Autoencoders vs GANs : Comparative Study of Deep Generative Models

Amit Asish Bhadra
6649-4087
University of Florida
amitbhadra@ufl.edu

Disha Nayar
6199-1035
University of Florida
dnayar@ufl.edu

Sagnik Ghosh
3343-6044
University of Florida
sagnik.ghosh@ufl.edu

Abstract

The Project Report compares the architecture and techniques of two different generative models, Variational Auto Encoders and Generative Adversarial Networks. Although there has been significant progress since the origination of the two models, we provide a basic understanding of the two. The report introduces the Generative Models catering to the needs of approximating probability distribution of data sets. Even though these models solve the same problem, the approaches are entirely different. We run our experiments using the MNIST data set on these two models.

1 Introduction

The machine learning problems are broadly divided into two categories, supervised learning and unsupervised learning. Supervised learning is comparatively easy, we build models that map input data to output labels. The challenge is to approximately estimate the mapping function so that given a never seen input data, the function can determine the correct label efficiently. Unsupervised learning on the other end is challenging, where the goal is to model the hidden structure in the input data. Clustering, Gaussian Mixture models, Dimensionality reduction are the basic unsupervised learning problems. Unsupervised Learning is a relatively unexplored area of research in the field of Machine Learning. There are a lot of open problems, if we are able to successfully learn and represent the underlying structure of data, we will possibly be able to understand the structure of the visual world.

We focus on the topic of generative models, a relatively unexplored field of Unsupervised Learning. A lot of recent developments have given a significant boost to this domain. Given training data, the goal of the generative models is to generate new samples from the same distribution. If the training data belongs to the probability distribution $p_{data}(x)$, generative models wants to generate samples say with distribution $p_{model}(x)$, and the distribution $p_{model}(x)$ is very similar to $p_{data}(x)$. To learn the complex distribution of data, these models use a deep multilayered neural network structure.

A generative model deals with the distribution of data and produces a likelihood of some action based on the data. What we mean by this is that they deal with the causal relations between different data-points. Causal relations are important because they generalize more. So, if we understand the generative process of how people behave during a pandemic in Asia, we can use that in the United States or in Europe. On the contrary, discriminative models are more catered towards learning boundaries of classes. Another great advantage of using generative models is that they do not rely on labeled data but impose strong assumptions on the data.

The class of generative models can be classified based on the density estimations, explicit density estimation, and implicit density estimation. The former is based on explicitly define $p_{model}(x)$ and solve it. The later can sample from $p_{model}(x)$ without explicitly defining it. PixelRNN/CNN and Variational Auto Encoders are examples of explicit density estimation, Generative Adversarial Network and Markov Chain are examples of implicit density estimation. In this paper, we wish to

decompose generative networks and study two approaches, one for each type. We chose Variational Auto Encoders and Generative Adversarial Networks. We evaluate both these approaches based on well-known MNIST digits data set.

2 Problem Statement

There are a lot of Machine Learning research applications, where the models don't have enough training data set. Therefore in those fields, the models don't generalize well. This is a significant issue in the field of Medical Imaging. To overcome this problem, we use generative models to generate data that looks like real-world data. The models that are used for this do not directly match the output to the training data but they estimate probability distribution which approximately maps to the distribution of true input data. This results in generating new data samples that are different than the ones that the model is trained on but belongs to the same probability distribution.

We look forward to understanding how the Generative Models, especially GAN and VAE succeeds in approximately estimating the true data distribution. How $p_{model}(x)$ becomes similar to $p_{data}(x)$?

3 Preliminaries

Before we start examining the algorithms for generative models, we need to have a sound understanding of the concepts of divergence. Below we discuss two metrics to quantify the similarities of two probability distributions.

3.1 Kullback–Leibler Divergence

The origin of KL divergence is Information Theory. This is defined on top of Entropy, which is a way to quantify information in data. The definition of entropy in any given probability distribution is the following:

$$H = \sum_{i=1}^{n} p(x_i) \cdot \log p(x_i)$$

The idea of KL divergence is to consider two probability distributions and to evaluate the difference between those. We consider the difference in logarithmic values between the distributions.

