A Study of Deep Learning Based Generative Models: VAE vs GAN

Shanlong Guo
University of Florida
sh.guo@ufl.edu

All code for this report has been uploaded to Github: https://github.com/Terry GSL/MLProject.git

Abstract

In this project report, I mainly learn the basic principles of two major generative models GAN and VAE, and use them to generate some new images, the main dataset used is mnist, from the comparison of the generated results I found that the images generated by VAE are more blurred, while the images generated by GAN are realistic, but its potential space may not have good structure.

Since the images generated using GAN are more noisy, I also studied a variant model of GAN, DCGAN, which optimizes the network structure based on GAN and makes the network easier to train. I first used it to generate some new mnist handwritten digital images and found that the generated images are of higher quality and more stable for training. Then I used a crawler to crawl some anime images in anime websites and crop them, then I used DCGAN to generate new anime avatars on top of that dataset, and the final avatars I got were very good.

Finally, I found a paper which describes a new hybrid generation model: CVAE-GAN, which combines the advantages of CVAE and GAN, using CVAE to encode the input and GAN to generate the image, and can use the prior probability distribution of CVAE to control the generated image properties, and also can improve the generated image by the adversarial loss function of GAN to improve the So I tried to reproduce the model of this paper to generate mnist handwritten digits, and I ended up with good results, and I was also able to generate the specified different styles of digits, and also observe how one digit is slowly converted into another one.

1 Introduction

With the rapid development of deep learning, generative models have attracted more and more attention in recent years. Among various generative models, generative adversarial networks (GAN) and variational autoencoders (VAE) are two widely used and studied models.

GANs can generate high-quality images similar to real images, but they suffer from training instability and pattern collapse problems. On the other hand, VAE can generate diverse images, but the image quality is lower. Therefore, researchers have been exploring how to combine the advantages of GAN and VAE to overcome their limitations.

In this project, we investigated the basic principles of GAN and VAE and used them to generate new images in the MNIST dataset. We found that although GAN generates more realistic images, its latent space may not have a well-organized structure. vae generates more diverse images, but the image quality is lower.

To improve the stability and image quality of GAN, we also explored the DCGAN model, which optimizes the network structure of GAN and shows better training stability and image quality. We applied DCGAN to a new anime face dataset and obtained high quality and diverse anime face images.

Finally, we investigate a new hybrid model, called CVAE-GAN, which combines the advantages of CVAE and GAN. The model uses CVAE to encode the input and GAN to generate the images, and the generated image attributes can be controlled by the prior probability distribution of CVAE. We implemented the CVAE-GAN model on the MNIST dataset and achieved good results, including the ability to generate different styles of figures and observe a gradual shift from one figure to another.

In summary, this project outlines the basic principles of GAN and VAE and their limitations, explores the DCGAN model for better image quality and stability, and investigates a new hybrid model, CVAE-GAN, which has the potential to generate high-quality images with controllable attributes.

2 Related Work

In recent years, there has been a growing interest in applying deep learning techniques to the field of image generation, and Generative Adversarial Network (GAN) and Variational Autoencoder (VAE) are two of the most widely used models.

GAN was proposed by Goodfellow et al. (2014), which showed impressive results because it can generate high quality images similar to real images. However, the training process of GAN is unstable, so it may lead to some crashes or low-quality image generation. To address these issues, several modifications to GAN architectures have been proposed, such as Wasserstein GAN (Arjovsky et al., 2017) and Spectral Normalization GAN (Miyato et al., 2018).

In contrast, VAE was introduced by Kingma and Welling (2013), it aim to learn a probabilistic distribution of the input data, and it allow for the generation of new samples from the learned distribution. While VAE are able to generate diverse samples, they often produce lower quality images compared to GAN due to the weaker reconstruction loss. Various modifications have been proposed to improve the quality of VAE-generated images, such as -VAE (Higgins et al., 2017) and Adversarial Autoencoders (Makhzani et al., 2016). VAE have also been used in various image generation tasks, such as 3D shape generation (Wu et al., 2016) and face generation (Larsen et al., 2015).

To leverage the strengths of both GANs and VAEs, hybrid models have been

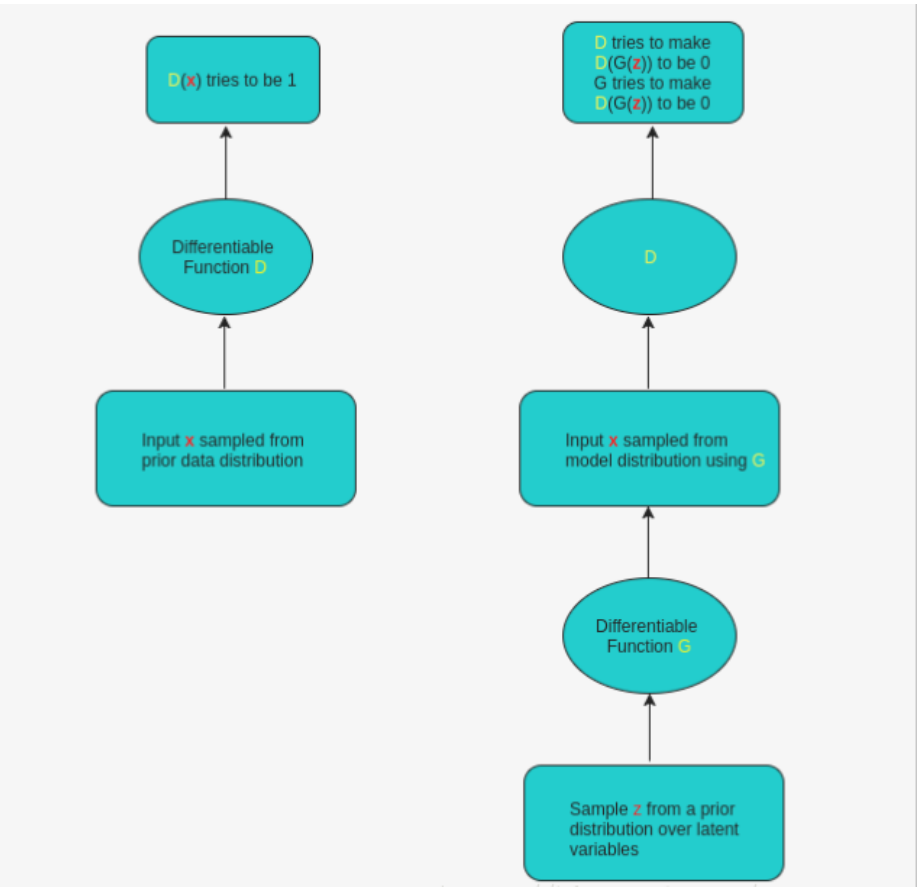
proposed, such as Adversarial Autoencoder (Makhzani et al., 2016) and Adversarial Variational Bayes (Mescheder et al., 2017). These models aim to learn a low-dimensional latent representation of the input data using the encoder component of a VAE, which is then used to generate new samples using the decoder component of a GAN. Another hybrid model, CVAE-GAN (Zhao et al., 2017), uses the prior distribution learned by the VAE to control the image attributes during the GAN's image generation process.

In summary, GAN and VAE have been extensively studied and have made significant progress in image generation tasks. Various modifications and hybrid models have been proposed to overcome the limitations of each individual model, and have shown impressive results in generating high-quality and diverse images.

3 Deep Generative Models Review

3.1 GAN

A discriminator and a generator work together in the GAN, an alternate method for producing data based on a given prior distribution.
The generator creates images from random noise, which is frequently referred to as an eigenvector or code and is typically derived from a uniform distribution (homogeneous distribution) or a Gaussian distribution, while the discriminator is used to distinguish "true" and "false" images.
The generator's job is to create a fakeable image that even the discriminator will have trouble identifying. In other words, the discriminator and the generator are at odds with one another. The generator does its best to produce more realistic images so that the discriminator would classify them as true, while the discriminator works very hard to discriminate between true and false images.



Let's say we have the networks G (generator) and D (discriminator), respectively. The generator network G aims to produce realistic visuals during training in an effort to trick the discriminator network D. D's objective is to attempt to distinguish between the fake images produced by G and the real ones. G and D create a dynamic "game process" in this manner.

This principle can be expressed in the following equation:

Suppose we have a training set $S = x(1), ..., x(m)$. Moreover, given any probability density function $pz(z)$ (of course, the simpler the corresponding probability distribution is, the better, while keeping the model complexity), we can use the random variable $Zpz(z)$ to sample m noisy samples $z(1), ..., z(m)$. ,z(m)}$.

