VAEs vs GANs : Comparison of Generative Models

Sai Manoj Jalam [1]

Abstract

Recent developments in generative models within the realm of machine learning have led to the emergence of Generative Adversarial Networks (GANs). GANs have found applications in various areas, including image denoising, generating medical images for research, and network security. In our project, we provide an overview of generative models before delving into two fascinating models: Variational Autoencoders and Generative Adversarial Networks. Both models can generate realistic, synthetic data, and we examine the distinct approaches they employ for generating new handwritten digits.

1. Introduction

Machine learning problems can be broadly categorized into supervised and unsupervised learning. Supervised learning involves creating models that map input data to output labels, while the challenge lies in estimating the mapping function to efficiently determine the correct label for unseen input data. On the other hand, unsupervised learning is more difficult, as it seeks to model the hidden structure within input data. Examples of unsupervised learning problems include clustering, Gaussian Mixture models, and dimensionality reduction. Unsupervised learning remains a relatively uncharted area in machine learning research, with many open problems to tackle. If we can successfully learn and represent data's underlying structure, we may be able to comprehend the visual world's structure.

Our focus is on generative models, an underexplored field within unsupervised learning that has seen significant advancements recently. Given training data, generative models aim to generate new samples from the same distribution. These models utilize deep multilayered neural networks to learn the complex distribution of data. Generative models handle data distribution and generate likelihoods for actions based on the data, focusing on causal relationships between data points. Understanding these causal relationships allows for better generalization, and generative models do not rely on labeled data but instead impose strong assumptions on it. Generative models can be classified based on density estimations, including explicit and implicit density estimation. Explicit density estimation involves explicitly defining $p_{model}(x)$ and solving it, while implicit density estimation can sample from $p_{data}(x)$ without an explicit definition. Examples of explicit density estimation include PixelRNN/CNN and Variational Autoencoders, while Generative Adversarial Networks and Markov Chains exemplify implicit density estimation. In this paper, we examine two generative network approaches, one from each type: Variational Autoencoders and Generative Adversarial Networks. We evaluate these approaches using the well-known MNIST digits dataset.

2. Problem Statement

In many machine learning research applications, there is a scarcity of sufficient training data, which results in models that do not generalize well. This issue is particularly prominent in the field of medical imaging. To address this problem, generative models are employed to create data resembling real-world data. Instead of directly matching the output to the training data, these models estimate a probability distribution that approximates the distribution of the actual input data, generating new data samples that, while distinct from the training data, belong to the same probability distribution.

Our goal is to comprehend how generative models, specifically GANs and VAEs, successfully approximate the true data distribution, and how $p_{model}(x)$ becomes akin to $p_{data}(x)$

3. Algorithms

First, we will delve into how GANs implicitly estimate density, followed by an examination of VAEs, which explicitly estimate density.

3.1. GAN

Before delving into the details and mathematical complexities of GANs, it is crucial to comprehend what they are. GAN stands for Generative Adversarial Network, and its objective is to produce data samples that are indistinguishable from the training data, a process known as generative modeling. The term ”adversarial” refers to the opposition between two adversaries, in this case, the Generator G and the Discriminator D. The generator creates fake samples, while the discriminator distinguishes between fake and real samples. Both the generator and discriminator are implemented as neural networks, as they are capable of modeling any possible function.

The discriminator’s loss is utilized to train both the Generator and Discriminator networks through backpropagation. The training process for the Discriminator D aims to maximize the likelihood of accurately predicting real and fake samples, while the Generator’s G training process seeks to maximize the probability of D making errors. Goodfellow referred to this adversarial framework as a minimax game in his paper, in which D attempts to minimize classification errors while G strives to increase them. Subsequently, we will examine the network architecture, the value function for the minimax game, and the function’s optimality conditions.

3.1.1. The Cost Function

In order to enable G to learn the probability distribution of x, random noise z is introduced from the latent space, with $G(z, \theta_g)$ representing the mapping to the data space. G is a differentiable function, implemented as a neural network, and is trained on θ_g. The second neural network, $G(x, \theta_g)$, is responsible for classifying x.

