

# Synthesising Images by Imagination

CHEN, Ta-Yu

This dissertation was submitted in part fulfilment of requirements for the degree of  
MSc Advanced Software Engineering

DEPT. OF COMPUTER AND INFORMATION SCIENCES  
UNIVERSITY OF STRATHCLYDE

August 2019

## Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [☒] No [☐]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is

I confirm that I wish this to be assessed as a Type 1 2 3 4 ☒ Dissertation (please circle)

Signature:

Handwritten signature in Chinese characters, reading '陳大星' (Chen Daxing).

Date: 19/08/2019

## Abstract

In this dissertation, four GANs models (vanilla GANs, DCGAN, WGAN, and WGAN-GP) were applied to three different training datasets including MNIST, NLVR, and Oxford-102 flowers. The models were successfully implemented through Keras and the code of the project can be efficiently executed by Python scripts. The limitations and the evolution of vanilla GANs were explored in the experiments of the dissertation. We can find that firstly DCGAN alleviated the non-convergence problem and produced good quality images on MNIST and Oxford-102 flowers; next, WGAN mitigated the mode collapse problem but failed on Oxford-102 flowers, using unsuitable method to restrict its discriminator. Lastly, WGAN-GP overcame all limitations and synthesised compelling images. In the end, DCGAN produced the best results of Oxford-120 flowers among all models and yielded Fréchet Inception Distance (FID) of 79 and a human error rate of 27.78%. On the other hand, WGAN-GP synthesised the best quality images on MNIST and NLVR. MNIST got FID of 6 and an error rate of 72.22%. NLVR obtained FID of 94 and an error rate of 19.44%.

## **Acknowledgements**

I would first like to thank my dissertation supervisor Dr. Dmitri Roussinov for his constructive guidance and professional suggestions helped me finish this work. Finally, I want to thank my parents who supported me during my studies. Thank you.

# Contents

List of Figures .....	viii
List of Tables .....	xi
Chapter 1: Introduction .....	1
1.1    Background of the project.....	1
1.2    Research Objectives.....	4
Chapter 2: Literature Review .....	5
2.1    Background of Generative Models .....	5
2.1.1    Pixel Recurrent Neural Networks (PixelRNN) .....	5
2.1.2    Variational Autoencoder (VAE) .....	6
2.1.3    Generative Adversarial Networks (GANs) .....	8
2.2    Evolution of GANs .....	12
2.2.1    The Main Problems of Vanilla GANs.....	12
2.2.2    Structure Improvement: Deep Convolutional GAN (DCGAN).....	14
2.2.3    New Estimating Distance: Wasserstein GAN (WGAN).....	14
2.3    Significant Components of deep neural networks .....	18
2.3.1    Introduction of Common Activation function in Neural Networks .....	18
2.3.2    Introduction of Common Optimizers in Neural Networks .....	20
2.4    Evaluation of Synthetic Images .....	22
Chapter 3: Research Method.....	24
3.1    Details of GANs models .....	24
3.1.1    Original GANs .....	24
3.1.2    DCGAN .....	25
3.1.3    WGAN .....	26
3.1.4    WGAN-GP.....	27
3.2    Experimental dataset .....	28
3.2.1    MNIST .....	28
3.2.2    NLVR Dataset.....	28
3.2.3    Oxford-102 Flowers .....	29
3.3    The Design of Experiments.....	30

3.3.1	Experiment 1: Generate handwritten digits.....	30
3.3.2	Experiment 2: Generate Geometric Shapes .....	31
3.3.3	Experiment 3: Generate Flowers.....	31
3.4	Implement GANs' models by Keras .....	31
3.4.1	Important components of Keras .....	32
3.4.2	Build a stacked model by Keras.....	33
3.4.3	Train GANs model by Keras .....	35
3.4.4	Build DCGAN's model by Keras .....	36
3.4.5	Instructions of Running Code .....	38
3.5	Evaluation Methods .....	39
3.5.1	Evaluating Synthesised Images by Human Judgment.....	39
3.5.2	Evaluating Synthesised Images by Fréchet Inception Distance (FID).....	39
Chapter 4: The Analyse of the Experiments .....		40
4.1	The 1 <sup>st</sup> Experiment: Generate Handwritten Digits.....	40
4.1.1	Vanilla GANs on MNIST .....	40
4.1.2	DCGAN on MNIST .....	42
4.1.3	WGAN on MNIST.....	43
4.1.4	WGAN-GP on MNIST .....	45
4.1.5	Model Comparisons on MNIST.....	46
4.2	The 2 <sup>nd</sup> Experiment: Generate Geometric Graphics .....	49
4.2.1	Vanilla GANs on NLVR.....	49
4.2.2	DCGAN on NLVR.....	51
4.2.3	WGAN on NLVR .....	53
4.2.4	WGAN-GP on NLVR.....	54
4.2.5	Model Comparisons on NLVR .....	56
4.3	The 3 <sup>rd</sup> Experiment: Generate Flower Images .....	59
4.3.1	Vanilla GANs on Oxford-102 Flowers.....	59
4.3.2	DCGAN on Oxford-102 Flowers.....	60
4.3.3	WGAN on Oxford-102 Flowers .....	61

4.3.4	WGAN-GP on Oxford-102 Flowers .....	62
4.3.5	Model Comparisons on Oxford-102 Flowers .....	63
Chapter 5: Conclusions and Recommendations.....		66
5.1	Conclusions.....	66
5.1.1	Problems of Vanilla GANs .....	66
5.1.2	The improvement in GANs training stability.....	66
5.1.3	The improvement in Estimating Method .....	66
5.2	Recommendations.....	67
Bibliography .....		68
Appendix A.....		72

## List of Figures

Figure 1-1. Progressively-growing GANs' concept and its compelling results (Karras et al., 2017).....	2
Figure 1-2. The result of CycleGAN presented by Zhu et al. (2017) .....	3
Figure 1-3. The functionalities of FaceApp and the images are created by FaceApp .....	4
Figure 2-1. A brief structure of AE.....	6
Figure 2-2. The comparison between AE and VAE structure which was reproduced from Lee (2017) .....	7
Figure 2-3. The concept of GANs. The handwritten digits were produced by the project's model .....	8
Figure 2-4. The difference between $\log(1 - DG(z))$ and $-\log(DG(z))$ when $DG(z)$ changes.....	11
Figure 2-5. A value function $V_{x,y} = x^2 - y^2$ with a saddle point ( $x = 0, y = 0$ ) .....	12
Figure 2-6. The demonstration of the model collapse. It was produced by the project's models.....	13
Figure 2-7. Two simple discrete distributions with five possible states .....	15
Figure 2-8. A transport plan of transforming $\mathbb{P}_g$ to $\mathbb{P}_r$ .....	15
Figure 2-9. The illustration of $Ppenalty$ . The figure was reproduced from (Lee, 2018) .....	18
Figure 2-10. ReLU function.....	19
Figure 2-11. Leaky ReLU function.....	19
Figure 3-1. The structure of the original GAN which is used in the project.....	24
Figure 3-2. The overall architecture of DCGAN which was reproduced from Radford et al. (2015) .....	26
Figure 3-3. Handwritten digits which were sampled from MNIST (LeCun et al., 1998).....	28
Figure 3-4. Geometric images which were sampled from NLVR dataset (Suhr et al., 2017) .....	29
Figure 3-5. An original image from NLVR was divided into three parts .....	29
Figure 3-6. The images which were sampled from the final training dataset .....	29
Figure 3-7. Images which were sampled from Oxford-102 flowers (Nilsback and Zisserman, 2008).....	30
Figure 3-8. The training process of an original GANs model from the code of the project .....	35
Figure 3-9. A DCGAN's generator built by Keras and it is captured from the project's code.....	36
Figure 3-10. A DCGAN's discriminator built by Keras and it was captured from the project's code .....	37
Figure 3-11. The instruction of the training module of the project.....	38
Figure 3-12. The instruction of the evaluating module of the project .....	38
Figure 3-13. A question of the project's questionnaire.....	39
Figure 4-1. The structures of GAN's model for MNIST .....	40
Figure 4-2. The losses of vanilla GAN trained on MNIST.....	41
Figure 4-3. Handwritten digits generated by vanilla GANs of the project .....	41
Figure 4-4. The structures of DCGAN's model for MNIST.....	42
Figure 4-5. The losses of DCGAN trained on MNIST .....	42



Figure 4-6. Handwritten digits generated by DCGAN of the project .....	43
Figure 4-7. The structures of WGAN's model for MNIST .....	43
Figure 4-8. The losses of WGAN trained on MNIST .....	44
Figure 4-9. Handwritten digits generated by WGAN of the project .....	44
Figure 4-10. The losses of WGAN-GP trained on MNIST .....	45
Figure 4-11. Handwritten digits generated by WGAN-GP of the project .....	45
Figure 4-12. The JS divergences of vanilla GANs and DCGAN in MNIST .....	46
Figure 4-13. The image similarity comparison of GANs in MNIST .....	47
Figure 4-14. The image similarity comparison of DCGAN in MNIST .....	47
Figure 4-15. The Wasserstein distances of WGAN and WGAN-GP in MNIST .....	48
Figure 4-16. The image similarity comparison of WGAN in MNIST .....	48
Figure 4-17. The image similarity comparison of WGAN-GP in MNIST .....	48
Figure 4-18. The structures of vanilla GANs' model for colorful images with 64×64 pixels .....	49
Figure 4-19. The losses of vanilla GANs trained on NLVR .....	50
Figure 4-20. Geometric graphics generated by vanilla GANs of the project .....	50
Figure 4-21. The structures of DCGAN model for colorful images with 64×64 pixels .....	51
Figure 4-22. The losses of DCGAN trained on NLVR .....	52
Figure 4-23. Geometric graphics generated by DCGAN of the project .....	52
Figure 4-24. The structures of WGAN model for colorful images with 64×64 pixels .....	53
Figure 4-25. The losses of WGAN trained on NLVR .....	53
Figure 4-26. Geometric graphics generated by WGAN of the project .....	54
Figure 4-27. The structures of WGAN-GP model for colorful images with 64×64 pixels .....	54
Figure 4-28. The losses of WGAN-GP trained on NLVR .....	55
Figure 4-29. Geometric graphics generated by WGAN-GP of the project .....	55
Figure 4-30. The JS divergences of the vanilla GANs and DCGAN in NLVR .....	56
Figure 4-31. The image similarity comparison of DCGAN in NLVR .....	57
Figure 4-32. The Wasserstein distances of WGAN and WGAN-GP in NLVR .....	57
Figure 4-33. The image similarity comparison of WGAN in NLVR .....	58
Figure 4-34. The image similarity comparison of WGAN-GP in NLVR .....	58
Figure 4-35. The losses of vanilla GANs trained on Oxford-102 flowers .....	59
Figure 4-36. Flower images generated by vanilla GANs of the project .....	59
Figure 4-37. The losses of DCGAN trained on Oxford-102 flowers .....	60
Figure 4-38. Flower images generated by DCGAN of the project .....	60
Figure 4-39. The losses of WGAN trained on Oxford-102 flowers .....	61

Figure 4-40. Flower images generated by WGAN of the project .....	61
Figure 4-41. The losses of WGAN-GP trained on Oxford-102 flowers .....	62
Figure 4-42. Flower images generated by WGANG-GP of the project.....	62
Figure 4-43. The JS divergences of vanilla GANs and DCGAN in Oxford-102 flowers.....	63
Figure 4-44. The image similarity comparison of DCGAN in Oxford-102 flowers.....	64
Figure 4-45. The Wasserstein distances of WGAN and WGAN-GP in Oxford-102 flowers.....	64
Figure 4-46. The image similarity comparison of WGAN-GP in NLVR.....	65
Figure A-1. Instruction of the questionnaire .....	72
Figure A-2. Inviting participants to join the questionnaire .....	73
Figure A-3. The questions for choosing synthesised digit images of GANs and DCGAN .....	74
Figure A-4. The questions for choosing synthesised digit images of WGAN and WGAN-GP .....	75
Figure A-5. The questions for choosing synthesised geometric graphics of GANs and DCGAN .....	76
Figure A-6. The questions for choosing generated geometric graphics of WGAN and WGAN-GP .....	77
Figure A-7. The questions for choosing generated flower images of GANs and DCGAN .....	78
Figure A-8. The questions for choosing generated flower images of WGAN and WGAN-GP .....	79

## List of Tables

Table 4-1. FID of vanilla GANs and DCGAN which were trained on MNIST .....	46
Table 4-2. FID of WGAN and WGAN-GP which were trained on MNIST .....	48
Table 4-3. FID of vanilla GANs and DCGAN which were trained on NLVR .....	57
Table 4-4. FID of WGAN and WGAN-GP which were trained on NLVR .....	58
Table 4-5. FID of vanilla GANs and DCGAN which were trained on Oxfor-102 flowers .....	63
Table 4-6. FID of WGAN and WGAN-GP which were trained on Oxfor-102 flowers .....	65

# Chapter 1: Introduction

## 1.1 Background of the project

Machine learning becomes a buzzword and its influence in many domains has grown rapidly in recent years. Practical applications of machine learning include face recognition and speech recognition (Lison, 2015). In general, high-quality features considerably affect the performance of machine learning models (Bantum et al., 2017). Nevertheless, extracting useful features from raw data needs a lot of domain knowledge. The difficulty of extracting features especially occurs in the domain where we are hard to describe their features by ourselves (LeCun et al., 2015). For instance, it is difficult to decide what kind of features could be used for categorizing the emotion in a speech.

The progress of deep learning overcomes the difficulty and forgoes extracting features from high-dimensional data. Deep learning is based on artificial neural networks (ANN) (Jain et al., 1996). Unlike ANN which has few hidden layers, the architectures of deep learning have multiple hidden layers. The well-known architectures of deep learning include convolution neural networks (CNNs) (LeCun and Bengio, 1995), recurrent neural networks (RNNs) (Rumelhart et al., 1988) and generative adversarial networks (GANs) (Goodfellow et al., 2014). Deep learning learns the representation of sophisticated data by computing the weights and the biases of deep learning models. Through the training of a deep learning model, it can obtain appropriate internal parameters for its task such as speech recognition. Deep learning makes huge progress in traditional applications of machine learning such as image classifications. Hu et al. (2018) proposed SENets, a the-state-of-the-art image classification model, which won the championship of the 2017 ILSVRC.

Recently, researchers have shown an increased interest in generative models with deep learning. A generative model is a model which takes a training dataset and learns the representation of the data distribution (Goodfellow et al., 2014). The generative model can generate more new data which conform with the distribution which learned from the training dataset. For instance, a generative model is fed with different kinds of flowers images and learns the representation of these images. After the model was trained, it can generate new flower images which may not exist in the training dataset but conformed with the distribution of the training dataset.

Famous generative models include Pixel Recurrent Neural Networks (PixelRNN) (Oord et al., 2016), Variational Autoencoders (VAE) (Kingma and Welling, 2013, Rezende et al., 2014, Kingma et al., 2016) and Generative adversarial networks (GANs) (Goodfellow et al., 2014). The project focuses on GANs, which is the most well-known breakthroughs in the domain of generative models recently. A discriminator and a generator which are neural networks constitute GANs. The functionality of the generator is generating

artificial data. And then, the discriminator takes the synthesised data and distinguish the real data from fake data. The objective of the generator is misleading the discriminator to do the wrong judgment. In other words, the purpose of the discriminator is increasing the accuracy of distinguishing fake data. Through the competition mechanism, GANs can learn the distribution of the training dataset. In respect of generating images, GANs can generate sharper images and produce them more quickly than VAE and PixelRNN respectively.

Along with the progress of GANs, it can be applied on different kinds of applications. There are two common applications demonstrated below. The first and the most common application is the image synthesising. Radford et al. (2015) proposed DCGAN which successfully introduced convolution neural networks into GANs. It provided a group of stable training architectures which can amplify the scale of GANs models in terms of the synthesising images. Because of the success of DCGAN, many GANs models applied the architecture of DCGAN on their models (Goodfellow, 2016). Progressively-growing GANs Karras et al. (2017), proposed in 2017, can synthesis images with  $1024 \times 1024$  pixels. It demonstrated the capability of generating compelling pictures with high-resolution. Its strategy is using the architecture which is composed of multiple GANs with different scales. In the beginning of the model, the generator generates low-resolution images with  $4 \times 4$  pixels. After the model converges, the scale of the model is increased to produce  $8 \times 8$  pixels images. Through the same process mentioned above, the resolution of generated images can be scaled up to  $1024 \times 1024$  pixels progressively. Figure 1-1 presented the concept of progressively-growing GANs and its compelling results.

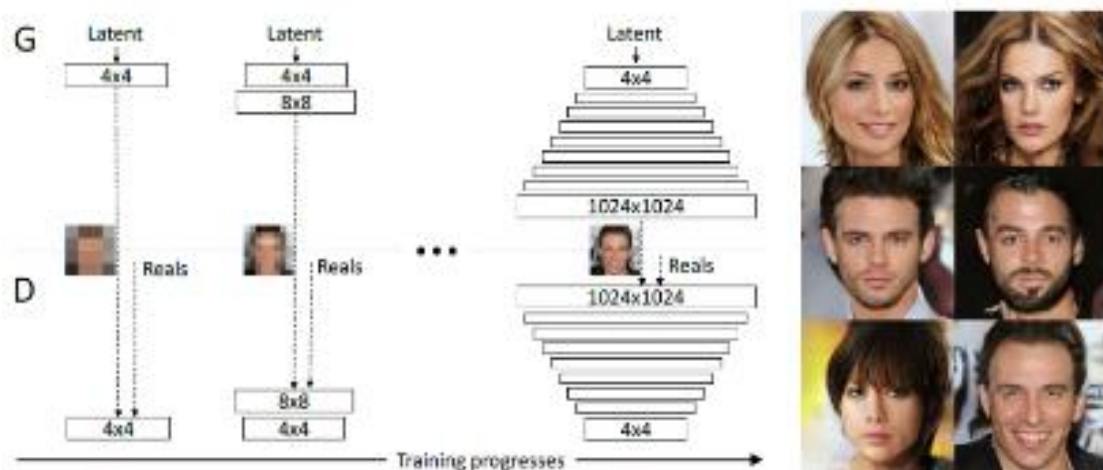


Figure 1-1. Progressively-growing GANs' concept and its compelling results (Karras et al., 2017)

The second application is the image-to-image translation. The application is translating input images into output images. The purpose of the task is to find the mapping relationship between input and output. Zhu et al. (2017) proposed CycleGAN, which is a successful method to learn the mapping between two image distributions. The architecture of CycleGAN is dual. An image *Input\_A* from domain  $D_A$  is fed to the first generator  $G_{A \rightarrow B}$  whose job is to transfer images from the domain  $D_A$  to the domain  $D_B$ . The output of  $G_{A \rightarrow B}$  is *Generated\_B* and it is fed into the second generator  $G_{B \rightarrow A}$  and the discriminator of  $D_B$  respectively. The output of  $G_{B \rightarrow A}$  is *Cycle\_A* and its constraint is as close to *Input\_A* as possible. On the other hand, an image *Input\_B* from the second domain  $D_B$  is fed into generator  $G_{B \rightarrow A}$  and is transferred to the domain  $D_B$  to get and image *Generated\_B*. The remaining steps are as same as above. The strength of CycleGAN is that it can be trained on unpair datasets. For instance, if you want to transfer horses to zebras, you don't need to prepare a bunch of horse images with corresponding zebra images. The image of horses and zebras can be collected independently.

Figure 1-2 demonstrated the fascinating result of CycleGAN.

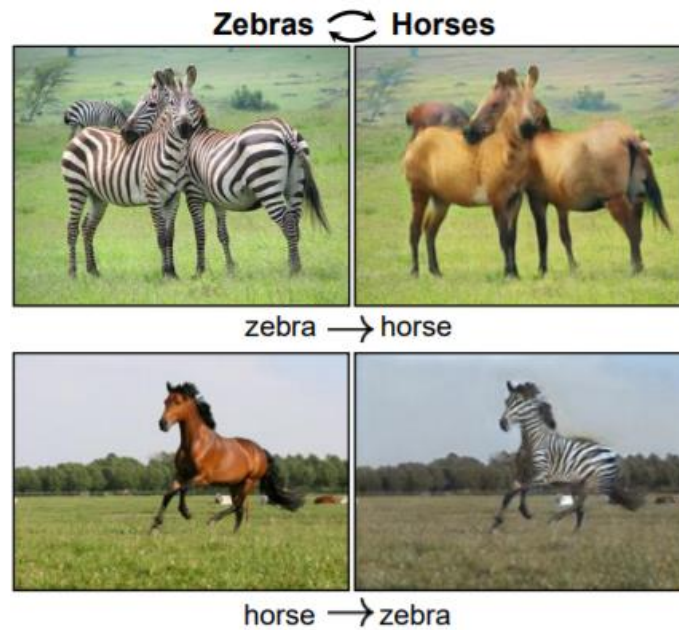


Figure 1-2. The result of CycleGAN presented by Zhu et al. (2017)

In the current market, a mobile application called FaceApp is a good example for applying the techniques of GANs in practice. This app is developed by Russian company Wireless Lab and uses machine learning to synthesising high-quality photographs in terms of face transformations (Vincent, 2017). Its functions include changing hair color, swapping genders, changing users' age and adding a smile on your face. The result is compelling and highly competitive in the market. Figure 1-3 illustrated the capability of FaceApp.



Figure 1-3. The functionalities of FaceApp and the images are created by FaceApp

## 1.2 Research Objectives

Although the rise of GANs brought a new perspective to generative models, it has two major problems. One is that the original GANs is hard to be trained. If the architecture of the model is not good enough, GANs may not converge. The other is the mode collapse problem. It means that models only generate a few types of data. To solve the problems, many advanced GANs-based models were proposed. The dissertation focused on utilizing these advanced GANs models to synthesis images and comparing their performance with the original GANs. The objectives of the dissertation are demonstrated below:

1. Implementing proposed GANs models including the original GAN, DCGAN, WGAN, and WGAN-GP
2. Implementing an automatic model training procedure which includes loading training datasets, training generative models, recoding training losses, generating synthesising images and evaluating models' performance.
3. Using vanilla GANs as a baseline and trained all GANs models on three datasets including MNIST, NLVR, and Oxford-102 flowers.
4. Executing the performance comparison amongst models included the quality of synthesising images and the convergence of the models

The dissertation was divided into five chapters. The 1st chapter as presented above. Next, the 2nd chapter presented the relevant literature. The literature review presented three common generative models including PixelRNN, VAE, and GANs, and the detail of GANs was emphasized. Then, the 3rd chapter demonstrated the implementation of the generative models and the designs of the experiments. After that, the results of the experiments were analyzed in the 4th chapter. Finally, the conclusions and the recommendations were presented in the 5th chapter.

## Chapter 2: Literature Review

The chapter reviewed three common generative models including PixelRNN, VAE and GANs. Moreover, the details about the evolution and the evaluation method of GANs were demonstrated.

### 2.1 Background of Generative Models

This section demonstrated three common generative models which used the deep learning technique to construct their structures and trained their models. All of the models adopted different perspectives to learn the data distribution which they wanted to catch.

#### 2.1.1 Pixel Recurrent Neural Networks (PixelRNN)

PixelRNN is an explicit density generative model and its likelihood can be computed tractably (Oord et al., 2016). It is based on fully visible belief networks (FVBNs) which can transfer an  $n$ -dimensional probability distribution  $x$  into a product of 1-dimensional probability distributions based on the chain rule of probability (Frey et al., 1996). The task of PixelRNN is generative colorful images with  $n^2$  pixels and the probability of each image  $p(x)$  can be shown as equation (2-1). The distribution of each pixel  $p(x_i|x_1, \dots, x_{i-1})$  is composed of the distribution of previous pixels with three channels including Red (R), Green (G), Blue (B). Hence, the distribution of the  $i$ -th pixel can be rewritten as equation (2-2).

$$p(x) = \prod_{i=1}^{n^2} p(x_i|x_1, \dots, x_{i-1}) \quad (2-1)$$

$$p(x_i|x_{<i}) = p(x_i, R|x_{<i}) p(x_i, G|x_{<i}, x_i, R) p(x_i, B|x_{<i}, x_i, R, x_i, G) \quad (2-2)$$

PixelRNN has twelve two-dimensional Long Short-Term Memory (LSTM) layers. LSTM is a variation of Recurrent Neural Network (RNN) to overcome the gradient vanishing problem (Hochreiter and Schmidhuber, 1997). The layers of LSTM be composed of memory cells which be connected recurrently. Memory cells consist of three gate units including input gate, forget gate and output gate and the mechanism of LSTM makes models have the capability for capturing long-range context over time. To obtain the interdependence between pixels in mages, PixelRNN adopts spatial LSTM (Graves and Schmidhuber, 2009) with two dimensions.

PixelRNN provides two types of LSTM layers including Row LSTM and Diagonal BiLSTM. In Row LSTM, the probability of the  $i$ -th pixel is composed of the input-to-state component and the recurrent state-to-state component with one-dimensional convolution which is size is  $k \times 1$  ( $k \geq 3$ ). The Row LSTM can compute features row by row. However, it only refers to partial previous pixels instead of the whole



previous pixels. In Diagonal BiLSTM, the probability of  $i$ -th pixel is composed of the input-to-state component with  $1 \times 1$  convolution kernel and the state-to-state component with  $2 \times 1$  convolution kernel. Diagonal BiLSTM can compute features along the entire diagonal of an image at once and captures the information of whole previous states for the  $i$ -th pixel. Hence, the training speed of the Diagonal BiLSTM model is faster than the model based on Row LSTM. Oord et al. (2016) also propose a simplified architecture, PixelCNN which is based on Convolutional Neural Networks (CNN) and shares the same core components in the PixelRNN. PixelCNN has fifteen fully convolution layers without any pooling layers. The advantage of PixelCNN is computing the whole features of an image at once and has the fastest training speed among the models mentioned above. Nevertheless, PixelCNN only refers to the neighbor input-to-state component. The main drawbacks of both PixelRNN and PixelCNN are a new image must be generated pixel by pixel and it cost a lot of time.

