PROBABILITY DISTRIBUTION CONSTRUCTION VIA DEEP LEARNING

HANNAH TAN E-LING

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Science (Honours) Applied Mathematics with Computing

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

Sept 2023

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : ____________________________________

Name : ____________________________________

ID No. : ____________________________________

Date : ____________________________________

APPROVAL FOR SUBMISSION

I certify that this project report entitled "**PROBABILITY DISTRIBUTION CONSTRUCTION VIA DEEP LEARNING**" was prepared by **HANNAH TAN E-LING** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Science (Honours) Applied Mathematics with Computing at Universiti Tunku Abdul Rahman.

Approved by,

Signature : ______________________________

Supervisor : ______________________________

Date : ______________________________

Signature : ______________________________

Co-Supervisor : ______________________________

Date : ______________________________

The copyright of this report belongs to the author under the terms of the copyright Act 1987 as qualified by Intellectual Property Policy of Universiti Tunku Abdul Rahman. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

© 2023, HANNAH TAN E-LING. All rights reserved.

ABSTRACT

The pursuit of estimating probability distributions of complex data is an ongoing challenge. Existing traditional methods impose a ceiling to the true resemblance of the targeted data distribution, due to their assumptions on the shape of the targeted data distribution. Recently, generative models have garnered substantial attention for its ability to replicate high-resolution images, thereby learning the distribution of high-complexity data. Inspired by this paradigmatic approach to learn a distribution without relying on an assumption about the shape of the target data distribution, this project explores the bridging of Deep Learning and Statistics within the area of distribution generation methods.

This paper provides the overall context of the research problem in Chapter 1, elaborates on existing literature and related works in Chapter 2, discusses the methodology and execution plan of this project in Chapter 3, mentions the results from what was executed in Chapter 4 and lastly concludes in Chapter 5.

TABLE OF CONTENTS

DECLARATION **1**
APPROVAL FOR SUBMISSION **2**
ABSTRACT **4**
TABLE OF CONTENTS **5**
LIST OF TABLES **7**
LIST OF FIGURES **8**
LIST OF SYMBOLS / ABBREVIATIONS **10**
LIST OF APPENDICES **11**

CHAPTER

1 **INTRODUCTION** **12**
1.1 General Introduction 12
1.2 Importance of Study 12
1.3 Problem Statement 13
1.4 Aim and Objectives 14
1.5 Scope and Limitation of Study 14

2 **LITERATURE REVIEW** **16**
2.1 Probability Distribution Estimation 16
2.2 Deep Generative Models 17
2.2.1 Variational Autoencoders (VAE) 17
2.2.2 Generative Adversarial Network (GAN) 20
2.2.3 Wasserstein Generative Adversarial Network (WGAN) 22
2.3 Applications of Deep Generative Models for Non-Image Data 28
2.4 Summary 31

3 **METHODOLOGY AND WORK PLAN** **34**
3.1 Introduction 34

	3.2	Overview of Procedure	34
	3.3	Software Library and Integrated Development Environment	34
	3.4	Data and Training Progression	35
	3.5	WGAN-GP Architecture and Algorithm	35
	3.6	Experimental Implementation	37
4		**RESULTS AND DISCUSSION**	**41**
	4.1	Introduction	41
	4.2	Training Dataset	41
	4.3	Model Setup	41
	4.4	Training Results	42
		4.4.1 Sinusoidal Data	43
		4.4.2 Linear Data	48
		4.4.3 Sigmoidal Data	50
		4.4.4 Annular Data	51
	4.5	Summary	52
	4.6	Discussion	53
5		**CONCLUSION**	**55**
	5.1	Conclusion	55
	5.2	Recommendation for Future Improvements	55
APPENDICES			**60**

LIST OF TABLES

Table 2.1: Comparison between generative models 32

Table 3.1: Types of Dataset 35

Table 3.2: Initial WGAN-GP Hyperparameters 36

Table 4.1: List of Training Dataset 41

Table 4.2: Initial WGAN-GP Hyperparameters 42

LIST OF FIGURES

Figure 2.1: Illustration of autoencoder architechture. 17
Figure 2.2: Illustration of variational autoencoder structure. 18
Figure 2.3: Three Distributions with Varying Distances 24
Figure 2.4: Illustration of heatmap of critic values (Phillips et al., 2022). 31