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i) \cdot (\log p(x_i) - \log q(x_i))$$

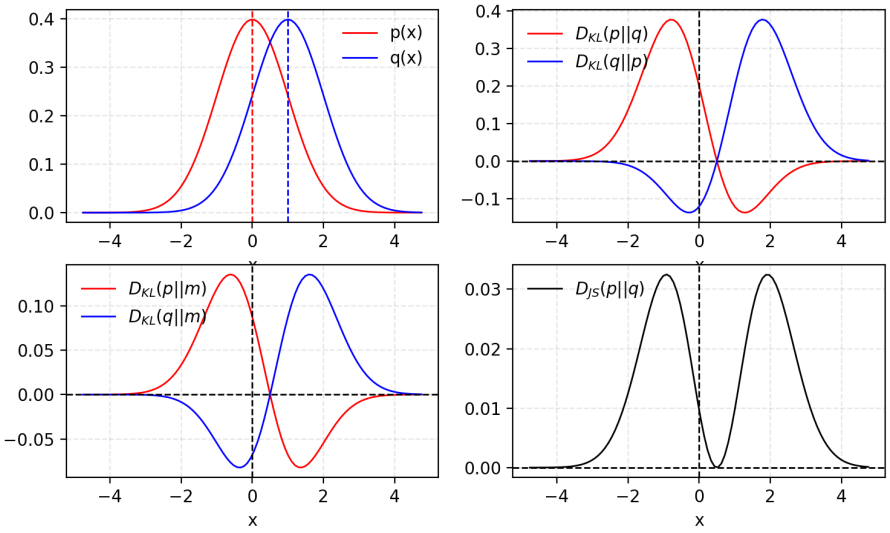

Figure 1: Given two Normal distribution, p with $\mu = 0$ and $\sigma = 1$ and q with $\mu = 1$ and $\sigma = 1$. The average of two distributions is labelled as $m = (p + q)/2$.

With KL divergence, we can calculate how much the actual probability distribution differs from the approximating one. A common mistake is to relate it to the distance between two distributions, however, that's not the case. Unlike distance metric, this metric is not symmetrical. $D_{KL}(p||q) \neq D_{KL}(q||p)$ Below is the most common representation of KL divergence.

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i) \cdot \log \frac{p(x_i)}{q(x_i)} \tag{1}$$

3.2 Jensen–Shannon Divergence

JS divergence is another metric for comparing two probability distributions. JS divergence is defined in terms of KL divergence but is symmetrical. This can be thought of as total divergence from the average of two probability distributions.

$$D_{JS}(p||q) = \frac{1}{2} D_{KL}(p||\frac{p+q}{2}) + \frac{1}{2} D_{KL}(q||\frac{p+q}{2}) \tag{2}$$

4 Algorthims

We would first elaborate and explore how GAN implicitly estimates density and then will explore VAE, which estimates density explicitly.

4.1 GANs

Before diving deep into the structure and mathematical intricacies of GAN, it is important to understand what GANs are. GAN stands for Generative Adversarial Network. The goal of the GAN is to generate data samples that are indistinguishable from training data. This is called Generative Modelling. The term adversarial corresponds to conflict or opposition and in our case, there are two adversaries, the Generator *G* and the Discriminator *D*. The generator generates fake samples and the discriminator distinguishes fake samples from real samples. Both the generator and discriminator are implemented as neural networks as we know Neural Networks can model every possible function.

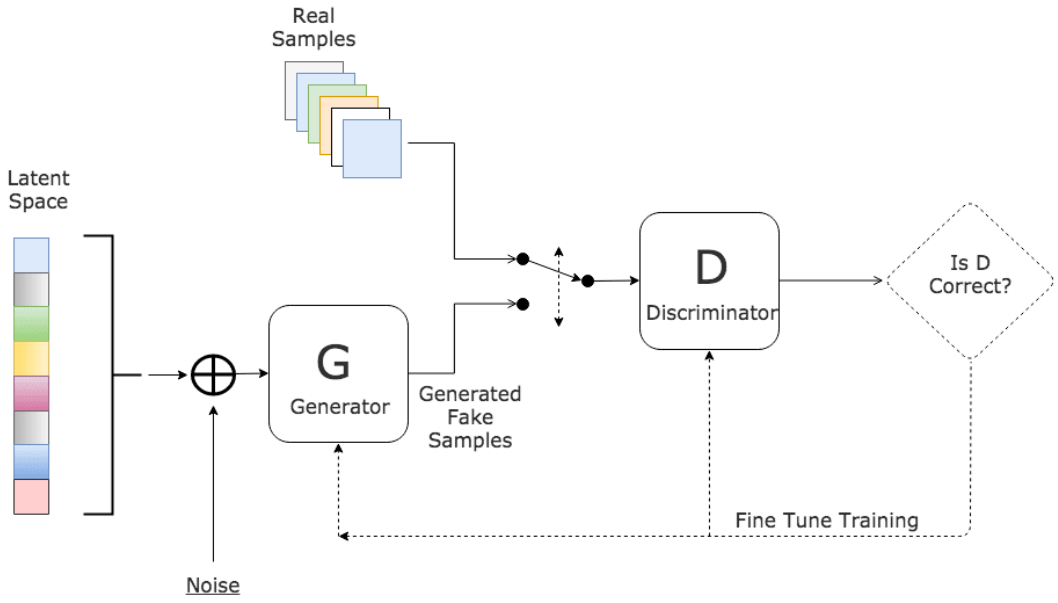

Figure 2: GAN Architecture [1]