From this, we can obtain the likelihood function:

$$
\begin{aligned}
&L(x^{(1)}, \ldots, x^{(m)}, z^{(1)}, \ldots, z^{(m)} \mid \theta_g, \theta_d) \\
&= \prod_{i=1}^{m} D(x^{(i)})^{I\{x^{(i)} \in \text{Data}\}} (1 - D(x^{(i)}))^{I\{x^{(i)} \notin \text{Data}\}} \\
&\qquad \times \prod_{j=1}^{m} D(G(x^{(j)}))^{I\{G(x^{(j)}) \in \text{Data}\}} (1 - D(G(x^{(j)})))^{I\{G(x^{(j)}) \notin \text{Data}\}} \\
&= \prod_{i=1}^{m} D(x^{(i)}) \prod_{j=1}^{m} (1 - D(G(x^{(j)})))
\end{aligned}
$$

Further, the log-likelihood is obtained as:

$$
\begin{aligned}
\log L &= \log \left(\prod_{i=1}^{m} D\left(x^{(i)}\right) \prod_{j=1}^{m} \left(1 - D\left(G\left(x^{(j)}\right)\right)\right) \right) \\
&= \sum_{i=1}^{m} \log D\left(x^{(i)}\right) + \sum_{j=1}^{m} \log \left(1 - D\left(G\left(z^{(j)}\right)\right)\right)
\end{aligned}
$$

By the law of large numbers, when mß, we approximate the expected loss by the empirical loss and obtain:

$$
\log L \approx \mathrm{E}_{x \sim p_{\text{data}}\ (x)}[\log D(x)] + \mathrm{E}_{z \sim p_z(z)}[\log(1 - D(x))] \mid
$$

On the one hand, we want to optimize the learnable parameters of the discriminator to maximize the log-likelihood function, and on the other hand, we want to optimize them to reduce the log-likelihood function. Formalizing this leads to the following optimization goal:

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

- There are just two terms in the entire equation. In the following, "x" stands for the actual image, "z" for the noise input to the G network, and "G(z)" for the image produced by the G network.
- Since x is the real image, the closer this value is to 1 for D, the more likely the D network is to identify that **the real picture is real**. And D(G(z)) is the likelihood that the **D network will decide whether or not the G-generated picture is accurate.**
- The goal of G: as stated above, D(G(z)) is the likelihood** that the D network will discern whether the image produced by G is real or not. In other words, since V(D, G) will be tiny, G wants D(G(z)) to be as large as feasible. Thus, we can see that min_G is the equation's top notation.
- The goal of D: the larger D(x) and the smaller D(G(x)) should be, the more capable D is. The size of V(D,G) will increase at this point. In order to determine the maximum value for D, the equation is (max_D).

Here, D and G are trained using the stochastic gradient descent approach. This is how the algorithm works:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

The first phase is training D. D is adding the gradient (ascending) in an effort to make V(G, D) as big as it can be. The gradient is subtracted (descending) in the second step of our training process for G such that V(G, D) is as minimal as possible. Alternating training methods are used throughout.

3.2 DCGAN

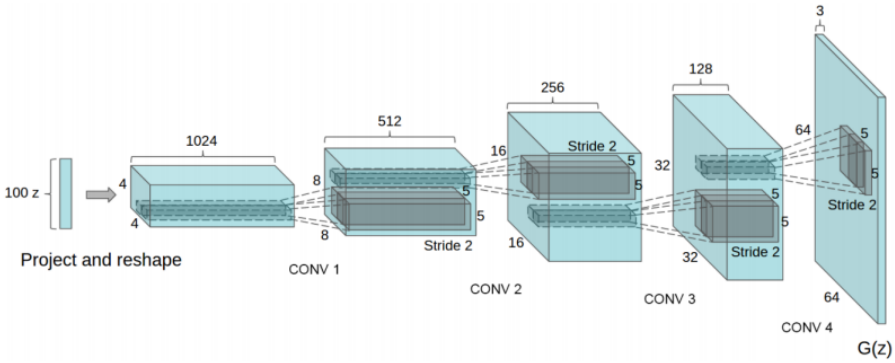
CNN and GAN are combined to create DCGAN. Convolutional networks are incorporated into the generative model to do unsupervised training, and the powerful feature extraction capabilities of convolutional networks are used to enhance the generative network's learning capabilities.

The basic idea behind DCGAN is the same as that of GAN, with the exception that two convolutional neural networks (CNNs) take the place of G and D in GAN, and DCGAN modifies the design of convolutional neural networks to enhance sample quality and convergence speed:

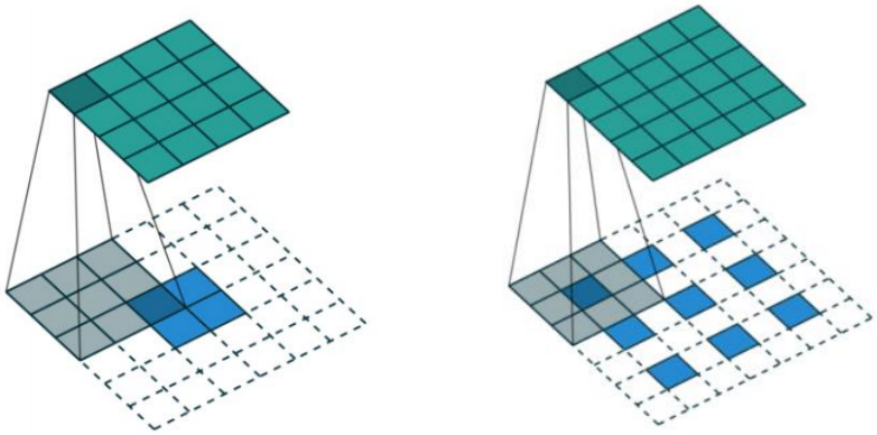
- There are no longer any pooling layers. The G network uses a transposed convolutional layer for upsampling, and the D network substitutes stride convolution for pooling.
- Both D and G use batch normalization.
- To convert the network into a completely convolutional network, remove the FC layer.

- Tanh is employed in the bottom layer while ReLU is used as the activation function in the G network.
- The activation function in the D network is LeakyReLU.

Schematic representation of the G network in DCGAN:



As can be seen, the input of the generator is a 100-dimensional noise, and the middle will pass through 4 convolution layers, with each convolution layer halving the number of channels and doubling the length and width to produce a $64*64*3$ size image output. It should be noted that many papers citing DCGAN make the mistaken assumption that the four convolutional layers are **Wide Convolution** when, in fact, they are **Fractionally Strided Convolution**. The difference between the two is illustrated in the figure below:



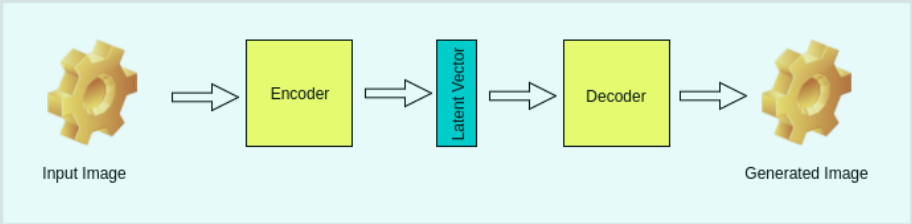
In contrast to micro-step amplitude convolution, which divides the input matrix and adds zeros around each individual pixel point, broad convolution adds zeros around the whole input matrix.

The above two **convolutional** operations that map from low-dimensional features to high-dimensional features are called **transpose convolution**, also known as

deconvolution.

3.3 VAE

A priori data distributions can be modeled using the variational self-encoder. It is made up of an encoder and a decoder. A latent vector is a low-level representation of the data that is created by the encoder by mapping high-level properties of the data distribution. The data's low-level representations are ingested by the decoder, which then produces its high-level counterparts.



3.3.1 Variational Lower bound

Additionally, the posterior probability is typically maximized to optimize the created model. That is, using the Bayesian formulation:

$$p(z \mid X) = \frac{p(X \mid z)p(z)}{\int_z p(X \mid z)p(z)\mathrm{d}z}$$

We want to replace the posterior probability $p(z|X)$ with the new function $q(z)$, then the two probability distributions need to be as similar as possible, and here the KL scatter is still chosen to measure the closeness of the two. According to the KL formula then we have:

$$\begin{aligned}\mathrm{KL}(q(z)\|p(z \mid X)) &= \int q(z) \log \frac{q(z)}{p(z \mid X)}\mathrm{d}z \\ &= \int q(z)[\log q(z) - \log p(z \mid X)]\mathrm{d}z\end{aligned}$$

Transformation according to the Bayesian formula yields:

$$\begin{aligned}&= \int q(z) \left[\log q(z) - \log \frac{p(X \mid z)p(X)}{p(z)}\right] \mathrm{d}z \\ &= \int q(z)[\log q(z) - \log p(X \mid z) - \log p(z) + \log p(X)]\mathrm{d}z\end{aligned}$$

Since the objective of the integration is z, the items that are not related to z are then taken out of the integration notation to obtain:

$$= \int q(z)[\log q(z) - \log p(X \mid z) - \log p(z)]\mathrm{d}z + \log p(X)$$

Swapping the equation left and right yields the following equation:

$$\log p(X) - \mathrm{KL}(q(z)\|p(z \mid X)) = \int q(z) \log p(X \mid z) \mathrm{d}z - \mathrm{KL}(q(z)\|p(z))$$

The goal of our training is to want KL(q(z)||p(z|X)) to be as small as possible, which is equivalent to making the right side of the equal sign as large as possible. The first term on the right side of the equal sign is actually based on the log-likelihood expectation of the probability of $q(z)$, and the second term is again a negative KL scatter, so we can assume that in order to find a good $q(z)$ that makes it as similar as possible to $p(z|X)$ and achieve the final optimization goal, the optimization objective will become:

$$\max \int q(z) \log p(X \mid z) \mathrm{d}z$$
$$\min \mathrm{KL}(q(z)||p(z))$$

3.3.2 Reparameterization Trick

The variational function $q(z)$, which, to approximate the posterior probability, actually represents the distribution of z given some X, would be $q(z|X)$ if its probability form were written in its entirety. We need to find a way to abstract it out of X. This conditional probability can be split into two parts, one is an observed variable $g(X)$, which represents the deterministic part of the conditional probability and its value is similar to the expectation of a random variable; the other part is the random variable , which is responsible for the random part. If $z(i) = g(X + (i))$, then $q(z(i)) = p((i))$, so the above formula on the derivation of the variance can be turned into the following one:

$$\log p(X) - \mathrm{KL}(q(z)\|p(z \mid X)) = \int p(\varepsilon) \log p\left(X \mid g_\phi(X, \varepsilon)\right) \mathrm{d}z - \mathrm{KL}(q(z \mid X, \varepsilon)\|p(z))$$

This assumption obeys a multidimensional and independent Gaussian distribution in each dimension. Also, the prior and posterior of z are assumed to be a multidimensional and independent Gaussian distribution in each dimension. The final form of the two optimization objectives is shown below.