Figure 3.1: Flow chart of proposed coding structure of WGAN-GP. 38

Figure 4.1: General WGAN-GP Generator Structure. 42
Figure 4.2: General WGAN-GP Critic Structure. 42
Figure 4.3: 500 Epochs of SGR01 Ordered from Left-to-right, Top-to-bottom 43
Figure 4.4: SGR01 Loss 44
Figure 4.5: SGR02 Loss and Results at 300 Epochs 45
Figure 4.6: SGR03 Loss and Results at 500 Epochs 45
Figure 4.7: SGR03 Loss and Results Beyond 500 Epochs 46
Figure 4.8: Comparison Between Varying Learning Rates and Gradient Penalty Values at 500 Epochs 47
Figure 4.9: SGR03 Loss and Results with $\lambda = 9$ and batch size = 200 at 800 Epochs 48
Figure 4.10: L01 Loss and Results with $\lambda = 9$ and batch size = 200 at 100 Epochs 49
Figure 4.11: L01 Loss and Results with $\lambda = 9$ and batch size = 200 at 300, 350 and 400 Epochs 50
Figure 4.12: Sigmoid Loss and Results with $\lambda = 9$ and batch size = 200 at 300, 700 and 1100 Epochs 51
Figure 4.13: Annular Data WGAN-GP Loss and Results with $\lambda = 9$ and batch size = 200 at 400, 900 and 1300 Epochs from Left to Right 52
Figure 4.14: Summary between Best Outputs of Different Datatypes 53

Figure 5.1: Code Part 1 60

Figure 5.2: Code Part 2 61
Figure 5.3: Code Part 3 61
Figure 5.4: Code Part 4 61
Figure 5.5: Code Part 5 62
Figure 5.6: Code Part 6 62
Figure 5.7: Code Part 7 63
Figure 5.8: Code Part 8 63
Figure 5.9: Code Part 9 63
Figure 5.10: Code Part 10 64
Figure 5.11: Code Part 11 64
Figure 5.12: Code Part 12 65
Figure 5.13: Code Part 13 65
Figure 5.14: Code Part 14 66
Figure 5.15: Code Part 15 66

LIST OF SYMBOLS / ABBREVIATIONS

$p(z)$	probability of generating latent code
$\phi(x_i\|z)$	probabilistic decoder
$q_\theta(z\|x_i)$	probabilistic encoder
$D(x)$	discriminator's probability that x is a real data sample
$G(x)$	generated data sample
γ	transport plan
$p_r = p_0$	distribution of real/train data
$p_g = p_\theta$	distribution of generated data
$f_w(x)$	critic function with parameter w
$\tilde{x}$	generated sample
x	real sample
$\hat{x}$	linear interpolation between real and generated samples
$f(x)$	critic function of sample type x
λ	gradient penalty coefficient
∇	gradient
β_1, β_2	hyperparameter for optimiser
$F_{1,n}, F_{1,n}$	empirical distribution functions

LIST OF APPENDICES

APPENDIX A: Code Screenshots 48

CHAPTER 1

INTRODUCTION

1.1 General Introduction

A probability distribution is a mathematical function that defines all the potential values and its respective likelihood of occurrence for a random variable within a specified range (Di Paola et al., 2018). From a Bayesian perspective, this relates closely to the evaluation of uncertainty, as every uncertain event can be expressed in a probabilistic manner. To learn a probability distribution is to learn the probability density, and in doing so, it provides for statistical inference, aids in decision making, and future-oriented planning. A common method to achieve the above is through estimating the parameters that give the best fit to the observed data, otherwise known as Maximum Likelihood Estimation (MLE). However, this approach presupposes that the observed data is independent and identically distributed. Unfortunately, this goes against characteristics of most real-world data. It also assumes the underlying distribution of the observed data, besides the fact that in some cases, this underlying distribution may not exist (Arjovsky and Bottou, 2017). It is worth noting that in a continuous case, if data samples which possess comparable characteristics to the observed data could be generated, this would be asymptotic to having the ability to find the observed data's density function and hence its probability distribution. A particular product of Deep Learning called generative models has exhibited the potential to realize that exact aim, among other impressive abilities. More precisely, generative models Generative Adversarial Networks (GANs), and Variational Autoencoders (VAE). This research explores the structure, training techniques, and applications of the GANs and propose its use to construct probability distributions.