### 2.1.2 Variational Autoencoder (VAE)

Before illustrating the idea of variational autoencoders (VAE), the concept of autoencoders needs to be explained first. Traditionally, an autoencoder (AE) consists of an encoder and a decoder and they are neural networks. An encoder takes a high-dimensional vector  $X$  as input and produces a lower-dimensional vector *Code* as output after going through a neural network encoder. *Code* is fed into a decoder which can output a vector  $\hat{X}$  which its dimension is as the same as the encoder's input. A brief structure is shown in Figure 2-1. There are some constraints such as the dimension of *Code* is smaller than input data  $X$ . A undercomplete autoencoder is an example of this type of constraint. This constraint makes autoencoders extract significant components of input data rather than just simply copy the input. The undercomplete autoencoders use linear activation functions and apply mean-square error (MSE) between  $X$  and  $\hat{X}$  as a reconstruction loss which needs to be minimized. Traditionally, AE is used to reduce the data's dimension and makes a training process for a machine learning model more efficiently. The technique of AE has been developed since the 1980s (Bourlard and Kamp, 1988, Hinton and Zemel, 1994) and it has been expanded with deep neural networks (Hinton and Salakhutdinov, 2006).

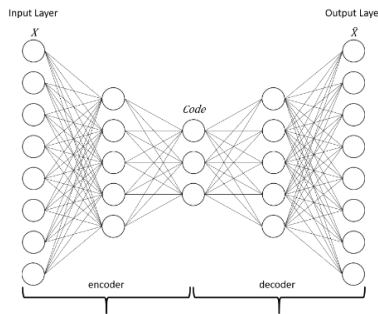


Figure 2-1. A brief structure of AE

Variational autoencoders (VAE) are the variations of autoencoders and famous as a generative model. Figure 2-2 demonstrates the difference between AE and VAE in terms of structures. VAE introduces three new terms including a mean code  $\mu$ , a standard deviation  $\sigma$  and a noise being sampled from a normal distribution to concrete the code layer. Each component of the code can be demonstrated as equation (2-3). Besides minimizing a reconstruction loss, VAE needs to minimize a latent loss. According to the derivation in the paper of Kingma and Welling (2013), the latent loss function can be demonstrated as shown in equation (2-4) when noise is drawn from a normal distribution. Both decoders of AE and VAE can generate new instances but VAE can create more various instances because of being a probabilistic model.

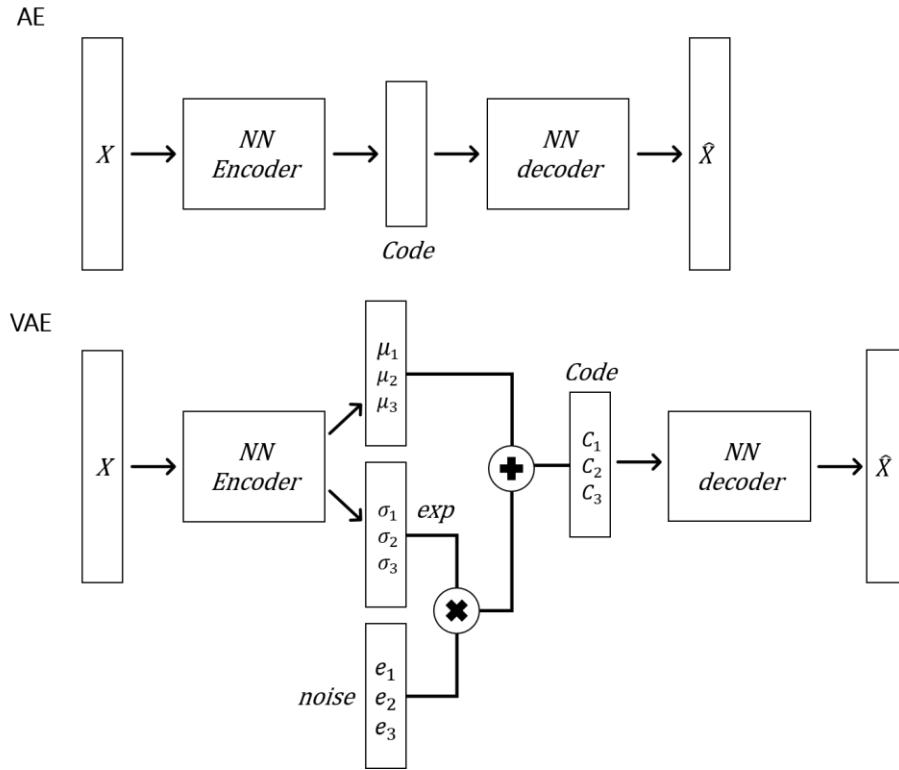


Figure 2-2. The comparison between AE and VAE structure which was reproduced from Lee (2017)

$$c_i = \exp(\sigma_i) \times e_i + \mu_i \quad (2-3)$$

$$\sum_{i=1}^3 (\exp(\sigma_i) - (1 + \sigma_i) + (\mu_i)^2) \quad (2-4)$$

### 2.1.3 Generative Adversarial Networks (GANs)

GANs is a deep learning generative framework proposed by Goodfellow et al. (2014). GANs consists of two multilayer perceptrons including a generator and a discriminator. The goal of the generator is capturing the distribution of the training data and the task of the discriminator is telling real data which is from training dataset from fake data which is generated by a generator. Figure 2-3 demonstrates the concept of GANs. The generator is fed with a noise vector which is sampled from a normal distribution and generates a fake image. Meanwhile, a real image is sampled from the training dataset and the discriminator tell which one is real by assigning real images to 1 and fake images to 0. In the training process, these two models compete iteratively until the generator fully learns the representative of training data and the discriminator gets 50% accuracy on telling fake data. The above description is a simple way to illustrate the idea of GANs. A further illustration is demonstrated below.

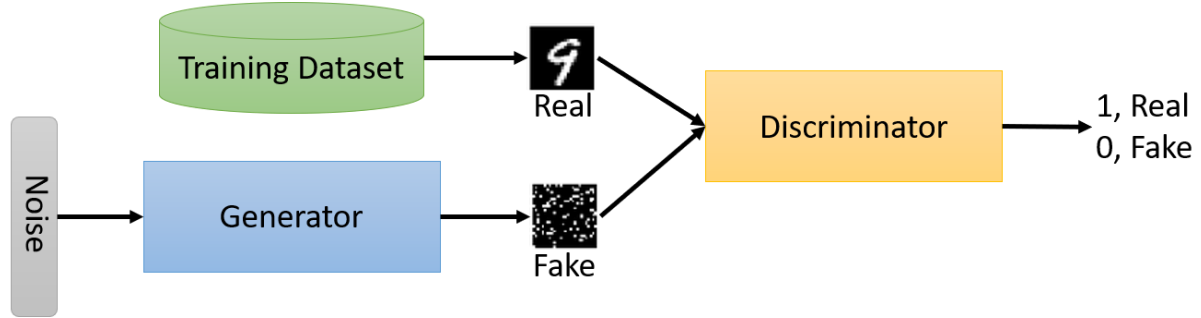


Figure 2-3. The concept of GANs. The handwritten digits were produced by the project's model

The distribution of the generator is  $P_g$  which learns from data  $x$  with the input noise vector  $z$  drawn from noise prior distribution  $p_z(z)$ . Through the generator  $G(z; \theta_g)$  with parameters  $\theta_g$ , an input noise vector  $z$  can be mapped to data space. Further, the discriminator  $D(x; \theta_d)$  with parameter  $\theta_d$  produces a single scalar to show the probability of a data  $x$  from training dataset  $p_{data}$ .  $D(x)$  is trained to have the best capability which can assign 1 to the data from  $p_{data}$  and 0 to the data from  $P_g$ . In the meanwhile,  $G(z)$  is trained to minimize the value of  $\log(1 - D(G(z)))$ . It means that the purpose of  $G(z)$  is maximizing the probability of  $D(x)$  been cheated. Obviously, GANs is not a simple optimization problem with a single objective. Instead, it is a minimax optimization problem with a value function  $V(D, G)$  and can be mathematically demonstrated as equation (2-5).

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \quad (2-5)$$

To solve this minimax optimization problem,  $G$  is fixed and we focus on  $\max_D V(D, G)$  firstly. The following derivation demonstrates what is the global optimum for the discriminator  $G$ .

$$\begin{aligned} V(D, G) &= \mathbb{E}_{x \sim P_{data}(x)} [\log D(x)] + \mathbb{E}_{x \sim P_g(x)} [\log(1 - D(x))] \\ &= \int_x P_{data}(x) \log D(x) dx + \int_x P_g(x) \log(1 - D(x)) dx \\ &= \int_x [P_{data}(x) \log D(x) + P_g(x) \log(1 - D(x))] dx \end{aligned}$$

To get maximal  $V(D, G)$ , the optimal  $D^*$  needs to maximize  $P_{data}(x) \log D(x) + P_g(x) \log(1 - D(x))$  with given  $x$ . To make the following derivation more readable,  $P_{data}(x)$ ,  $P_g(x)$  and  $D(x)$  are replaced as  $a$ ,  $b$  and  $D$  respectively. Hence, we can get an equation  $f(D) = a \log(D) + b \log(1 - D)$ . To get the optimal  $D^*$  which makes  $f(D)$  is maximal, we can calculate the derivative of  $f(D)$  with respect to  $D$  and find  $D^*$  which makes the derivative is zero.

$$\begin{aligned} \frac{df(D)}{dD} = 0 &\Leftrightarrow a \times \frac{1}{D} + b \times \frac{1}{1-D} \times (-1) = 0 \\ &\Leftrightarrow a \times \frac{1}{D^*} = b \times \frac{1}{1-D^*} \\ &\Leftrightarrow a(1 - D^*) = bD^* \\ &\Leftrightarrow D^* = \frac{a}{a+b} \\ &\Leftrightarrow D^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_g(x)} \end{aligned}$$

From above derivation, we can get **Proposition 1**.

**Proposition 1.** When  $G$  is fixed, the optimal  $D$  is

$$D^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_g(x)} \quad (2-6)$$

Therefore,  $\max_D V(D, G)$  is equal to  $V(D^*, G)$  and  $V(D^*, G)$  can be rewritten as the following:

$$\begin{aligned} V(D^*, G) &= \mathbb{E}_{x \sim P_{data}(x)} \left[ \log \frac{P_{data}(x)}{P_{data}(x) + P_g(x)} \right] + \mathbb{E}_{x \sim P_g(x)} \left[ \log \frac{P_g(x)}{P_{data}(x) + P_g(x)} \right] \\ &= \int_x P_{data}(x) \log \frac{P_{data}(x)}{P_{data}(x) + P_g(x)} dx + \int_x P_g(x) \log \frac{P_g(x)}{P_{data}(x) + P_g(x)} dx \\ &= \int_x P_{data}(x) \log \frac{P_{data}(x) \times \frac{1}{2}}{(P_{data}(x) + P_g(x)) \times \frac{1}{2}} dx + \int_x P_g(x) \log \frac{P_g(x) \times \frac{1}{2}}{(P_{data}(x) + P_g(x)) \times \frac{1}{2}} dx \end{aligned}$$

$$= -2\log 2 + \int_{\mathbf{x}} P_{\text{data}}(\mathbf{x}) \log \frac{P_{\text{data}}(\mathbf{x})}{(P_{\text{data}}(\mathbf{x}) + P_g(\mathbf{x})) \times \frac{1}{2}} d\mathbf{x} + \int_{\mathbf{x}} P_g(\mathbf{x}) \log \frac{P_g(\mathbf{x})}{(P_{\text{data}}(\mathbf{x}) + P_g(\mathbf{x})) \frac{1}{2}} d\mathbf{x}$$

We can observe that the above second and third term are Kullback–Leibler divergence (KL) respectively.

$$\begin{aligned} V(D^*, G) &= -2\log 2 + KL\left(P_{\text{data}} \left\| \frac{P_{\text{data}} + P_g}{2} \right\| \right) + KL\left(P_g \left\| \frac{P_{\text{data}} + P_g}{2} \right\| \right) \\ &= -2\log 2 + 2JSD(P_{\text{data}} \| P_g) \end{aligned}$$

The above second term is Jensen-Shannon divergence (JSD) between two distribution  $P_{\text{data}}$  and  $P_g$ . After finding the optimal discriminator  $D^*$ , we can obtain the optimal  $G^*$  by minimizing  $V(D^*, G)$ . Because Jensen-Shannon divergence is non-negative, and it is zero when two distributions are equal, the global minimum  $V(D^*, G)$  occurs when  $P_{\text{data}}$  is equal to  $P_g$ .

The high-level overview of the training process is shown below:

1. Initialize the discriminator  $D_0$  and the generator  $G_0$  with parameters  $\theta_d^0$  and  $\theta_g^0$  respectively.
2. Train GAN with  $n$  times iteration and do the below step within each iteration:
  - Block 1: repeat  $k$  times
    - i. Sampling  $m$  examples  $x^1, x^2, \dots, x^m$  from the data distribution  $P_{\text{data}}(x)$
    - ii. Sampling  $m$  noise vectors  $z^1, z^2, \dots, z^m$  from the noise prior distribution  $P_z(z)$
    - iii. Updating the parameters of discriminator by ascending the gradient of  $\widetilde{V}_d$  with the learning rate  $\eta$ :

$$\widetilde{V}_d = \frac{1}{m} \sum_{i=1}^m \left[ \log D(x^i) + \log \left( 1 - D(G(z^i)) \right) \right]$$

$$\theta_d = \theta_d + \eta \nabla \widetilde{V}_d(\theta_d)$$

- Block 2: execute once
  - i. Sampling  $m$  noise vectors  $z^1, z^2, \dots, z^m$  from the noise prior distribution  $P_z(z)$
  - ii. Updating the parameters of generator by descending the gradient of  $\widetilde{V}_g$  with the learning rate  $\eta$ :

$$\widetilde{V}_g = \frac{1}{m} \sum_{i=1}^m \left[ \log \left( 1 - D(G(z^i)) \right) \right]$$

$$\theta_g = \theta_g - \eta \nabla \widetilde{V}_g(\theta_g)$$

Nevertheless, at the beginning of the training process, the gradient of  $D(G(z))$  is small because  $G$  is not good enough to defraud  $D$ . In other words, the changing of  $\log(1 - D(G(z)))$  is slight and it makes the training process inefficient. To learn the model quickly in the early stage,  $\log(1 - D(G(z)))$  can be replaced with  $-\log(D(G(z)))$  in practice. Figure 2-4 illustrates the difference these two functions when  $D(G(z))$  changes and it can be observed that the changing of the function  $-\log(D(x))$  is more severe initially. Hence, minimizing  $-\log(D(G(z)))$  is more productive than minimizing  $\log(1 - D(G(z)))$ .

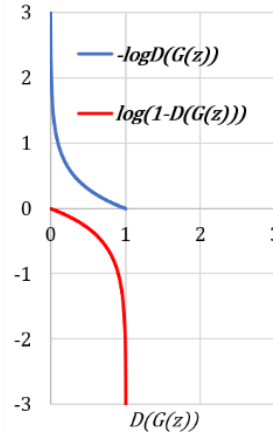


Figure 2-4. The difference between  $\log(1 - D(G(z)))$  and  $-\log(D(G(z)))$  when  $D(G(z))$  changes

Along with GANs arises, it provides solutions to problems encountered by existing generative model:

1. Complex Markov chains don't be used in the training process. Instead, GANs can be simply trained with backpropagation.
2. GANs can generate more sharp images than those created by traditional models such as VAE.
3. GANs is unsupervised machine learning and it can be widely applied in unsupervised and semi-supervised machine learning domain.
4. The generator of GANs can generate images in parallel unlike traditional model such PixelRNN which needs to generate images pixel by pixel.

However, vanilla GANs still has some issues:

1. The training of GANs model may not converge. Sometimes, the gradient descent may not work well in the training process. It means that the training of GANs is more unstable than some traditional models such as VAE and PixelRNN.
2. GANs models encounter gradient vanishing and mode collapse problems.

To solve these issues and improve the performance of GANs, many novel GANs-based models were proposed. The following chapters illustrated how do these models deal with the problems of GANs.

## 2.2 Evolution of GANs

### 2.2.1 The Main Problems of Vanilla GANs

#### I. Non-convergence

The training objective of GANs is finding a Nash equilibrium in a minmax game. Figure 2-5 demonstrates a value function  $V(x, y)$  with a saddle point  $(x = 0, y = 0)$  is a solution of a Nash equilibrium. When  $G$  is fixed,  $D$  cannot get more benefit as it changes around the saddle point, and vice versa. However, in the training process of GANs, both the update of the generator and discriminator may eliminate each other's progress. It makes the loss of models fluctuate and the training process hard to converge. At the early stage of GANs' research, researchers used human monitoring to check the quality of images at a certain interval and terminate the model training if necessary. The method was not a scientific method and did not solve the root cause.

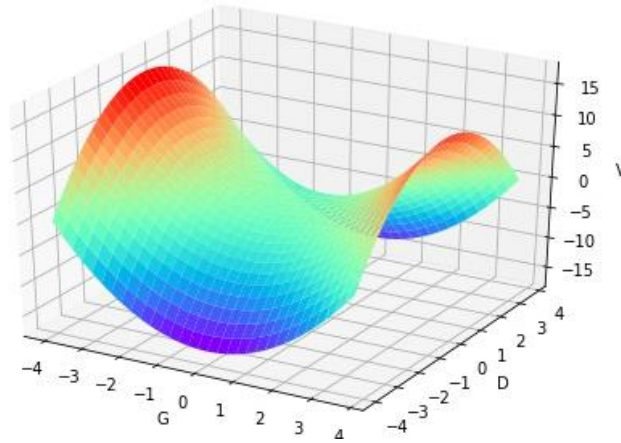


Figure 2-5. A value function  $V(x, y) = x^2 - y^2$  with a saddle point  $(x = 0, y = 0)$

#### II. Mode collapse

Another problem of vanilla GANs is mode collapse. It means that the generator only generates one or a few modes of the data. A complete mode collapse seldom occurs, but a fractional mode collapse happens often in practice. For instance, a GANs model learns a digital image representation from MNIST (LeCun et al., 1998) dataset which contains digits from 0 to 9. As shown in Figure 2-6, a model with mode collapse generated numerous number eight instead of the other nine different number. Instead, a model without mode collapse can generate all digits. This kind of mode collapse is inter-class mode collapse and another type is intra-class (Huang et al., 2018). There are many writing styles in each digit. However, an intra-class mode collapse model only generates a certain writing style instead of every style.

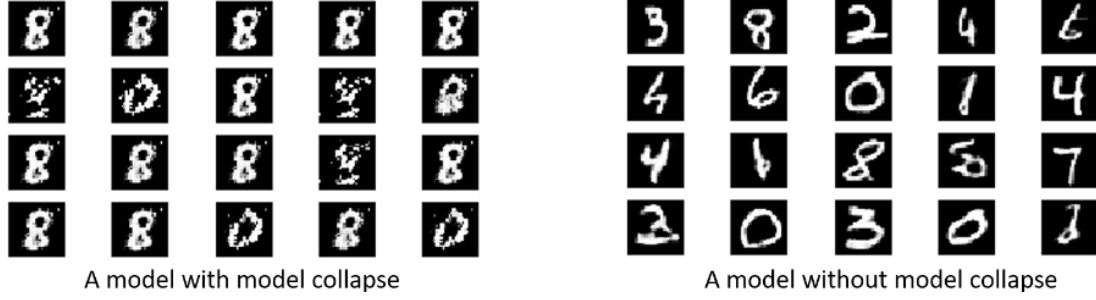


Figure 2-6. The demonstration of the model collapse. It was produced by the project's models

Goodfellow (2016) gives a primary explanation to illustrate why mode collapse happens. An original training process of GANs is minimax optimization and an optimal generator is obtained (equation (2-7)). In practice, the training process may like maxmini optimization which is shown in equation (2-8) sometimes.

$$G^* = \min_G \max_D V(D, G) \quad (2-7)$$

$$G^* = \max_D \min_G V(D, G) \quad (2-8)$$

In the maxmini training process, the generator is updated multiple times in a loop and may map all input noise vector  $z$  into the same generated data. In that way, the generator can be easy to cheat the discriminator. Because the optimal generator already been insensitive to input noise  $z$  before training the discriminator at next round, the gradient descent method makes the discriminator move to a worse state. Furthermore, the responsibility of the discriminator is only telling real data and fake data without examining the diversity of fake data. Eventually, GANs model only generates very few types of data and mode collapse occurs. In the original GANs, it suggests that the generator should avoid making too much progress without updating the discriminator (Goodfellow et al., 2014). Another useful method called mini-batch discriminator is proposed to alleviate the mode collapse problem (Salimans et al., 2016). Minibatch discriminator takes multiple input data at once instead of one by one. Through inner layers of the discriminator, each input can be transformed into a vector of features. The discriminator multiplies these vectors with a tensor and gets feature matrixes of each input data. By computing L1-distance between feature matrixes row by row, the discriminator can calculate the similarity of input data. Hence, the minibatch discriminator not only distinguishes real and fake data but also compute the diversity of data. Through the mechanism, the generator reduces the intention to generate single-mode data to fool the discriminator and the mode collapse problem is alleviated. The following sections demonstrated advanced GANs models which applied different strategies to solve the issues of GANs.



### **2.2.2 Structure Improvement: Deep Convolutional GAN (DCGAN)**

Convolutional neural networks (CNNs) (LeCun et al., 1998) which can capture different spatial features of images through different kernels have shown well performance on the classification domain in recent years. DCGAN successfully adapts the strength of CNNs and modified it to build up a more stable GANs-based model comparing with the original GANs (Radford et al., 2015).

The traditional architecture of CNNs has four main parts including the convolution layer, the pooling layer, the flatten layer and the fully connected layer (LeCun et al., 1998). CNNs utilizes different convolution layers to extract interesting features and reduce computation amount by pooling layers. After that, all feature maps generated by pooling layers are flattened and connected with fully connected layers as general deep neural networks.

DCGAN does not use the architecture of the original CNNs directly, it adopts a variation of CNNs called the all convolution net (Springenberg et al., 2014). Instead of using pooling layers, the all convolution net applies strided convolution layers to implement spatially dimension reduction. There are three main emphases in the architecture of DCGAN. Firstly, it is just like the all convolution net. Pooling layers are abandoned, and fractional-strided and strided convolution layers are used in the generator and discriminator respectively. Strided convolutions layers make spatial downsampling be learned by the generator and the discriminator respectively. Secondly, there is no fully connected layer both in the generator and the discriminator. It increases the efficiency of the model convergence. Finally, Batch normalization is added after every layer except the output layer of the generator and the input of the discriminator. It improves the efficiency of training in deep neural networks and prevents the generator from the mode collapse problem. The reason for not applying all layers with batch normalization is preventing GANs model from instability.

In terms of activation functions, three activation functions, including ReLU, Tanh and Leaky ReLU, are applied in the networks. All layers of the generator use ReLU activation function except for the output layer which adopts Tanh activation function. Leaky ReLU is applied to all layers of the discriminator. Through experiments and the observations of the paper (Radford et al., 2015), these setup of activation functions enhance the speed of the training process and make models reach the convergence.

### **2.2.3 New Estimating Distance: Wasserstein GAN (WGAN)**

Arjovsky et al. (2017) proposed Wasserstein GAN (WGAN) which provided solutions to no-convergence and mode collapse problems observed in the original GAN. WGAN introduced the Earth-Mover (EM) distance which can be called as Wasserstein-1 distance into GAN to replace the KL and the JS divergence which evaluate the distance between real and generated data distribution in the vanilla GAN. Before

illustrating EM distance, we can consider two simple discrete distributions  $\mathbb{P}_g$  and  $\mathbb{P}_r$  with five possible states  $x$  and  $y$  respectively which is shown in Figure 2-7. There are many transport plans which can transform  $\mathbb{P}_g$  to the target distribution  $\mathbb{P}_r$  and each of transport plan has a cost. The cost of transport plans can be defined as  $\sum_{x,y} \gamma(x,y) \|x - y\|$ . For instance, Figure 2-8 demonstrates a transport plan  $\gamma(x,y)$  with the transport cost 8.