3.3.3 Encoder and Decoder formula

Getting the second term $KL(q(z)||p(z)))$ on the right side of Equation as small as possible is our second optimization goal. The prior of z is currently assumed to be a multidimensional and independent Gaussian distribution in each dimension, and a stronger assumption can be made here that the mean of each dimension of this Gaussian distribution is zero and the covariance is the unit matrix. From this point, the previously mentioned KL scatter formula begins with:

$$\mathrm{KL}(p1\|p2) = \frac{1}{2}\left[\log \frac{\det\left(\Sigma_2\right)}{\det\left(\Sigma_1\right)} - \mathrm{d} + \mathrm{tr}\left(\Sigma_2^{-1}\Sigma_1\right) + \left(\mu_2 - \mu_1\right)^{\mathrm{T}} \Sigma_2^{-1}\left(\mu_2 - \mu_1\right)\right]$$

Simplified as:

$$\mathrm{KL}(p1\|N(0, I)) = \frac{1}{2}\left[-\log\left[\det\left(\Sigma_1\right)\right] - \mathrm{d} + \mathrm{tr}\left(\Sigma_1\right) + \mu_1^{\mathrm{T}}\mu_1\right]$$

A vector 1 to represent the major diagonal of the covariance matrix is all that is required in the calculation, eliminating the necessity to describe the covariance as the shape of a matrix. The formula will then be further reduced as follows:

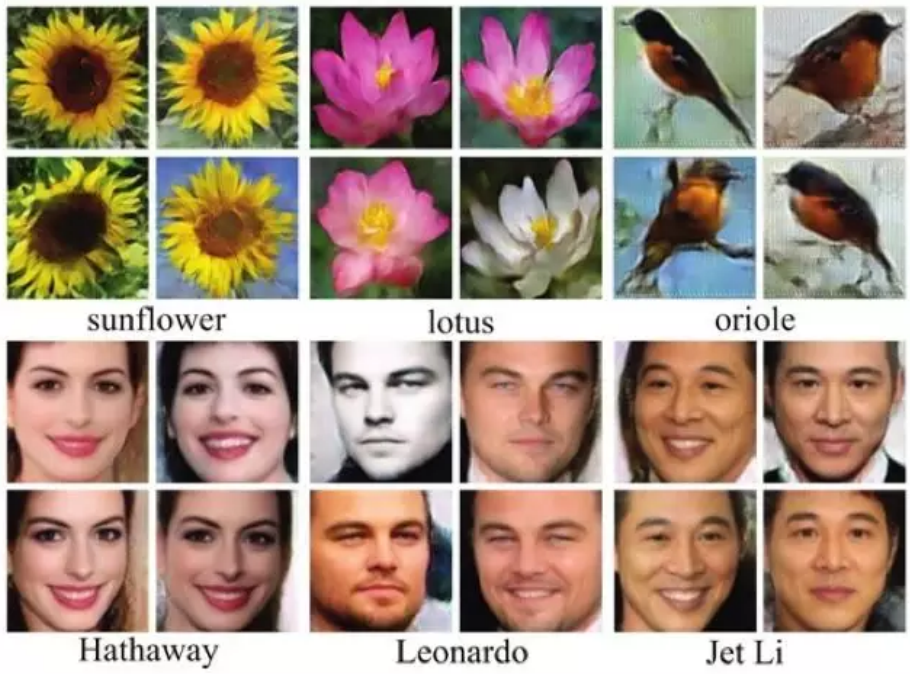
$$\mathrm{KL}\left(p1\left(\mu_1, \sigma_1\right) \| N(0, I)\right) = \frac{1}{2}\left[-\sum_i \log\left[\left(\sigma_{1i}\right)\right] - \mathrm{d} + \sum_i \left(\sigma_{1i}\right) + \mu_1^{\mathrm{T}} \mu_1\right]$$

This paradigm is known as the encoder model since the function $g()$ carries out the transformation from the observed data to the implied data.

The next optimization objective, which is to maximize the likelihood expectation of the first term on the left side of Equation. This part is relatively simple. Since the previous Encoder model has already calculated a batch of observed variables X corresponding to the implied variables z, another deep model can be built here, modeled according to the likelihood, with the input being the implied variables z and the output being the observed variables X. If the output image is similar to the previously generated image, then the likelihood is considered to be maximized. This model is called Decoder.

3.4 CVAE-GAN

The CVAE-GAN model combines the advantages of CVAE and GAN with better control, stability and diversity, and is a very effective generative model. Where C stands for the ability to generate images for a specified classification using the classification as input. It generates images quite well on each classification as shown in the following figure.



CVAE-GAN is mainly composed of the following 4 neural networks:

- E: Encoder (Encoder), input image x, output encoding z.
- G: Generator. Input encoding z, output image x.
- C: Classifier. Input image x, output category c.
- D: Discriminator. Input image x, determine its truthfulness.

The architecture of CVAE-GAN is shown in the following figure:

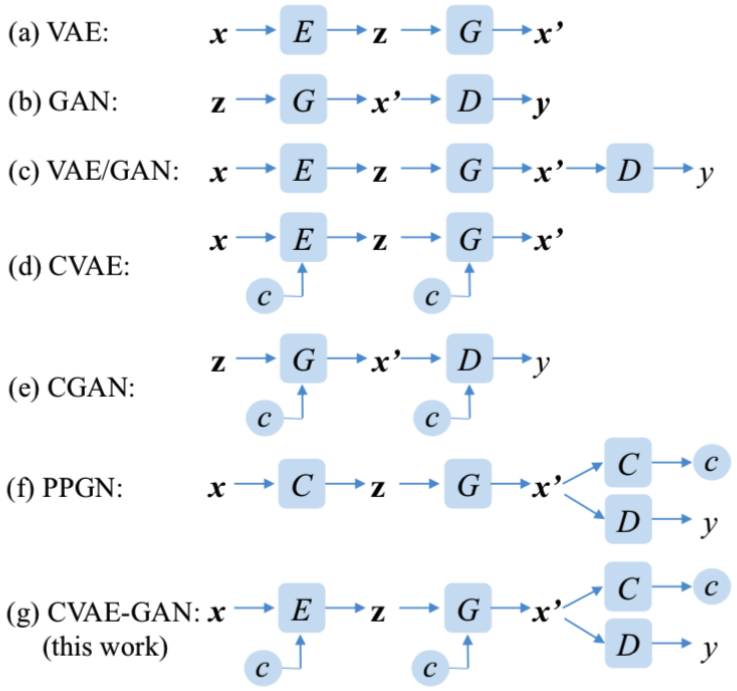


Figure 2. Illustration of the structure of VAE [12, 31], GAN [8], VAE/GAN [15], CVAE [34], CGAN [18], PPGN [23] and the proposed CVAE-GAN. Where $\boldsymbol{x}$ and $\boldsymbol{x}'$ are input and generated image. E,G,C,D are encoder, generative, classification, and discriminative network, respectively. $\boldsymbol{z}$ is the latent vector. y is a binary output which represents real/synthesized image. c is the condition, such as attribute or class label.

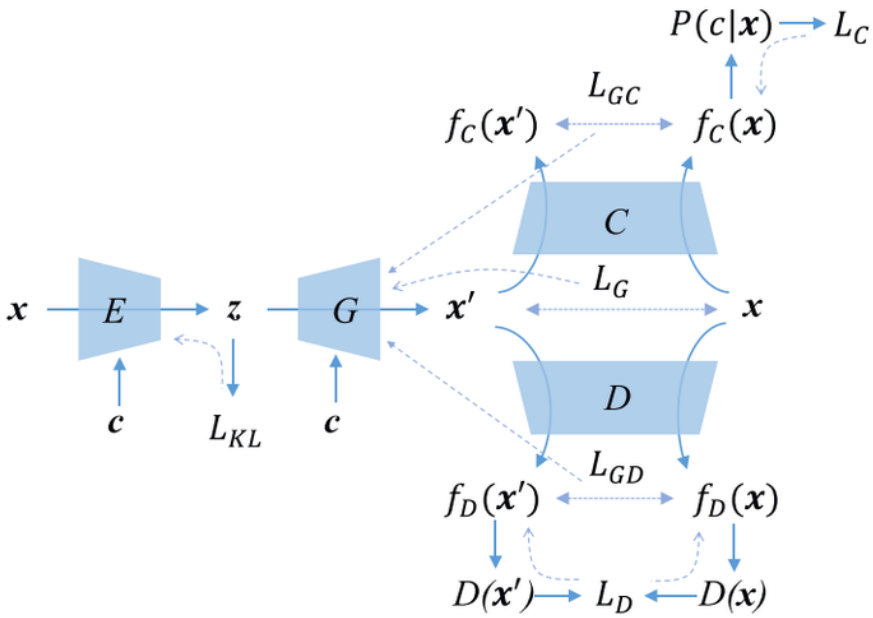
The G of VAE is better than the G of GAN, so the structure of VAE is in front. Then, since VAE's criterion for judging x' and x similarity is not good enough, we have to add D again to judge it. Finally, we need to ensure that the generated image belongs to the c-category, so C is also added.

where the Loss of G has three main parts:

- For z generated from x, G should be able to restore x' close to x (pixel-wise close)
- The image generated by G should be identifiable by D as belonging to the real image
- the image generated by G should be identifiable by C as belonging to category c

The resulting z can be fairly well carved out of the image.