The loss of the discriminator is used to train both the Generator and Discriminator networks via backpropagation. The training procedure for *D* is to maximize the probability of the Discriminator predicting true and fake samples correctly. The training procedure for G is to maximize the probability

of D of making a mistake. Goodfellow in his paper [2] has labeled this competitive framework as a minimax game between two players, where D tries to minimize the error in classification and the G tries to maximize it. In the following, we would see the network architecture, the value function for the minimax game, and the optimality condition of the function.

4.1.1 The Cost Function

To make *G* learn the probability distribution of x, a random noise z is introduced from the latent space, and $G(z, \theta_g)$ represent the mapping to data space. G is a differentiable function, implemented as a neural network and this is trained on θ_g. The second neural network $D(x, \theta_d)$ classifies x.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))] \tag{3}$$

The first term in the equation corresponds to Discriminator's prediction on the real data and the second term corresponds to the prediction on the data generated by Generator. The logarithm is used to scale the numbers. $D(x)$ represents the average number of samples classified as real on samples taken from original data. Therefore, the goal of the discriminator is to maximize this term, so that it can classify real samples with more confidence. $D(G(z))$ represents the average of samples classified as real on samples generated from random noise z by Generator. The goal of the Discriminator is to minimize this term as it doesn't want to classify fake samples as real. We take $(1 - D(G(z)))$ as minimizing $D(G(z))$ is the same as maximizing $(1 - D(G(z)))$. While training the parameters for the discriminator θ_g, we do gradient ascent over the cost function as we want to maximize the function. The goal of the generator is to maximize $D(G(z))$, so that the discriminator classifies more samples from $G(z)$ as fake. This is equivalent to minimizing $1 - D(G(z))$, therefore to minimize this we perform gradient descent on the same cost function for training the Generator, θ_g.

Algorithm 1: k decides how many steps we make the Discriminator perform, and thus is a hyper parameter.

for *every training iteration do* **do**
  while $k > 0$ **do**
    Sample n noise samples from noise prior $p_g(z)$, [$z^{(1)}$,...,$z^{(n)}$]
    Sample n random samples from training data distribution $p_{data}(x)$, [$x^{(1)}$,...,$x^{(n)}$]
    Update the discriminator by performing gradient ascent:

$$\nabla\theta_d \frac{1}{n} \sum_{i=1}^{n} [logD(x^i) + log(1 - D(G(z^i)))]$$

    $k \leftarrow k - 1$
  end
  Sample n noise samples from noise prior $p_g(z)$, [$z^{(1)}$,...,$z^{(n)}$]
  Update the generator by performing gradient descent:

$$\nabla\theta_g \frac{1}{n} \sum_{i=1}^{n} [log(1 - D(G(z^i)))]$$

end

Choice of distribution of latent variable The noise z is drawn from prior probability distribution p_z. A common and sensible choice is to draw z from a zero-centered unit-variance Gaussian distribution. The unit variance allows each element in random vector to be uncorrelated to other elements and these end up picking different features of the generated samples. And with zero centered density distribution, choosing any point will make sure the generated image will be valid.

4.1.2 Theoretical Results

Optimal Value for D We consider the optimal value for D by fixing Generator G. Expanding the expectation values in terms of their integral form and by changing variables we get,

$$V(G, D) = \int_x \Big(p_{data}(x) \log(D(x)) + p_g(x) \log(1 - D(x)) \Big) dx$$

Replacing $p_{data}(x) = a$, $p_g(x) = b$ and $D(x) = y$, setting the derivative equal to zero we get,

$$\frac{\delta}{\delta y}(a\log(y) + b\log(1-y)) = 0$$

Solution of the equation is $y = \frac{a}{a+b}$. Now replacing back the values of a, b and y we get,

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} \tag{4}$$

When the training is optimal, $p_{data}(x)$ gets very close to $p_g(x)$, $D^*(x) = \frac{1}{2}$. This signifies that the Discriminator at this optimal point has to guess the result, which is intuitive.