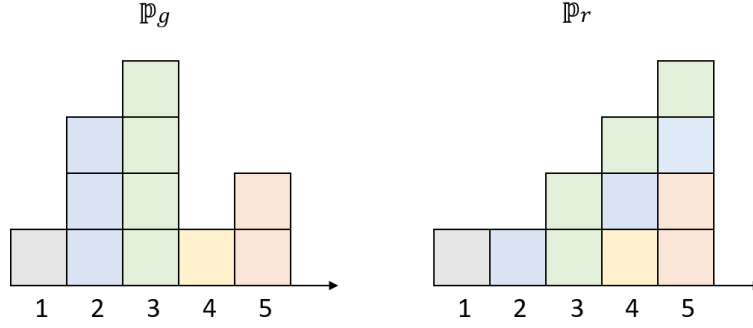


Figure 2-7. Two simple discrete distributions with five possible states

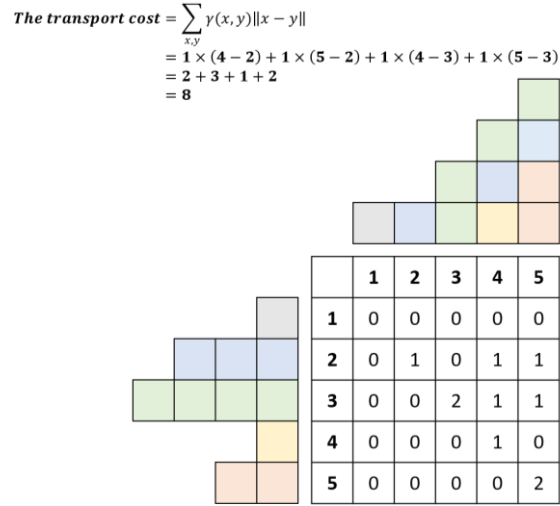


Figure 2-8. A transport plan of transforming  $\mathbb{P}_g$  to  $\mathbb{P}_r$

The EM distance is the optimal transport plan which has the smallest cost which is shown in equation (2-9). Comparing with the JS and the KL divergence, The Wasserstein distance provides more smooth gradients all over the data space and prevents model training from the gradient vanishing problem.

$$W(\mathbb{P}_g, \mathbb{P}_r) = \inf_{\gamma \in (\mathbb{P}_g, \mathbb{P}_r)} \sum_{x,y} \gamma(x,y) \|x - y\| = \inf_{\gamma \in (\mathbb{P}_g, \mathbb{P}_r)} \mathbb{E}_{(x,y) \sim \gamma} \|x - y\| \quad (2-9)$$

Based on the Wasserstein distance, WGAN proposed a new objective function (equation (2-10)) to evaluate the Wasserstein distance between real data  $P_{data}$  and generated data  $P_g$ .  $f$  is a critic, and its functionality is evaluating the quality of fake data generated by the generator. Unlike the discriminator in the original GAN which gives a real data possibility of the data generated by the generator, the critic in WGAN told the generator about its progress and help the generator to learn better. There is a constraint of  $f$  which it must be a 1-Lipschitz function. The Lipschitz function is defined as equation (2-11). It means that  $f$  should be a smooth function.

$$W(P_{data}, P_g) = \max_{f \in 1-Lipschitz} \mathbb{E}_{x \sim P_{data}}[f(x)] - \mathbb{E}_{x \sim P_g}[f(x)] \quad (2-10)$$

$$\|f(x_1) - f(x_2)\| \leq K \|x_1 - x_2\| \quad (2-11)$$

$f$  is a 1-Lipschitz function as  $K = 1$

However, the 1-Lipschitz function is hard to be tracked in a high dimensional space. Hence, WGAN simply uses weight clipping to constrain the critic. The weight clipping is forcing parameters  $\theta$  of the critic to be in a specific region. If  $\theta$  is bigger than constant  $c$  and the program makes  $\theta$  be equal to  $c$ . Furthermore, if  $\theta$  is smaller than  $-c$ , the program makes  $\theta$  be equal to  $-c$ .

The following illustrates the high-level pseudo code of WGAN:

1. Initialize the critic  $f_0$  and the generator  $G_0$  with parameters  $\theta_f^0$  and  $\theta_g^0$  respectively.
2. Train WGAN with  $n$  times iteration and do the below step within each iteration:
  - Block 1: repeat  $k$  times
    - i. Sample  $m$  examples  $x^1, x^2, \dots, x^m$  from the data distribution  $P_{data}(x)$
    - ii. Sample  $m$  noise vectors  $z^1, z^2, \dots, z^m$  from the noise prior distribution  $P_z(z)$
    - iii. Update the parameters of the critic by ascending the gradient of  $\widetilde{W}_f$  with the learning rate  $\eta$  and restrict  $\theta_f$  by weight clipping:

$$\widetilde{W}_f = \frac{1}{m} \sum_{i=1}^m [f(x^i) - f(G(z^i))]$$

$$\theta_f = \theta_f + \eta \nabla \widetilde{W}_f(\theta_f)$$

$$\theta_f = clip(\theta_f, -c, c)$$

- Block 2: execute once
  - i. Sample  $m$  noise vectors  $z^1, z^2, \dots, z^m$  from the noise prior distribution  $P_z(z)$

- ii. Update the parameters of generator by descending the gradient of  $\widetilde{W}_g$  with the learning rate  $\eta$ :

$$\begin{aligned}\widetilde{W}_g &= -\frac{1}{m} \sum_{i=1}^m f(G(z^i)) \\ \theta_g &= \theta_g - \eta \nabla \widetilde{W}_g(\theta_g)\end{aligned}$$

Moreover, WGAN removes sigmoid activation function in the last layer of the critic and uses RMSProp as an optimizer instead of momentum-based optimizers such as momentum and Adam.

Nevertheless, there is a problem with WGAN. As the authors said, weight clipping is not good enough to constrain the critic to be a 1-Lipschitz function. Because the setting of  $c$  is tricky. If  $c$  is too large, it may make the critic hard to be trained optimally. If  $c$  is too small, the gradient vanishing problem may arise (Arjovsky et al., 2017).

To more specifically constrain the critic, Gulrajani et al. (2017) proposed WGAN-GP with the gradient penalty to enforce the 1-Lipschitz constraint on the critic. The authors observe that “A differentiable function is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere” (Gulrajani et al., 2017). Therefore, the objective function is added a gradient penalty (equation (2-12)) to be an alternative way to constrain the critic. Because it is hard to calculate the norm of gradients on entire data space, WGAN-GP only compute the gradient’s norm of data which is from the penalty distribution  $P_{penalty}$ . Eventually, the objective function of WGAN-GP can be rewritten as equation (2-13).  $P_{penalty}$  is composed of the middle points of a pair of data sampled from  $P_{data}$  and  $P_g$  respectively which is shown in Figure 2-9. From the experiments of WGAN-GP, using gradient penalty only with data from  $P_{penalty}$  shows the performance of the model is better than WGAN and can restrict the critic more appropriately (Gulrajani et al., 2017).

$$W(P_{data}, P_g) = \max_f (\mathbb{E}_{x \sim P_{data}} [f(x)] - \mathbb{E}_{x \sim P_g} [f(x)] - \lambda \int_x \max(0, \|\nabla_x f(x)\| - 1) dx) \quad (2-12)$$

$$W(P_{data}, P_g) \approx \max_f (\mathbb{E}_{x \sim P_{data}} [f(x)] - \mathbb{E}_{x \sim P_g} [f(x)] - \lambda \mathbb{E}_{x \sim P_{penalty}} [\max(0, \|\nabla_x D(x)\| - 1)]) \quad (2-13)$$

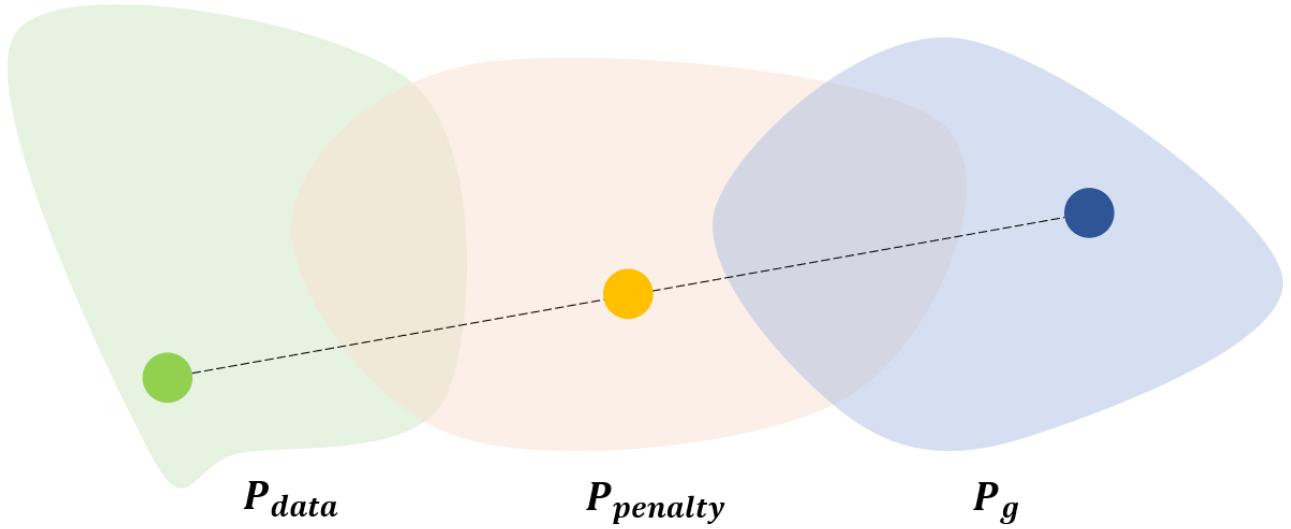


Figure 2-9. The illustration of  $P_{penalty}$ . The figure was reproduced from Lee (2018)

## 2.3 Significant Components of deep neural networks

In the section, significant components of deep neural networks were illustrated. These components were also used in GANs models which were implemented in the project.

### 2.3.1 Introduction of Common Activation function in Neural Networks

The purpose of activation functions is introducing non-linear feature into neural networks. It makes neural networks solve complex problems by using a small number of nodes. In general, multiple inputs multiplied with their weights are summed up and the summation is fed into activation function to compute an input for the next layer. The section demonstrates two activation functions which are used in the research.

- **ReLU**

ReLU was proposed by Nair and Hinton (2010) and defined as  $R(x) = \max(0, x)$ . It gives output the same value as the input if the input is bigger than 0 (Figure 2-10). Otherwise, it outputs 0. Its computation effort is less than sigmoid and tanh. It also introduces sparsity into models by making some node inactivated. ReLU is broadly applied to hidden layers to prevent models from the overfitting problem. The advantages of ReLU are alleviating the gradient vanishing problem and enhancing the efficiency of the training process. Its disadvantage is causing some nodes dead because of the existence of 0 regions. It could make part of neural networks stop working as known as dying ReLU.

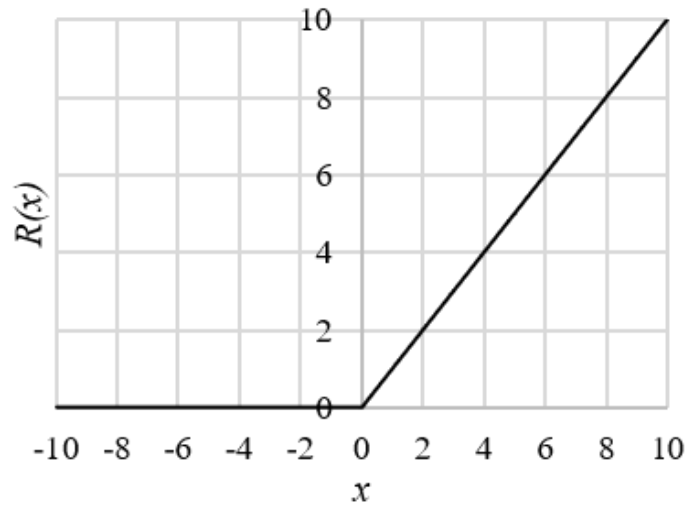


Figure 2-10. ReLU function

- **Leaky ReLU**

Leaky ReLU is a variation of ReLU (Maas et al., 2013). Unlike ReLU, leaky ReLU has no 0 regions. It makes inputs which are smaller than 0 multiplied with a small ratio such as 0.01 (Figure 2-11). Leaky ReLU has the same advantages of ReLU but prevents models from the dying ReLU problem. Leaky ReLU is also frequently applied on hidden layers of neural networks.

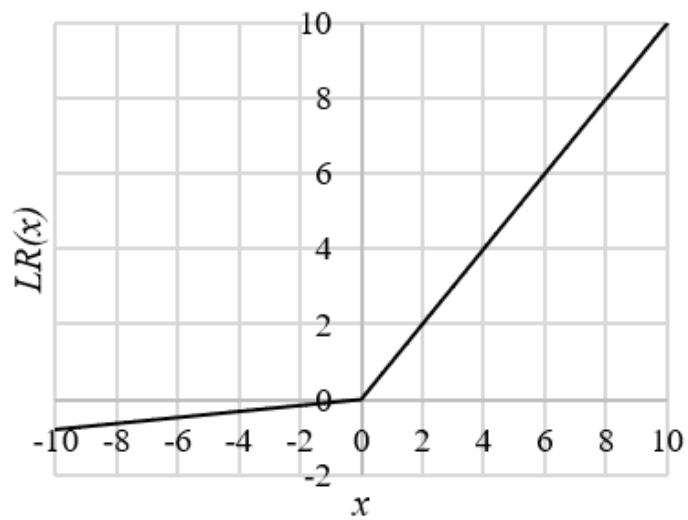


Figure 2-11. Leaky ReLU function

### 2.3.2 Introduction of Common Optimizers in Neural Networks

The training of deep neural networks is based on the gradient descent method which computes the gradients of models with training dataset and updates models' weights toward the opposite direction of gradients. Common optimizers include SGD and Momentum. The former updates the gradient data by data and the latter introduces a physical mechanism to SGD. This section introduced another two optimizers which were used in the project to improve the efficiency of the gradient's computation.

- **Adagrad**

The learning rate of the training process plays an import role in the training process in terms of its efficiency and convergence. If the learning rate is too big, it may make models hard to converge because of the vibration of the cost. If the learning rate is too small, the speed of models' convergence is too slow. Hence, it is hard to choose an appropriate learning rate in the beginning. Furthermore, the learning rate may be different in different stages of the training process. At the beginning of a model training, a big learning rate can make the model have huge progress. When the model almost reaches the global minimum, a small learning rate helps it reach the target decently. Adagrad introduces a changeable learning rate to SGD (Duchi et al., 2011). An adjustable term  $\frac{1}{\sqrt{n+\epsilon}}$  is added. Hence, the learning rate can change by time.  $n$  is the sum of past gradients' square and  $\epsilon$  prevents the value in root computation from being 0. Through the mechanism, the learning rate can be changed by time and improve the efficiency of the model training. Nevertheless, the learning rate becomes very small along with the accumulation in the denominator. This main disadvantage makes models stop learning knowledge from their training datasets.

$$W_{t+1} \leftarrow W_t - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L_t}{\partial W_t}$$
$$n = \sum_{r=1}^t \left( \frac{\partial L_r}{\partial W_r} \right)^2$$

- **RMSprop**

Root Mean Square Propagation (RMSprop) was invented by Tieleman and Hinton (2012). The method was not published on a paper but was taught on Tieleman and Hinton's lecture. The format of RMSprop is shown below and is similar to Adagrad. However, the former adds  $v_t$  which is the combination of the exponential average of the square of past gradients and the square of the current gradient.  $\rho$  is moving average parameter and is suggested as 0.9. RMSprop can solve small learning rate problem of Adagrad. RMSprop also prevents training processes from the oscillation of the cost by slowing down learning tempo automatically.

$$W_{t+1} \leftarrow W_t - \eta \frac{1}{\sqrt{v_t + \epsilon}} \frac{\partial L_t}{\partial W_t}$$

$$v_t = \rho v_{t-1} + (1 - \rho) \left( \frac{\partial L_t}{\partial W_t} \right)^2$$

- **Adam**

Adaptive Moment Estimation (Adam) is a machine learning optimizer which also adopts an adaptive learning rate (Kingma and Ba, 2014). It combines the advantages of Momentum and RMSprop. Momentum refers to previous gradients and changes the range of weights updating. RMSprop adjusts the current gradient based on its intensity. Adam computes the estimates of the first moment of gradients  $m_t$  and the second moment of gradients  $v_t$ . These are biases on these two terms. Hence, Adam uses  $\hat{m}_t$  and  $\hat{v}_t$  to alleviate them. In the end, the updating of the models' weights can be shown below. In practice,  $\beta_1$ ,  $\beta_2$  and  $\epsilon$  are set as 0.9, 0.999 and  $10^{-8}$  respectively. The settings work well according to the authors' experiments (Kingma and Ba, 2014).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L_t}{\partial W_t}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left( \frac{\partial L_t}{\partial W_t} \right)^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$W_{t+1} \leftarrow W_t - \eta \frac{1}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Some of the above optimizers seem to have complex computation. However, they are easy to be applied through current open-source neural network libraries such as Keras (Chollet, 2015). Developers simply use functions of neural network libraries to choose specified optimizers for their models.



## 2.4 Evaluation of Synthetic Images

An intuitive evaluation method is conducting a questionnaire about asking participants to distinguish images which are created by generative models. Salimans et al. (2016) used Amazon Mechanical Turk (MTurk) to perform the subjective evaluation on GANs. AMT is an internet platform built by Amazon and hiring people to do tasks which are hard for computers but effortless for human.

Besides the qualitative evaluation method, quantitative methods are also used to evaluate the performance of generative models. There are two widely used quantitative evaluation methods including and Fréchet Inception Distance (FID) (Heusel et al., 2017).

The first one is Inception Score (IS) which was proposed by Salimans et al. (2016). The authors applied a pre-trained image classifier, Inception, which can classify input images and give the top-5 most likely class (Szegedy et al., 2016) to calculate IS. IS evaluates a generative model based on two criteria. The first standard is a generative model can produce meaningful images. It means that a conditional label distribution  $p(y|x)$  has low entropy and. The second standard is a great variety of images can be generated via the model. In the other word, the entropy of the marginal  $p(y) = \int p(y|x = G(z)) dz$  is high and the equation can be changed as  $p(y) = \frac{1}{N} \sum_{i=1}^N p(y|x^i)$  practically. Furthermore,  $p(y|x)$  is a distribution with a pick and  $p(y)$  is a mean distribution. It means that if the KL divergence between these two distributions is bigger, the performance of models is better. In conclusion, IS can be defined as equation (2-14).

$$IS = \exp(\mathbb{E}_x KL(p(y|x) \parallel p(y))) \quad (2-14)$$

However, IS still has some limitations. Barratt and Sharma (2018) concluded two types of IS's limitations including IS-itself type and usage scenario type. In IS-itself type, the first limitation is that IS is sensitive to inner weights of pre-trained classifiers. For instance, the value of IS in the Inception model trained by TensorFlow and the Inception model trained by Keras are different. The difference could overweight the improvement claimed by new proposed models. The second limitation in this type is the calculation procedure of Inception Score. The common calculation process is generating 50000 images, dividing them into 10 equal parts, calculating Inception Score of each part and calculating the mean and the variance. However, each part only contains 5000 images and is not enough to represent the marginal  $p(y)$ . The recommended method is calculating Inception Score of 50000 images at once. In usage scenario type, the first limitation is Inception Score is unreasonable if the training dataset of generative models is not same as the training dataset of pre-trained classifier. Hence, it is better to train Inception model with same dataset which is used by generative model. The second limitation is Inception Score is only a rough guideline and

cannot be used to optimize generative models directly. The last limitation is Inception Score cannot detect the overfitting phenomenon of generative models.

The other evaluation method is Fréchet Inception Distance (Heusel et al., 2017). It is based on the idea of Fréchet Distance which is used to calculate the distance between two multivariate normal distributions (Dowson and Landau, 1982). FID extracts feature both of generated images and realistic images respectively from the last pooling layer in Inception and calculate the Fréchet Distance between them. From the last pooling layer, the mean of feature  $m$  and the covariance matrix  $C$  can be obtained. Hence, equation (2-15) shows FID between real images ( $m_r, C_r$ ) and generated images ( $m_g, C_g$ ).

$$FID = \|m_r - m_g\|^2 + Tr(C_r + C_g - 2(C_r C_g)^{\frac{1}{2}}) \quad (2-15)$$

Comparing with IS, FID is more consistent with judgment conducted by humans and more capable to resist the effect of noise. Furthermore, FID can detect the intra-class mode collapse and IS cannot fulfill the same thing (Heusel et al., 2017). FID compares features from generated and real images and it makes FID more reasonable than IS. However, both cannot detect the overfitting of generative models. In this study, FID is applied to evaluate generative models because it has more advantages than IS as mentioned above.

## Chapter 3: Research Method

The project used four GANs models including original GANs, DCGAN, WGAN, and WGAN-GP to generate images from the different training dataset. These models were evaluated based on the speed of models' convergence, the quality of generated images and the variety of images.

### 3.1 Details of GANs models

The details of GANs models were divided into three parts in terms of the structures of GANs models, the loss function of the model and the training process.

#### 3.1.1 Original GANs

- **The structure of the model**

The generator of the original GANs was a simple fully-connected neural. There were four hidden layers in the networks with a 100-dimensional input layer and an output layer with the same shape as training images. All hidden layers had 512 neural nodes and applied Leaky ReLU activation functions except for the fourth layer which used Tanh. From the first to fourth hidden layers, the number of their nodes are 256, 512, 1024 and the product of training images' shape respectively.

The discriminator was a binary classifier. It was also a simple neural network which had two hidden layers with Leaky ReLU activation functions. The discriminator took an image which was flattened as an input and outputs a value which was between 0 to 1 to present the possibility of real images.

In practice, the generator and the discriminator were created individually. The discriminator was trained firstly. Then, these two neural networks were combined as a generative adversarial network and it was trained with the fixed discriminator to get the optimal generator. Figure 3-1 illustrated the overall structure of the original GANs.

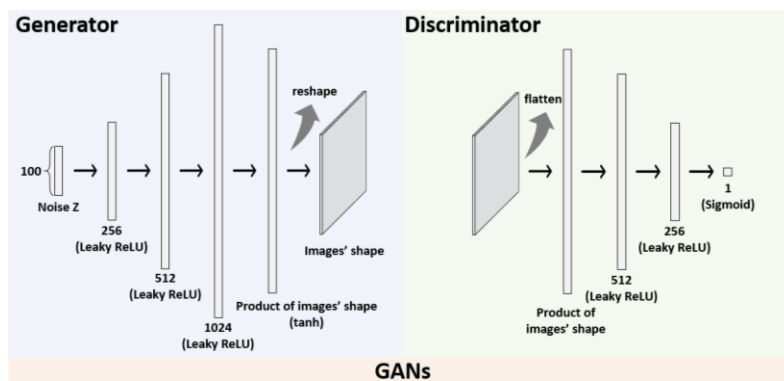


Figure 3-1. The structure of the original GAN which is used in the project

- **Loss function**

In the original GANs, both the discriminator and the combined neural network used binary cross-entropy as loss functions. Intuitively, it was reasonable for the discriminator to use binary cross-entropy as its loss function because the task which the discriminator wanted to solve was a binary classification problem. For the combined neural network, it took a noise input and the input was fed into the generator. The generator outputted an image and it was the input of the discriminator. After training the discriminator, the generator can be optimized by training the combined neural network with the fixed discriminator. Hence, the combined neural network also solved a binary classification problem.

- **Training process**

The training process was like the process illustrated in section 2.1.3. In the original paper (Goodfellow et al., 2014), the discriminator can be trained multiple times before optimizing the generator. In the research, both discriminator and the generator are trained one time in each iteration. Adam optimizer is used in the neural networks for optimizing the speed of model training.

### **3.1.2 DCGAN**

- **The structure of the model**

As its name, DCGAN adopted convolution neural networks to build its architecture. Unlike traditional convolution neural networks, there was no pooling layer in DCGAN. Fractional-strided and strided convolution layers were applied on the generator and the discriminator respectively. To more easily explain DCGAN's structure, we assumed that training dataset was colorful images with  $64 \times 64$  pixels. As usual, the input of the generator was a 100-dimensional noise which was sampled from a normal distribution. The noise was connected by a layer with  $4 \times 4 \times 1024$  nodes and the layer was followed by a ReLU activation function. The output of the layer was reshaped to three-dimensional space and its size was  $4 \times 4 \times 1024$ . After this layer, the following layers used fractional-strided convolution to perform the upsampling of data with batch normalization and ReLU function. It was worth to know that the batch normalization and ReLU were not used on the output layer. It only used Tanh function and outputs images with  $64 \times 64 \times 3$  pixels. 3 meant that a colorful image had R, G, and B channels. From the structure of the generator, it can be observed that the generator's output was upsampled layer by layer. It meant that DCGAN provided flexibility for different training datasets with different sizes.

The structure of the generator can be extended by users' need. In terms of the structure of the discriminator, it was almost the reverse of the generator. The discriminator took an input with  $64 \times 64 \times 3$  pixels and it used strided convolution with 2 steps to downsample the data layer by layer. All convolution layers used

Leaky ReLU as activation functions and they were followed by batch normalization except for the first convolution layer. In the end, the last convolution layer was connected fully with the out layer which has a node. Like the original GAN, the discriminator outputted a value to show the probability of real images.

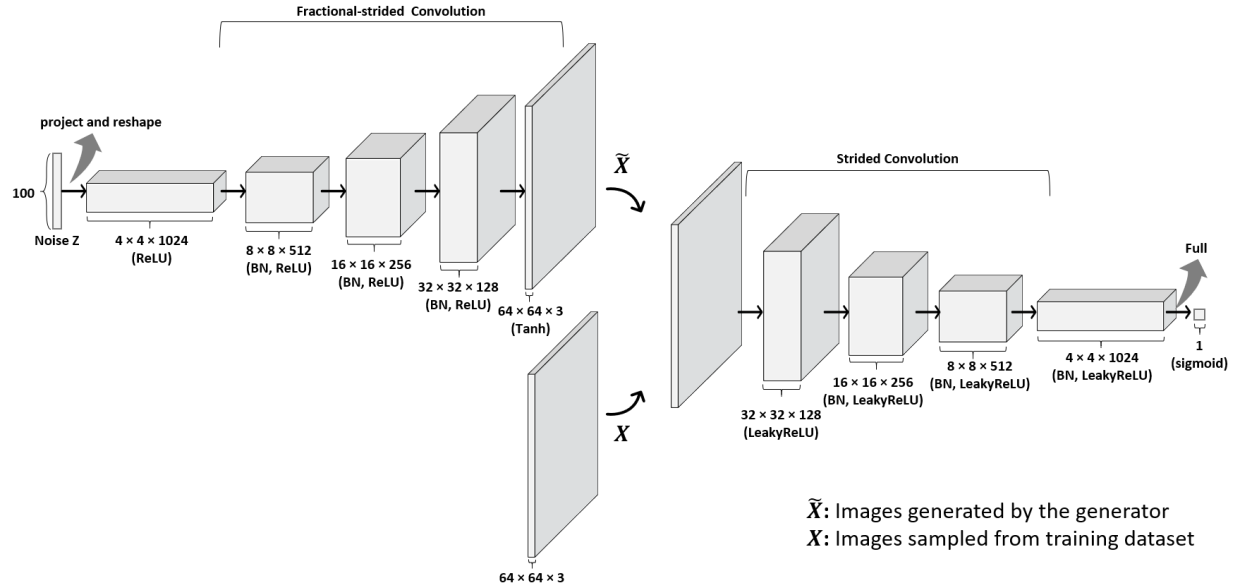


Figure 3-2. The overall architecture of DCGAN which was reproduced from Radford et al. (2015)

- **Loss function and training process**

The loss function of DCGAN was binary cross-entropy. The training process of the original GANs can be applied in DCGAN directly and Adam optimizer was used in the deep neural networks.

