Comparative performance analysis of GAN vs VAE

Nikhil Tank[*]
Department of Computer Science
University of Florida
Gainesville, FL 32608
`nikhil.tank@ufl.edu`

Abstract

The objective of the project is to perform a comparative performance analysis of Generative Adversarial Network (GAN) vs Variational Autoencoder (VAE) in generating Human Handwritten Gray-scale images of numbers. Both of the approaches are unsupervised machine learning techniques and are used to study an underlying data distribution. The models are trained on the MNIST Dataset, with an end goal of generating visually appealing handwritten number images.

1 Introduction

The GAN and VAE are an exciting new addition to the powerful unsupervised machine Paradigm. They study the underlying data distribution so well that they can generate a random, new output that looks indistinguishably similar to training data. They are unsupervised in the sense that they do not any label information to train the neural network model. The model creates new data instances identical to training data without any prior information of the correct answer.

Generative Adversarial Network (GAN)[1] is a class of machine learning framework originally designed as a type of generative model for unsupervised learning by Ian Goodfellow in 2014. The approach work by taking noise as an input parameter and generate candidates using "Generative Network", which is evaluated by "Discriminative network" for its originality in comparison to the real image from the training dataset. A good model will procedure novel candidates that the discriminator thinks are not synthesized (meaning to "fool" the system that the candidate is from the true data distribution instead of fake or synthesized). The network works by fine-tuning both the Generator and Discriminator over each epoch (iteration) and each works towards improving each other's performance. Since the two networks are always fighting against each other, that's why they can be called adversaries, because of this they are termed as "Generative Adversarial Network".

Variational Autoencoder (VAE)[2] is an autoencoder that regularizes and breaks-down (encode) the input to its latent space such that during the decoding phase some new data can be generated from it while keeping the good properties (principal component) of the input. The approach work by taking the input parameter and setting encoder and decoder as a neural network and learns the best encoding-decoding scheme over iteration and optimization steps. Thus, eventually, the encoder and decoder create a bottleneck for the data and feature in the middle that ensures only the principal component or the key features of the input to get through and feeding it to be reconstructed. This is in contrast with the GAN, as GAN generates the novel candidate in the middle of the process and the True/False as an output of the model.

The beauty of machine learning algorithms lies in large and equally spread data set. 60,000 hand-written Gray-scale number images from MNIST[3] dataset were used for the training of these two models. Additional 10,000 images were taken as a test dataset for the VAE. The images of the dataset

[*](https://www.linkedin.com/in/nikhiltank94, https://github.com/NikhilTank94)

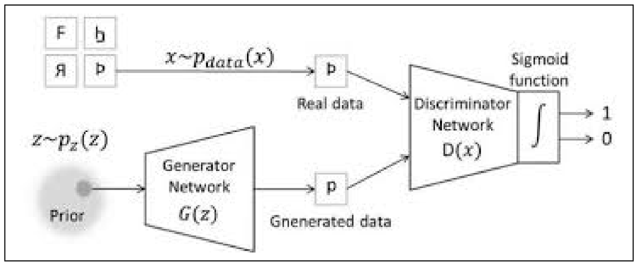


Figure 1: GAN block diagram

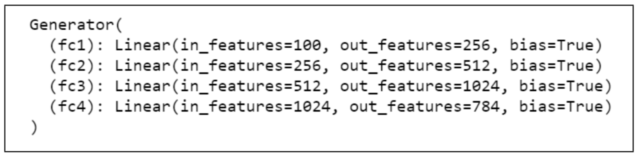

```
Generator(
  (fc1): Linear(in_features=100, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=1024, bias=True)
  (fc4): Linear(in_features=1024, out_features=784, bias=True)
)
```

Figure 2: Generator NN layer

can be imported from Pytorch's an open-source machine learning library called Pytorch. Pytorch is a widely used computer vision and natural language processing applications.

2 Generative Adversarial Network

2.1 Structure of GAN

The Neural Network for Generative Adversarial Network has 2 components Generator and Discriminator.

2.1.1 Generator Neural Network

The Generator network is made from the composition of the 4 layers of the neural network, with each layer's output being passed through a leaky ReLU function. The benefit of using leaky ReLU is that it allows small negative values to the output when the input is less than zero. As a result, it speeds up the training and convergence process over standard ReLU. The slope of the Leaky ReLU is set to be 0.2 in every layer of the Generator. The final output of this NN is passed through a Tanh function for classification.