The detailed architecture of CVAE-GAN is shown in Fig:



The training algorithm of CVAE-GAN is shown in the figure, where each item is intuitive. Note that there is also an important trick used in it, which is to expect x' and x to have similar features in the middle layers of the network for D and C. This helps stabilize the network: x' and x' have similar features in the middle layers. This helps to stabilize the network:

Algorithm 1 The training pipeline of the proposed CVAE-GAN algorithm.

Require: m, the batch size. n, class number. θ_E, initial E network parameters. θ_G, initial G network parameters. θ_D, initial D network parameters. θ_C, initial C network parameters.

while θ_G has not converged **do**
  Sample $\{x_r, c\} \sim P_r$ a batch from the real data;
  $\mathcal{L}_C \leftarrow -\log(P(c|x_r))$
  $z \leftarrow E(x_r, c)$
  Sample $z_p \sim P_{\boldsymbol{z}}$ a batch of prior samples;
  $\mathcal{L}_{KL} \leftarrow KL(q(z|x_r, c)||P_{\boldsymbol{z}})$
  $x_f \leftarrow G(z, c)$
  $x_p \leftarrow G(z_p, c)$
  $\mathcal{L}_D \leftarrow -(\log(D(x_r)) + \log(1 - D(x_f)) + \log(1 - D(x_p)))$
  Calculate x_r feature center $\frac{1}{m}\sum_i^m f_D(x_r)$ and x_p feature center $\frac{1}{m}\sum_i^m f_D(x_p)$;
  $\mathcal{L}_{GD} \leftarrow \frac{1}{2}||\frac{1}{m}\sum_i^m f_D(x_r) - \frac{1}{m}\sum_i^m f_D(x_p)||_2^2$
  Calculate each class c_i feature center $f_C^{c_i}(x_r)$ for x_r and $f_C^{c_i}(x_p)$ for x_p using moving average method;
  $\mathcal{L}_{GC} \leftarrow \frac{1}{2}\sum_{c_i}||f_C^{c_i}(x_r) - f_C^{c_i}(x_p)||_2^2$
  $\mathcal{L}_G \leftarrow \frac{1}{2}(||x_r - x_f||_2^2 + ||f_D(x_r) - f_D(x_f)||_2^2 + ||f_C(x_r) - f_C(x_f)||_2^2)$
  $\theta_C \xleftarrow{+} -\nabla_{\theta_C}(\mathcal{L}_C)$
  $\theta_D \xleftarrow{+} -\nabla_{\theta_D}(\mathcal{L}_D)$
  $\theta_G \xleftarrow{+} -\nabla_{\theta_G}(\mathcal{L}_G + \mathcal{L}_{GD} + \mathcal{L}_{GC})$
  $\theta_E \xleftarrow{+} -\nabla_{\theta_E}(\mathcal{L}_G + \mathcal{L}_{KL})$
end while

By using the four major networks E+G+C+D, CVAE-GAN achieves a fairly

satisfactory generative model.

4 Results

In total, we tested GAN, DCGAN, VAE, CVAE-GAN on handwritten digital datasets and also extraordinarily used DCGAN on top of our own crawled anime dataset and got good results. All the experimental codes are implemented using Pytorch, and the initial size of our mnist dataset images is 28×28, and the initial size of anime images is 256×256.

Here are some specific experimental results to show and some of our hyperparameter choices.

4.1 GAN

Hyper-parameters :