### 3.1.3 WGAN

- **The structure of the model**

Because the structure of DCGAN made the training process more stable, it was widely used in many GANs as a base structure. Hence, deep convolution neural networks were used in the discriminator and the generator of WGAN. However, there was a changing in WGAN. The sigmoid was removed from the output layer of the discriminator. Consequently, the discriminator produced a value which was not constrained between 1 to 0.

- **Loss function**

Unlike, the vanilla GAN and DCGAN, there was no log function in WGAN. Hence the loss of WGAN was easier to compute.  $y_{true}$  was ground truth and  $y_{pred}$  was the value produced by the discriminator in WGAN. The loss of WGAN can be formulated as equation (3-1 in practice.

$$Wasserstein\ Loss = \frac{1}{m} \sum_{i=1}^m [y_{true} \times y_{pred}] \quad (3-1)$$

- **Training process**

The main structure of WGAN's training process was similar to GANs' but there were three differences in the former. Firstly, we trained the discriminator once in each iteration. Nevertheless, the discriminator of WGAN was trained 5 times in each iteration. Secondly, the optimizer was RMSProp instead of momentum-based optimizers such as momentum and Adam. Thirdly, there was a weight clipping procedure before updating the weights of the model. The parameter  $c$  of the weight clipping was 0.01 and it constrained the weights between -0.01 to 0.01.

### 3.1.4 WGAN-GP

- **The structure of the model**

In this project, WGAN-GP followed the structure of WGAN's model. Like WGAN, there was no sigmoid in the output layer. There was a difference between WGAN and WGAN-GP. There was no batch normalization in the discriminator of WGAN-GP. Although there were many GANs' models which applied the batch normalization to stabilize the training of the models, the process of the batch normalization went against the gradient penalty in WGAN-GP because the gradient penalty treated inputs one by one instead of the entire batch of inputs. Through the experiments of WGAN-GP, the model did well without the gradient vanishing problem in spite of a lack of the batch normalization.

- **Loss function**

The main progress of WGAN-GP was utilizing the gradient penalty to constrain the discriminator to be a 1-Lipschitz function more appropriately. Besides the Wasserstein loss, there was a gradient penalty loss in WGAN-GP.

- **Training process**

The training process of WGAN-GP was pretty much like WGAN's except for the former used Adam instead of RMSProp to be an optimizer.

## 3.2 Experimental dataset

There were three datasets employed in the project to analyse and evaluate the performance of these GANs model.

### 3.2.1 MNIST

MNIST was a dataset of various handwritten style digits which modified from NIST dataset which was shown in Figure 3-3 (LeCun et al., 1998). MNIST was extensively used in machine learning domains such as classification (Cireřan et al., 2012, Deng, 2012) and generative models (Kingma and Welling, 2013, Mirza and Osindero, 2014, Maaløe et al., 2016). It contained 60000 training data and 10000 test data, and both included digital images from 0 to 9. These images were grayscale and normalized to  $28 \times 28$  pixels. In practical, we can utilize TensorFlow's function to download MNIST from the internet. Each image was linearized and represented by a vector of size  $1 \times 784$ . The dimension of these images was quite smaller than other dataset and suitable for primary experiments of generative models.

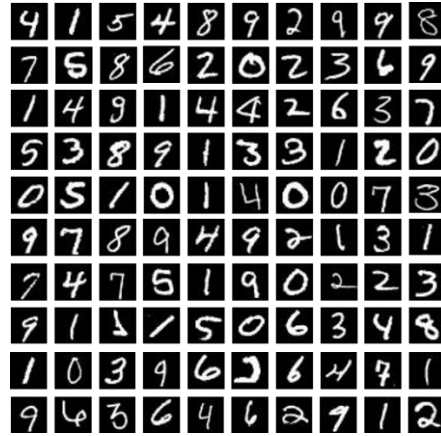


Figure 3-3. Handwritten digits which were sampled from MNIST (LeCun et al., 1998)

### 3.2.2 NLVR Dataset

NLVR dataset was proposed by Suhr et al. (2017) and created for natural language processing (NLP). Although we did not perform NLP in the project, the dataset contained many geometric images and was suitable for exploring the capability of GANs models. Images of NLVR dataset were composed of three boxes which contained different type, size and color geometric shapes (Figure 3-4). The boxes were separated by two gray bars which were shown in Figure 3-5. Because we only focused on generating different combinations of geometric shapes, the boxes were isolated from original images of NLVR and were resized to  $64 \times 64$  pixels. After these image processing, the training dataset contained 9,306 images. Figure 3-6 demonstrated the geometric images which were sampled from the final training dataset.

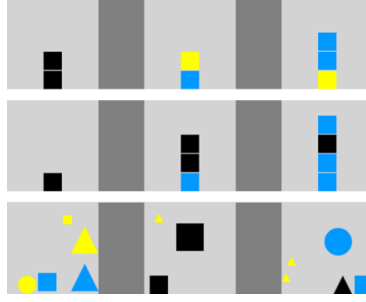


Figure 3-4. Geometric images which were sampled from NLVR dataset (Suhr et al., 2017)

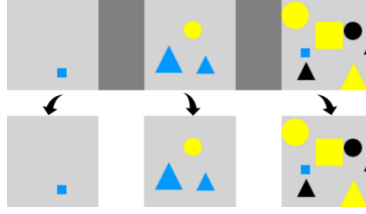


Figure 3-5. An original image from NLVR was divided into three parts

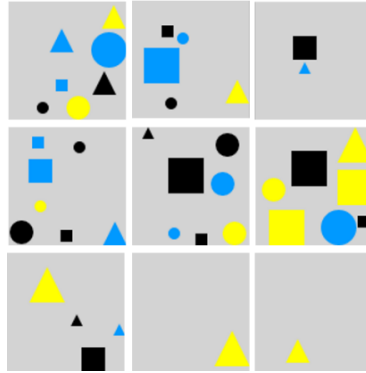


Figure 3-6. The images which were sampled from the final training dataset

### 3.2.3 Oxford-102 Flowers

Oxford-102 flowers database was composed of 102 flower categories which occur in the UK frequently (Nilsback and Zisserman, 2008). The number of flowers' images was 8,189 and each category comprised between 40 and 258 colorful images. The database was used in many experiments of GAN to evaluate their performance (Reed et al., 2016, Zhang et al., 2017, Bodnar, 2018). These images had different scales, poses, and lights which is shown in Figure 3-7. In the report, the images were resized into  $64 \times 64$  pixels and fed to generative adversarial networks.





Figure 3-7. Images which were sampled from Oxford-102 flowers (Nilsback and Zisserman, 2008)

### 3.3 The Design of Experiments

This section demonstrated three experiments which explored the performance of different GAN models on interesting datasets such as Oxford-102 Flowers.

#### 3.3.1 Experiment 1: Generate handwritten digits

The first experiment was using MNIST to train GAN models and generated handwritten digits. The iteration of the model training was 30,000 times. The losses of the generator and the discriminator were recorded and performed further analysis. The models were saved every 10,000 iterations. Furthermore, the generator generated 64 images every 200 iterations. Saving images in different stages of the model training can help us to monitor the state of the models. Theoretically, the models which were trained on MNIST should generate recognizable images. Hence, the first experiment can confirm the correctness of models' structures and their hyperparameters such as optimizers.

It was worth to mention that the structure of DCGAN was not the same as the structure demonstrated in section 3.1. Because the size of the images was  $28 \times 28$  pixels in MNIST, the structure of DCGAN was simpler. For the generator, the input was reshaped to  $7 \times 7 \times 128$  and upsampled to  $14 \times 14 \times 128$ . After that, the following hidden layer kept up-sampling the input to  $28 \times 28 \times 64$  and the last convolution layer made the output be  $28 \times 28 \times 1$ . 1 meant a gray-scale image has only a channel. For the discriminator, it took an image as input and its dimension was  $28 \times 28 \times 1$ . The first and second strided convolution layer made the data be  $14 \times 14 \times 1$  and  $7 \times 7 \times 1$  respectively. To make the computation easier, a zero padding was used for making the data be  $8 \times 8 \times 1$ . And then, the third strided convolution layer can make the data

be  $4 \times 4 \times 1$ . In the end, the flatten layer made the output be a digit. The structures of WGAN and WGAN-GP also were changed based on DCGAN for MNIST.

### **3.3.2 Experiment 2: Generate Geometric Shapes**

After training GANs models with simple gray-scale image dataset, it was exciting to expand the scale of GANs models to generate more complex images. The second experiment was training GANs models based on geometric shapes. NLVR dataset was used in the experiment. Images of NLVR was colorful and bigger than images of MNIST. It meant that GANs models were more complicated and had more parameters. The training iteration was 30,000 times, and the generator also generated 64 images every 200 iterations. Models were saved every 10,000 iterations. Because images of NLVR were resized to  $64 \times 64$  pixels, the structure of DCGAN was the same as the structure shown in section 3.1.2.

### **3.3.3 Experiment 3: Generate Flowers**

Although NLVR dataset was more complex than MNIST, its images were not as complicated as images we saw in the world. Hence, Oxford-102 Flowers was used for training more sophisticated generative models for creating real-world images. Images of Oxford-102 Flowers had different scales, poses, and lights. It meant that there were more latent features in the dataset. Therefore, it was more challenge for generative models to generate meaningful images. Because the dataset was more complex, the iteration of the training process was 40,000 times. The models were also saved every 10,000 iterations and generate images every 200 iterations. We also used the same models' structures shown in section 3.1.2.

## **3.4 Implement GANs' models by Keras**

Keras was an open-source machine learning library which was proposed by Chollet (2015). It was a high-level API written by Python and designed as the top of prevalent neural-networks such as TensorFlow and Theano. Keras provided a user-friendly modeling API for deep learning developers. Users did not worry about building fundamental structures of models such as making tensors. Keras used a component called Backend to transfer low-level computations to TensorFlow or Theano. Keras let developers implement their deep learning models rapidly and helped them get initial results quickly. In this project, TensorFlow was used as the Backend of Keras. Because the structure of DCGAN was applied intensively in the project, we mainly used DCGAN to illustrate how to use Keras to build models.

### 3.4.1 Important components of Keras

- **Sequential**

The first step for building deep learning models was creating a “**Sequential**” object. Sequential was the main component to connect all elements in Keras. Developers can add multiple layers to construct their neural networks.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

model = Sequential()
```

Models utilized “**add**” function to stack multiple neural network layers. For the first layer, developers needed to specify the dimension of input data.

```
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dense(64, activation='relu'))
```

- **Dense**

Dense was a fully connected neural network layer. Usually, there were two important parameters. One was “**units**” which was a positive number to determine the dimension of the output. The other was “**activation**” which specified used activation functions such as **sigmoid**.

```
Dense(units, activation=sigmoid)
```

- **Conv2D**

Conv2D performed 2D convolution computation. The following function demonstrated four crucial parameters of Conv2D. “**filters**” was the depth of output. “**kernel\_size**” was the size of the 2D convolution window. “**strides**” was the step of 2D convolution window in terms of row and column. “**padding**” was a mechanism for handling the boundary problem in convolution computation. There were two options in “**padding**”. One was “**same**” and the other was “**valid**”. “**same**” used zero-padding which added 0 value around the boundary of images. It made convolution computation can fit the size of images. If “**valid**” was used, convolution computation abandons elements which did not meet with the convolution window and its step. In the project, we set this parameter as “**same**”. In that way, the size of outputs is was the size of inputs divided by the number of convolution step.

```
Conv2D(filters, kernel_size, strides=(1, 1), padding='same')
```

- **Conv2DTranspose**

Conv2DTranspose performed up-sampling intelligently and magnified the size of images. “strides” controlled the size of an image. The following code demonstrated an image’s size was doubled. The function was used in the generator of DCGAN

```
Conv2D(filters, kernel_size, strides=(2, 2), padding='same')
```

- **Dropout**

Dropout algorithm randomly set some nodes of layers as 0. It can alleviate the overfitting problem and be used in convolution neural networks frequently. “rate” was the ratio of dropped nodes.

```
Dropout(rate)
```

- **BatchNormalization**

BatchNormalization performed the batch normalization algorithm. It can prevent models from gradient vanishing problem and stabilize the training process. Developers can set the momentum of the moving mean and the moving variance through “momentum”.

```
BatchNormalization(momentum=0.99)
```

### 3.4.2 Build a stacked model by Keras

Because a GANs model was composed of a discriminator and a generator, we needed to stack two models through **Model** class of Keras. Firstly, we created a Sequential object which had two hidden layers.

```
self.latent_dimension = 100
self.images_shape = (28, 28, 1)

generator = Sequential()

generator.add(Dense(512, input_dim=self.latent_dimension))
generator.add(BatchNormalization())
generator.add(LeakyReLU(alpha=0.2))
generator.add(Dense(np.prod(self.images_shape), activation='tanh'))
generator.add(Reshape(self.images_shape))
```

Then, we used a **Model** object to wrap this model which took a 100-dimensional input and outputted an image with the size  $28 \times 28 \times 1$ .

```
random_noise = Input(shape=(self.latent_dimension,))

generated_image = generator (random_noise)

self.generator = Model(random_noise, generated_image)
```

After instantiating the generator, we can use the same elements to create the discriminator.

```
discriminator = Sequential()

discriminator.add(Flatten(512, input_shape=self.image_shape))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dense(1, activation='sigmoid'))
```

Unlike the generator, the discriminator took an image as an input and outputted a fraction.

```
images = Input(shape= self.images_shape)

validity = model_d(images)
self.discriminator = Model(images, validity)
```

In Keras, models needed to be configured by “**compile**” function. The model of the discriminator was compiled firstly, and the optimizer of the model was Adam. There were three main parameters including the learning rate,  $\beta_1$ , and  $\beta_2$  in Adam. In compile function, you can determine the loss function, the training optimizer, and evaluation metric.

```
optimizer = Adam(0.0002, 0.5, 0.999)
self.discriminator.compile(loss='binary_crossentropy',
                           optimizer=optimizer,
                           metrics=['accuracy'])
```

After compiling the discriminator, we set up the input “**z**” of the generator and instance the output of the generator “**image**”.

```
z = Input(shape=(self.latent_dim,))
image = self.generator(z)
```

Then, we made the discriminator untrainable before stacking models because the discriminator was fixed when the generator was trained. The discriminator took the product of the generator as its input. The following code built the connection between these two models.

```
self.discriminator.trainable = False
validity = self.discriminator(image)
```

After connecting the generator and the discriminator, we built a complete GANs model as below.

```
self.combined = Model(z, validity)
self.combined.compile(loss='binary_crossentropy', optimizer=optimizer)
```

### 3.4.3 Train GANs model by Keras

The section demonstrated how to use Keras to train GANs models. The GANs model was trained 1000 times in the example. In each iteration, the discriminator was trained firstly, and then the stacked model was trained. Before training the models, the ground truth of the data needed to be specified. If an image was real, it was label 1. If an image was fake, it was label 0. The following code created label vectors and their length was the same as the batch size.

```
valid = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))
```

The training process can be divided into five steps which were shown in Figure 3-8. Firstly, we sampled a batch of images from a training dataset and created a noise tensor randomly. Secondly, the generator produced a batch of fake images. Thirdly, the discriminator was trained with a whole batch of real images and fakes images respectively. “**train\_on\_batch**” function can perform a gradient update algorithm with a batch of data. Fourthly, we created a noise tensor for training generator. Finally, the generator can be trained through training the stacked model. Because the discriminator in the stacked model was untrainable, we can train the generator with a fixed discriminator. After training the discriminator and the generator, presenting their loss could be good for observing the state of models.

```
for epoch in range(epochs):

    # -----
    # Train Discriminator
    # -----

    # Select a random batch of images
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]

    noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

    # Generate a batch of new images
    gen_imgs = self.generator.predict(noise)

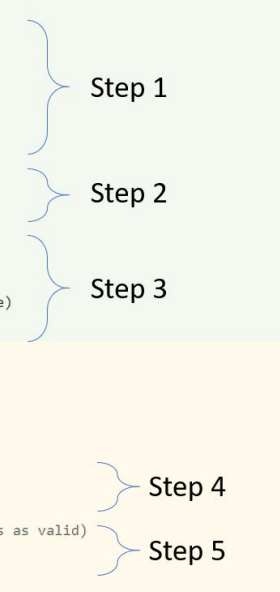
    # Train the discriminator
    d_loss_real = self.discriminator.train_on_batch(imgs, valid)
    d_loss_fake = self.discriminator.train_on_batch(gen_imgs, fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # -----
    # Train Generator
    # -----

    noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

    # Train the generator (to have the discriminator label samples as valid)
    g_loss = self.combined.train_on_batch(noise, valid)

    # Plot the progress
    print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100*d_loss[1], g_loss))
```



Step 1

Step 2

Step 3

Step 4

Step 5

Figure 3-8. The training process of an original GANs model from the code of the project

### 3.4.4 Build DCGAN's model by Keras

Section 3.4.2 only demonstrated how to build an original GANs model by Keras. This section further illustrated a DCGAN's structure which was stacked by Keras.

We can explore the detail of DCGAN from its generator. As usual, the generator was instanced as a Sequential. A dense object was used to concrete the first hidden layer which took a 100-dimensional noise and outputted a tensor with the size  $1024 \times 4 \times 4$ . The tensor was the start of fractional-strided convolution stack. Conv2DTranspose was used from the second to fifth hidden layers. The stride of Conv2DTranspose was 2 and it can upsample the data intelligently. The size of the feature maps in fractional-strided convolution layers was 5 by 5. Following the paper of DCGAN (Radford et al., 2015), activation functions of layers were ReLU except for the output layer which used Tanh. As shown in Figure 3-9, a generator with a deep convolution neural network can be built easily and orderly.

```
def build_generator(self):  
    model = Sequential()  
  
    model.add(Dense(1024 * 4 * 4, activation="relu", input_dim=self.latent_dim))  
    model.add(Reshape((4, 4, 1024)))  
  
    #4 x 4 -> 8 x 8  
    model.add(Conv2DTranspose(512, kernel_size=5, strides=2, padding='same'))  
    model.add(BatchNormalization(momentum=0.8))  
    model.add(Activation("relu"))  
  
    #8 x 8 -> 16 x 16  
    model.add(Conv2DTranspose(256, kernel_size=5, strides=2, padding='same'))  
    model.add(BatchNormalization(momentum=0.8))  
    model.add(Activation("relu"))  
  
    #16 x 16 -> 32 x 32  
    model.add(Conv2DTranspose(128, kernel_size=5, strides=2, padding='same'))  
    model.add(BatchNormalization(momentum=0.8))  
    model.add(Activation("relu"))  
  
    #32 x 32 -> 64 x 64  
    model.add(Conv2DTranspose(64, kernel_size=5, strides=2, padding='same'))  
    model.add(BatchNormalization(momentum=0.8))  
    model.add(Activation("relu"))  
  
    model.add(Conv2D(self.channels, kernel_size=5, padding="same"))  
    model.add(Activation("tanh"))  
  
    model.summary()  
  
    noise = Input(shape=(self.latent_dim,))  
    img = model(noise)  
  
    return Model(noise, img)
```

Figure 3-9. A DCGAN's generator built by Keras and it is captured from the project's code

A discriminator built by Keras was shown in Figure 3-10. From the first hidden layer, we can see the discriminator took an image as an input and used convolution computation with 2 strides to perform the downsampling of the input. The first hidden layer was followed by three convolution layers which kept downsampling the data with the batch normalization. In the end, a flatten layer was used to transfer the data into a  $1 \times n$  vector.  $n$  was the number of the last convolution layer's output. The flatten layer was followed by a fully-connected layer. The output of the full-connected layer was the probability of real data. The dropout algorithm was employed in all convolution layers to prevent the model from the overfitting problem. We applied LeakyReLU in all hidden layers besides the output layer which employed Sigmoid. After building the models of DCGAN, the same training procedure which was shown in section 3.4.3 can be used to training DCGAN.

```
def build_discriminator(self):
    model = Sequential()

    #64 x 64->32 x 32
    model.add(Conv2D(32, kernel_size=5, strides=2, input_shape=self.img_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    #32 x 32->16 x 16
    model.add(Conv2D(64, kernel_size=5, strides=2, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    #16 x 16->8 x 8
    model.add(Conv2D(128, kernel_size=5, strides=2, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    #8 x 8->4 x 4
    model.add(Conv2D(256, kernel_size=5, strides=2, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    model.summary()

    img = Input(shape=self.img_shape)
    validity = model(img)

    return Model(img, validity)
```

Figure 3-10. A DCGAN's discriminator built by Keras and it was captured from the project's code



### 3.4.5 Instructions of Running Code

The code of the project was composed of the training module and the evaluating module. Figure 3-11 demonstrated how to use “GAN\_main.py” to train different types of GANs on three different training datasets. Furthermore, Figure 3-12 illustrated how to utilize “GAN\_evaluation.py” to evaluate GANs by calculating their FID and synthesising corresponding images.

```
usage: GAN_main.py [-h]
                    model_name training_dataName epochs batch_size
                    sample_interval save_model_interval

positional arguments:
  model_name            Specify the type of model. (GAN, DCGAN, WGAN, WGANGP)
  training_dataName     Training data for GANs. (mnist, nlvr, 102flowers)
  epochs                Training epochs for the training process.
  batch_size            Batch size for the training process.
  sample_interval       The interval of sampling synthesised images
  save_model_interval   The interval of saving GANs models

optional arguments:
  -h, --help            show this help message and exit

Example:

python3 GAN_main.py WGANGP 102flowers 40000 64 1000 10000
```

Figure 3-11. The instruction of the training module of the project

```
usage: GAN_evaluation.py [-h]
                        model_directory FID_sample_number
                        Generate_sample_number

positional arguments:
  model_directory       The directory of GANs models
  FID_sample_number     The number of generated images for caculating FID
  Generate_sample_number
                        The number of generated images

optional arguments:
  -h, --help            show this help message and exit

Example:

