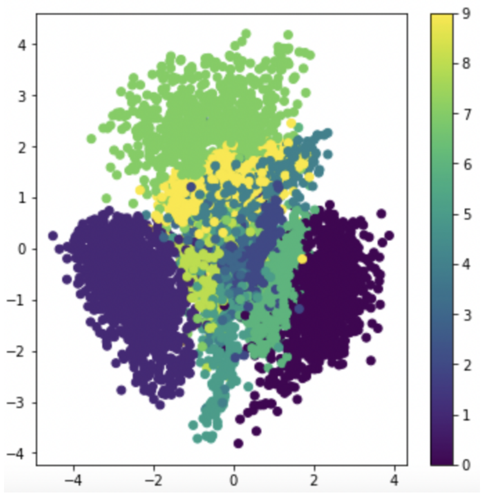
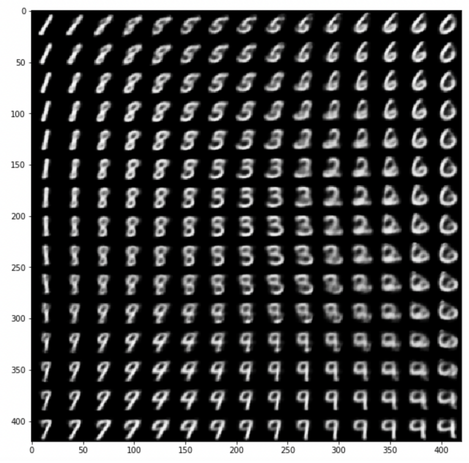


Figure 5: Digit classes plot

5.1.4 Digits output

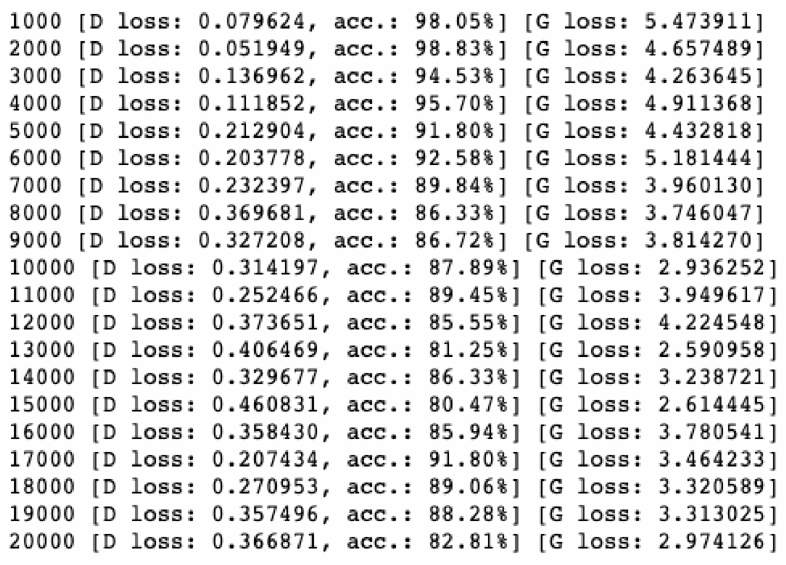
Following is a 2-dimensional manifold of the output of the digits generated. 15 * 15 digits are present in the figure generated below.