$\min_G max_D(D, G) = E_{x\sim pdata(x)}[logD(x)] + E_{z\sim pz(z)}[log(1 - D(G(z)))]$

The equation’s first term corresponds to the Discriminator’s prediction on real data, while the second term relates to its prediction on data generated by the Generator. Logarithms are used to scale the numbers. $D(x)$ represents the average number of samples classified as real from the original data. The Discriminator’s objective is to maximize this term to confidently classify real samples. $D(G(z))$ denotes the average number of samples classified as real from those generated by the Generator using random noise z. The Discriminator aims to minimize this term to avoid misclassifying fake samples as real. Minimizing $D(G(z))$ is equivalent to maximizing $(1 - D(G(z)))$, so we take $(1 - D(G(z)))$.

When training the Discriminator’s parameters $\theta_g)$, gradient ascent is performed on the cost function since the goal is to maximize it. On the other hand, the Generator’s objective is to maximize $D(G(z))$ so that the Discriminator classifies more samples from $G(z)$ as fake. This is equivalent to minimizing $1 - D(G(z))$. To achieve this, we perform gradient descent on the same cost function while training the Generator.

Choice of distribution of latent variable The noise z is sampled from a prior probability distribution p_z. A typical and reasonable choice is to draw z from a Gaussian distribution with zero mean and unit variance. The unit variance ensures that each element in the random vector is uncorrelated with other elements, allowing them to capture different features of the generated samples. Additionally, with a zero-centered density distribution, selecting any point ensures that the generated image will be valid.

3.1.2. Theoretical Results

Optimal Value for D We evaluate the optimal value for the Discriminator D by holding the Generator G fixed. By expressing the expectation values in their integral form and applying variable transformations, we obtain the following:

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

Replacing $p_{data}(x) = a, p_g(x) = b$ and $D(x) = y$, setting the derivative equal to zero we get, 4

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

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

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

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: By recalculating the value of D, we can try to obtain the global optimal.

$C(G) = max_D V(D, G) = E_{x\ pdata(x)}[log\, D(x)] + E_{x\ pg(z)}[log\,(1 - D(x))]$

$C(G) = E_{x\ pdata(x)}[log\frac{1}{2}] + E_{x\ p_g(z)}[log\frac{1}{2}] = -log\,4$

The importance of the loss function: Writing the loss function in terms of the expression for D^*

$C(G) = \max DV(D, G) = E_{x\ p_{data}(x)}[\log \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}] + Ex\ p_g(z)[\log \frac{p_g(x)}{p_{data}(x)+p_g(x)}]$

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

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

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.

This can be represented using equation (2) in JS divergence, $C(G) = 2D_{JS}(p_{data}(x)\|p_g(x)) - \log 4$

In summary, when the discriminator is at its optimal configuration $D*$, the loss function reflects the similarity between the generated data distribution $p_{data}(x)$ and the real sample distribution $p_g(x)$ using the JS divergence. The optimal generator $G*$ that replicates the real data distribution results in a JS divergence of 0. Consequently, with an optimal generator and optimal discriminator, the minimum value for $V(D^*; G^*)$ i= -log(4), which is consistent with equation mentioned earlier.

3.2. GANs

Since the inception of GANs, numerous implementations have been developed for neural networks, primarily based on various applications. GAN, which stands for Deep Convolutional Generative Adversarial Network, is a popular and successful GAN architecture. At its core, the GAN utilizes a standard CNN architecture for the discriminative model. For the generator, convolutions are replaced with upconvolutions (also known as transposed convolutions), successively enlarging the representation at each layer as it maps from a low-dimensional latent vector to a high-dimensional image.

In our experiment to generate samples resembling the MNIST dataset, we implement the GAN architecture as suggested in the referenced paper [4].