The Global Optimal The cost function can be recomputed by applying the optimal value of D,

$$C(G) = \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D^*(x)] + \mathbb{E}_{x\sim p_g(z)}[\log(1 - D^*(x))]$$

$$C(G) = \mathbb{E}_{x\sim p_{data}(x)}[log\frac{1}{2}] + \mathbb{E}_{x\sim p_g(z)}[log\frac{1}{2}] = -\log 4 \tag{5}$$

The Significance of the Loss Function Writing the loss function in terms of the expression for D^*,

$$C(G) = \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}] + \mathbb{E}_{x\sim p_g(z)}[\log \frac{p_g(x)}{p_{data}(x)+p_g(x)}]$$

The equation can be rewritten by dividing the denominators inside log and taking log 2 outside.

$$C(G) = \mathbb{E}_{x\sim p_{data}(x)}[\log \frac{p_{data}(x)}{\frac{p_{data}(x)+p_g(x)}{2}}] + \mathbb{E}_{x\sim p_g(z)}[\log \frac{p_g(x)}{\frac{p_{data}(x)+p_g(x)}{2}}] - 2\log 2$$

The terms inside the Expectation are the measurements of KL divergence for the true data distribution or the generated data distribution to the average distribution.

$$C(G) = D_{KL}(p_{data}(x)||\frac{p_{data}(x)+p_g(x)}{2}) + D_{KL}(p_g(x)||\frac{p_{data}(x)+p_g(x)}{2}) - 2\log 2$$

This can be represented using equation (2) in JS divergence,

$$C(G) = 2D_{JS}(p_{data}(x)||p_g(x)) - \log 4 \tag{6}$$

To summarize, when the discriminator is at optimal setting D^*, the loss function represents the similarity between the generative data distribution $p_{data}(x)$ and the real sample distribution $p_g(x)$ by JS divergence. The optimal generator G^* that replicates the real data distribution, sets the JS divergence term to 0. Therefore, with optimal Generator and Optimal Discriminator, this leads to the minimum $V(D^*, G^*) = -\log 4$ which is aligned with equation (5) above.

4.2 DCGANs

Since, the invention of GAN, a lot of implementations has been done for the neural nets mostly depending on the various application. DCGAN is one of the popular and successful network design for GAN. The core of the DCGAN architecture uses a standard CNN architecture on the discriminative model. For the generator, convolutions are replaced with upconvolutions, so the representation at each layer of the generator is actually successively larger, as it makes from a low-dimensional latent vector onto a high-dimensional image.

In our experiment for generating samples identical to MNIST data-set, we implement the DCGAN architecture suggested in the paper. [4].

Replacing Pooling Layers The output of the convolutional layers, that are the feature maps are sensitive to the location of the features in the input. To overcome this sensitivity, we downsample the feature maps. Pooling layers are used for this downsampling. In DCGAN however, we want the generator and discriminator to learn it's own spatial upsampling and downsampling. Hence, the pooling layers are replaced by fractional strided convolutions and strided convolutions.

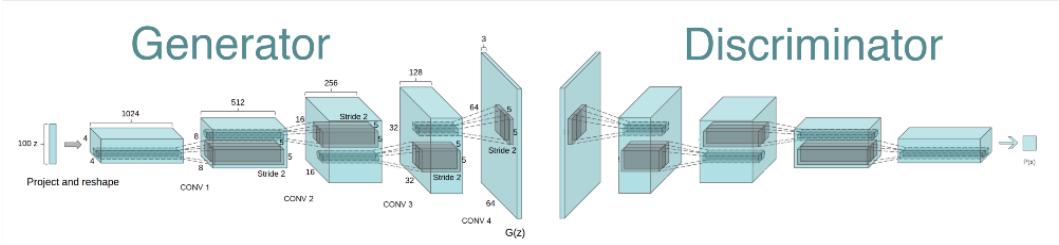

Figure 3: DCGAN Architecture [3]

Removing fully connected layers In GAN, the output from the global average pooling layer is shaped into a 1D vector and then fed to a fully connected layer. Global average pooling increased model stability but hurts the convergence speed. In DCGAN we do the following instead. For the discriminator, we flatten the last convolution layer and then feed it into a single sigmoid output. For the generator, the first layer that uses random noise as the input is generally treated as a fully connected layer, here the result is reshaped into a 4 dimension tensor and is fed to the first convolution layer, ready for upscaling.

Batch Normalization GAN is subject to falling into mode collapse, the generator may collapse to a space such that it always generates same outputs. The generator might succeed in tricking the discriminator, but it fails to learn and understand the complex nature of the data distribution. To avoid this, DCGAN uses batch normalization. The input was normalized to have zero mean and unit variance. This allowed for deeper models to work without falling into mode collapse. Batch normalization also helps in issues that were faced because of poor initialization of parameters.