The lowest level of the Generator network starts with 100 in-features and diverges out to 784 out-features (28X28), each value corresponding to a pixel value in the GAN's generated image. The biases are set to True and are calculated using the back-word propagation step. The Generator model is generating the synthesized version of the Handwritten numbers by taking the randomly varying tensor as an input.

For measuring the weight of the Generator network we are using nn.BCELoss as a metric. The BCE-Loss is calculated between the Discriminator's capability of distinguishing between the actual MNIST image and the image generated by the Generator. In a way, the Discriminator NN is training the Generator NN. The learning rate is kept at 0.0002 to avoid overtraining the network.

2.1.2 Discriminator Neural Network

The Discriminator network is made from the composition of the 4 layers of the neural network, with each layer's output being passed through a leaky ReLU and a drop out function. The drop-out value is set to be 0.3, this means that model will throw away 30 percent of the information from the training step. This is an elective way of avoiding overtraining. The final output of this NN is passed through a Sigmoid activation function for classifying the image into real or fake.

The lowest level of the Discriminator network starts with 784 in-features (each value corresponding to a pixel value from the generated image) and converging to 1. The biases are set to True and are

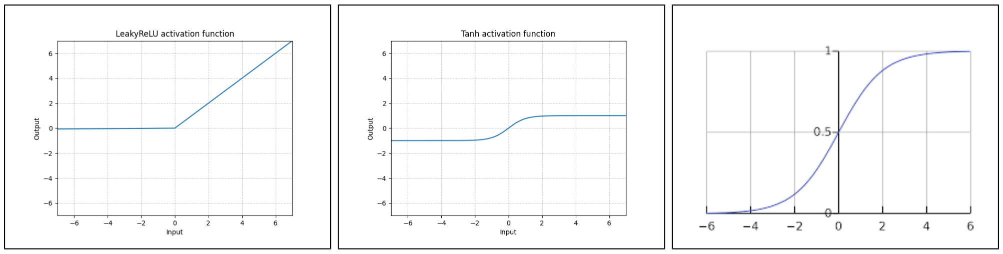
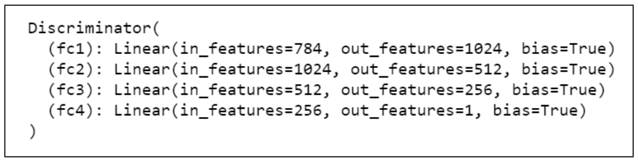


Figure 3: LeakyReLU funtion, Tanh funtion, Sigmoid Funtion



```
Discriminator(
  (fc1): Linear(in_features=784, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=256, bias=True)
  (fc4): Linear(in_features=256, out_features=1, bias=True)
)
```

Figure 4: Discriminator NN layer

calculated using the back-word propagation step. The Discriminator returns possibilities, a number between 0 and 1, for both real and fake data, 1 corresponding to real and 0 correspondings to fake.

For measuring the weight of the Discriminator network we are adding BCELoss for distinguishing between an image from the MNIST data set and BCE-Loss for distinguishing between fake images made by Generator. In process of minimizing the BCE-Loss, both Generator NN and Discriminator NN are training each other. The learning rate is kept at 0.0002 to avoid overtraining this network.

2.2 Problem faced with GAN

While working with GAN I have faced multiple problems. After reading about them, I found out that these problems are areas of active research. In the earlier phase of the project, I was trying to generate a wide spectrum of numbers from the model, but the model was mostly generating '8's and '6's. I can know that the generator was always trying to generate an output that was most plausible to the discriminator. This is was causing due to local minima, the network was somehow stuck in it and was generating repetitive output, the most plausible ones. I have to increase drop out to work around the problem. This form of GAN failure is called Mode Collapse. This can be seen in the GIF generated from the images (refer GAN implementation file on github)