python3 GAN_evaluation.py gan_models/test 1000 500
```

Figure 3-12. The instruction of the evaluating module of the project

## 3.5 Evaluation Methods

To evaluate the quality of generated images, qualitative and quantitative evaluation methods were applied. The former was employing questionnaires to perform human judgment, and the latter was computing Fréchet Inception Distance (FID) to obtain objective scores (Heusel et al., 2017).

### 3.5.1 Evaluating Synthesised Images by Human Judgment

The questionnaire was modified from experiments performed by Salimans et al. (2016). It was divided into three parts according to different datasets. Firstly, each question demonstrated real and fake images which were sampled from the corresponding dataset and generative models respectively (Figure 3-13). And then, there were 4 images which were composed of 2 real and 2 generated images. Participants were required to annotate fake images. In the process, the role of participants was similar to a discriminator in a GANs model. A good generator should make human hard to discriminate the difference between real and generated images. The higher the error rate of discriminating fake images is, the better the generative model is. The full questionnaire was presented in Appendix A.

We demonstrate you pictures that are either generated by a computer or sampled from real images. Your task is to choose which **two** were generated by a computer

**Q1.**













Examples of real images	Examples of images generated by a computer
   	   
<input type="checkbox"/> 	<input type="checkbox"/> 
<input type="checkbox"/> 	<input type="checkbox"/> 

Figure 3-13. A question of the project's questionnaire

### 3.5.2 Evaluating Synthesised Images by Fréchet Inception Distance (FID)

A generative model generated 5,000 images each time and its FID was calculated. The smaller the FID is, the better the generative model is. FID can also detect the mode collapse in models. Through the objective evaluation, the quality of numerous synthesised images can be examined automatically.

## Chapter 4: The Analyse of the Experiments

This chapter demonstrated the results of experiments and comparing the performance of the models with each other.

### 4.1 The 1<sup>st</sup> Experiment: Generate Handwritten Digits

The first experiment was using GANs models to generate binary handwritten images. The structures of models were modified to take images with  $28 \times 28$  pixels as inputs. All models were trained with 128 batch size and 30,000 epochs.

#### 4.1.1 Vanilla GANs on MNIST

The structures of the generator and the discriminator was presented in Figure 4-1. Both models are built with multilayer perceptron (MLP). There are 533, 505 and 1,493,520 parameters in the discriminator and the generator respectively.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0	dense_4 (Dense)	(None, 256)	25856
dense_1 (Dense)	(None, 512)	401920	leaky_re_lu_3 (LeakyReLU)	(None, 256)	0
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0	batch_normalization_1 (Batch Normalization)	(None, 256)	1024
dense_2 (Dense)	(None, 256)	131328	dense_5 (Dense)	(None, 512)	131584
leaky_re_lu_2 (LeakyReLU)	(None, 256)	0	leaky_re_lu_4 (LeakyReLU)	(None, 512)	0
dense_3 (Dense)	(None, 1)	257	batch_normalization_2 (Batch Normalization)	(None, 512)	2048
Total params: 533,505			dense_6 (Dense)	(None, 1024)	525312
Trainable params: 533,505			leaky_re_lu_5 (LeakyReLU)	(None, 1024)	0
Non-trainable params: 0			batch_normalization_3 (Batch Normalization)	(None, 1024)	4096
			dense_7 (Dense)	(None, 784)	803600
			reshape_1 (Reshape)	(None, 28, 28, 1)	0
			Total params: 1,493,520		
			Trainable params: 1,489,936		
			Non-trainable params: 3,584		

Figure 4-1. The structures of GAN's model for MNIST

From the line graphic (Figure 4-2), the loss of the discriminator increased dramatically firstly. After around 300 epochs, its figure remained steady with a small vibration. For the loss of the generator, it decreased rapidly in the beginning and start to climb. It reached a peak of 1.2 in 1500 epochs and kept steady after 4000 epochs. Both the losses of the generator and the discriminator had small vibration along with the training progressing. The phenomena implied that these two neural networks competed in the maxmin game.

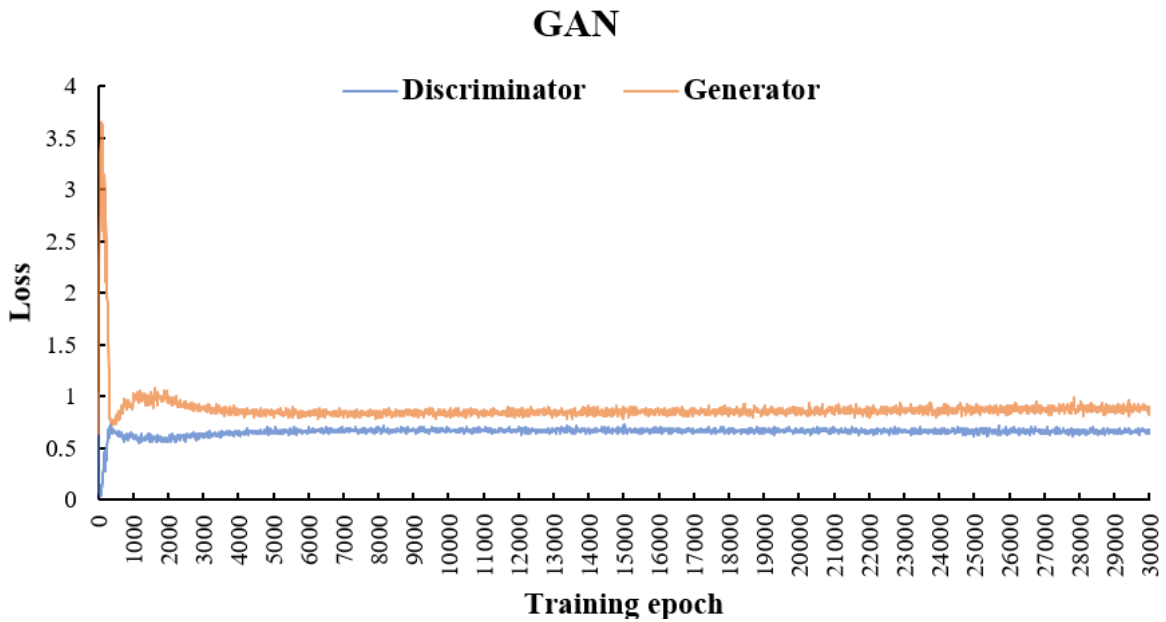


Figure 4-2. The losses of vanilla GAN trained on MNIST

Figure 4-3 demonstrated generated images in different training stages from 10,000 to 30,000 epochs. After observing, the generator can produce a higher quality of images along with the increase of the training epochs. However, there were some limitations such as the white noise around digits, incomplete shape, and rouge boundaries.



Figure 4-3. Handwritten digits generated by vanilla GANs of the project

## 4.1.2 DCGAN on MNIST

DCGAN was composed of two convolution neural networks. Their structures were simpler than standard DCGAN because the size of the images in the training dataset was smaller. Their structure was presented in Figure 4-4.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
conv2d_28 (Conv2D)	(None, 14, 14, 32)	320	dense_21 (Dense)	(None, 6272)	633472
leaky_re_lu_32 (LeakyReLU)	(None, 14, 14, 32)	0	reshape_6 (Reshape)	(None, 7, 7, 128)	0
dropout_12 (Dropout)	(None, 14, 14, 32)	0	up_sampling2d_9 (UpSampling2D)	(None, 14, 14, 128)	0
conv2d_29 (Conv2D)	(None, 7, 7, 64)	18496	conv2d_32 (Conv2D)	(None, 14, 14, 128)	147584
zero_padding2d_1 (ZeroPadding2D)	(None, 8, 8, 64)	0	batch_normalization_26 (Batch Normalization)	(None, 14, 14, 128)	512
batch_normalization_23 (Batch Normalization)	(None, 8, 8, 64)	256	activation_11 (Activation)	(None, 14, 14, 128)	0
leaky_re_lu_33 (LeakyReLU)	(None, 8, 8, 64)	0	up_sampling2d_10 (UpSampling2D)	(None, 28, 28, 128)	0
dropout_13 (Dropout)	(None, 8, 8, 64)	0	conv2d_33 (Conv2D)	(None, 28, 28, 64)	73792
conv2d_30 (Conv2D)	(None, 4, 4, 128)	73856	batch_normalization_27 (Batch Normalization)	(None, 28, 28, 64)	256
batch_normalization_24 (Batch Normalization)	(None, 4, 4, 128)	512	activation_12 (Activation)	(None, 28, 28, 64)	0
leaky_re_lu_34 (LeakyReLU)	(None, 4, 4, 128)	0	conv2d_34 (Conv2D)	(None, 28, 28, 1)	577
dropout_14 (Dropout)	(None, 4, 4, 128)	0	activation_13 (Activation)	(None, 28, 28, 1)	0
conv2d_31 (Conv2D)	(None, 4, 4, 256)	295168	Total params: 856,193		
batch_normalization_25 (Batch Normalization)	(None, 4, 4, 256)	1024	Trainable params: 855,809		
leaky_re_lu_35 (LeakyReLU)	(None, 4, 4, 256)	0	Non-trainable params: 384		
dropout_15 (Dropout)	(None, 4, 4, 256)	0			
flatten_6 (Flatten)	(None, 4096)	0			
dense_20 (Dense)	(None, 1)	4097			
Total params: 393,729					
Trainable params: 392,833					
Non-trainable params: 896					

Figure 4-4. The structures of DCGAN's model for MNIST

The losses of the generator and the discriminator were demonstrated in Figure 4-5. The figures for these two neural networks decreased rapidly in the beginning and then kept steady with small vibrations.

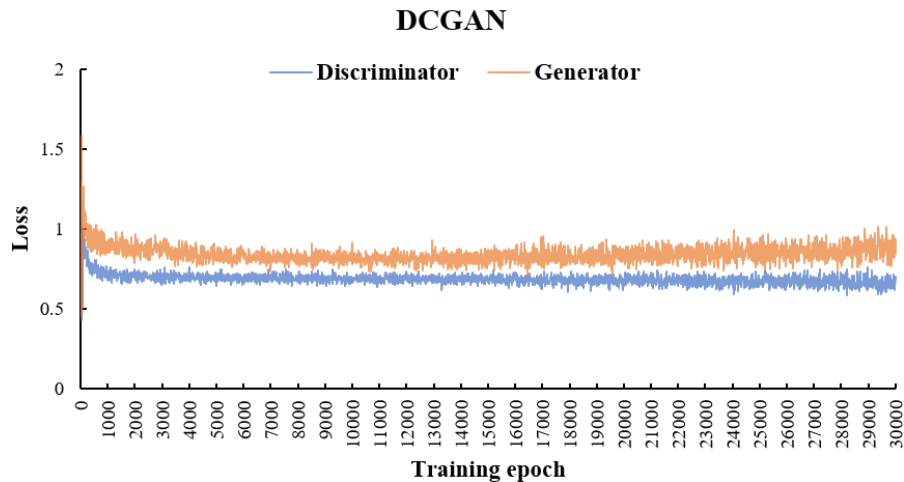


Figure 4-5. The losses of DCGAN trained on MNIST

Generated images of DCGAN were presented in Figure 4-6. The quality of generated images was improved when the number of training epochs increased. After 30,000 epochs, DCGAN generated high-quality images. There was very little noise around digits and the shape of digits was smooth and complete.

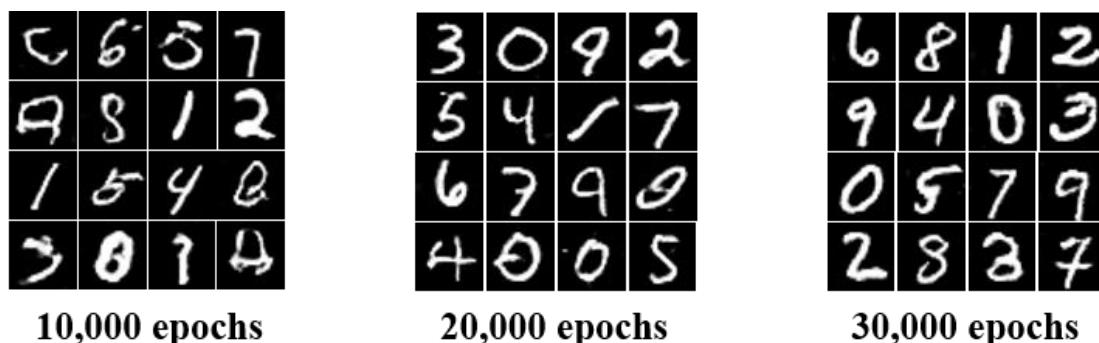


Figure 4-6. Handwritten digits generated by DCGAN of the project

### 4.1.3 WGAN on MNIST

The structures of the generator and the discriminator in WGAN were based on DCGAN. There were two differences in the discriminator of WGAN. One was no batch normalization layers and the other was no sigmoid activation function in the output layer. Figure 4-7 demonstrated the full structures of the discriminator and the generator.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
conv2d_15 (Conv2D)	(None, 28, 28, 64)	1664	dense_16 (Dense)	(None, 1024)	103424
leaky_re_lu_19 (LeakyReLU)	(None, 28, 28, 64)	0	leaky_re_lu_23 (LeakyReLU)	(None, 1024)	0
dropout_5 (Dropout)	(None, 28, 28, 64)	0	dense_17 (Dense)	(None, 6272)	6428800
conv2d_16 (Conv2D)	(None, 12, 12, 128)	204928	batch_normalization_12 (Batch Normalization)	(None, 6272)	25088
leaky_re_lu_20 (LeakyReLU)	(None, 12, 12, 128)	0	leaky_re_lu_24 (LeakyReLU)	(None, 6272)	0
dropout_6 (Dropout)	(None, 12, 12, 128)	0	reshape_4 (Reshape)	(None, 7, 7, 128)	0
conv2d_17 (Conv2D)	(None, 6, 6, 128)	409728	conv2d_transpose_3 (Conv2DTranspose)	(None, 14, 14, 128)	409728
leaky_re_lu_21 (LeakyReLU)	(None, 6, 6, 128)	0	batch_normalization_13 (Batch Normalization)	(None, 14, 14, 128)	512
dropout_7 (Dropout)	(None, 6, 6, 128)	0	leaky_re_lu_25 (LeakyReLU)	(None, 14, 14, 128)	0
flatten_4 (Flatten)	(None, 4608)	0	conv2d_18 (Conv2D)	(None, 14, 14, 64)	204864
dense_14 (Dense)	(None, 1024)	4719616	batch_normalization_14 (Batch Normalization)	(None, 14, 14, 64)	256
leaky_re_lu_22 (LeakyReLU)	(None, 1024)	0	leaky_re_lu_26 (LeakyReLU)	(None, 14, 14, 64)	0
dense_15 (Dense)	(None, 1)	1025	conv2d_transpose_4 (Conv2DTranspose)	(None, 28, 28, 64)	102464
Total params: 5,336,961			batch_normalization_15 (Batch Normalization)	(None, 28, 28, 64)	256
Trainable params: 5,336,961			leaky_re_lu_27 (LeakyReLU)	(None, 28, 28, 64)	0
Non-trainable params: 0			conv2d_19 (Conv2D)	(None, 28, 28, 1)	1601
			Total params: 7,276,993		
			Trainable params: 7,263,937		
			Non-trainable params: 13,056		

Figure 4-7. The structures of WGAN's model for MNIST

In Figure 4-8, the loss of the generator fluctuated dramatically before 1000 epochs. After that, the figure for the generator climbed near to 0 and stayed stable. For the loss of the discriminator, there was the lowest point around 400 epochs, and the loss increased near to 0 and remained steady. Overall, the losses of these two neural networks converged as the training progressed.

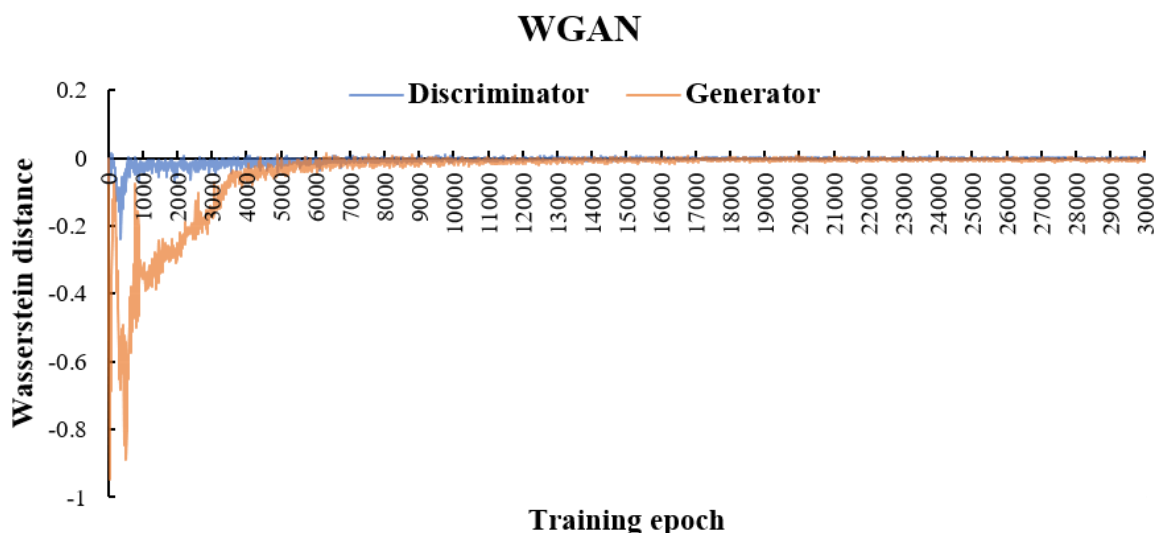


Figure 4-8. The losses of WGAN trained on MNIST

WGAN can synthesise compelling handwritten digits (Figure 4-9). After 10,000 training epochs, the generator already produced good-quality images.

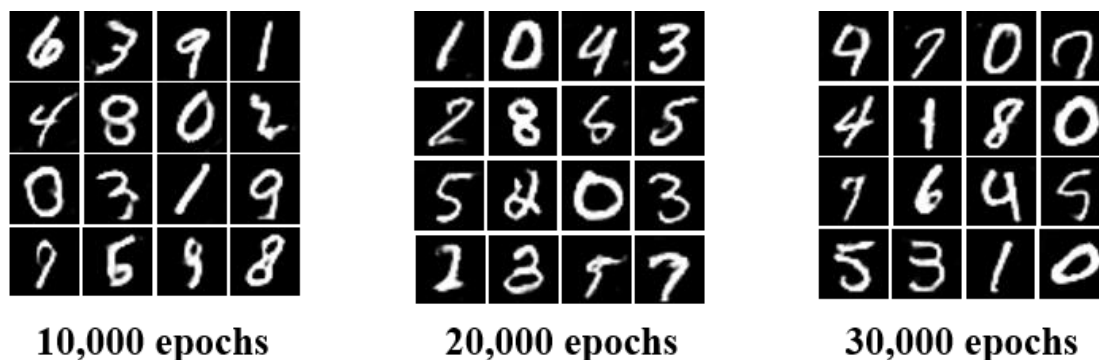


Figure 4-9. Handwritten digits generated by WGAN of the project



#### 4.1.4 WGAN-GP on MNIST

The generator and the discriminator of WGAN-GP had the same structures in WGAN. Figure 4-10 demonstrated the changing of the model's losses. The loss of the discriminator climbed rapidly firstly and remained steady with small vibration between -3 to -2. The loss of the generator had more fluctuation than the figure for the discriminator, but it tended to converge.

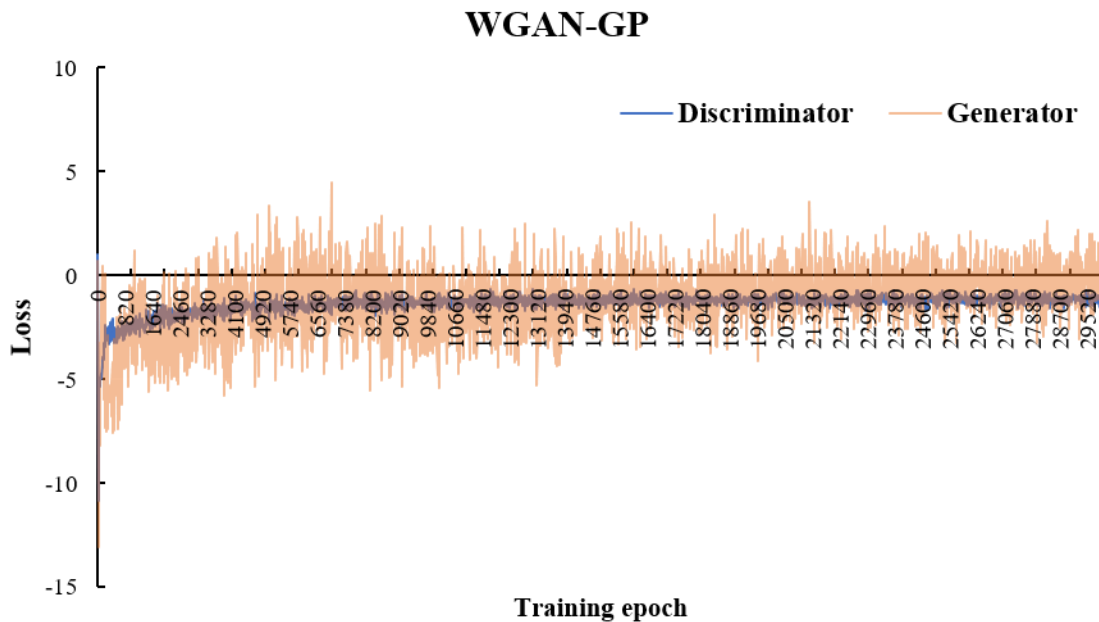


Figure 4-10. The losses of WGAN-GP trained on MNIST

Computer-generated digits of WGAN-GP were presented in Figure 4-11. Along with the increment of the training epoch, the quality of the images was improved.

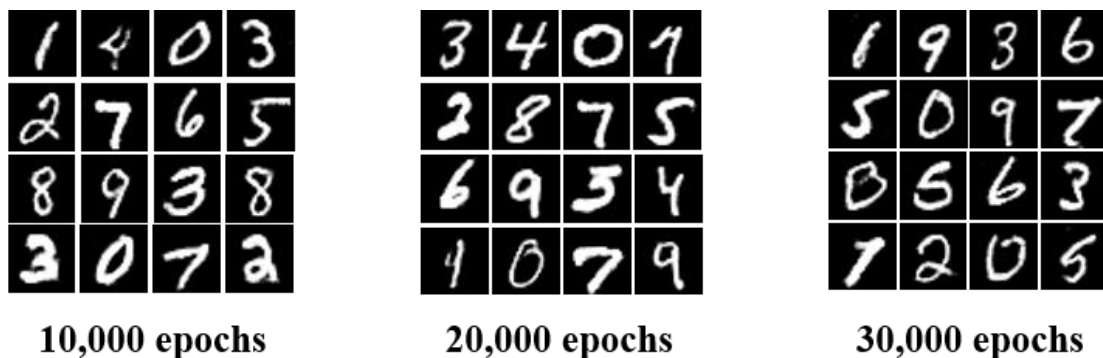


Figure 4-11. Handwritten digits generated by WGAN-GP of the project



### 4.1.5 Model Comparisons on MNIST

Vanilla GANs and DCGAN applied the JS divergence to estimate the similarity between the real data distribution and the generated data distribution. By contrast, the Wasserstein distance was adopted in WGAN and WGAN-GP to estimate the distance between these two distributions. Consequently, these four models were grouped into 2 sets. The first group was vanilla GANs and DCGAN, and the other group was WGAN and WGAN-GP.

Figure 4-12 presented the JS divergence of vanilla GANs and DCGAN. The figure for vanilla GANs was more fluctuating than the one of DCGAN. It ascended firstly and reached a peak of 1.1 and decreased to the lowest point. After that, it increased again and remained steady after 8,000 training epochs. On the other hand, the JS divergence of DCGAN decreased consistently along with the training progressing. It can imply that DCGAN was more stable than vanilla GANs.

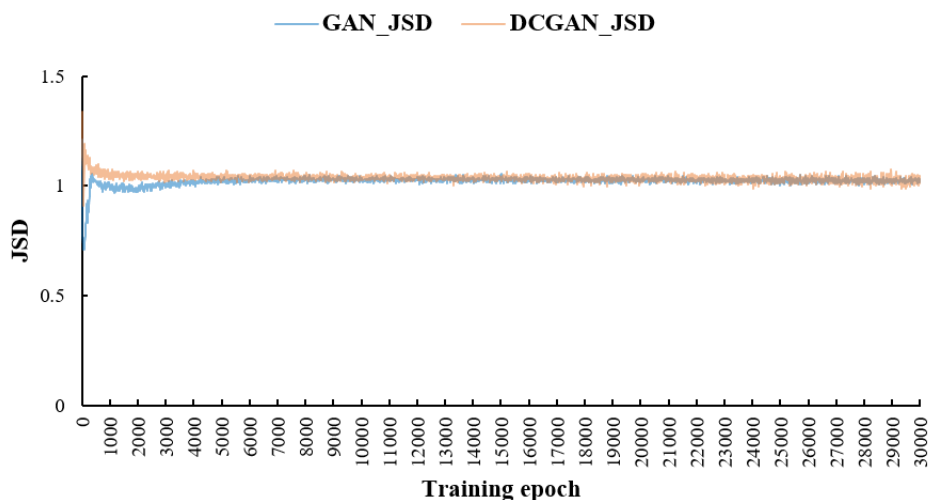


Figure 4-12. The JS divergences of vanilla GANs and DCGAN in MNIST

In terms of the objective evaluation, DCGAN got better FID in each training stage which can be found in Table 4-1. FID of DCGAN was at least 10 times less than the one of vanilla GANs at each training stages. Furthermore, the speed of DCGAN's convergence was also faster than vanilla GANs because the former got a satisfied FID after only 10,000 epochs.

FID			
	10,000 epochs	20,000 epochs	30,000 epochs
Vanilla GANs	114	91	85
DCGAN	10	7	8

Table 4-1. FID of vanilla GANs and DCGAN which were trained on MNIST

Figure 4-13 and Figure 4-14 demonstrated the comparisons between the generated and the corresponding similar real handwritten digits in GAN and DCGAN respectively. They illustrated that the generative models can synthesis new images rather than copying the data from the training dataset.

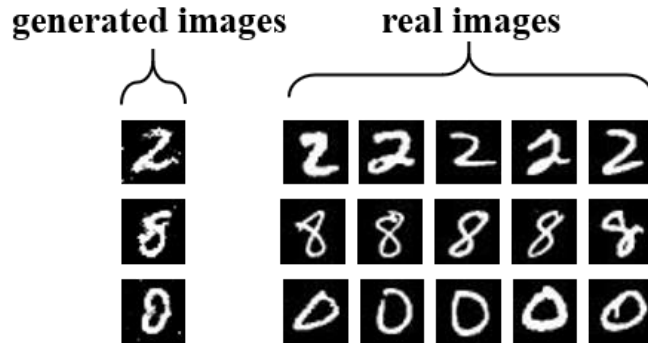


Figure 4-13. The image similarity comparison of GANs in MNIST

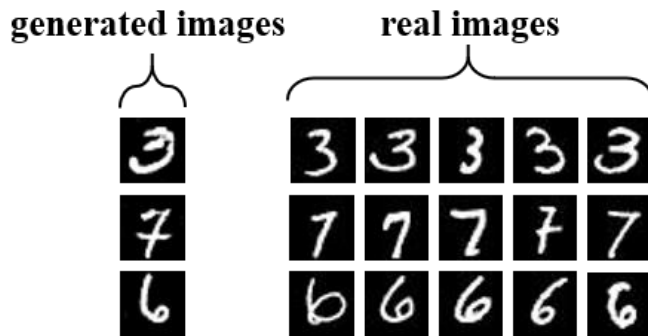


Figure 4-14. The image similarity comparison of DCGAN in MNIST

In the subjective evaluation, vanilla GANs and GDCGAN got error rates of 5.56% and 44.44% respectively on the human judgment. It meant that the generated images from DCGAN were more capable of fooling human through their compelling quality.

Figure 4-15 presented the Wasserstein distance of WGAN and WGAN-GP. The figure for WGAN was much less than the one of WGAN-GP. However, its changing can be observed after enlarging the sub picture in Figure 4-15. It decreased consistently and was gradually close to 0. In the meanwhile, the Wasserstein distance of WGAN had similar behavior. It went down firstly and tent to converge as the training progressed. Table 4-2 demonstrated FID of these two generative models at different training stages. When the models experienced more training, their performance became better. From Figure 4-15 and Table 4-2, we can find a positive correlation between the images' quality and the Wasserstein distance. From Figure 4-16 and Figure 4-17, we can observe that the images which were synthesised by WGAN and WGAN-GP were not exactly the same as the images from the training dataset.