Activation Function The generator uses the RELU activation function in all layers except the output layer. The output layer uses the tanh activation function. RELU is used because it helps resolve the vanishing gradient problem. It also promotes sparse activation. The discriminator uses LeakyReLu as its activation function. Since leakyRelu has a small negative slope, this allows a small gradient signal for negative values. Hence, while updating weights during backpropagation, it passes a small negative gradient. This makes the gradient flow from the discriminator to the generator much stronger and helps in better weight updates.

4.3 Autoencoders

The basic architecture of autoencoders is a combination of an encoder system and a decoder system, both of which are deep neural networks with a latent space in between them. The entire structure works in a way such that we feed the input data using the encoders, there is some dimension reduction as it passes through the layers and updates values in the latent space. Now, we pass the data in the latent space and pass it through our decoder and get some meaningful output. This iterative encoding and decoding updates our latent space with the help of backpropagation. From this, it sounds that the process is similar to the dimension reduction techniques. In fact, had we taken a single layer network instead of a deep neural network, this might very well be compared to that of PCA. The robustness of the system depends on the number of layers we choose for our encoder and decoder system, which should be the identical reverse of each other. It also depends on the dimensions of the latent space such that we are able to correctly classify the different inputs to separate spaces without loss of data.

But this acts as a bad generative system as the process of mapping the inputs to a latent space overfits the data. This is why we need Variational Autoencoders [5] - VAEs regularize the training such that there is no overfitting and the latent space has properties of a good generative process. In VAE, instead of encoding an input as a single point, we encode it as a distribution over the latent space.

Latent variables are hidden variables that are part of our program but which we don't directly observe. We denote this by **z**. In autoencoders, the **z space** acts as a prior distribution, which we denote by $p_\theta(z)$.

From Fig 4, we see that the job of the encoder is to encode the data to the latent variable Z with lesser dimensions than the data x. The encoder outputs parameters to a Gaussian probability density

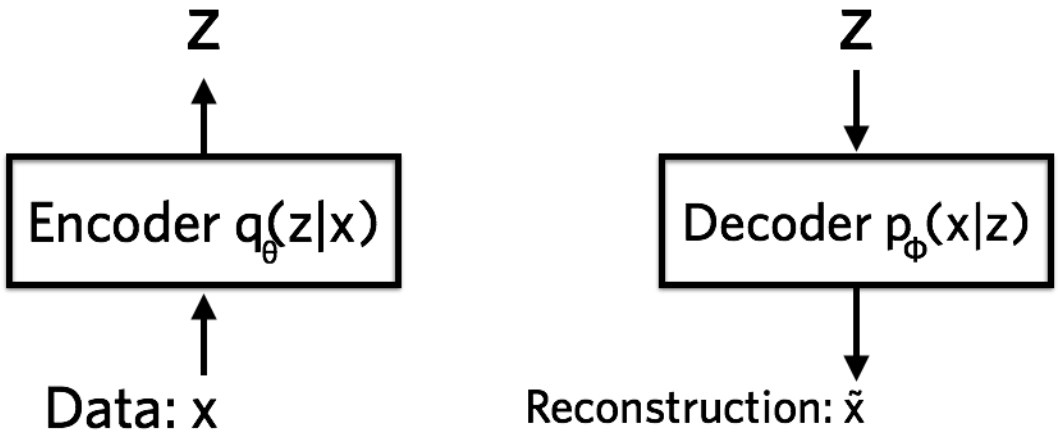

Figure 4: Encoder Decoder Architecture

function $q_\theta(z|x)$. The decoder is denoted by $p_\phi(x|z)$. Its input is z and it produces the probability distribution of the data and has weights denoted by ϕ.

4.3.1 Loss Function

The Loss function for VAE is given by:

$$l_i(\theta, \phi) = -\mathbb{E}_{z\sim q_\theta(z|x_i)}[logp_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z)) \tag{7}$$

The loss function is taken as the negative log-likelihood with the regularizer term. The total loss is thus $\sum_{i=1}^{N} l_i$, where N is the total number of input data.

The first term is the reconstruction loss and it is taken from the expectation of the encoder. This is then used by the decoder to reconstruct the original data. So, if the decoder is unable to do this well, the loss incurred is large.

The regularization term is the Kullback-Leibler divergence as explained above. Here we compare $q_\theta(z|x)$ and $p(z)$ to see how close these two distributions are.

By definition, we consider p as a Normal distribution with $p(z) = Normal(0, 1)$. Now we see a clash between the two terms of the equation where the reconstruction term penalizes if the latent space, z is different from p and the regularizer tries to keep the distribution diverse.