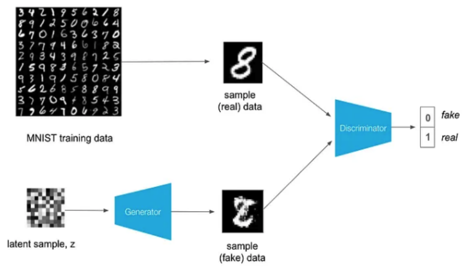

Figure 1. GAN Architecture

Replacing Pooling Layers The feature maps produced by convolutional layers are sensitive to the location of features in the input. To address this sensitivity, feature maps are downsampled using pooling layers. However, in GAN, the goal is for the generator and discriminator to learn their own spatial upsampling and downsampling. As a result, pooling layers are replaced with fractional-strided convolutions (for upsampling) and strided convolutions (for downsampling).

Removing fully connected layers In traditional GANs, the output from the global average pooling layer is reshaped into a 1D vector and then passed to a fully connected layer. Although global average pooling improves model stability, it adversely affects convergence speed. In GAN, a different approach is taken. For the discriminator, the last convolution layer is flattened and then fed into a single sigmoid output. For the generator, instead of using a fully connected layer for the first layer that takes random noise as input, the result is reshaped into a 4-dimensional tensor and then passed to the first convolution layer, where it is ready for upscaling.

Batch Normalization GANs can be prone to mode collapse, where the generator collapses into a space that consistently generates identical outputs. Although the generator may deceive the discriminator, it fails to learn and comprehend the complexity of the data distribution. To mitigate this issue, GAN employs batch normalization. By normalizing the input to have a zero mean and unit variance, deeper models can function without succumbing to mode collapse. Additionally, batch normalization helps address problems arising from poor parameter initialization.

Activation Function In the generator, the ReLU activation function is used in all layers except the output layer, which employs the tanh activation function. ReLU is chosen because it helps address the vanishing gradient problem and encourages sparse activation. On the other hand, the discriminator uses the LeakyReLU activation function. Since LeakyReLU has a small negative slope, it allows a slight gradient signal for negative values. Consequently, during backpropagation, it passes a small negative gradient. This strengthens the gradient flow from the discriminator to the generator, leading to more effective weight updates.

3.3. VAE

Autoencoders consist of an encoder and a decoder, both of which are deep neural networks, with a latent space between them. The architecture operates by feeding input data through the encoder, reducing dimensions as it progresses through the layers and updating values in the latent space. The data in the latent space is then passed through the decoder to produce meaningful output. This iterative encoding and decoding process updates the latent space using backpropagation.

Although this process may seem similar to dimensionality reduction techniques, such as PCA, the robustness of the system depends on the number of layers chosen for the encoder and decoder, which should be mirror images of each other. It also depends on the dimensions of the latent space to allow for accurate classification of various inputs without data loss. However, autoencoders can overfit the data and do not function well as generative systems.

To address this issue, Variational Autoencoders (VAEs) [5] are introduced. VAEs regularize the training to prevent overfitting and ensure that the latent space has the properties of a good generative process. In VAEs, an input is encoded as a distribution over the latent space rather than a single point. Latent variables are hidden variables within the model that are not directly observed, denoted by z. In autoencoders, the **Z space** acts as a prior distribution, represented by $p_\theta(z)$.

As shown in Fig 1, the encoder's role is to encode the data into the latent variable Z with fewer dimensions than the data x. The encoder outputs parameters for a Gaussian probability density

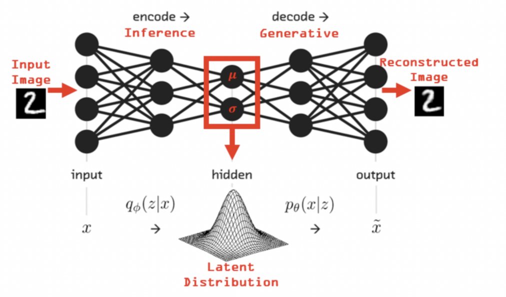

Figure 2. Encoder Decoder Architecture

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

3.3.1. LOSS FUNCTION

The Loss function for VAE is given by:

$l_i(\theta, \phi) = -E_{z\ q_\theta(z|x_i)}[log\, p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$

The loss function for VAEs consists of the negative log-likelihood combined with a regularization term, resulting in a total loss of $\sum_{i=1}^{N} l_i$, where N represents the total number of input data points.