5.2 GAN

5.2.1 Training

- Number of iterations = 20000
- Batch size = 128
- Sample Interval = 1000



```
1000 [D loss: 0.079624, acc.: 98.05%] [G loss: 5.473911]
2000 [D loss: 0.051949, acc.: 98.83%] [G loss: 4.657489]
3000 [D loss: 0.136962, acc.: 94.53%] [G loss: 4.263645]
4000 [D loss: 0.111852, acc.: 95.70%] [G loss: 4.911368]
5000 [D loss: 0.212904, acc.: 91.80%] [G loss: 4.432818]
6000 [D loss: 0.203778, acc.: 92.58%] [G loss: 5.181444]
7000 [D loss: 0.232397, acc.: 89.84%] [G loss: 3.960130]
8000 [D loss: 0.369681, acc.: 86.33%] [G loss: 3.746047]
9000 [D loss: 0.327208, acc.: 86.72%] [G loss: 3.814270]
10000 [D loss: 0.314197, acc.: 87.89%] [G loss: 2.936252]
11000 [D loss: 0.252466, acc.: 89.45%] [G loss: 3.949617]
12000 [D loss: 0.373651, acc.: 85.55%] [G loss: 4.224548]
13000 [D loss: 0.406469, acc.: 81.25%] [G loss: 2.590958]
14000 [D loss: 0.329677, acc.: 86.33%] [G loss: 3.238721]
15000 [D loss: 0.460831, acc.: 80.47%] [G loss: 2.614445]
16000 [D loss: 0.358430, acc.: 85.94%] [G loss: 3.780541]
17000 [D loss: 0.207434, acc.: 91.80%] [G loss: 3.464233]
18000 [D loss: 0.270953, acc.: 89.06%] [G loss: 3.320589]
19000 [D loss: 0.357496, acc.: 88.28%] [G loss: 3.313025]
20000 [D loss: 0.366871, acc.: 82.81%] [G loss: 2.974126]
```

Figure 6: 20 epochs

5.2.2 Training loss for Generator and Discriminator

Following is the plot of training loss for generator and discriminator.