Finally, we train VAE using gradient descent to get the optimum values of our weights and bias, θ and ϕ. So, for step size ρ, the formula is as follows -

$$\theta \leftarrow \theta - \rho\frac{\partial l}{\partial \theta} \tag{8}$$

4.4 VAE as a probability model

Generating Image: Our model is defined on two parameters, the data x and the latent variables z. To generate images using latent variables, we define joint probability over x and z given as : $p(x, z) = p(x|z)p(z)$ where $p(x|z)$ is the likelihood. For the MNIST dataset, the images are black and white. Each pixel can take a value of 0 or 1. Hence, we can represent the likelihood using Bernoulli distribution.

Inference for the model: Given an input image x, we want to convert this into latent variables z that have much smaller dimension. We want to come up with good values got the latent variable given observed data x. This can be calculated as:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{9}$$

where $p(x) = \int p(x|z)p(z)dz$

We need to calculate $p(x)$ over all the configurations of z and hence it takes exponential time. Therefore, it is not practical to calculate $p(z|x)$ directly and we try to approximate it. We use a family of distributions to approximate the posterior probability. This is represented as $q_\theta(z|x)$, where θ represents the parameters. If q was Gaussian, then we will have θ for all data points given by $\theta_{xi} = (\mu_{xi}, \sigma^2_{xi})$ Since we our doing approximation, there will be some information loss. We use KL divergence to measure this loss in information.

$$KL(q_\theta(z|x)||p(z|x)) = E_q[logq_\theta(z|x)] - E_q[logp(x,z)] + logp(x) \tag{10}$$

where we find θ such that we minimize this equation.

Hence the ideal approximation is given by: $q^*_\theta(z|x) = argmin_\theta KL(q_\theta(z|x)||p(z|x))$

The above equation also uses $p(x)$ and takes exponential time to calculate, which we don't want! We therefore turn to Evidence Lower Bound(ELBO) to approximate the posterior probability. This is defined as:

$$ELBO(\theta) = E_q[logp(x,z)] - E_q[logq_\theta(z|x)] \tag{11}$$

Using (10) and (11), we we can write $p(x)$ as:

$$logp(x) = ELBO(\theta) + KL(q_\theta(z|x)||p(z|x)) \tag{12}$$

We have $KLdivergence >= 0$ using Jensen's inequality as mentioned above. Hence minimizing this is equivalent to maximizing Evidence Lower Bound. Further, in VAE, we do not have any global latent variables and ELBO can be written as a sum, where each term represents a datapoint. To find $ELBO(\theta) = \sum_i ELBO_i(\theta)$ where $ELBO_i(\theta) = Eq_\theta(z|x_i)[logp(x_i|z)] - KL(q_\theta(z|x_i)||p(x))$

We can now apply stochastic gradient descent on this. Hence, to summarize everything, we replace θ by the parameters we will use in our neural network, which are θ and ϕ.

$$ELBO_i(\theta, \phi) = Eq_\theta(z|x_i)[logp_\phi(x_i|z)] - KL(q_\theta(z|x_i)||p(x)) \tag{13}$$

where θ is the parameter used by the encoder and ϕ by the decoder. Now, this equation is nothing but the negative loss function that we mentioned above in (7).

5 Experiments

We evaluate performance of both GAN and VAE on the MNIST handwritten digits dataset. [6]. Firstly, we implement GAN using TensorFlow on Python, we use deep convolutional neural networks to model our generator and discriminator. For our discriminator, the training data is an image of size 28×28. The model parameters are described in next section. Next, we implement VAE using Keras on Python. The same set of images is used in training this model. VAE experiment is described in section 5.2.

5.1 DCGAN

We implement the DCGAN with the below changes for better stability, output and convergence.

1. GAN is very sensitive to hyperparameters, for good results the below parameters are used.

a. Optimizer : adam optimizer

b. Learning rate : 0.0002

c. Momentum term (beta1) : 0.5

d. Batch size : 100

e. Number of Epochs : 20

2. Before training the network, we need to initialize the weights. Instead of initializing the weights randomly, we use a gaussian distribution to initialize them. The distribution has a mean of 0 and a standard deviation of 0.02

3. The images used for training (input to the discriminator) are scaled so that the pixel values are in the range [-1,1]. This is done so that the fake images from the generator and the real images fed into the discriminator both have the same range of pixel values.