The first term, the reconstruction loss, is derived from the encoder's expectation and used by the decoder to reconstruct the original data. If the decoder struggles to reconstruct the data accurately, the loss incurred will be significant.

The regularization term is the Kullback-Leibler divergence, which compares the distributions $q_\theta(z \mid x)$ and $p(z)$ to evaluate their similarity. By definition, $p(z)$ is considered a normal distribution with $p(z) = Normal(0, 1)$. This creates a tension between the two terms of the equation, as the reconstruction term penalizes if the latent space, z, differs from ρ, while the regularizer aims to maintain a diverse distribution.

Finally, VAEs are trained using gradient descent to optimize the weights and biases, denoted by θ and ϕ. For a given step size, denoted by ρ, the optimization formula is as follows:

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

3.4. 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 : $p(x, z) = p(x \mid z)p(z)$ where $p(x \mid z)$ 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:

$$p(z \mid x) = \frac{p(x \mid z)p(z)}{p(x)}$$

where $p(x) = \int p(x \mid z)p(z)dz$ 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 $p(z \mid x)$ directly and we try to approximate it. We use a family of distributions to approximate the posterior probability. This is represented as $q_\theta(z \mid x)$, where θ represents the parameters. If q was Gaussian, then we will have θ for all data points given by $\theta_{xi} = \left(\mu_{xi}, \sigma_{xi}^2\right)$ Since we our doing approximation, there will be some information loss. We use KL divergence to measure this loss in information.

$$KL\left(q_\theta(z \mid x) \| p(z \mid x)\right) = E_q\left[\log q_\theta(z \mid x)\right] - E_q[\log p(x, z)] + \log p(x)$$

where we find θ such that we minimize this equation. Hence the ideal approximation is given by: $q_\theta^*(z \mid x) = argmin_\theta KL\left(q_\theta(z \mid x) \| p(z \mid x)\right)$ 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:

$$ELBO(\theta) = E_q[\log p(x, z)] - E_q\left[\log q_\theta(z \mid x)\right]$$

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

$$\log p(x) = ELBO(\theta) + KL\left(q_\theta(z \mid x) \| p(z \mid x)\right)$$

We have K Ldivergence $>=$ 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 $ELBO_i(\theta) = Eq_\theta\left(z \mid x_i\right)\left[\log p\left(x_i \mid z\right)\right] - KL\left(q_\theta\left(z \mid x_i\right) \| p(x)\right)$ 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.

$$ELBO_i(\theta, \phi) = Eq_\theta\left(z \mid x_i\right)\left[\log p_\phi\left(x_i \mid z\right)\right] - KL\left(q_\theta\left(z \mid x_i\right) \| p(x)\right)$$

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).

4. Experiments

We assess the performance of both GAN and VAE on the MNIST handwritten digits dataset [6]. Initially, we implement GAN using TensorFlow in Python, employing deep convolutional neural networks for our generator and discriminator models. The training data for the discriminator consists of 28x28 images. The model parameters are detailed in the subsequent section. Following this, we implement VAE using Keras in Python, utilizing the same set of images for training. The VAE experiment is outlined in section 4.2.

4.1. GAN

We implement the GAN 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 : 0.001

c. Batch size : 100

d. Number of Epochs : 200

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.

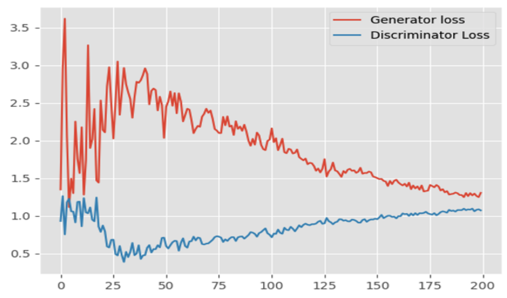

Figure 3. GAN losses across Epochs

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.