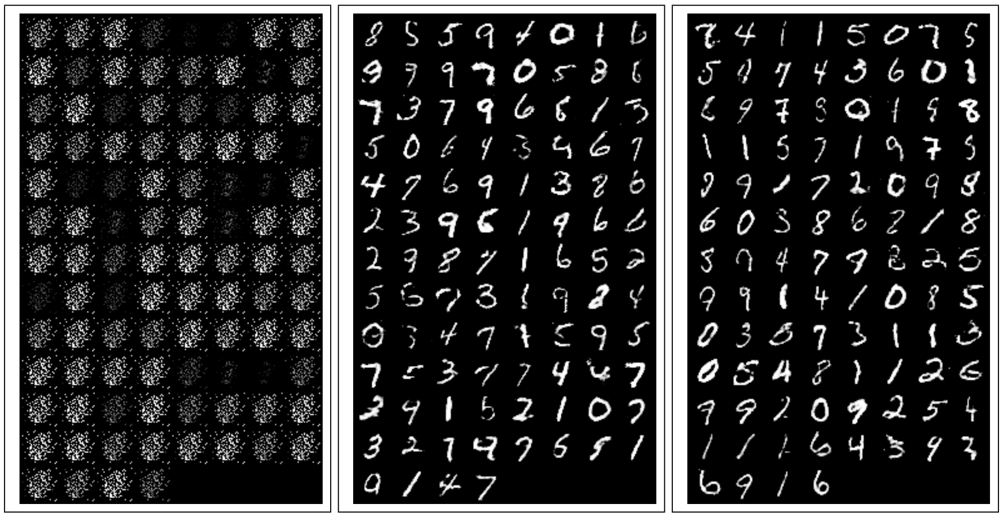


Figure 5: GAN output for 1, 500, 1000 epoch

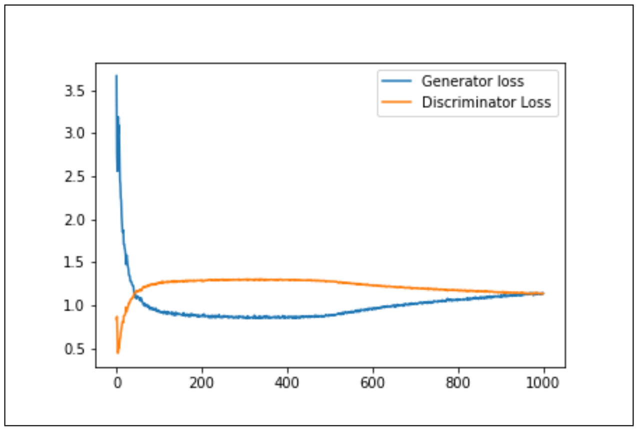
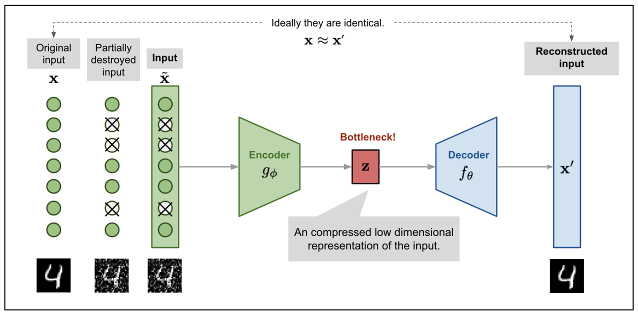


Figure 6: Generator loss and Discriminator loss over 1000 Epoch



Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Original input
$\mathbf{x}$
Partially destroyed input
Input
$\tilde{\mathbf{x}}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
Reconstructed input
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Figure 7: VAE block diagram

3 Variational Autoencoder

3.1 Structure of VAE

Variational Autoencoder (VAE) is an autoencoder that regularizes and breaks-down (encode) the input to its latent space such that during the decoding phase some new data can be generated from it while keeping the good properties (principal component) of the input. The VAE model contains two parts, namely Encoder and Decoder. The VAE model is based on a probabilistic approach, based on Bayesian Inference. The model tries to learn the true probability distribution of the data from the latest information on which it is trained. The trained model is used to can generate new data from that learned distribution.

The Encode network is made from the composition of the 4 layers of the neural network, with two stacked layers for output and one-one layer for mean and variance. This step is to call the Reparameterization step (refer to figure 8). The output from the primary layer NN is passed through Regular ReLU. The Decoder network is made similar to encode but in a bottom-up fashion. Three linear layers of NN with ReLU function at the output, followed by a Sigmoid function. The average training loss was calculated using summing BCE loss and KL Divergence for reconstructed image and image from MNIST dataset, this was used to back-propagate and derive the new weights of the NN model. The learning rate of 0.0001 after 3 trials with the training.