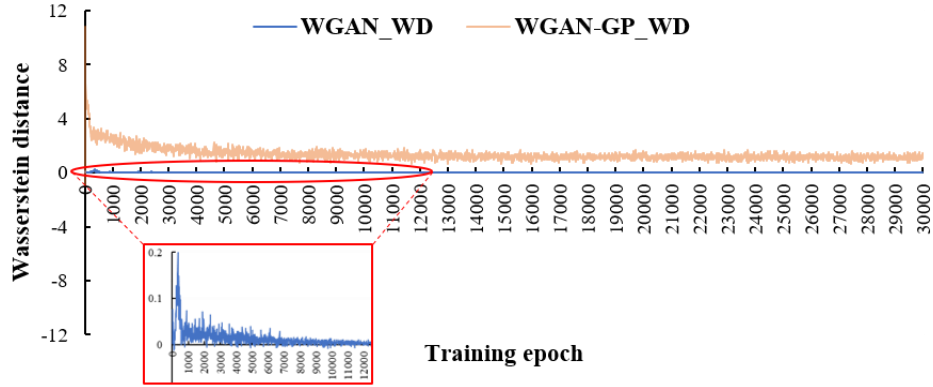


Figure 4-15. The Wasserstein distances of WGAN and WGAN-GP in MNIST

FID			
	10,000 epochs	20,000 epochs	30,000 epochs
WGAN	11	8	6
WGAN-GP	8	6	6

Table 4-2. FID of WGAN and WGAN-GP which were trained on MNIST

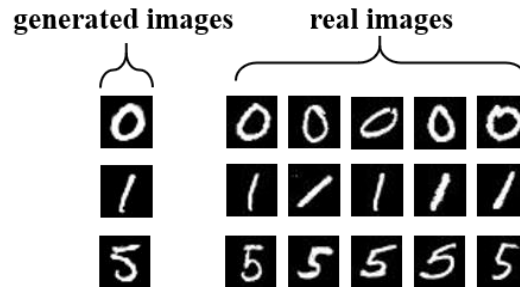


Figure 4-16. The image similarity comparison of WGAN in MNIST

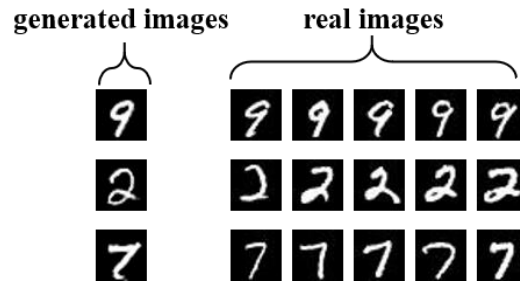


Figure 4-17. The image similarity comparison of WGAN-GP in MNIST

In terms of the subjective evaluation, WGAN and WGAN-GP yielded error rates of 52.78% and 72.22% respectively on the human judgment. It implied that both two models can generate good quality images to misleading the participants of the questionnaire.

In the first experiment, the performance of WGAN-GP was the best among these four models according to FID and human judgment. The worst quality of handwritten digits was generated by vanilla GANs. Furthermore, vanilla GANs was more unstable than DCGAN in the training because of its fluctuating the JS divergence.

## 4.2 The 2<sup>nd</sup> Experiment: Generate Geometric Graphics

After performing GANs on the simple gray-scale image dataset, MNIST, the 2<sup>nd</sup> experiment was conducted to explore the capability of GANs on more complex data. NLVR dataset was used in the second experiment and its images were resized to 64×64 pixels. All models were trained with 64 batch size and 30,000 epochs.

### 4.2.1 Vanilla GANs on NLVR

The structures of both the generator and the discriminator were 4-layer MLP with ReLU activation function and each hidden layer had 512 hidden units. Full structures of these two neural networks were presented in Figure 4-18.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
flatten_8 (Flatten)	(None, 12288)	0	dense_29 (Dense)	(None, 512)	51712
dense_24 (Dense)	(None, 512)	6291968	leaky_re_lu_44 (LeakyReLU)	(None, 512)	0
leaky_re_lu_40 (LeakyReLU)	(None, 512)	0	dense_30 (Dense)	(None, 512)	262656
dense_25 (Dense)	(None, 512)	262656	leaky_re_lu_45 (LeakyReLU)	(None, 512)	0
leaky_re_lu_41 (LeakyReLU)	(None, 512)	0	dense_31 (Dense)	(None, 512)	262656
dense_26 (Dense)	(None, 512)	262656	leaky_re_lu_46 (LeakyReLU)	(None, 512)	0
leaky_re_lu_42 (LeakyReLU)	(None, 512)	0	dense_32 (Dense)	(None, 512)	262656
dense_27 (Dense)	(None, 512)	262656	leaky_re_lu_47 (LeakyReLU)	(None, 512)	0
leaky_re_lu_43 (LeakyReLU)	(None, 512)	0	dense_33 (Dense)	(None, 12288)	6303744
dense_28 (Dense)	(None, 1)	513	reshape_8 (Reshape)	(None, 64, 64, 3)	0
Total params: 7,080,449			Total params: 7,143,424		
Trainable params: 7,080,449			Trainable params: 7,143,424		
Non-trainable params: 0			Non-trainable params: 0		

Figure 4-18. The structures of vanilla GANs' model for colorful images with 64×64 pixels

The losses of the discriminator and the generator were presented in Figure 4-19. The figure for the discriminator reached a peak of 8 after 60 training epochs and did not change anymore. The loss of the generator increased rapidly at the beginning and climbed gently after 1400 training epochs. It reached a peak of 16 after 4200 training epochs. The losses of these two neural networks saturated in the early stage of the training. It meant that two neural networks stopped competing and they did not improve themselves anymore.

After observing the synthesised images, we can find that the generator only generates whole gray color images which were presented in Figure 4-20. The full mode collapse problem occurred, and the generator only produced the same images to fool the generator. In this case, the discriminator had 50% accuracy on distinguishing synthesised images. Generally, 50% accuracy implied a GANs model met Nash equilibrium in a maxmini game and the model was good. However, it was only an illusion of a successful model because of the full mode collapse problem. The problem was common in vanilla GANs when it was applied on complicated data.

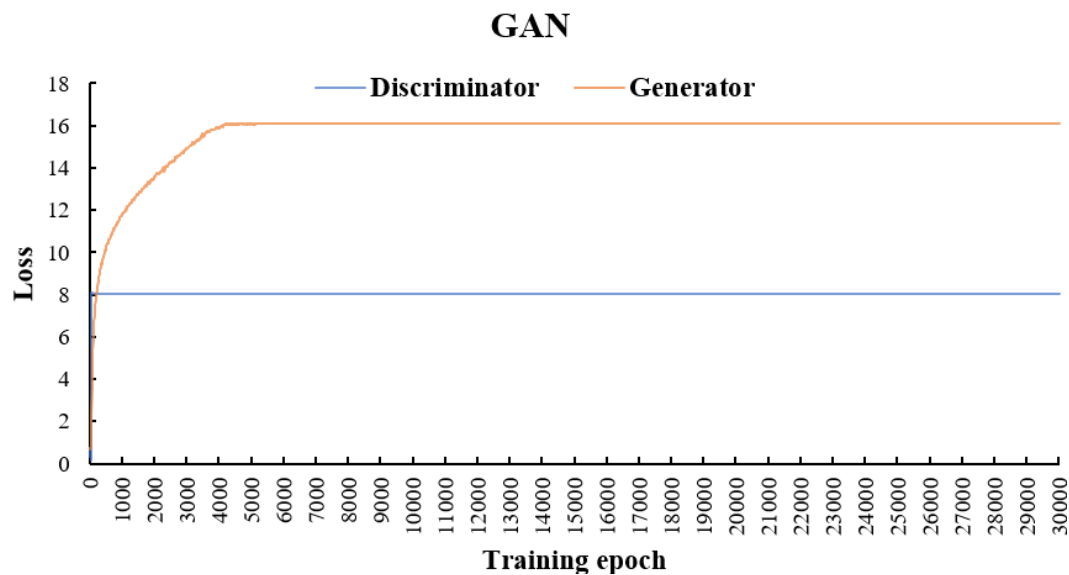


Figure 4-19. The losses of vanilla GANs trained on NLVR

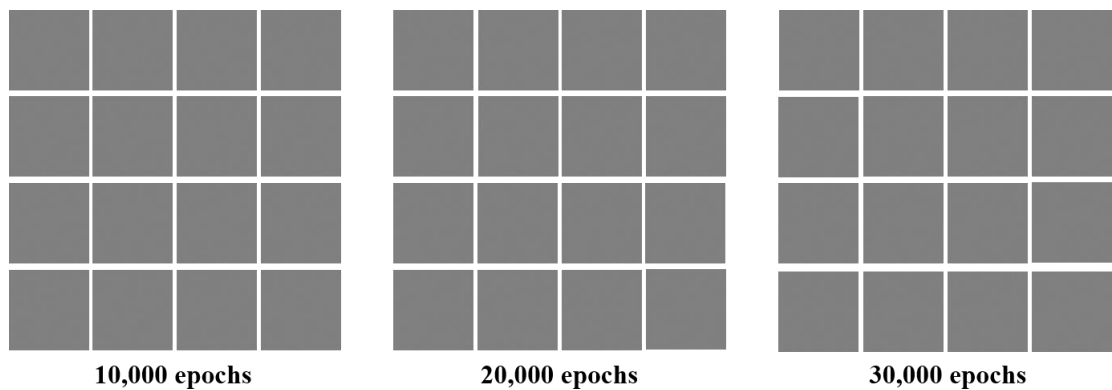


Figure 4-20. Geometric graphics generated by vanilla GANs of the project

## 4.2.2 DCGAN on NLVR

To accommodate colorful images with 64x64 pixels, the structures of the discriminator and the generator for MNIST were modified and the full structures were demonstrated in Figure 4-21.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
conv2d_35 (Conv2D)	(None, 32, 32, 64)	4864	dense_23 (Dense)	(None, 16384)	1654784
leaky_re_lu_36 (LeakyReLU)	(None, 32, 32, 64)	0	reshape_7 (Reshape)	(None, 4, 4, 1024)	0
dropout_16 (Dropout)	(None, 32, 32, 64)	0	up_sampling2d_11 (UpSampling)	(None, 8, 8, 1024)	0
conv2d_36 (Conv2D)	(None, 16, 16, 128)	204928	conv2d_39 (Conv2D)	(None, 8, 8, 512)	13107712
batch_normalization_28 (Batch Normalization)	(None, 16, 16, 128)	512	batch_normalization_31 (Batch Normalization)	(None, 8, 8, 512)	2048
leaky_re_lu_37 (LeakyReLU)	(None, 16, 16, 128)	0	activation_14 (Activation)	(None, 8, 8, 512)	0
dropout_17 (Dropout)	(None, 16, 16, 128)	0	up_sampling2d_12 (UpSampling)	(None, 16, 16, 512)	0
conv2d_37 (Conv2D)	(None, 8, 8, 256)	819456	conv2d_40 (Conv2D)	(None, 16, 16, 256)	3277056
batch_normalization_29 (Batch Normalization)	(None, 8, 8, 256)	1024	batch_normalization_32 (Batch Normalization)	(None, 16, 16, 256)	1024
leaky_re_lu_38 (LeakyReLU)	(None, 8, 8, 256)	0	activation_15 (Activation)	(None, 16, 16, 256)	0
dropout_18 (Dropout)	(None, 8, 8, 256)	0	up_sampling2d_13 (UpSampling)	(None, 32, 32, 256)	0
conv2d_38 (Conv2D)	(None, 4, 4, 512)	3277312	conv2d_41 (Conv2D)	(None, 32, 32, 128)	819328
batch_normalization_30 (Batch Normalization)	(None, 4, 4, 512)	2048	batch_normalization_33 (Batch Normalization)	(None, 32, 32, 128)	512
leaky_re_lu_39 (LeakyReLU)	(None, 4, 4, 512)	0	activation_16 (Activation)	(None, 32, 32, 128)	0
dropout_19 (Dropout)	(None, 4, 4, 512)	0	up_sampling2d_14 (UpSampling)	(None, 64, 64, 128)	0
flatten_7 (Flatten)	(None, 8192)	0	conv2d_42 (Conv2D)	(None, 64, 64, 64)	204864
dense_22 (Dense)	(None, 1)	8193	batch_normalization_34 (Batch Normalization)	(None, 64, 64, 64)	256
Total params: 4,318,337			activation_17 (Activation)	(None, 64, 64, 64)	0
Trainable params: 4,316,545			conv2d_43 (Conv2D)	(None, 64, 64, 3)	4803
Non-trainable params: 1,792			activation_18 (Activation)	(None, 64, 64, 3)	0
			Total params: 19,072,387		
			Trainable params: 19,070,467		
			Non-trainable params: 1,920		

Figure 4-21. The structures of DCGAN model for colorful images with 64×64 pixels

Figure 4-22 demonstrated the losses of DCGAN. Most of the discriminator's loss was near to 0 and the generator's loss had huge variance and spiking. The behavior implied the failure of the model. Furthermore, the loss of the generator climbed along with the training progressing overall. The synthesised images which were presented in Figure 4-23 showed the generator still learned a part of the distribution of the training dataset. It can generate different combinations of squares but was not good at producing triangles and circles. Nevertheless, the partial mode collapse problem was observed in Figure 4-23. We can discover that only a few types of geometric combinations were produced by the generator.

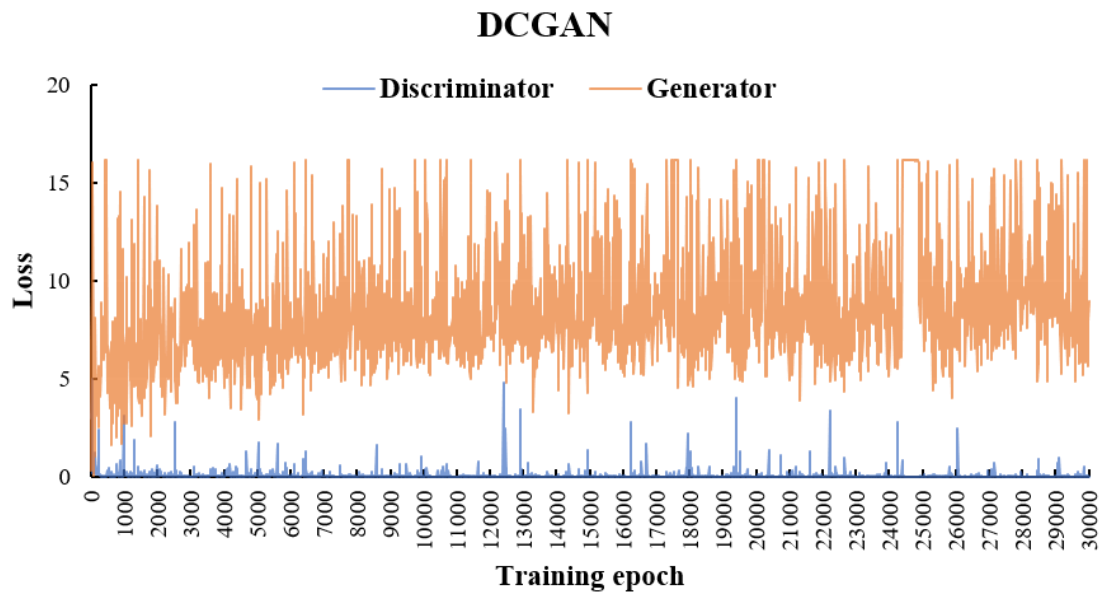


Figure 4-22. The losses of DCGAN trained on NLVR

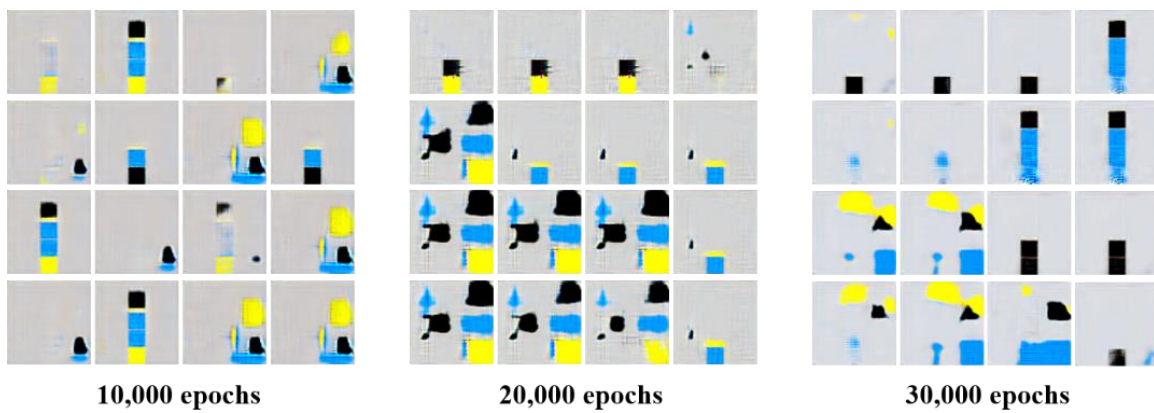


Figure 4-23. Geometric graphics generated by DCGAN of the project

### 4.2.3 WGAN on NLVR

WGAN utilized the similar structures of DCGAN but there was no sigmoid activation function in the discriminator. The full structure was presented in Figure 4-24.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
conv2d_20 (Conv2D)	(None, 32, 32, 64)	4864	dense_19 (Dense)	(None, 16384)	1654784
leaky_re_lu_28 (LeakyReLU)	(None, 32, 32, 64)	0	reshape_5 (Reshape)	(None, 4, 4, 1024)	0
dropout_8 (Dropout)	(None, 32, 32, 64)	0	batch_normalization_19 (Batch Normalization)	(None, 4, 4, 1024)	4096
conv2d_21 (Conv2D)	(None, 16, 16, 128)	204928	activation_6 (Activation)	(None, 4, 4, 1024)	0
batch_normalization_16 (Batch Normalization)	(None, 16, 16, 128)	512	up_sampling2d_5 (UpSampling2D)	(None, 8, 8, 1024)	0
leaky_re_lu_29 (LeakyReLU)	(None, 16, 16, 128)	0	conv2d_24 (Conv2D)	(None, 8, 8, 512)	13107712
dropout_9 (Dropout)	(None, 16, 16, 128)	0	batch_normalization_20 (Batch Normalization)	(None, 8, 8, 512)	2048
conv2d_22 (Conv2D)	(None, 8, 8, 256)	819456	activation_7 (Activation)	(None, 8, 8, 512)	0
batch_normalization_17 (Batch Normalization)	(None, 8, 8, 256)	1024	up_sampling2d_6 (UpSampling2D)	(None, 16, 16, 512)	0
leaky_re_lu_30 (LeakyReLU)	(None, 8, 8, 256)	0	conv2d_25 (Conv2D)	(None, 16, 16, 256)	3277056
dropout_10 (Dropout)	(None, 8, 8, 256)	0	batch_normalization_21 (Batch Normalization)	(None, 16, 16, 256)	1024
conv2d_23 (Conv2D)	(None, 4, 4, 512)	3277312	activation_8 (Activation)	(None, 16, 16, 256)	0
batch_normalization_18 (Batch Normalization)	(None, 4, 4, 512)	2048	up_sampling2d_7 (UpSampling2D)	(None, 32, 32, 256)	0
leaky_re_lu_31 (LeakyReLU)	(None, 4, 4, 512)	0	conv2d_26 (Conv2D)	(None, 32, 32, 128)	819328
dropout_11 (Dropout)	(None, 4, 4, 512)	0	batch_normalization_22 (Batch Normalization)	(None, 32, 32, 128)	512
flatten_5 (Flatten)	(None, 8192)	0	activation_9 (Activation)	(None, 32, 32, 128)	0
dense_18 (Dense)	(None, 1)	8193	up_sampling2d_8 (UpSampling2D)	(None, 64, 64, 128)	0
Total params: 4,318,337			conv2d_27 (Conv2D)	(None, 64, 64, 3)	9603
Trainable params: 4,316,545			activation_10 (Activation)	(None, 64, 64, 3)	0
Non-trainable params: 1,792			Total params: 18,876,163		
			Trainable params: 18,872,323		
			Non-trainable params: 3,840		

Figure 4-24. The structures of WGAN model for colorful images with 64×64 pixels

Figure 4-25 illustrated the losses of WGAN. The loss of the generator went down consistently. In the end, it vibrated at a little region from -0.0001 to 0.0001. The loss of the discriminator increased gently as the training progressed. After 12000 epochs, both these two losses converged.

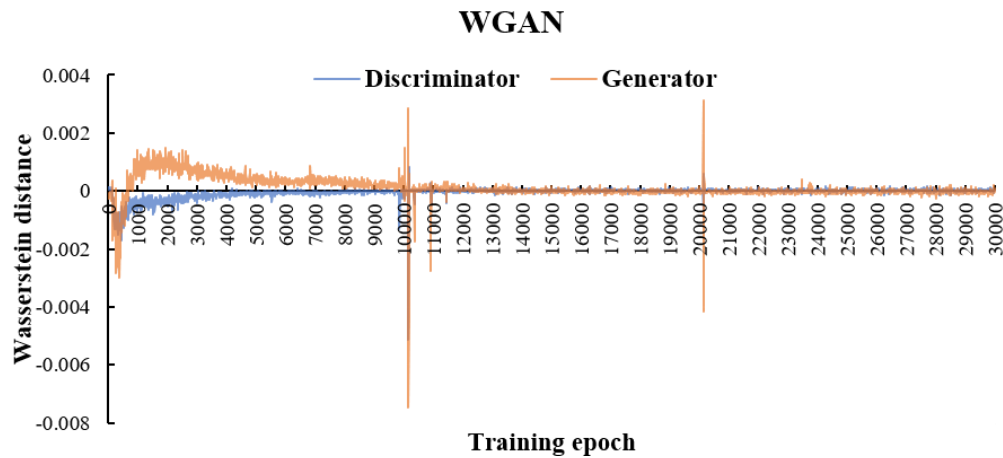


Figure 4-25. The losses of WGAN trained on NLVR



Figure 4-26 presented the synthesised images from WGAN. In the beginning of the training, the background was rough, and the boundaries of geometric graphics were not sharp. After 20,000 epochs, the synthesised images were clearer, and their background was even. The mode collapse problem was not observed in WGAN. The generative model can produce various types of geometric images rather than only a few types of geometric combinations.

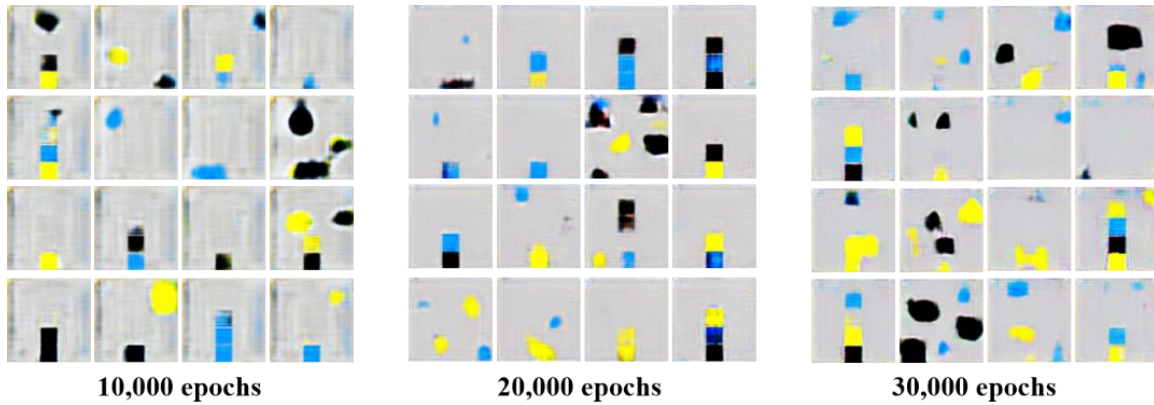


Figure 4-26. Geometric graphics generated by WGAN of the project

#### 4.2.4 WGAN-GP on NLVR

The structures of WGAN-GP were similar to WGAN's. Nevertheless, there was no batch normalization layer in the discriminator. The full structure was presented in Figure 4-27.

Discriminator			Generator		
Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 32, 32, 64)	4864	dense_12 (Dense)	(None, 16384)	1654784
leaky_re_lu_1 (LeakyReLU)	(None, 32, 32, 64)	0	reshape_3 (Reshape)	(None, 4, 4, 1024)	0
dropout_1 (Dropout)	(None, 32, 32, 64)	0	up_sampling2d_1 (UpSampling2D)	(None, 8, 8, 1024)	0
conv2d_7 (Conv2D)	(None, 16, 16, 128)	204928	conv2d_6 (Conv2D)	(None, 8, 8, 512)	13107712
leaky_re_lu_2 (LeakyReLU)	(None, 16, 16, 128)	0	batch_normalization_8 (Batch Normalization)	(None, 8, 8, 512)	2048
dropout_2 (Dropout)	(None, 16, 16, 128)	0	activation_1 (Activation)	(None, 8, 8, 512)	0
conv2d_8 (Conv2D)	(None, 8, 8, 256)	819456	up_sampling2d_2 (UpSampling2D)	(None, 16, 16, 512)	0
leaky_re_lu_3 (LeakyReLU)	(None, 8, 8, 256)	0	conv2d_7 (Conv2D)	(None, 16, 16, 256)	3277056
dropout_3 (Dropout)	(None, 8, 8, 256)	0	batch_normalization_9 (Batch Normalization)	(None, 16, 16, 256)	1024
conv2d_9 (Conv2D)	(None, 4, 4, 512)	3277312	activation_2 (Activation)	(None, 16, 16, 256)	0
leaky_re_lu_4 (LeakyReLU)	(None, 4, 4, 512)	0	up_sampling2d_3 (UpSampling2D)	(None, 32, 32, 256)	0
dropout_4 (Dropout)	(None, 4, 4, 512)	0	conv2d_8 (Conv2D)	(None, 32, 32, 128)	819328
flatten_1 (Flatten)	(None, 8192)	0	batch_normalization_10 (Batch Normalization)	(None, 32, 32, 128)	512
dense_2 (Dense)	(None, 1)	8193	activation_3 (Activation)	(None, 32, 32, 128)	0
Total params: 4,314,753			up_sampling2d_4 (UpSampling2D)	(None, 64, 64, 128)	0
Trainable params: 4,314,753			conv2d_9 (Conv2D)	(None, 64, 64, 64)	204864
Non-trainable params: 0			batch_normalization_11 (Batch Normalization)	(None, 64, 64, 64)	256
			activation_4 (Activation)	(None, 64, 64, 64)	0
			conv2d_10 (Conv2D)	(None, 64, 64, 3)	4803
			activation_5 (Activation)	(None, 64, 64, 3)	0
			Total params: 19,072,387		
			Trainable params: 19,070,467		
			Non-trainable params: 1,920		

Figure 4-27. The structures of WGAN-GP model for colorful images with 64×64 pixels

Figure 4-28 demonstrated the changing of two neural networks' losses. The loss of the discriminator increased and kept in a certain region after 1000 training epochs. On the other hand, the loss of the generator decreased rapidly in the beginning and vibrated after 500 training epochs. Overall, two neural networks competed to make WGAN-GP be improved. The synthesised images were presented in Figure 4-29. After 10,000 epochs, the images had even background and some geometric graphic already had sharp boundaries. It illustrated that the convergence speed of WGAN-GP was faster than the one of WGAN. In the 30,000 epochs, WGAN-GP can generate compelling images which contained simple square combinations. These images were saturated and clear.

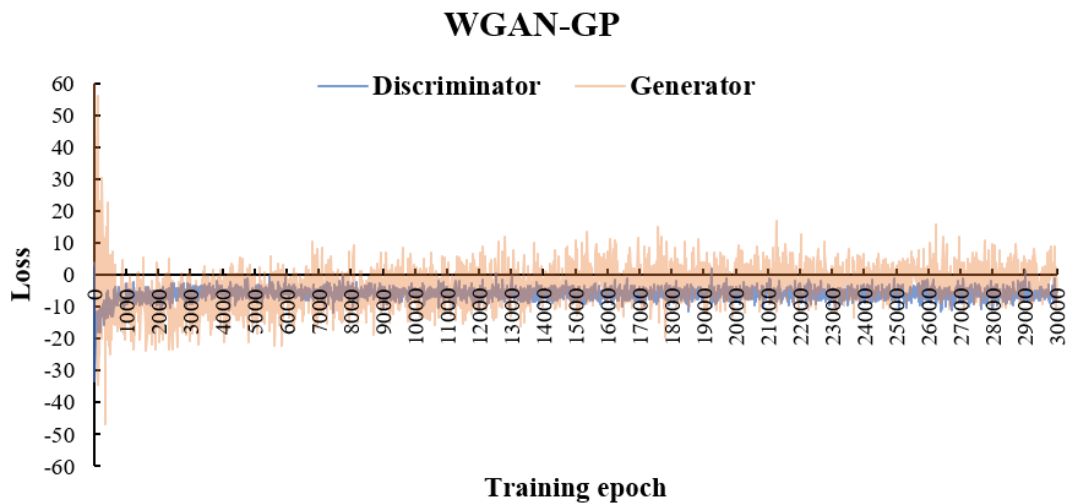


Figure 4-28. The losses of WGAN-GP trained on NLVR

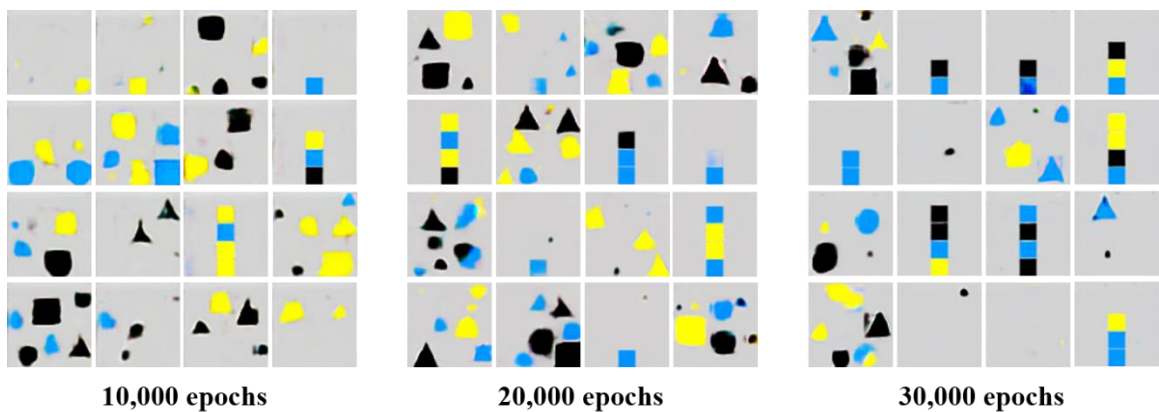


Figure 4-29. Geometric graphics generated by WGAN-GP of the project

## 4.2.5 Model Comparisons on NLVR

As explained in section 4.1.5 , we divided these four models into two groups. The first group was vanilla GANs and DCGAN and the other was WGAN and WGAN-GP.

Figure 4-30 demonstrated the JS divergence of vanilla GANs and DCGAN. The former saturated at the beginning of training. It meant that the model stopped progressing. The latter had surges sometimes and did not change frequently.