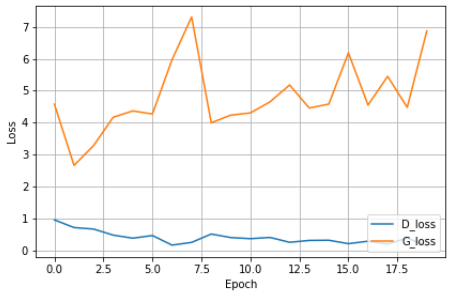

Figure 5: Discriminator and Generator Losses across training Epochs

Now we represent the data generated by our Generator network, we take only a few epochs to present how the Generator learns distribution of original data.

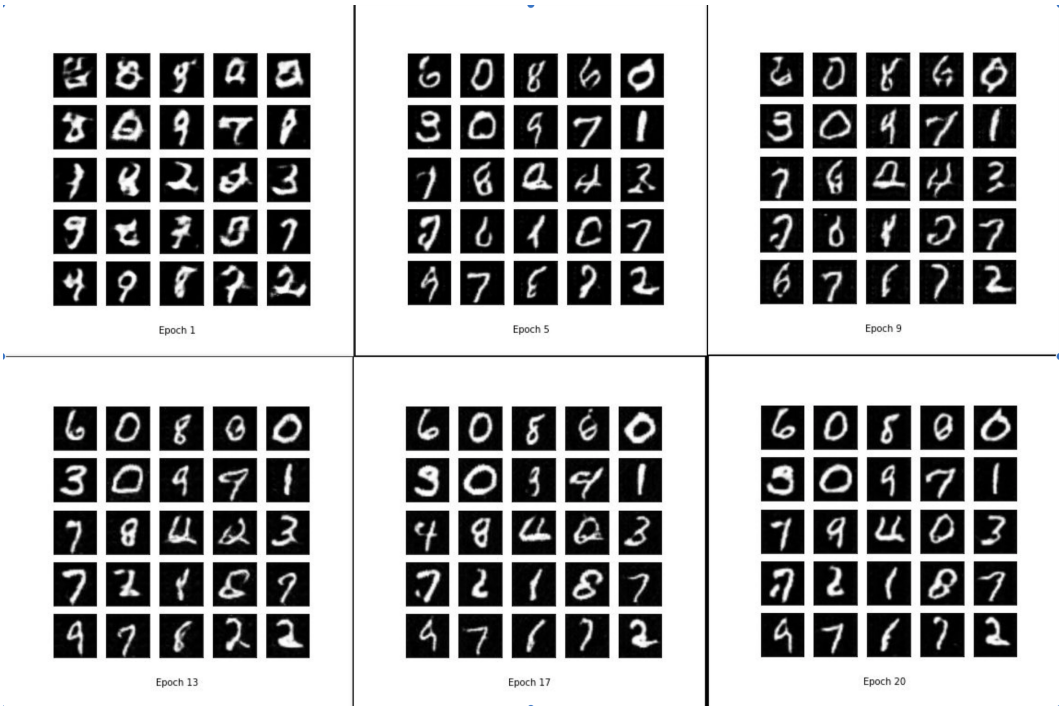

Figure 6: Improvement of Generated Samples across Epochs

We train for 20 epochs, however, we don't see much difference in the generated images after 10 epochs. The lack of evaluation metric is a problem. With the lack of evaluation metric, it's unclear as for how many epochs we should train the model.

5.2 Variational Autoencoder

We run the entire set of neural networks with the following hyperparameters.

a. Optimizer : rmsprop

b. Learning rate : 0.001

c. rho : 0.9

d. Batchsize : 65000

e. Number of epochs : 7

From our experiment, we can see that the loss is decreasing with every epoch until it stabilizes to an optimum loss. We can use saturation of loss as an evaluating parameter for stopping our training instead of running for fixed number of Epochs.

```
Train on 65000 samples, validate on 5000 samples
Epoch 1/7
65000/65000 [==============================] - 400s 6ms/step - loss: 0.2155 - val_loss: 0.1951
Epoch 2/7
65000/65000 [==============================] - 403s 6ms/step - loss: 0.1924 - val_loss: 0.1907
Epoch 3/7
65000/65000 [==============================] - 406s 6ms/step - loss: 0.1884 - val_loss: 0.1888
Epoch 4/7
65000/65000 [==============================] - 407s 6ms/step - loss: 0.1860 - val_loss: 0.1879
Epoch 5/7
65000/65000 [==============================] - 407s 6ms/step - loss: 0.1844 - val_loss: 0.1852
Epoch 6/7
65000/65000 [==============================] - 407s 6ms/step - loss: 0.1832 - val_loss: 0.1840
Epoch 7/7
65000/65000 [==============================] - 400s 6ms/step - loss: 0.1823 - val_loss: 0.1827
<keras.callbacks.callbacks.History at 0x7f5c2856e4a8>
```

Figure 7: Loss of VAE over 7 epochs

VAE tries to find similarities between the patterns of the images of the digits and clusters them close to each other, as a result we get a mapping shown below.