latent_size = 64

hidden_size = 256

image_size = 784

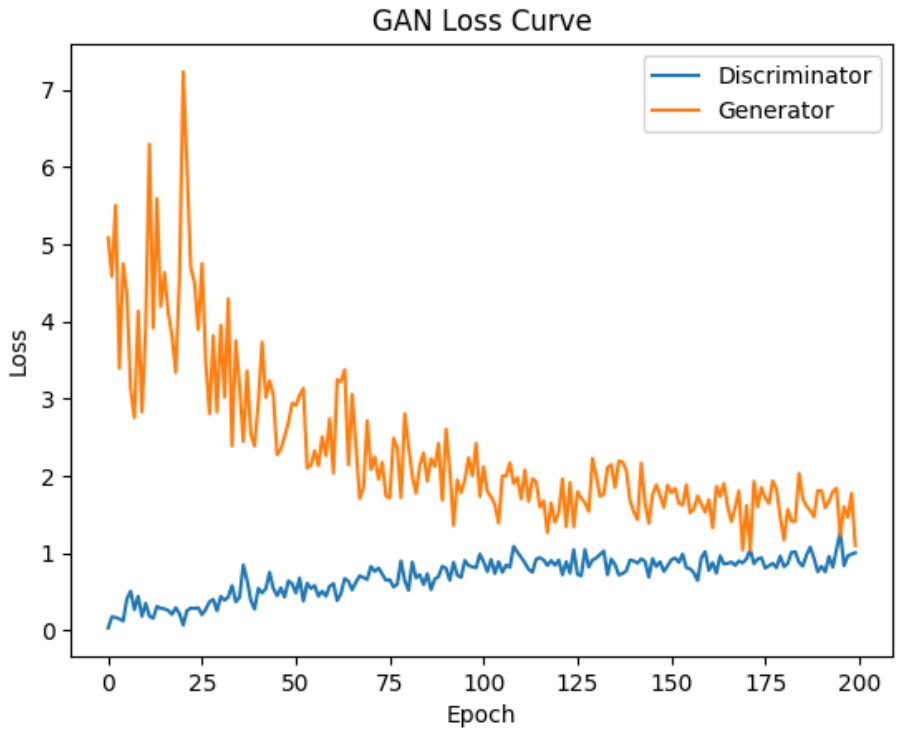
num_epochs = 200

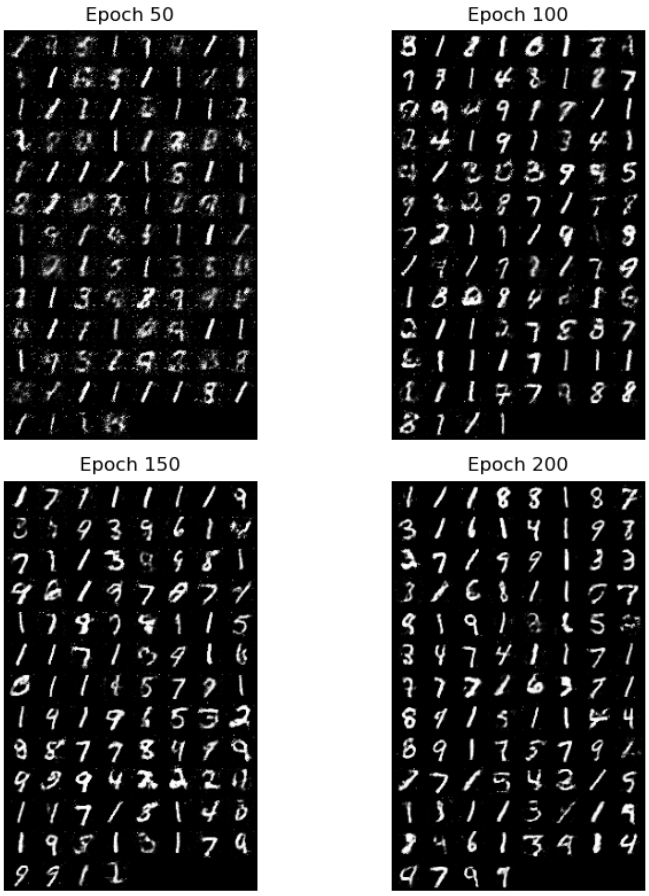
batch_size = 100

lr = 0.0002

Optimizer : adam optimizer

```
Output exceeds the size limit. Open the full output data in a text editor
Epoch [0/200], Step [600/600], d_loss: 0.0296, g_loss: 5.0803, D(x): 0.99, D(G(z)): 0.02
Epoch [1/200], Step [600/600], d_loss: 0.1741, g_loss: 4.5834, D(x): 0.96, D(G(z)): 0.10
Epoch [2/200], Step [600/600], d_loss: 0.1662, g_loss: 5.5027, D(x): 0.93, D(G(z)): 0.04
Epoch [3/200], Step [600/600], d_loss: 0.1475, g_loss: 3.3940, D(x): 0.94, D(G(z)): 0.05
Epoch [4/200], Step [600/600], d_loss: 0.1206, g_loss: 4.7457, D(x): 0.94, D(G(z)): 0.02
Epoch [5/200], Step [600/600], d_loss: 0.4063, g_loss: 4.3388, D(x): 0.92, D(G(z)): 0.17
Epoch [6/200], Step [600/600], d_loss: 0.5062, g_loss: 3.1278, D(x): 0.87, D(G(z)): 0.16
Epoch [7/200], Step [600/600], d_loss: 0.2673, g_loss: 2.7529, D(x): 0.89, D(G(z)): 0.05
Epoch [8/200], Step [600/600], d_loss: 0.4409, g_loss: 4.1331, D(x): 0.94, D(G(z)): 0.23
Epoch [9/200], Step [600/600], d_loss: 0.1815, g_loss: 2.8265, D(x): 0.95, D(G(z)): 0.09
Epoch [10/200], Step [600/600], d_loss: 0.3501, g_loss: 3.9315, D(x): 0.92, D(G(z)): 0.14
Epoch [11/200], Step [600/600], d_loss: 0.1818, g_loss: 6.2938, D(x): 0.95, D(G(z)): 0.03
Epoch [12/200], Step [600/600], d_loss: 0.1560, g_loss: 3.9150, D(x): 0.95, D(G(z)): 0.06
Epoch [13/200], Step [600/600], d_loss: 0.3099, g_loss: 5.5876, D(x): 0.89, D(G(z)): 0.02
Epoch [14/200], Step [600/600], d_loss: 0.2849, g_loss: 4.1917, D(x): 0.93, D(G(z)): 0.11
Epoch [15/200], Step [600/600], d_loss: 0.2777, g_loss: 4.6312, D(x): 0.92, D(G(z)): 0.06
Epoch [16/200], Step [600/600], d_loss: 0.2556, g_loss: 4.1131, D(x): 0.95, D(G(z)): 0.13
Epoch [17/200], Step [600/600], d_loss: 0.2067, g_loss: 3.8159, D(x): 0.94, D(G(z)): 0.10
Epoch [18/200], Step [600/600], d_loss: 0.2926, g_loss: 3.3388, D(x): 0.89, D(G(z)): 0.05
Epoch [19/200], Step [600/600], d_loss: 0.2170, g_loss: 4.5336, D(x): 0.93, D(G(z)): 0.04
Epoch [20/200], Step [600/600], d_loss: 0.0676, g_loss: 7.2281, D(x): 0.97, D(G(z)): 0.02
Epoch [21/200], Step [600/600], d_loss: 0.2557, g_loss: 6.0146, D(x): 0.90, D(G(z)): 0.02
Epoch [22/200], Step [600/600], d_loss: 0.2863, g_loss: 4.6929, D(x): 0.94, D(G(z)): 0.09
Epoch [23/200], Step [600/600], d_loss: 0.2822, g_loss: 4.4984, D(x): 0.97, D(G(z)): 0.16
Epoch [24/200], Step [600/600], d_loss: 0.2924, g_loss: 3.8953, D(x): 0.91, D(G(z)): 0.04
...
Epoch [196/200], Step [600/600], d_loss: 0.8356, g_loss: 1.5983, D(x): 0.70, D(G(z)): 0.25
Epoch [197/200], Step [600/600], d_loss: 0.9650, g_loss: 1.4631, D(x): 0.72, D(G(z)): 0.35
Epoch [198/200], Step [600/600], d_loss: 0.9860, g_loss: 1.7756, D(x): 0.67, D(G(z)): 0.28
Epoch [199/200], Step [600/600], d_loss: 1.0037, g_loss: 1.0970, D(x): 0.71, D(G(z)): 0.36
```





4.2 DCGAN

Dataset: mnist

Hyper-parameters :

batch_size = 128

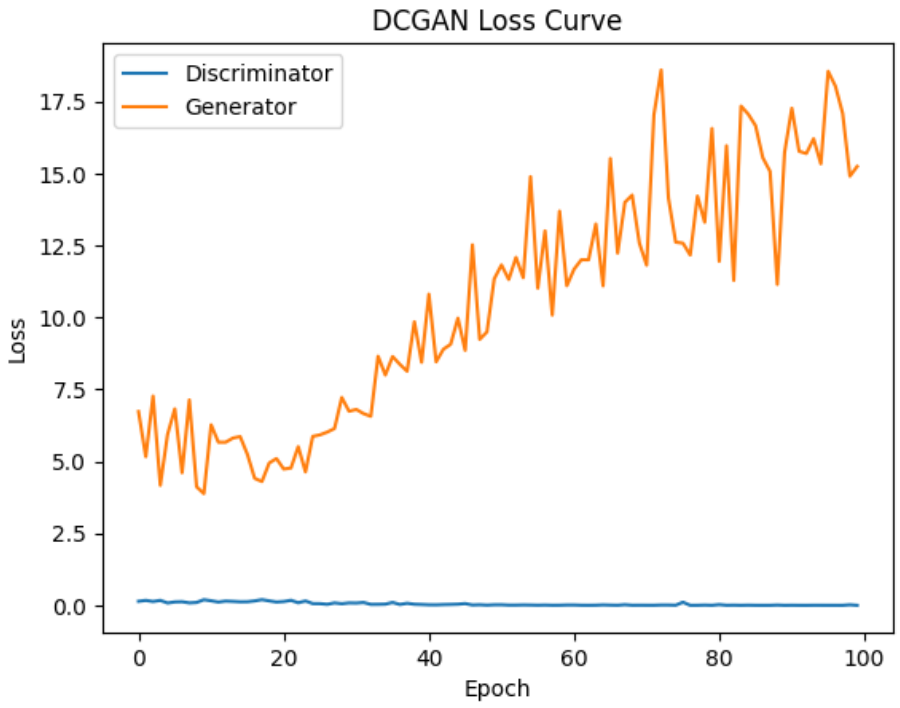
num_epoch = 100

z_dimension = 100

lr = 0.0003

Optimizer : adam optimizer

```
Output exceeds the size limit. Open the full output data in a text editor
Epoch[0/100],d_loss:0.140446,g_loss:6.732050 D real: 0.987276,D fake: 0.074173
Epoch[1/100],d_loss:0.172050,g_loss:5.164668 D real: 0.974520,D fake: 0.081085
Epoch[2/100],d_loss:0.133183,g_loss:7.271091 D real: 0.941956,D fake: 0.017553
Epoch[3/100],d_loss:0.172014,g_loss:4.168848 D real: 0.968638,D fake: 0.088267
Epoch[4/100],d_loss:0.081170,g_loss:5.937690 D real: 0.955697,D fake: 0.013355
Epoch[5/100],d_loss:0.117887,g_loss:6.826725 D real: 0.962372,D fake: 0.013424
Epoch[6/100],d_loss:0.124918,g_loss:4.606397 D real: 0.948755,D fake: 0.031349
Epoch[7/100],d_loss:0.085299,g_loss:7.136805 D real: 0.956387,D fake: 0.021232
Epoch[8/100],d_loss:0.099250,g_loss:4.117732 D real: 0.965669,D fake: 0.035886
Epoch[9/100],d_loss:0.195303,g_loss:3.880519 D real: 0.968290,D fake: 0.109111
Epoch[10/100],d_loss:0.159540,g_loss:6.272925 D real: 0.935438,D fake: 0.006777
Epoch[11/100],d_loss:0.113170,g_loss:5.665688 D real: 0.962646,D fake: 0.023689
Epoch[12/100],d_loss:0.148022,g_loss:5.666120 D real: 0.934990,D fake: 0.029748
Epoch[13/100],d_loss:0.136893,g_loss:5.809911 D real: 0.949200,D fake: 0.028227
Epoch[14/100],d_loss:0.122135,g_loss:5.863274 D real: 0.950459,D fake: 0.028711
Epoch[15/100],d_loss:0.124421,g_loss:5.243636 D real: 0.962375,D fake: 0.030054
Epoch[16/100],d_loss:0.154353,g_loss:4.412270 D real: 0.961781,D fake: 0.059659
Epoch[17/100],d_loss:0.197090,g_loss:4.299699 D real: 0.939031,D fake: 0.055516
Epoch[18/100],d_loss:0.155868,g_loss:4.944643 D real: 0.940691,D fake: 0.041951
Epoch[19/100],d_loss:0.114503,g_loss:5.098955 D real: 0.958998,D fake: 0.038819
Epoch[20/100],d_loss:0.131589,g_loss:4.735130 D real: 0.969750,D fake: 0.070791
Epoch[21/100],d_loss:0.174895,g_loss:4.770987 D real: 0.968188,D fake: 0.086378
Epoch[22/100],d_loss:0.089229,g_loss:5.513806 D real: 0.974610,D fake: 0.038539
Epoch[23/100],d_loss:0.152008,g_loss:4.634298 D real: 0.975191,D fake: 0.077039
Epoch[24/100],d_loss:0.060360,g_loss:5.869754 D real: 0.972191,D fake: 0.017060
...
Epoch[96/100],d_loss:0.001106,g_loss:18.038557 D real: 0.998928,D fake: 0.000006
Epoch[97/100],d_loss:0.000801,g_loss:17.091667 D real: 0.999329,D fake: 0.000117
Epoch[98/100],d_loss:0.018523,g_loss:14.908038 D real: 0.999996,D fake: 0.007515
Epoch[99/100],d_loss:0.000141,g_loss:15.248213 D real: 0.999892,D fake: 0.000033
```