The result of the vanilla GANs was not delightful. The full mode collapse problem occurred in the early stage of the training. By observing its losses, the Nash equilibrium of the model was met at the beginning. However, the fact was that the generator only synthesised images with a whole gray color to fool the discriminator. By observing the training data further, we can find that most images' background accounts for over 50% of the total. It could lead the generator to only learn how to synthesis background images.

In terms of the result of DCGAN, it illustrated that the generator can catch a part of real data's distribution and generated meaningful images. DCGAN can construct more reliable generative models than vanilla GANs. However, the partial mode collapse problem still happened in DCGAN. Figure 4-23 demonstrated a bunch of images generated by DCGAN, and only a few types of images were synthesised.

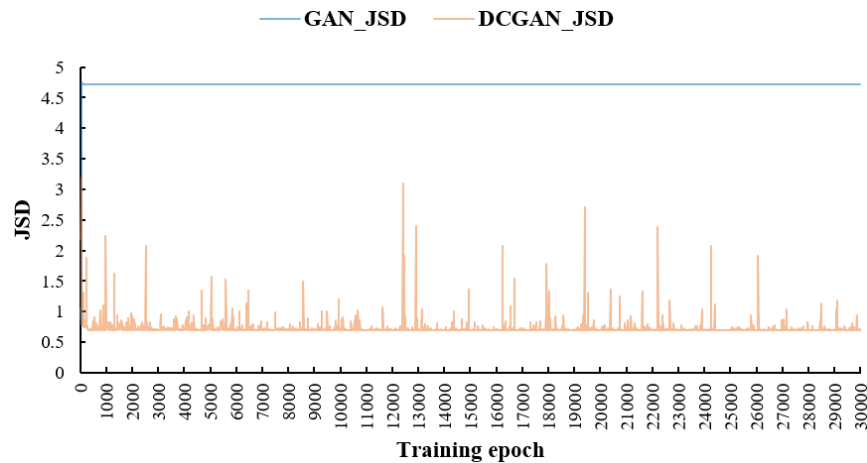


Figure 4-30. The JS divergences of the vanilla GANs and DCGAN in NLVR

Because of the full mode collapse problem, vanilla GANs got terrible FID at every training stage as shown in Table 4-3. Furthermore, the generator did not make any progress along with the training progressing because its FID did not change after 20,000 epochs. The performance of DCGAN was better than vanilla GANs according to FID. However, its figure was also not good enough to fool human. In the subjective evaluation, vanilla GANs and DCGAN gained 0.00% and 13.89% error rate respectively.

FID			
	10,000 epochs	20,000 epochs	30,000 epochs
Vanilla GANS	346	314	314
DCGAN	269	294	242

Table 4-3. FID of vanilla GANs and DCGAN which were trained on NLVR

Figure 4-31 demonstrated the generated images of DCGAN and the corresponding similar images of the training dataset. Although some generated images looked like the real images, they did not match perfectly. In terms of vanilla GANs, the similarity comparison was not presented because of the failure of the training.

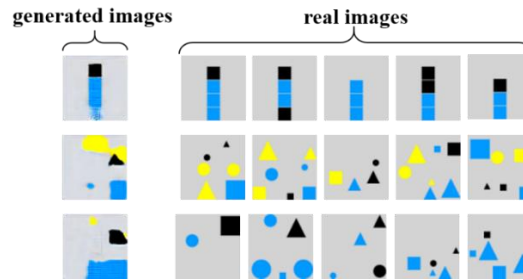


Figure 4-31. The image similarity comparison of DCGAN in NLVR

Figure 4-32 presented the Wasserstein distance of WGAN and WGAN-GP. The figures for these two models went down as the models experienced more training epochs. Table 4-4 illustrated the FID of the generators in these two models. It was clear that their FID decreased along with the training progressing. Form Figure 4-32 and Table 4-4, we also observed a positive correlation between the Wasserstein distance and the images' quality. The result of the questionnaire revealed that WGAN and WGAN-GP got 5.56% and 19.44% error rate on discriminating synthesised geometric graphics. Hence, WGAN-GP had better performance than WGAN.

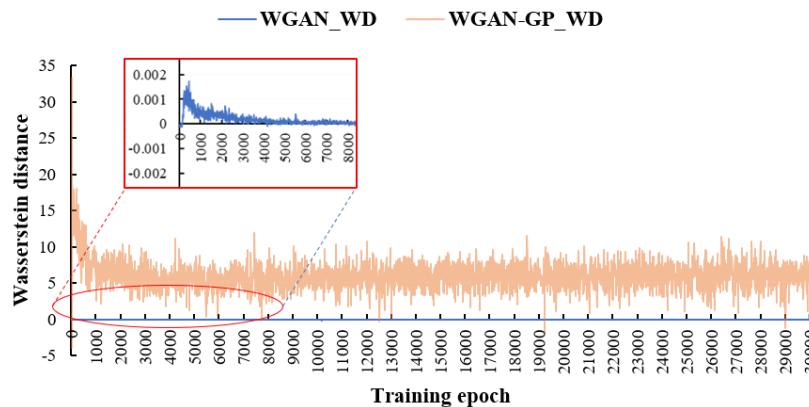


Figure 4-32. The Wasserstein distances of WGAN and WGAN-GP in NLVR

FID			
	10,000 epochs	20,000 epochs	30,000 epochs
WGAN	278	272	256
WGAN-GP	195	112	109

Table 4-4. FID of WGAN and WGAN-GP which were trained on NLVR

The similarity comparisons between the generated and real images were presented in Figure 4-33 and Figure 4-34. Both WGAN and WGAN-GP can synthesis new images which were different from the images of the training dataset.

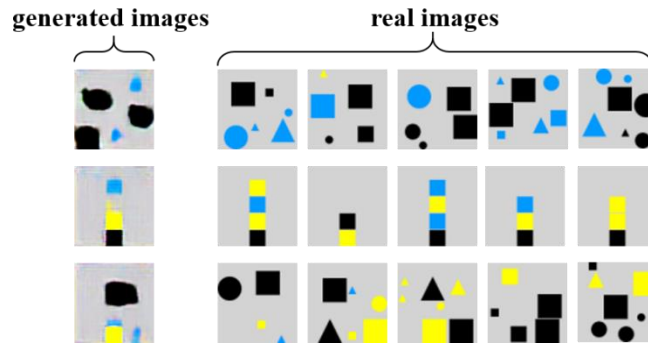


Figure 4-33. The image similarity comparison of WGAN in NLVR

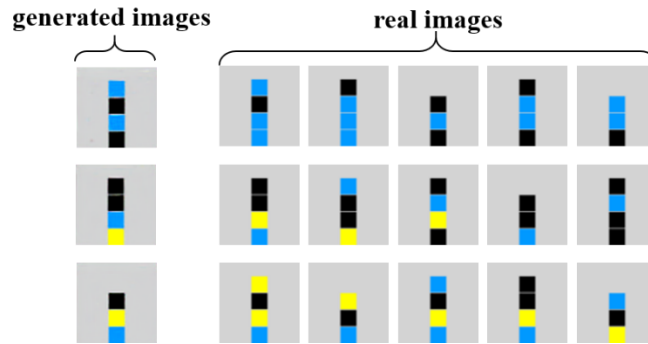


Figure 4-34. The image similarity comparison of WGAN-GP in NLVR

The performance of WGAN-GP was the best based on FID and the questionnaire. Both WGAN and WGAN-GP did not suffer from the mode collapse problem which occurred in vanilla GANs and DCGAN. And then, the correlation between the Wasserstein distance and the quality of the images was better than the one between the JS divergence and the quality of the images. Furthermore, WGAN and WGAN-GP were more stable than vanilla GANs and DCGAN in the training by observing the changing of their generator losses.

### 4.3 The 3<sup>rd</sup> Experiment: Generate Flower Images

This experiment employed Oxford-102 flowers as the training dataset to build generative models. All models were trained with 64 batch size and 40,000 epochs.

#### 4.3.1 Vanilla GANs on Oxford-102 Flowers

As presented in Figure 4-35, the loss of the generator declined in the beginning of the training and started to explode after 7000 epochs. In the meantime, the loss of the discriminator slowly decreased. From the observation, the discriminator cannot provide an effective guide to the generator. Hence, the loss of the generator consistently grew when the model experienced more training epochs.

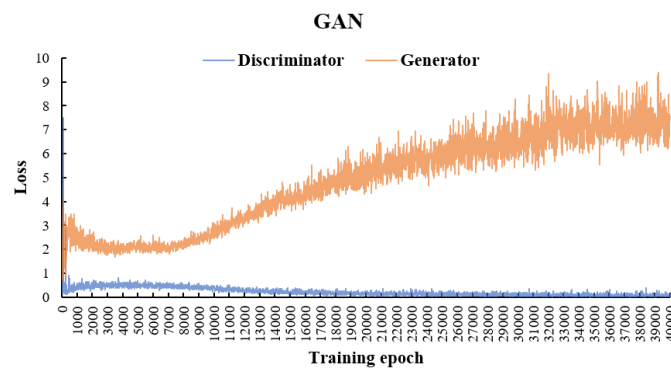


Figure 4-35. The losses of vanilla GANs trained on Oxford-102 flowers

Figure 4-36 demonstrated synthesised flowers in different training stages from 10,000 to 40,000 epochs. Unfortunately, the results were not well, and the generator cannot capture the detail of the training dataset. The generated images had very blur shape along with a tremendous amount of noise. All in all, vanilla GANs learned a little knowledge in the beginning but the loss of generator exploded too quickly to make further progress.

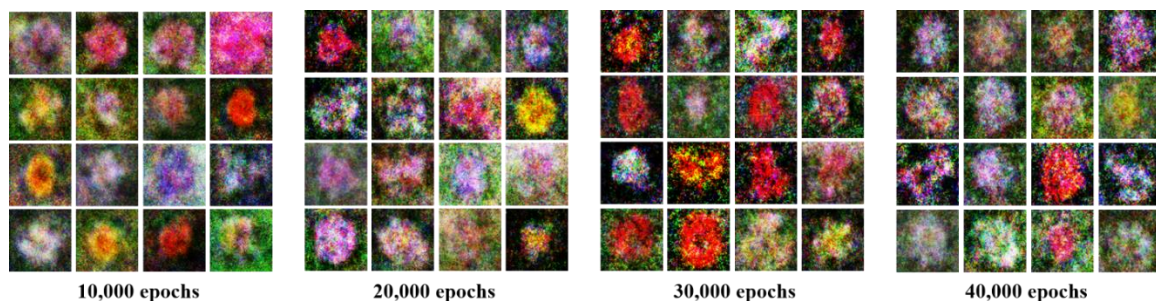


Figure 4-36. Flower images generated by vanilla GANs of the project



### 4.3.2 DCGAN on Oxford-102 Flowers

As shown in Figure 4-37, the loss of the discriminator declined. It meant that the distance between the distribution of training dataset and the distribution of synthesised images was shortened consistently. In contrast, the loss of the generator climbed gently along with the training progressing.

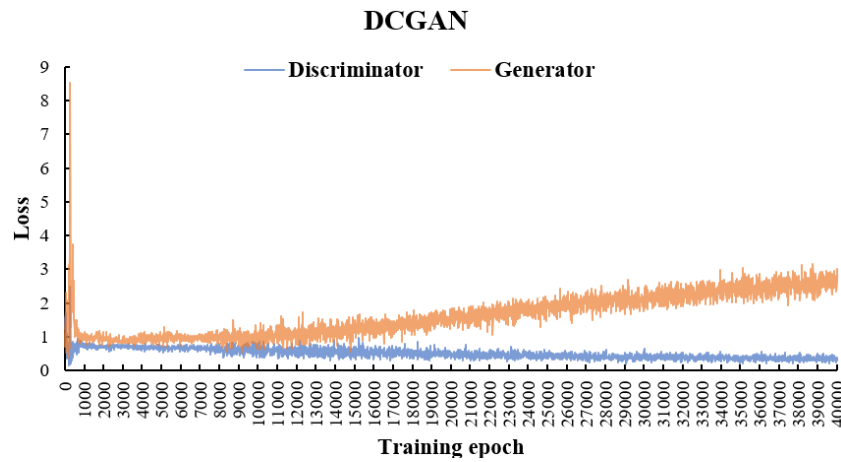


Figure 4-37. The losses of DCGAN trained on Oxford-102 flowers

Comparing with vanilla GANs, DCGAN can produce more realistic flower images which were shown in Figure 4-38. The boundaries of the synthesised images were sharper, and the images presented various types of flower shapes. Some of them displayed more flower detail such as petals and anthers. Furthermore, the depth of field can be observed in the images.

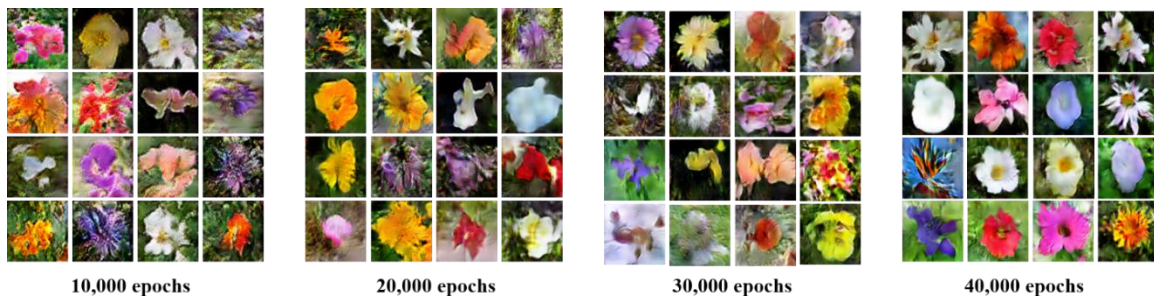


Figure 4-38. Flower images generated by DCGAN of the project

### 4.3.3 WGAN on Oxford-102 Flowers

Figure 4-39 showed the losses of WGAN. Both losses of the discriminator and the generator exploded toward the negative direction. It was not a good phenomenon in the training process because it meant that the losses did not converge.

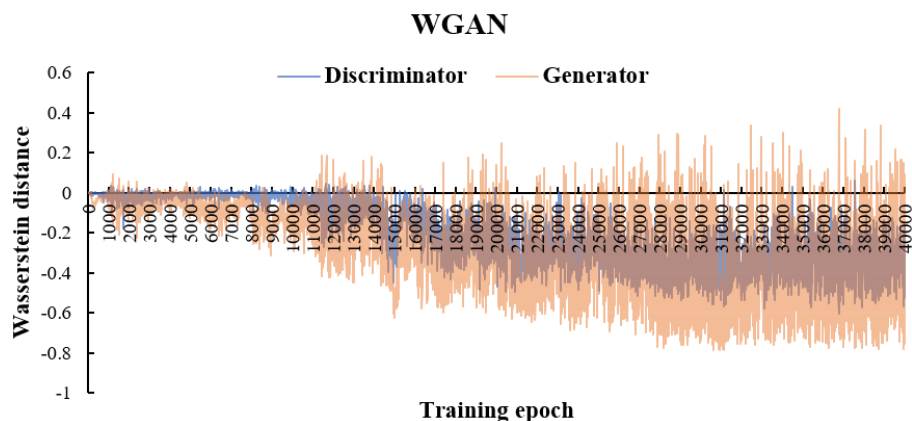


Figure 4-39. The losses of WGAN trained on Oxford-102 flowers

The exploding losses reflected the awful quality of generated images which were shown in Figure 4-40. Before 10,000 epochs, the model kept its variety. However, the models only generated one or two types of images when the model was trained more times. The mode collapse problem occurred.

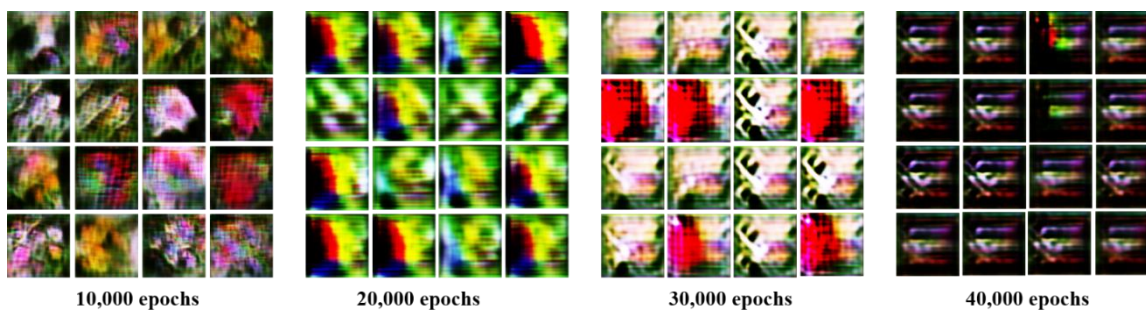


Figure 4-40. Flower images generated by WGAN of the project



### 4.3.4 WGAN-GP on Oxford-102 Flowers

The losses of WGAN-GP were illustrated in Figure 4-41. The loss of the discriminator increased rapidly and vibrated in a small range. In the meanwhile, the figure for the generator went down quickly and reached the lowest point. And then, it climbed slowly along with the vibration.

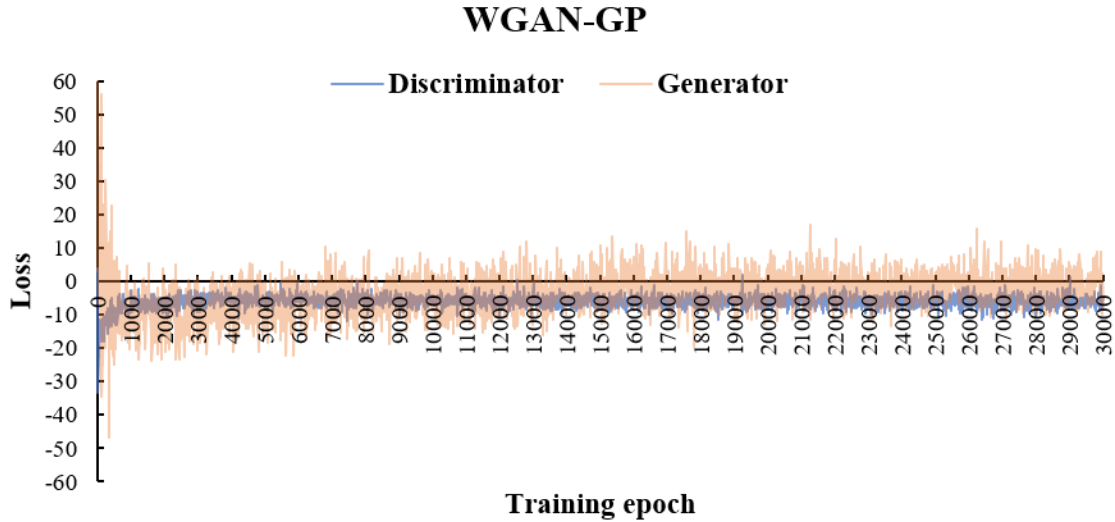


Figure 4-41. The losses of WGAN-GP trained on Oxford-102 flowers

The synthesised images of WGAN-GP were shown in Figure 4-42. We can find that the quality of the synthesised images was improved consistently. As the results of DCGAN, the images presented variant flower shapes and the detail of flowers.

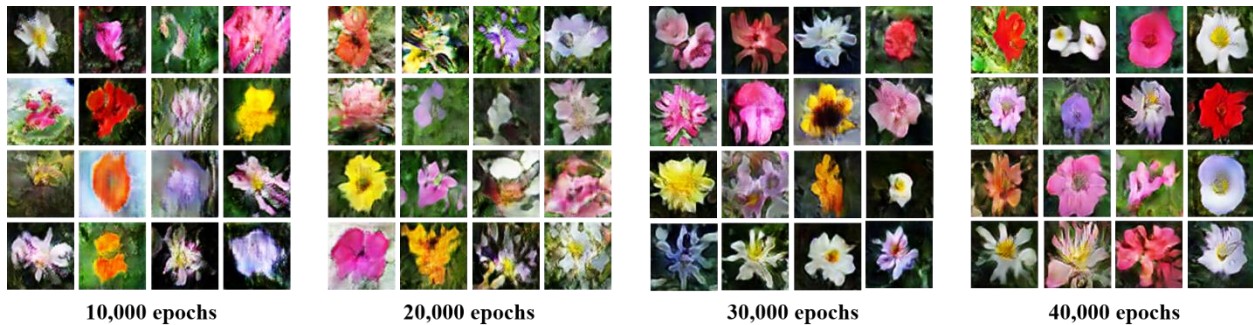


Figure 4-42. Flower images generated by WGAN-GP of the project

### 4.3.5 Model Comparisons on Oxford-102 Flowers

Figure 4-43 illustrated the JS divergence of vanilla GANs and DCGAN. The JS divergence of these two models went down consistently. In general, the quality of synthesised images was better if a GANs model had a small JS divergence. However, the reducing of the JS divergence in vanilla GANs was not relevant to the quality of its synthesised images positively. In fact, the quality of the images was getting worse. On the other hand, the JS divergence of DCGAN had a positive correlation with the quality of the synthesised images.

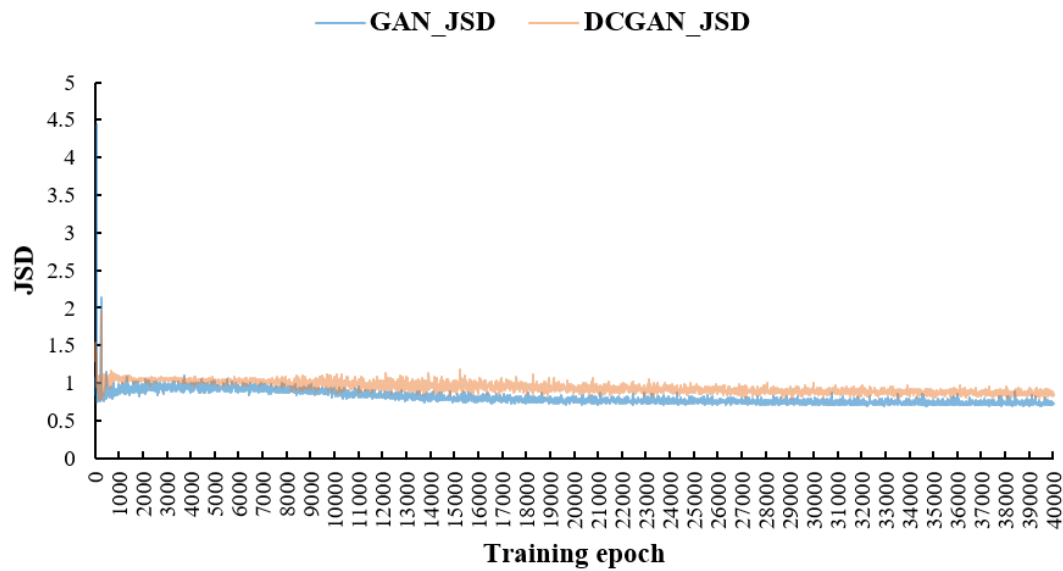


Figure 4-43. The JS divergences of vanilla GANs and DCGAN in Oxford-102 flowers

Table 4-5 demonstrated the objective evaluation of vanilla GANs and DCGAN. The figure for vanilla GANs was getting bigger along with the training progressing. By contrast, FID of DCGAN declined along with the training epoch increasing. In terms of the subjective evaluation, the images of vanilla GANs were too terrible to fool the participants. Hence, it got 0.00% error rate on human judgment. On the other hand, DCGAN can generate better flower images to mislead the participants and it yielded a 27.78% error rate.

FID				
	10,000 epochs	20,000 epochs	30,000 epochs	40,000 epochs
Vanilla GANs	331	341	345	353
DCGAN	144	102	90	79

Table 4-5. FID of vanilla GANs and DCGAN which were trained on Oxfor-102 flowers

Figure 4-44 illustrated the similarity comparison of DCGAN. It showed that DCGAN can generate new flower images. The similarity comparison of vanilla GANs was not presented because of the failure of the training.

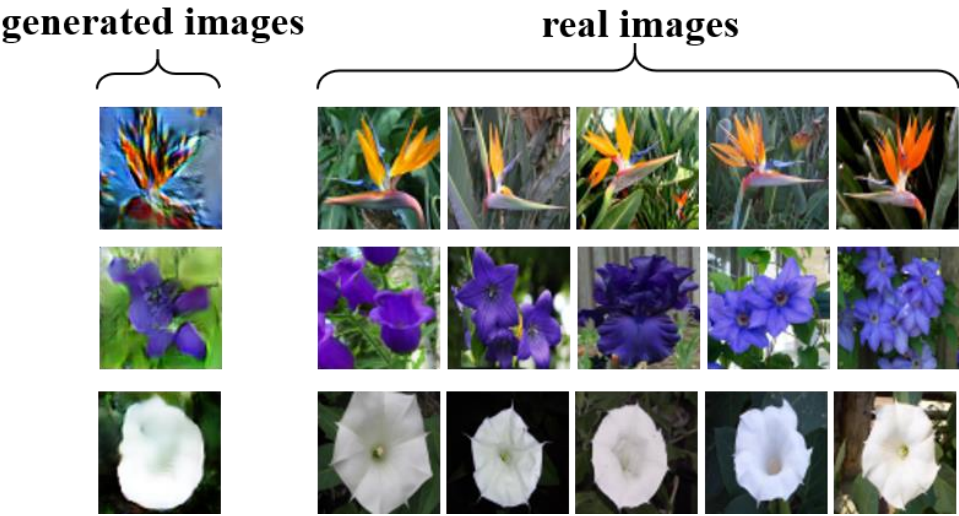


Figure 4-44. The image similarity comparison of DCGAN in Oxford-102 flowers

Figure 4-45 demonstrated the Wasserstein distance of WGAN and WGAN-GP. The figure for WGAN-GP had large amplitude, but it tended to decrease on the whole. In contrast, the figure for WGAN increased consistently. Form the observation, we can find that there was a high correlation between the Wasserstein distance and the quality of the synthesised images. The performance of WGAN was getting worse along with the increment of its Wasserstein distance and WGAN-GP can generate better flower images when its Wasserstein distance declined.

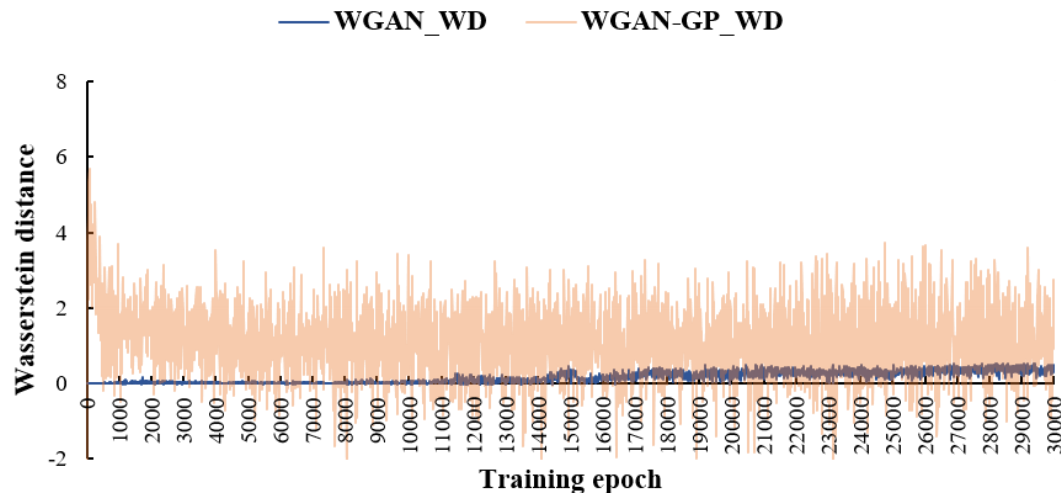


Figure 4-45. The Wasserstein distances of WGAN and WGAN-GP in Oxford-102 flowers

The result of the objective evaluation was positively relevant to the result of the questionnaire. No participant did the wrong judgment on the images which were synthesised by WGAN. However, WGAN-GP got 11.11% error rate on human judgment.

FID				
	10,000 epochs	20,000 epochs	30,000 epochs	40,000 epochs
WGAN	272	364	386	363
WGAN-GP	139	120	115	104

Table 4-6. FID of WGAN and WGAN-GP which were trained on Oxfor-102 flowers

Figure 4-46 demonstrated the images which were synthesised by WGAN-GP were different from the images of the training dataset.

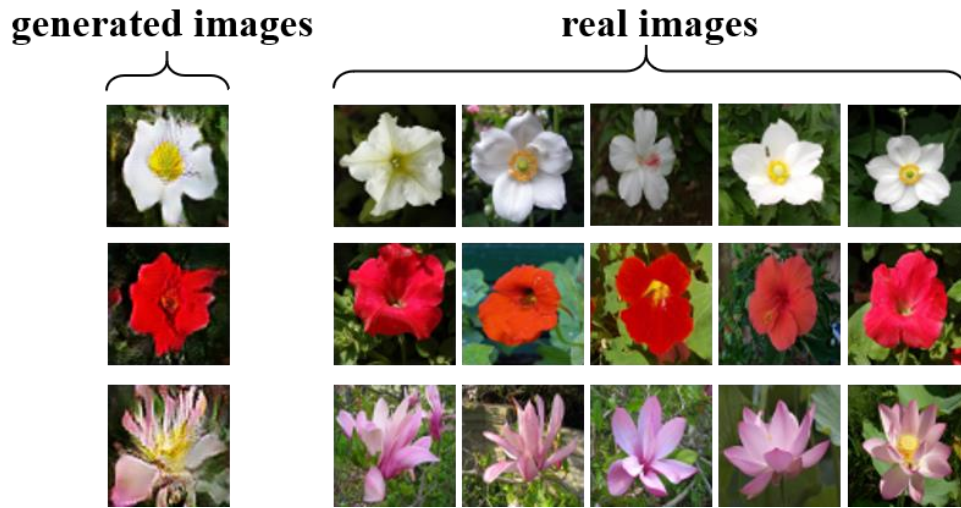


Figure 4-46. The image similarity comparison of WGAN-GP in NLVR

In the third experiment, the synthesised images by DCGAN got the smallest FID and the most participants selected wrong answers in the corresponding question. It was obvious that vanilla GANs which were built by MLP was hard to be applied to the complicated training data. Vanilla GANs only learned a little part of the target distribution as shown in section 4.3.1. WGAN also failed to capture the distribution of the training data. Furthermore, a serious mode collapse happened in WGAN. Conversely, DCGAN and WGAN-GP performed more stably.

## **Chapter 5: Conclusions and Recommendations**

### **5.1 Conclusions**

#### **5.1.1 Problems of Vanilla GANs**

Generative adversarial networks demonstrated advantages on synthesising images. One was that the synthesised images were sharper than traditional generative models. The other was that the speed of generating images was fast. However, the non-convergence problem and the mode collapse problem made vanilla GANs hard to be trained.

From the experiment of generating handwritten digits, the result of vanilla GANs was acceptable and the problems mentioned above were not observed. The reason for vanilla GANs' success could be credited to the dimension of MNIST was small. Hence, the model can capture a large portion of the training data's distribution. However, the problems of vanilla GANs happened when it was applied to more complex training data including NLVR and Oxford-102 flowers. The full mode collapse problem occurred in NLVR. The model only synthesised a type of images. In the experiment of synthesising flower images, non-convergence was observed. From the analysis in section 4.3.1, the loss of vanilla GANs' generator exploded along with the training progressing.

#### **5.1.2 The improvement in GANs training stability**

To adopt the idea of GANs on higher dimensional data, the stability of GANs models needed to be meliorated. In the 3rd experiments, DCGAN improved the stability of the training. The quality of the images synthesised by DCGAN kept improving as the model experienced more training epochs. It also got the best score both in the objective and the subjective evaluation. Nevertheless, the fractional mode collapse was observed in DCGAN in the 2<sup>nd</sup> experiment.

#### **5.1.3 The improvement in Estimating Method**

From the experiments, we can find a high correlation between the Wasserstein distance and the performance of the models. If the Wasserstein distance was reduced, the quality of synthesised images was improved. By contrast, the performance of the model cannot be improved if the Wasserstein distance cannot be diminished. The 3<sup>rd</sup> experiment demonstrated this point effectively. The Wasserstein distance of WGAN in the 3<sup>rd</sup> experiment increased consistently rather than converging. Hence, the generator of WGAN cannot produce compelling flower images. On the other hand, the JS divergence was not consistent with the performance of the models.

In theatrically, WGAN can improve stability of the training process and alleviate the mode collapse problem. However, WGAN failed to capture the distribution of Oxford-102 flowers in the 3<sup>rd</sup> experiment. The main reason for WGAN's unstable could be caused by the parameter  $c$  in weight clipping. In the original paper (Arjovsky et al., 2017), the author said that weight clipping was a temporary method to restrict the discriminator. From the experiments conducted by Gulrajani et al. (2017), if  $c$  was too big, it could cause the loss of the generator to explode.