The input to the VAE model is 784 features (28X28), which the encoder tries to reduce to 2 features and from these 2 features, a new image is generated by the decoder. The reconstructed image output was poor and blurry (Refer Figure 7)and the trained model was not able to create any recognizable handwritten digit image from random sample space. With less number of the epoch, the model was not able to approximate the image in the input and because of that, the reconstructed image was very blur and unrecognizable. Even a ’0’ was indistinguishable from ’7’ or ’6’. By increasing epoch the images were sharp but still, a slightly hazy appearance was present. The result further improved by reducing the batch size of the model.

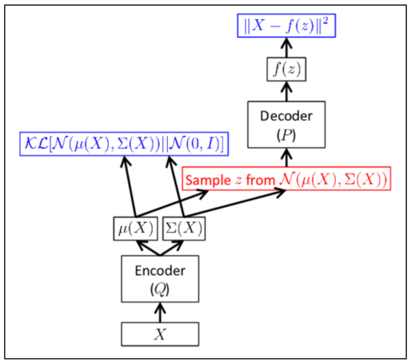
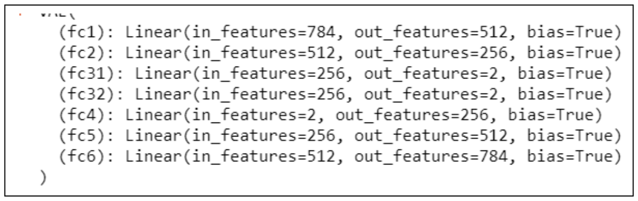

$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)]$
Sample z from $\mathcal{N}(\mu(X), \Sigma(X))$
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
X

Figure 8: Reparameterization step VAE [7]



```
VAE(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc31): Linear(in_features=256, out_features=2, bias=True)
  (fc32): Linear(in_features=256, out_features=2, bias=True)
  (fc4): Linear(in_features=2, out_features=256, bias=True)
  (fc5): Linear(in_features=256, out_features=512, bias=True)
  (fc6): Linear(in_features=512, out_features=784, bias=True)
)
```