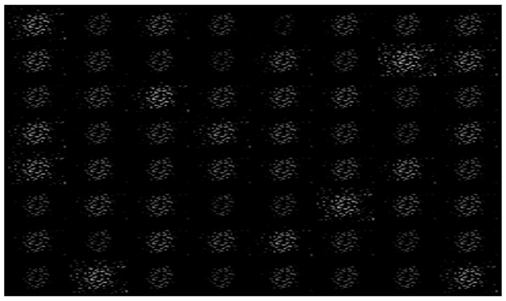

Figure 4. Generated outputs from 1st epoch

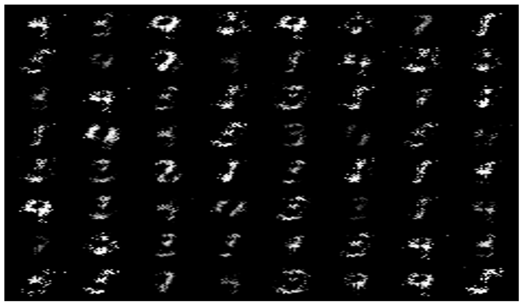

Figure 5. Generated outputs from 55th epoch

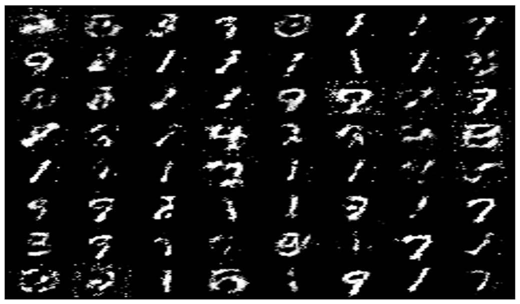

Figure 6. Generated outputs from 100th epoch

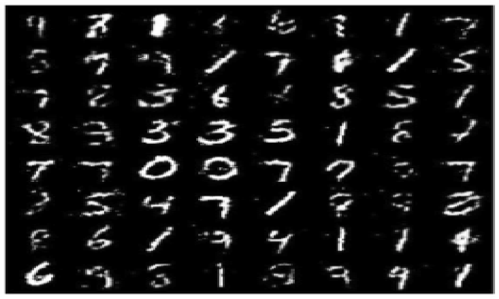

Figure 7. Generated outputs from 200th epoch

We discuss some challenges encountered while training the GAN on the MNIST dataset. One issue is the absence of an evaluation metric. We ran the model for 200 epochs, but there was not much noticeable difference in the generated images after 10 epochs. Due to the lack of an evaluation metric, it's difficult to determine when the training should be halted.

4.2. Variational Autoencoder

We run the entire set of neural networks with the following hyper-parameters.

a. Optimizer : adam

b. Learning rate : 0.001

c. latent dim : 2

d. Batchsize : 100

e. Number of epochs : 50

From our experiment, we can see that the loss is decreasing with every epoch until it stabilizes to an optimum loss.

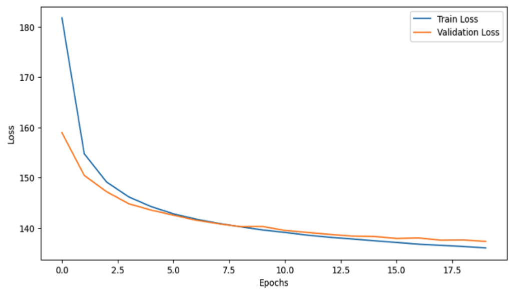

Figure 8. VAE Error

VAE aims to identify common patterns within digit images and group them closely together, resulting in a mapping as depicted below.