In contrast to the result of WGAN, WGAN-GP performed stably in every experiment. Wasserstein distance of WGAN-GP decreased consistently in every experiment and it can synthesis better quality images. It could be credited to the gradient penalty which restricts the discriminator to be a 1-Lipschitz function more appropriately.

Comparing with the other three models, WGAN can accommodate different kinds of training dataset and provides stability in the training process in overall.

## 5.2 Recommendations

After conducting the project, there were five recommendations for further research. First, GANs models can be applied on more complex training data such as LSUN (Yu et al., 2015), CelebA (Liu et al., 2015) and ImageNet (Russakovsky et al., 2015). These datasets contain more training data and were full of variety. Next, the training iterations can be increased if the computation power was enough. In general, the model can be improved by increasing the training iterations. It was possible to increase computation power by training the models in a multi-GPU environment. Then, the image size of the training data can be enlarged. In the project, we restricted the size of the images to 64×64 pixels. For instance, the image size of the training can be resized to 128×128 pixels to producing more compelling images. Finally, the standard questionnaire was not efficiency enough because it was hard to gather numerous effective responses. The suggested way was posting the questions on Amazon Mechanical Turk (MTurk) to get more responses.

## Bibliography

- ARJOVSKY, M., CHINTALA, S. & BOTTOU, L. (2017) Wasserstein gan. *arXiv*. Available at: <https://arxiv.org/pdf/1701.07875.pdf> (Accessed: 27 June 2019).
- BANTUM, E. O. C. *et al.* (2017) ‘Machine learning for identifying emotional expression in text: Improving the accuracy of established methods’. *Journal of Technology in Behavioral Science*, 2 (1), pp.21-27.
- BARRATT, S. & SHARMA, R. (2018) A note on the inception score. *arXiv*. Available at: <https://arxiv.org/pdf/1801.01973.pdf> (Accessed: 3 June 2019).
- BODNAR, C. (2018) Text to image synthesis using generative adversarial networks. *arXiv*. Available at: <https://arxiv.org/pdf/1805.00676.pdf> (Accessed: 15 June 2019).
- BOURLARD, H. & KAMP, Y. (1988) ‘Auto-association by multilayer perceptrons and singular value decomposition’. *Biological cybernetics*, 59 (4-5), pp.291-294.
- CHOLLET, F. (2015). Keras. Available at: <https://github.com/fchollet/keras> (Accessed: 28 May 2019).
- CIREŞAN, D., MEIER, U. & SCHMIDHUBER, J. (2012) Multi-column deep neural networks for image classification. *arXiv*. Available at: <https://arxiv.org/pdf/1202.2745.pdf> (Accessed: 12 June 2019).
- DENG, L. (2012) ‘The MNIST database of handwritten digit images for machine learning research [best of the web]’. *IEEE Signal Processing Magazine*, 29 (6), pp.141-142.
- DOWSON, D. C. & LANDAU, B. V. (1982) ‘The Fréchet distance between multivariate normal distributions’. *Journal of multivariate analysis*, 12 (3), pp.450-455.
- DUCHI, J., HAZAN, E. & SINGER, Y. (2011) ‘Adaptive subgradient methods for online learning and stochastic optimization’. *Journal of Machine Learning Research*, 12(Jul), pp.2121-2159.
- FREY, B. J., HINTON, G. E. & DAYAN, P. (1996) ‘Does the wake-sleep algorithm produce good density estimators?’. In *NIPS*, pp.661-667.
- GOODFELLOW, I. (2016) NIPS 2016 tutorial: Generative adversarial networks. *arXiv*. Available at: <https://arxiv.org/pdf/1701.00160.pdf> (Accessed: 27 May 2019).
- GOODFELLOW, I. *et al.* (2014) ‘Generative adversarial nets’. In *NIPS*, pp.2672-2680.
- GRAVES, A. & SCHMIDHUBER, J. (2009) ‘Offline handwriting recognition with multidimensional recurrent neural networks’. In *NIPS*, pp.545-552.
- GULRAJANI, I. *et al.* (2017) ‘Improved training of wasserstein gans’. In *NIPS*, pp.5767-5777.
- HEUSEL, M. *et al.* (2017) ‘Gans trained by a two time-scale update rule converge to a local nash equilibrium’. In *NIPS*, pp.6626-6637.

- HINTON, G. E. & SALAKHUTDINOV, R. R. (2006) ‘Reducing the dimensionality of data with neural networks’. *science*, 313 (5786), pp.504-507.
- HINTON, G. E. & ZEMEL, R. S. (1994) ‘Autoencoders, minimum description length and Helmholtz free energy’. In *NIPS*, pp.3-10.
- HOCHREITER, S. & SCHMIDHUBER, J. (1997) ‘Long short-term memory’. *Neural computation*, 9 (8), pp.1735-1780.
- HU, J., SHEN, L. & SUN, G. (2018) ‘Squeeze-and-excitation networks’. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp.7132-7141.
- HUANG, H., YU, P. S. & WANG, C. (2018) An introduction to image synthesis with generative adversarial nets. *arXiv*. Available at: <https://arxiv.org/pdf/1803.04469.pdf> (Accessed: 27 June 2019).
- JAIN, A. K., MAO, J. & MOHIUDDIN, K. M. (1996) ‘Artificial neural networks: A tutorial’. *Computer*, 3, pp.31-44.
- KARRAS, T., AILA, T., LAINE, S. & LEHTINEN, J. (2017) Progressive growing of gans for improved quality, stability, and variation. *arXiv*. Available at: <https://arxiv.org/pdf/1710.10196.pdf> (Accessed: 27 May 2019).
- KINGMA, D. P. & BA, J. (2014) Adam: A method for stochastic optimization. *arXiv*. Available at: <https://arxiv.org/pdf/1412.6980.pdf> (Accessed: 27 June 2019).
- KINGMA, D. P. *et al.* (2016) Improved variational inference with inverse autoregressive flow. In *NIPS*, pp.4743-4751.
- KINGMA, D. P. & WELLING, M. (2013) Auto-encoding variational bayes. *arXiv*. Available at: <https://arxiv.org/pdf/1312.6114.pdf> (Accessed: 29 May 2019).
- LECUN, Y. & BENGIO, Y. (1995) ‘Convolutional networks for images, speech, and time series’. *The handbook of brain theory and neural networks*, 3361 (10).
- LECUN, Y., BENGIO, Y. & HINTON, G. (2015) ‘Deep learning’. *nature*, 521 (7553), pp.436.
- LECUN, Y., BOTTOU, L., BENGIO, Y. & HAFNER, P. (1998) ‘Gradient-based learning applied to document recognition’. *Proceedings of the IEEE*, 86 (11), pp.2278-2324.
- LEE, H. Y. (2017) *Generative Adversarial Network (GAN)*. [PowerPoint presentation to Machine Learning]. Available at: [http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS\\_2017/Lecture/GAN%20\(v11\).pdf](http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2017/Lecture/GAN%20(v11).pdf) (Accessed: 22 May 2019).
- LEE, H. Y. (2018) *Tips for Improving GAN*. [PowerPoint presentation to Machine learning]. Available at: [http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS\\_2018/Lecture/WGAN%20\(v2\).pdf](http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2018/Lecture/WGAN%20(v2).pdf) (Accessed: 27 May 2019).
- LISON, P. (2015) An introduction to machine learning. *Language Technology Group (LTG)*, 1 (35).



- LIU, Z., LUO, P., WANG, X. & TANG, X. (2015) ‘Deep learning face attributes in the wild’. In *Proceedings of the IEEE International Conference on Computer Vision*, pp.3730-3738.
- MAALØE, L., SØNDERBY, C. K., SØNDERBY, S. K. & WINTHER, O. (2016) Auxiliary deep generative models. *arXiv*. Available at: <https://arxiv.org/pdf/1602.05473.pdf> (Accessed: 22 June 2019).
- MAAS, A. L., HANNUN, A. Y. & NG, A. Y. (2013) ‘Rectifier nonlinearities improve neural network acoustic models’. In *Proc. icml*, 30 (1), pp.3.
- MIRZA, M. & OSINDERO, S. (2014) Conditional generative adversarial nets. *arXiv*. Available at: <https://arxiv.org/pdf/1411.1784.pdf> (Accessed: 10 June 2019).
- NAIR, V. & HINTON, G. E. (2010) ‘Rectified linear units improve restricted boltzmann machines’. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp.807-814.
- NILSBACK, M.-E. & ZISSERMAN, A. (2008) ‘Automated flower classification over a large number of classes’. In *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pp.722-729.
- OORD, A. V. D., KALCHBRENNER, N. & KAVUKCUOGLU, K. (2016) Pixel recurrent neural networks. *arXiv*. Available at: <https://arxiv.org/pdf/1601.06759.pdf> (Accessed: 30 May 2019).
- RADFORD, A., METZ, L. & CHINTALA, S. (2015) Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv*. Available at: <https://arxiv.org/pdf/1511.06434.pdf> (Accessed: 12 June 2019).
- REED, S., AKATA, Z., YAN, X., LOGESWARAN, L., SCHIELE, B. & LEE, H. (2016) Generative adversarial text to image synthesis. *arXiv*. Available at: <http://proceedings.mlr.press/v48/reed16.pdf> (Accessed: 29 May 2019).
- REZENDE, D. J., MOHAMED, S. & WIERSTRA, D. (2014) Stochastic backpropagation and approximate inference in deep generative models. *arXiv*. Available at: <https://arxiv.org/pdf/1401.4082.pdf> (Accessed: 22 June 2019).
- RUMELHART, D. E., HINTON, G. E. & WILLIAMS, R. J. (1988) ‘Learning representations by back-propagating errors’. *Cognitive modeling*, 5 (3), pp.1.
- RUSSAKOVSKY, O. *et al.* (2015) ‘Imagenet large scale visual recognition challenge’. *International journal of computer vision*, 115 (3), pp.211-252.
- SALIMANS, T. *et al.* (2016) ‘Improved techniques for training gans’. In *NIPS*, pp.2234-2242.
- SPRINGENBERG, J. T., DOSOVITSKIY, A., BROX, T. & RIEDMILLER, M. (2014) Striving for simplicity: The all convolutional net. *arXiv*. Available at: <https://arxiv.org/pdf/1412.6806.pdf> (Accessed: 28 May 2019).
- SUHR, A., LEWIS, M., YEH, J. & ARTZI, Y. (2017) ‘A corpus of natural language for visual reasoning’. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2, pp.217-223.

- SZEGEDY, C. *et al.* (2016) ‘Rethinking the inception architecture for computer vision’. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp.2818-2826.
- TIELEMAN, T. & HINTON, G. (2012) Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. [PowerPoint presentation to Neural networks for machine learning]. Available at: [https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) (Accessed: 22 June 2019).
- Vincent, J. (2017) *This app uses neural networks to put a smile on anybody’s face*. Available at: <https://www.theverge.com/tldr/2017/1/27/14412814/faceapp-neural-networks-ai-smile-image-manipulation> (Accessed: 2 August 2019).
- YU, F. *et al.* (2015) Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv*. Available at: <https://arxiv.org/pdf/1506.03365.pdf> (Accessed: 27 June 2019).
- ZHANG, H. *et al.* (2017) ‘Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks’. In *Proceedings of the IEEE International Conference on Computer Vision*, pp.5907-5915.
- ZHU, J.-Y., PARK, T., ISOLA, P. & EFROS, A. A. (2017) ‘Unpaired image-to-image translation using cycle-consistent adversarial networks’. In *Proceedings of the IEEE International Conference on Computer Vision*, pp.2223-2232.

## Appendix A

The following images demonstrated the full questionnaire used for evaluating GANs models.

### **Introduction**

My name is TA-YU, CHEN and a postgraduate student from advanced software engineering in the university of Strathclyde. I invite you to complete a short questionnaire about distinguishing the difference between synthesising and real images.

### **What is the purpose of this investigation?**

Techniques of synthesising images have developed for decades. Computer graphics are considerably employed in this domain. Along with the progress of machine learning, researchers devote their energy to explore the possibility of synthesising images by the technique of machine learning. In the domain, Generative adversarial network (GAN) is a well-known deep learning method which can train a model to generate meaningful images. The purpose of this investigation is evaluating the quality of synthesising images from GAN models and helping me to analyse the performance of these models.

### **Do you have to take part?**

Participation is completely voluntary. You also have a right to withdraw without detriment in the period of the investigation.

### **What will you do in the project?**

In the investigation, you will need to complete a questionnaire with 12 questions about distinguishing the difference between synthesising and real images. It takes participants for around 5 minutes.

Figure A-1. Instruction of the questionnaire

**Why have you been invited to take part?**

The investigation focusses on people who have a little knowledge about synthesising images through machine learning.

**Would you like to take part in the survey?**

NO

☐

Yes

☐

Figure A-2. Inviting participants to join the questionnaire

We demonstrate you pictures that are either generated by a computer or sampled from real images. Your task is to choose which **two** were generated by a computer

Q1.

Examples of real images



Examples of images generated by a computer



☐ 3

☐ 8

☐ 0

☐ 2

Q2.

Examples of real images



Examples of images generated by a computer



☐ 5

☐ 0

☐ 3

☐ 4

Figure A-3. The questions for choosing synthesised digit images of GANs and DCGAN

Q3.

Examples of real images



Examples of images generated by a computer



☐ 7

☐ 2

☐ 5

☐ 4

Q4.

Examples of real images



Examples of images generated by a computer



☐ 0

☐ 5

☐ 9

☐ 6



Figure A-4. The questions for choosing synthesised digit images of WGAN and WGAN-GP

Q5.

Examples of real images



Examples of images generated by a computer



Q6.

Examples of real images



Examples of images generated by a computer



Figure A-5. The questions for choosing synthesised geometric graphics of GANs and DCGAN

Q7.

Examples of real images



Examples of images generated by a computer

☐☐☐☐

Q8.

Examples of real images



Examples of images generated by a computer

☐☐☐☐

Figure A-6. The questions for choosing generated geometric graphics of WGAN and WGAN-GP

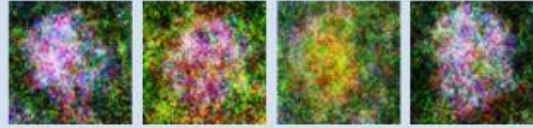


Q9.

Examples of real images



Examples of images generated by a computer

☐☐☐☐

Q10.

Examples of real images



Examples of images generated by a computer

☐☐☐☐

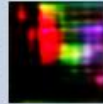
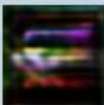
Figure A-7. The questions for choosing generated flower images of GANs and DCGAN

Q11.

Examples of real images



Examples of images generated by a computer

☐☐☐☐

Q12.

Examples of real images



Examples of images generated by a computer

☐☐☐☐

Figure A-8. The questions for choosing generated flower images of WGAN and WGAN-GP