Figure 9: VAE NN layer

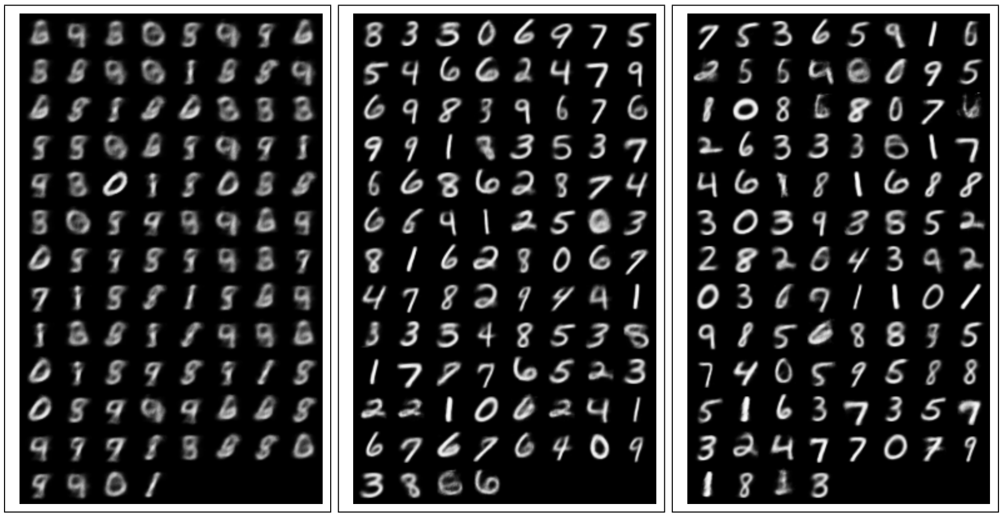


Figure 10: VAE output for 1, 500, 1000 epoch

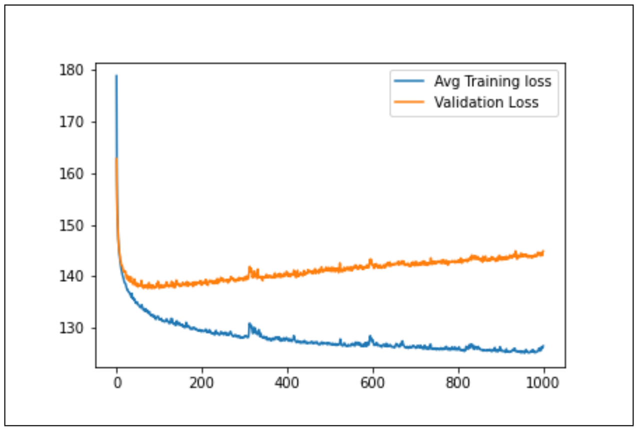


Figure 11: Average Training loss and Validation loss over Epochs

3.2 Problem faced with VAE

Using the data from the average training and validation losses, we can see that they are diverging from each other by increasing epoch (Refer figure 11). This could be a result of over training the data on the less equally spread data. The model trains fast but with generated blur images. Some of the number generated were consistently poor. This can be seen in the GIF generated from the images (refer VAE implementation file on github)

4 Results

The draw a clear picture and to clearly distinguish between the GAE and VAE, I have implemented the models on the Linear NN. The results can be further improved in both of the approaches by using convolutions layers and FID score, but by doing so the model will loose their originality.

In the early stage of the project I was using a CPU for training my model, but it used to take 20-30 mins on 50 epochs of VAE, whereas 4-5hrs for 200 epochs of GAN. In this project I got access to a GPU, because of that I was able to run 1000 epoch with a batch size of 100 images in just 4 hrs. By running only 50 epochs, VAE was able to generate better visually pleasing results than GAN in a lesser number of epochs. Even though the images generated by the VAE model were a little blur, the VAE gave better computation time.

After running 600+ epoch, I saw something strange happened. The Discriminator loss and Generator loss started to converge. The improvement in the images was more prominent, there was less Salt-pepper noise. Images generated before 500 epochs were slightly broken and were less distinguishable. But after 600 epochs they were sharp and complete. I came across 2 techniques to evaluate my GAN model, the Inception model, and Fréchet Inception Distance (FID) score. But the Inception model expects a 3-channel image, for that one solution was to triple the MNIST single-channel and modify the generator to output in the same 3-channel space. But doing this we would be changing the network and we would be no longer generating the MNIST images.

5 Conclusion

The Final GAN Model was able to reconstruct images such that it was hard to tell the real image from the generated fake image and the new images constructed from random sample space were identical to the handwritten digits. But, the computation time required for GAN was almost 4 times that of VAE. Refer to figure 5, after training the GAN for 1000 epoch, as out of 100 generated images 95 we so clear, sharp, and without artifact. Therefore, the model can be said 95 percent accurate in generating handwritten images. Similarly, for the 1000 epoch, the VAE output was recorded, most of the images were still blur and smudgy, but still readable. Only 2 images had artifacts. Hence, we can say GAN was better at generating handwritten fake images.

Implementation: Code

GAN: https://github.com/NikhilTank94/GAN

VAE: https://github.com/NikhilTank94/VAE

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, & Sherjil Ozair (2014) Generative Adversarial Nets (eds.) Departement d'informatique et de recherche op ´ erationnelle ´ Universite de Montr ´ eal Montreal, QC H3C 3J7

[2] Kingma, Diederik Welling, Max. (2019). An Introduction to Variational Autoencoders. Foundations and 149 Trends® in Machine Learning. 12. 307-392. 10.1561/2200000056.

[3] http://yann.lecun.com/exdb/mnist/

[4] https://debuggercafe.com/generating-mnist-digit-images-using-vanilla-gan-with-pytorch/

[5] https://debuggercafe.com/getting-started-with-variational-autoencoder-using-pytorch/

[6] https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html

[7] https://medium.com/analytics-vidhya/generative-modelling-using-variational-autoencoders-vae-and-beta-vae

[8] https://www.researchgate.net/figure/The-schematic-diagram-of-the-generative-adversarial-network-GAN_fig1_338550526