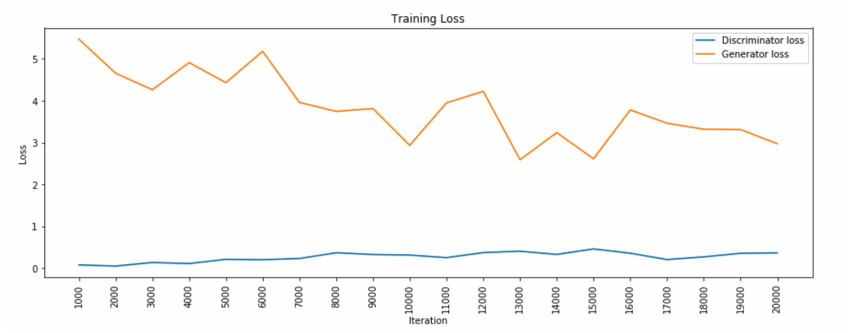


Figure 7: Training Loss for GAN

5.2.3 Images generated by GAN

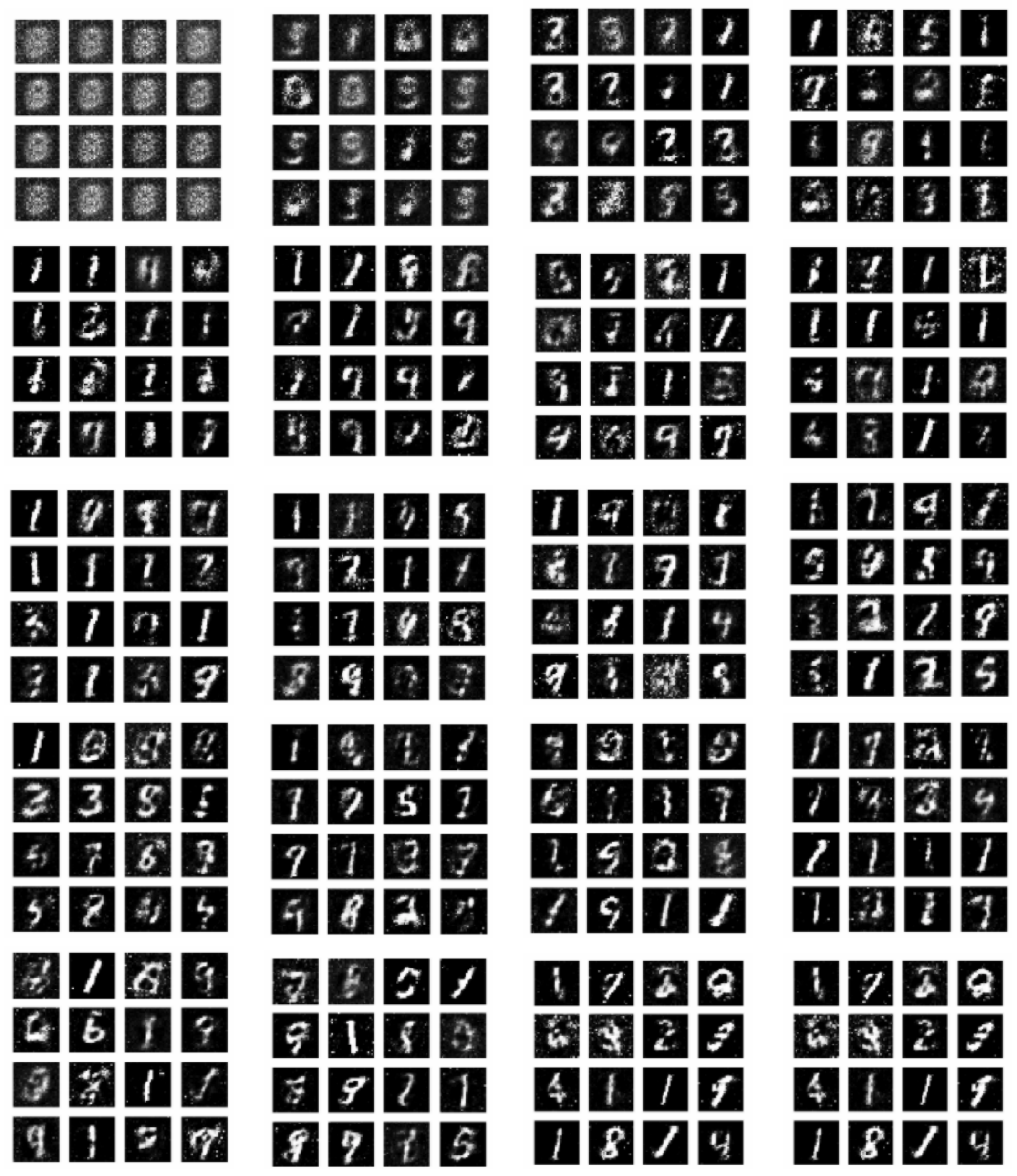


Figure 8: 20 images generated in each of the epochs

5.2.4 Discriminator Accuracy

Following is a plot of Discriminator Accuracy.

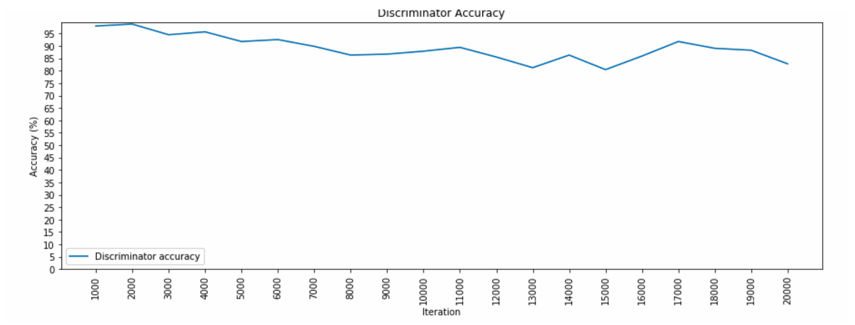


Figure 9: Discriminator Accuracy

6 Conclusion

Following are the conclusions I drew from my experiments on the performance of VAEs and GANs and their comparative study.

- Variational Autoencoders do not produce as sharp images as GANs. This is because VAE draws on Gaussian distribution.
- GAN is very sensitive to hyperparameters and some times does not train properly, especially in the initial rounds. This is because the loss is high because sampling is done through random noise.
- Non-convergence is an issue for GANs, and a field for active research. GAN produces better results than VAE on a limited set of values.
- Due to the game-theoretic nature of GANs, the generator and discriminator have to learn at similar rates. If either of them outperforms the other, the other has trouble learning because of low update of the gradient. This phenomenon is called Mode collapsing.

References

[1] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.

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

[3] Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media, 2019.

[4] Langr, Jakub, and Vladimir Bok. GANs in Action: Deep Learning with Generative Adversarial Networks. Manning Publications, 2019.

[5] Chollet, Francois. Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek. MITP-Verlags GmbH Co. KG, 2018.