Dataset: anime

Random Seed: 999

workers = 2

batch_size = 128

image_size = 64

since images are RGB they have 3 channels: nc = 3

generator input size: nz = 100

generator's feature maps: ngf = 64

discriminator's feature maps: ndf = 64

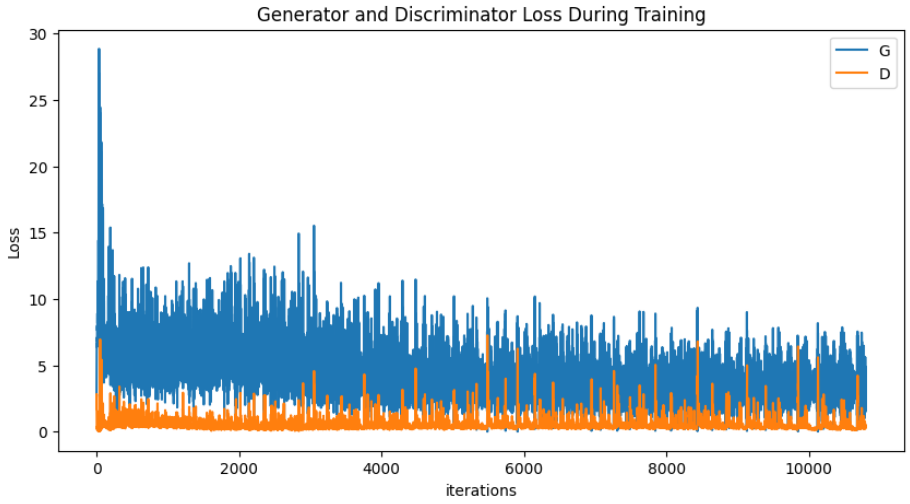
num_epochs = 50

lr = 0.0002

beta1 = 0.5

Optimizer : adam optimizer

```
Output exceeds the size limit. Open the full output data in a text editor
Starting Training Loop...
[0/50][0/216]   Loss_D: 1.7402  Loss_G: 2.7207  D(x): 0.3710    D(G(z)): 0.3621 / 0.0872
[0/50][50/216]  Loss_D: 6.9571  Loss_G: 24.4530 D(x): 0.9935    D(G(z)): 0.9613 / 0.0000
[0/50][100/216] Loss_D: 0.8723  Loss_G: 7.1864  D(x): 0.6143    D(G(z)): 0.0130 / 0.0031
[0/50][150/216] Loss_D: 0.5579  Loss_G: 5.2083  D(x): 0.7486    D(G(z)): 0.0296 / 0.0078
[0/50][200/216] Loss_D: 0.4323  Loss_G: 5.9701  D(x): 0.7792    D(G(z)): 0.0372 / 0.0048
[1/50][0/216]   Loss_D: 1.9490  Loss_G: 9.0669  D(x): 0.3771    D(G(z)): 0.0002 / 0.0006
[1/50][50/216]  Loss_D: 1.0592  Loss_G: 7.9741  D(x): 0.8970    D(G(z)): 0.4685 / 0.0013
[1/50][100/216] Loss_D: 0.5057  Loss_G: 4.8977  D(x): 0.8859    D(G(z)): 0.2745 / 0.0158
[1/50][150/216] Loss_D: 0.7840  Loss_G: 9.5407  D(x): 0.9174    D(G(z)): 0.4428 / 0.0002
[1/50][200/216] Loss_D: 0.4973  Loss_G: 5.1126  D(x): 0.8075    D(G(z)): 0.1648 / 0.0104
[2/50][0/216]   Loss_D: 0.6915  Loss_G: 4.5089  D(x): 0.8686    D(G(z)): 0.3605 / 0.0267
[2/50][50/216]  Loss_D: 0.5036  Loss_G: 7.4418  D(x): 0.9286    D(G(z)): 0.3058 / 0.0014
[2/50][100/216] Loss_D: 1.2548  Loss_G: 7.6999  D(x): 0.7718    D(G(z)): 0.5072 / 0.0016
[2/50][150/216] Loss_D: 1.4882  Loss_G: 10.7720 D(x): 0.9017    D(G(z)): 0.6692 / 0.0001
[2/50][200/216] Loss_D: 0.5070  Loss_G: 6.3069  D(x): 0.9286    D(G(z)): 0.2963 / 0.0059
[3/50][0/216]   Loss_D: 0.4017  Loss_G: 5.4653  D(x): 0.7718    D(G(z)): 0.0384 / 0.0102
[3/50][50/216]  Loss_D: 0.3400  Loss_G: 3.9451  D(x): 0.8688    D(G(z)): 0.1029 / 0.0276
[3/50][100/216] Loss_D: 0.4540  Loss_G: 5.9376  D(x): 0.9200    D(G(z)): 0.2680 / 0.0048
[3/50][150/216] Loss_D: 0.6395  Loss_G: 4.0553  D(x): 0.6705    D(G(z)): 0.0810 / 0.0289
[3/50][200/216] Loss_D: 0.7837  Loss_G: 4.0758  D(x): 0.5949    D(G(z)): 0.0475 / 0.0395
[4/50][0/216]   Loss_D: 0.5524  Loss_G: 3.2333  D(x): 0.8004    D(G(z)): 0.1797 / 0.0783
[4/50][50/216]  Loss_D: 0.9554  Loss_G: 6.3494  D(x): 0.8185    D(G(z)): 0.3972 / 0.0093
[4/50][100/216] Loss_D: 1.6145  Loss_G: 10.2168 D(x): 0.8901    D(G(z)): 0.6970 / 0.0003
[4/50][150/216] Loss_D: 0.7935  Loss_G: 8.2887  D(x): 0.9618    D(G(z)): 0.4617 / 0.0011
...
[49/50][50/216] Loss_D: 0.2515  Loss_G: 2.4077  D(x): 0.8259    D(G(z)): 0.0410 / 0.1285
[49/50][100/216]        Loss_D: 2.4724  Loss_G: 7.2594  D(x): 0.9930    D(G(z)): 0.8607 / 0.0017
[49/50][150/216]        Loss_D: 0.3184  Loss_G: 2.4241  D(x): 0.8816    D(G(z)): 0.1509 / 0.1368
[49/50][200/216]        Loss_D: 0.2032  Loss_G: 3.2884  D(x): 0.9091    D(G(z)): 0.0916 / 0.0593
```