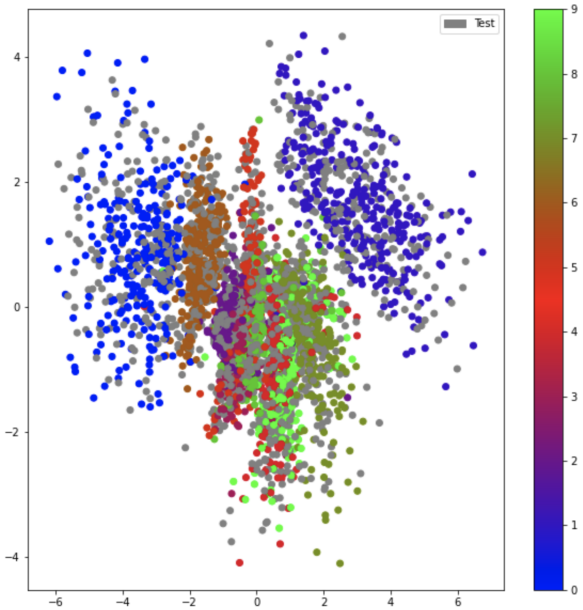

Figure 8: Cluster of MNIST digits using VAE

Now, that we see the cluster in terms of datapoints, we map this to the images of the digits produced. These are **new** digits produced by our program.

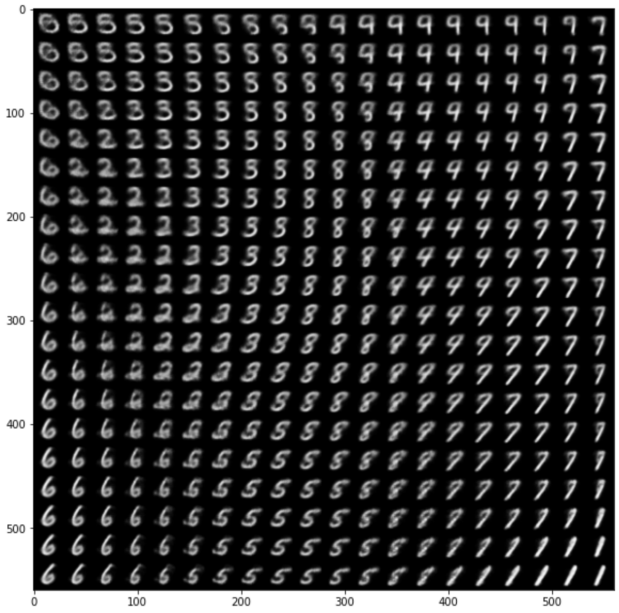

Figure 9: New digits produced by VAE

5.3 Discussion

Comparing VAE and GAN, in their practicality VAE produced blurry images compared to the images generated by GAN. We also notice that some generated digits by VAE models, are hard to distinguish. For example, images generated for 9 , 4 and 7 , 1 are similar in their appearance. GAN is computationally heavy and requires more epochs to train. Despite this, due to its ability to produce crisp images, it is popular where the requirement is to generate higher resolution images. The advantage of VAE over GAN is, it maps input sample to latent space and then generates images from it. Recent developments in the field of Generative Models have proposed to assemble both GAN and VAE to bring benefits of both. The idea is to replace the Generator in GAN by a VAE and Discriminator is used on the images generated by this. The loss is backpropagated to train the new Generator in a similar fashion. Larsen [7] has suggested this implementation, and we consider this as our future work for this project.

6 Conclusions

In this report, we have proposed two models that are used for image generation. With continuous improvements, scaled up data sets and training procedures, we can expect the generative models to succeed in creating data samples which will be indistinguishable from the training samples. With higher computation powers available, recent developments in both VAE and GAN can generate higher resolution images. Through generative models, we will be able to address situations where there is a lack of training data for creating models. Although, the concepts that we discussed are the basic building blocks, both the GAN and VAE are continuously improving. Due to the limitations on the resources available to us to train our model, we were not able to focus on more complicated applications. In our project report, we provided a platform for every Machine Learning enthusiast to understand generative modeling and two latest most techniques.

References

[1] N. Sharma, *heartbeat.fritz.ai*.

[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[3] *sigmoidal.io*. Roman Trusov.

[4] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015.

[5] D. P. Kingma, M. Welling, *et al.*, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[6] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[7] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," 2015.