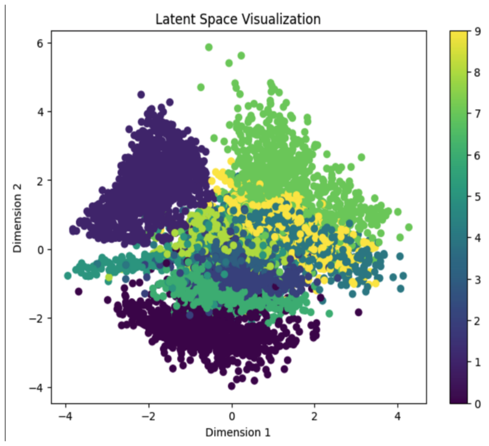

Figure 9. Cluster of MNIST data

By visualizing the outputs for different epochs, we can observe how the quality of the generated samples evolves during the training process. This helped me identify when the model starts to generate meaningful samples and if there is any overfitting or mode collapse.

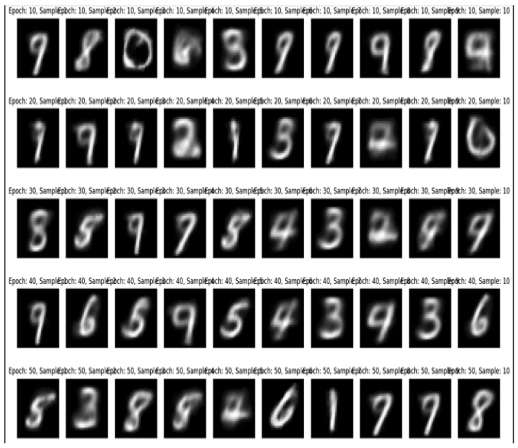

Figure 10. Improvement of generated samples across Epochs

Now, that we see the cluster we can use this to produce the images of the digits produced. The following are new digits produced by my model.

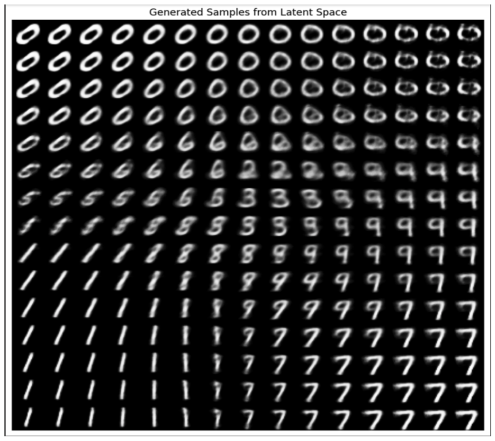

Figure 11. New Digits produced by VAE

4.3. Observation

When comparing VAE and GAN in practical applications, VAE tends to produce blurrier images compared to those generated by GAN. Additionally, some of the images generated by VAE models can be difficult to distinguish, such as those for digits 9 and 4 or 7 and 1. Although GAN is computationally intensive and requires more training epochs, its ability to produce sharper images makes it popular for generating high-resolution images.

One advantage of VAE over GAN is its ability to map input samples to a latent space before generating images. Recent advancements in generative models have proposed combining both GAN and VAE to leverage their respective strengths. This approach involves replacing the Generator in GAN with a VAE and using a Discriminator on the images generated by the new system. The loss is then backpropagated to train the updated Generator similarly to the original GAN setup. Larsen [7] has suggested this implementation, which could be considered as future work for our project.

5. Conclusions

In this report, we have presented two models employed for image generation. As advancements continue in training procedures and larger datasets become available, generative models are expected to successfully create data samples that are indistinguishable from training samples. With increased computational power, recent developments in both VAE and GAN can generate higher resolution images. Generative models can help address situations where there is a scarcity of training data for building models. Although we discussed the fundamental concepts, both GAN and VAE are continuously evolving. Due to resource limitations in training our models, we could not explore more complex applications. Nevertheless, this project report provides a foundation for machine learning enthusiasts to understand generative modeling and two of the latest techniques in the field.

6. 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.

7. Code

[1]https://colab.research.google.com
/drive/1ejTWDqXVJ01Cs2onYdagQw31h0NjiL1Y?usp=sharing