4.3 VAE

latent_dim = 2

input_dim = 28 * 28

inter_dim = 256

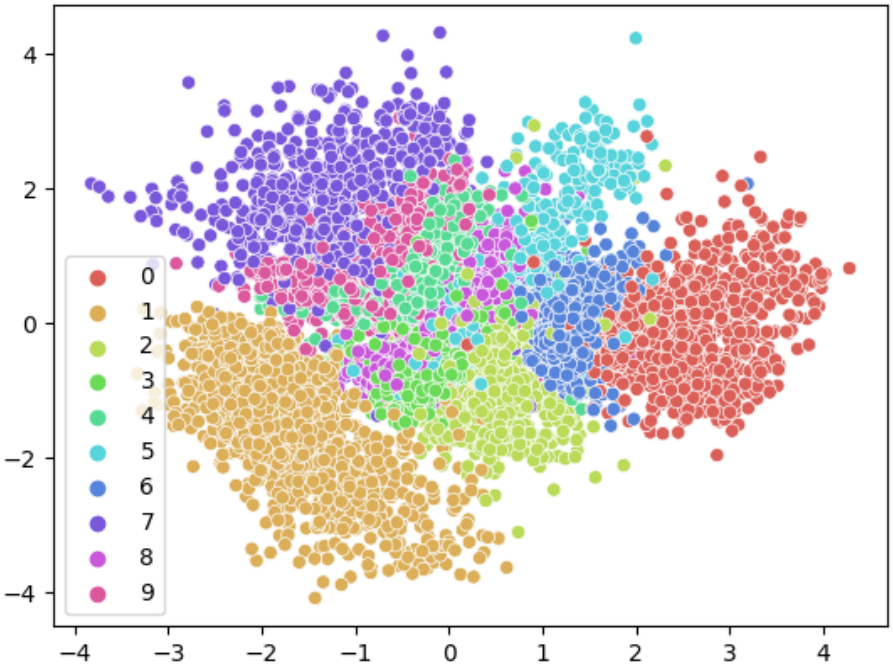
epochs = 200

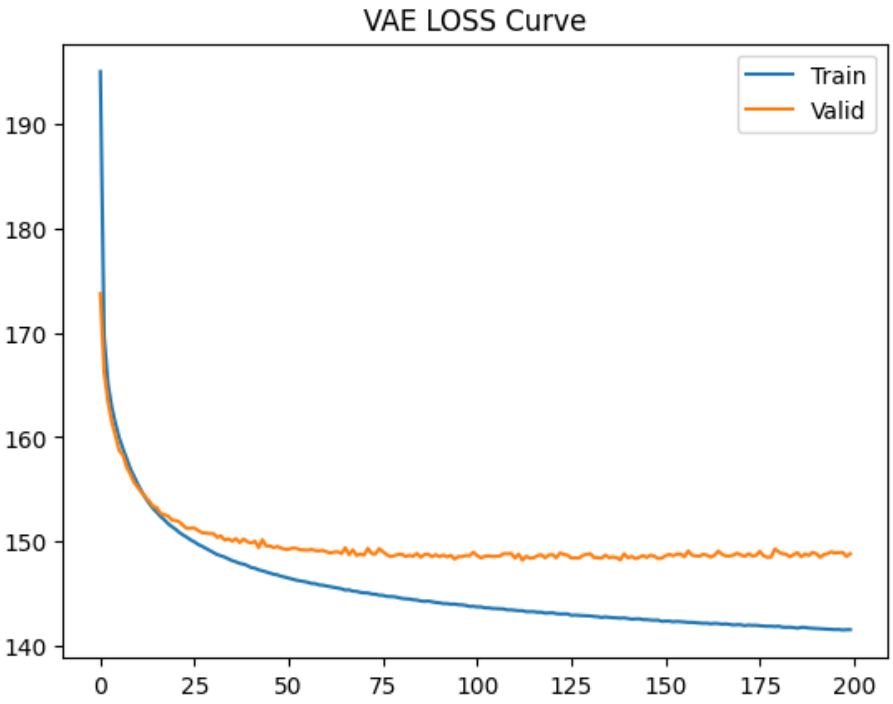
batch_size = 128

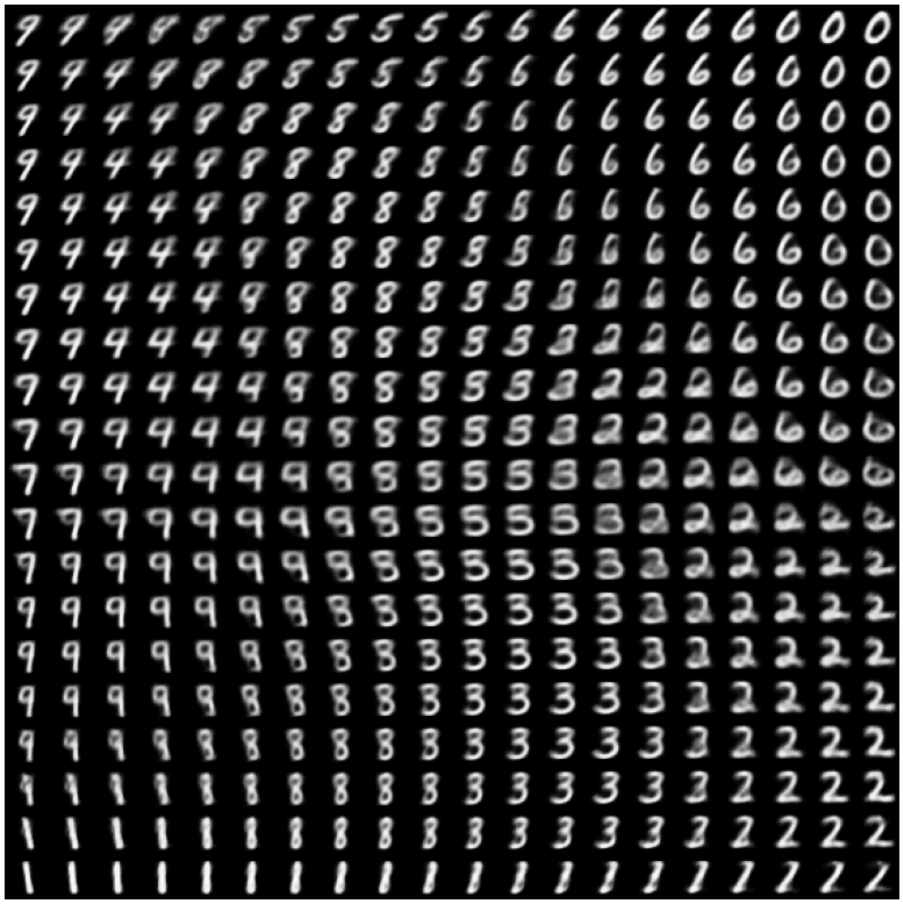
lr = 0.0001

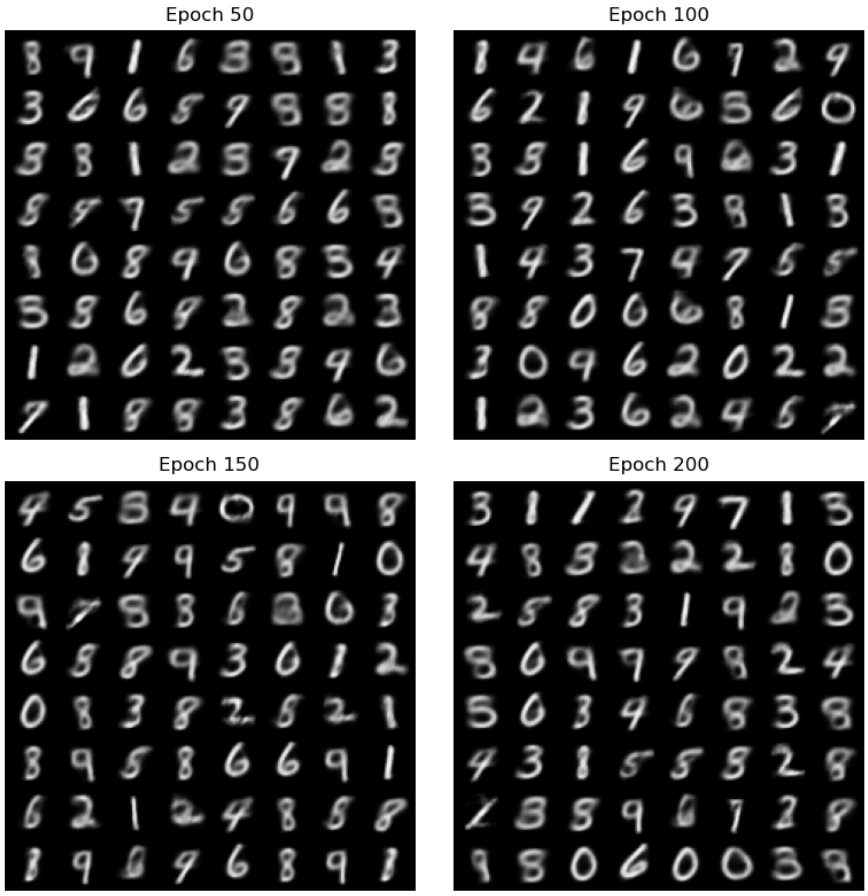
Optimizer : adam optimizer

```
Output exceeds the size limit. Open the full output data in a text editor
Training loss  194.923   Recon  187.345          KL  7.578 in Step 100
Training loss  181.586   Recon  175.466          KL  6.120 in Step 200
Training loss  184.401   Recon  178.980          KL  5.421 in Step 300
Training loss  180.530   Recon  175.729          KL  4.800 in Step 400
Valid loss  173.751      Recon  168.837          KL  4.914 in epoch 0
Model saved
Epoch 1
Training loss  167.473   Recon  162.389          KL  5.084 in Step 0
Training loss  176.803   Recon  172.067          KL  4.736 in Step 100
Training loss  171.387   Recon  166.446          KL  4.941 in Step 200
Training loss  166.377   Recon  161.253          KL  5.125 in Step 300
Training loss  165.067   Recon  160.031          KL  5.036 in Step 400
Valid loss  166.230      Recon  161.084          KL  5.146 in epoch 1
Model saved
Epoch 2
Training loss  168.586   Recon  163.434          KL  5.152 in Step 0
Training loss  169.467   Recon  164.222          KL  5.245 in Step 100
Training loss  169.844   Recon  164.919          KL  4.925 in Step 200
Training loss  161.312   Recon  156.184          KL  5.127 in Step 300
Training loss  165.208   Recon  159.957          KL  5.251 in Step 400
Valid loss  163.443      Recon  158.274          KL  5.169 in epoch 2
Model saved
Epoch 3
Training loss  166.361   Recon  161.443          KL  4.918 in Step 0
Training loss  165.392   Recon  160.324          KL  5.069 in Step 100
...
Training loss  140.463   Recon  133.726          KL  6.737 in Step 200
Training loss  139.468   Recon  132.923          KL  6.545 in Step 300
Training loss  137.144   Recon  130.661          KL  6.483 in Step 400
Valid loss  148.831      Recon  142.171          KL  6.661 in epoch 199
```


4
2
0
−2
−4
−4
−3
−2
−1
0
1
2
3
4
0
1
2
3
4
5
6
7
8
9


VAE LOSS Curve
Train
Valid
190
180
170
160
150
140
0
25
50
75
100
125
150
175
200





4.4 CVAE-GAN

batchSize = 128

imageSize = 28

nz = 100

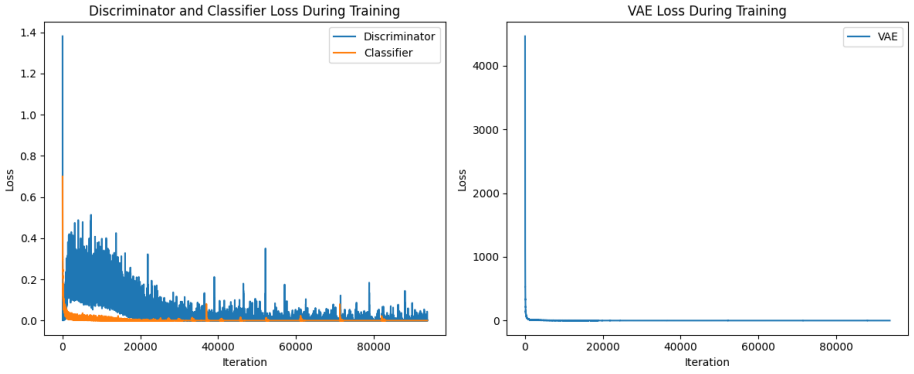
nepoch = 200

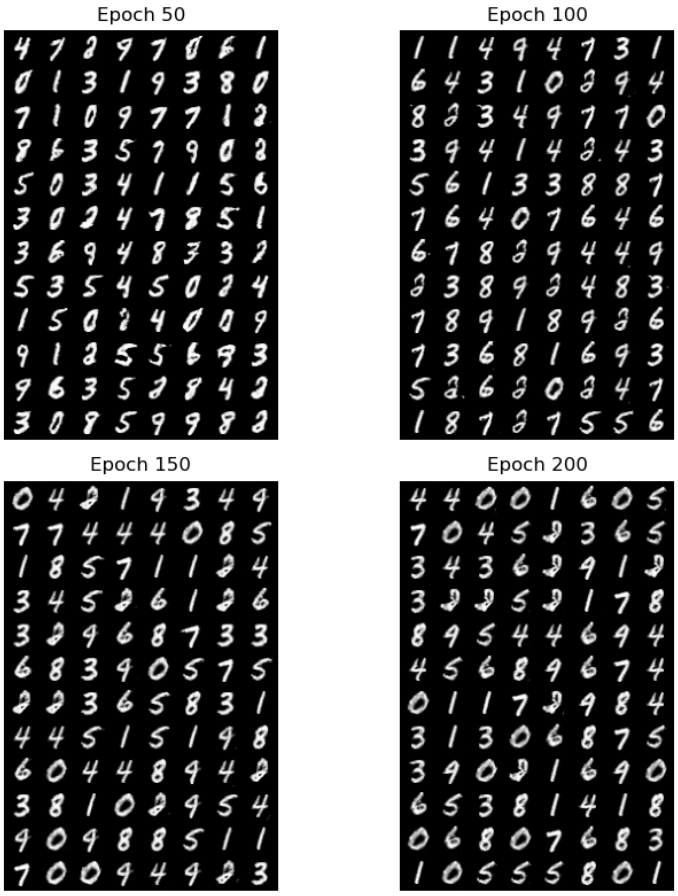
Random Seed: 88

lr = 0.0001

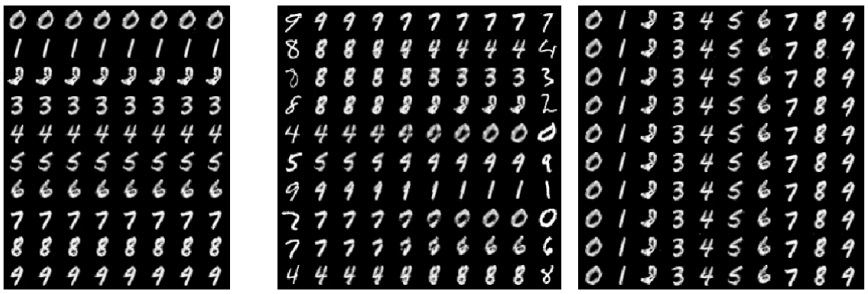
Optimizer : adam optimizer

```
Output exceeds the size limit. Open the full output data in a text editor
[0/200][0/469] Loss_D: 1.3814 Loss_C: 0.6999 Loss_G: 4462.5200
[0/200][100/469] Loss_D: 0.0014 Loss_C: 0.2063 Loss_G: 255.7935
[0/200][200/469] Loss_D: 0.0014 Loss_C: 0.1100 Loss_G: 105.2340
[0/200][300/469] Loss_D: 0.0048 Loss_C: 0.0801 Loss_G: 70.3400
[0/200][400/469] Loss_D: 0.0085 Loss_C: 0.0523 Loss_G: 53.1841
tensor([2, 8, 5, 4, 3, 4, 2, 2, 2, 1, 6, 1, 3, 3, 9, 0, 2, 4, 6, 6, 5, 9, 7, 6,
        2, 8, 1, 1, 9, 2, 0, 1, 9, 3, 7, 1, 2, 8, 1, 3, 8, 1, 1, 1, 5, 0, 5, 7,
        9, 4, 0, 5, 4, 7, 0, 2, 9, 9, 5, 1, 8, 8, 8, 0, 1, 5, 7, 0, 9, 7, 5, 6,
        5, 6, 8, 2, 9, 1, 8, 9, 5, 9, 3, 4, 9, 9, 0, 2, 2, 9, 3, 5, 3, 5, 5, 7])
[1/200][0/469] Loss_D: 0.0063 Loss_C: 0.0485 Loss_G: 44.6728
[1/200][100/469] Loss_D: 0.0063 Loss_C: 0.0356 Loss_G: 34.0157
[1/200][200/469] Loss_D: 0.0161 Loss_C: 0.0473 Loss_G: 30.8625
[1/200][300/469] Loss_D: 0.0578 Loss_C: 0.0366 Loss_G: 26.2031
[1/200][400/469] Loss_D: 0.1147 Loss_C: 0.0404 Loss_G: 21.1455
tensor([0, 2, 8, 8, 3, 3, 4, 7, 7, 9, 5, 0, 2, 2, 9, 7, 7, 8, 9, 7, 5, 3, 9, 3,
        7, 9, 2, 6, 0, 7, 2, 1, 2, 5, 6, 4, 6, 4, 0, 4, 0, 4, 5, 3, 1, 7, 7, 9,
        0, 7, 3, 4, 2, 1, 3, 1, 7, 0, 3, 7, 8, 0, 9, 6, 3, 7, 7, 6, 4, 2, 3, 7,
        1, 7, 8, 6, 5, 6, 7, 1, 1, 6, 2, 5, 1, 3, 6, 1, 9, 8, 7, 5, 8, 7, 6, 8])
[2/200][0/469] Loss_D: 0.0894 Loss_C: 0.0404 Loss_G: 18.4746
[2/200][100/469] Loss_D: 0.0717 Loss_C: 0.0317 Loss_G: 18.1008
[2/200][200/469] Loss_D: 0.1430 Loss_C: 0.0197 Loss_G: 15.2165
[2/200][300/469] Loss_D: 0.2348 Loss_C: 0.0241 Loss_G: 13.5849
[2/200][400/469] Loss_D: 0.2460 Loss_C: 0.0260 Loss_G: 12.6696
tensor([4, 5, 8, 7, 5, 8, 9, 6, 8, 2, 0, 9, 6, 7, 3, 1, 4, 7, 6, 4, 2, 2, 6, 7,
        2, 4, 6, 6, 9, 9, 1, 2, 8, 1, 2, 8, 4, 0, 8, 8, 5, 8, 9, 5, 1, 8, 7, 3,
...
tensor([4, 4, 0, 0, 1, 6, 0, 5, 7, 0, 4, 5, 2, 3, 6, 5, 3, 4, 3, 6, 2, 9, 1, 2,
        3, 2, 2, 5, 2, 1, 7, 8, 8, 9, 5, 4, 4, 6, 9, 4, 4, 5, 6, 8, 9, 6, 7, 4,
        0, 1, 1, 7, 2, 9, 8, 4, 3, 1, 3, 0, 6, 8, 7, 5, 3, 9, 0, 2, 1, 6, 9, 0,
        6, 5, 3, 8, 1, 4, 1, 8, 0, 6, 8, 0, 7, 6, 8, 3, 1, 0, 5, 5, 5, 8, 0, 1])
```





Specify the generated image:



5 Conclusion

We evaluate and compare the performance of GAN, DCGAN, VAE, and CVAE-GAN on the MNIST dataset as well as on our own collection of anime face

datasets. The results show that although GANs produce high-quality images that are visually similar to real images, they suffer from instability and pattern collapse problems during training. On the other hand, VAEs produce diverse images but with lower image quality. the DCGAN model shows better image quality and stability during training. the CVAE-GAN model combines the advantages of both models to produce high-quality images with controllable properties.

From the experimental results, we found that the DCGAN model outperforms the traditional GAN model in terms of image quality and training stability. The generated anime face images show high diversity and quality, which demonstrates the effectiveness of the DCGAN architecture in learning complex image features.

In addition, we observe that the VAE model is able to generate diverse images but with lower image quality compared to the GAN and DCGAN models. This is because the VAE model is optimized for reconstruction loss rather than for image generation. Nevertheless, the VAE model provides a useful tool to explore the latent space and generate new images from the learned distributions.

The CVAE-GAN model shows impressive results in generating high-quality images with controllable properties. By using the prior distribution learned by CVAE, we are able to control the image properties during image generation by GAN. This allows us to generate different styles of figures on the MNIST dataset and create smooth transitions between two different figures.

To achieve these results, we faced several challenges in the training process. A major issue is the instability of the GAN training process, which causes the model to produce low-quality images or collapse into a single pattern. To address this issue, we tried different modifications to the GAN architecture, such as DCGAN, and found that these modifications improved the stability of the training process.

Another challenge is to choose the appropriate hyperparameters for each model, such as learning rate and batch size. We performed a grid search to find the best values of these hyperparameters, which allowed us to achieve the best results for each model.

In addition, our anime face dataset is relatively small and contains a wide range of styles, which makes it difficult for us to achieve good image quality and diversity. We addressed this issue by pre-processing the images and using data enhancement techniques to increase the size of the dataset.

In summary, this project provides insights into the strengths and limitations of GAN, VAE and their hybrid models. We successfully applied these models to generate high-quality images on the MNIST dataset and our own anime face dataset. our experimental results demonstrate the potential of deep learning models in generating high-quality and diverse images. We also identified and overcame several challenges, including training instability and hyperparameter selection. Our results demonstrate the potential of these models in generating high quality and diverse images with controlled attributes.

References

[1] Goodfellow, Ian, et al. "Generative adversarial networks." *Communications of the ACM* 63.11 (2020): 139-144.

[2] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." *International conference on machine learning*. PMLR, 2017.

[3] Miyato, Takeru, et al. "Spectral normalization for generative adversarial networks." *arXiv preprint arXiv:1802.05957* (2018).

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

[5] Higgins, Irina, et al. "beta-vae: Learning basic visual concepts with a constrained variational framework." *International conference on learning representations*. 2017.

[6] Larsen, Anders Boesen Lindbo, et al. "Autoencoding beyond pixels using a learned similarity metric." *International conference on machine learning*. PMLR, 2016.

[7] Wu, Jiajun, et al. "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling." *Advances in neural information processing systems* 29 (2016).

[8] Makhzani, Alireza, et al. "Adversarial autoencoders." *arXiv preprint arXiv:1511.05644* (2015).

[9] Mescheder, Lars, Sebastian Nowozin, and Andreas Geiger. "Adversarial variational bayes: Unifying variational autoencoders and generative adversarial networks." *International conference on machine learning*. PMLR, 2017.

[10] Bao, Jianmin, et al. "CVAE-GAN: fine-grained image generation through asymmetric training." *Proceedings of the IEEE international conference on computer vision*. 2017.