1.2 Importance of Study

The potential to construct probability distributions without relying on a presumed distribution highlights the significance of researching and enhancing prob-

ability distribution construction using generative models like GANs, breaking away from traditional approaches. For many applications, such as speech and image recognition, natural language processing, and anomaly detection, accurate data distribution estimate is crucial. Poor distribution estimation has shown in multiple instances to cause terrible calamities. The financial crisis in 2008 resulted from inaccurate assumptions made from the housing market distribution (Kang and Liu, 2014). In a separate case, the Space Shuttle Challenger tragedy that took place in 1986, where the space shuttle orbiter exploded due to the falsely estimated probability distribution of the likelihood of O-ring failure (Dalal et al., 1989). Accurate probability distribution estimation is a fundamental topic in statistics and machine learning. By adding to the body of work in the area of statistics and machine learning, this study can help advance the field and lead to new insights and techniques for alternative data modeling and analysis. Additionally, by providing another use-case of GANs beyond image-based data, of which GANs have exhibited excellent performance in, this study serves as a means to provide a notion to the future development and utilization of GANs.

1.3 Problem Statement

Maximum Likelihood Estimation (MLE) may not be an optimal method for estimating complex data probability distributions, especially when these data are non-regular and violate the standard assumption of MLE that observed data is independent and identically distributed (Rao, 1957). Complex data often display temporal correlation among its observations. Coupled with its dependency on various external factors, multiple assumptions about this data must be made for the nice properties of MLE, such as consistency, efficiency, and asymptotic normality to exist. Various modifications to MLE have been proposed to address these issues, such as mixed-effects models, maximum a posteriori estimation, and generalized linear mixed models, which include additional information and assumptions to improve accuracy and robustness of parameter estimation. Despite these efforts, estimating probability distributions for complex data remains a challenging task as MLE functions on the basis that the accuracy towards observed data is limited by its assumptions. Besides, MLE has the tendency to

produce biased estimations, especially when the volume of the dataset is small.

In some cases, maximum likelihood functions can be too complex to evaluate or not exist in a closed form. Thus, simulation-based estimation methods have brought rise to the use of machine learning in for estimation in a statistical context. A paper by Lewis and Syrgkanis elaborates efforts to select moment conditions using an adversarial approach. Wei and Jiang, 2021, used neural networks to direct mapping from data to parameter estimates and statistical accuracy of these estimates (Wei and Jiang, 2022). The direction of previous efforts of machine learning were scattered within statistics, used in nuanced and non-nuanced parameter model estimation, such as boosted trees, lasso regression and random forest parameters (Chernozhukov et al., 2018). There has been success in utilising artificial neural networks and boosted decisions to generate probability distribution functions (Sadeh et al., 2016).

However, only few papers made use of generative models specifically for probability distribution generation. The lack of a universal method for estimating complex data distributions highlights the need for further research to develop more effective and efficient methods for estimating complex data distributions. Thus, this study contributes to the expanding body of research that combines machine learning with statistics.

1.4 Aim and Objectives

The aim of this project is to make use of the ability of GANs to create a generative model with the ability to generate standard probability distributions. The key objectives of this project are as follows:

1. Provide a thorough review on GAN
2. Create a generative model capable of generating probability distributions
3. Evaluate the effectiveness of the generated distributions

1.5 Scope and Limitation of Study

For the purpose of this study, the scope of this project narrows down the category of deep learning, focusing mainly on GANs as the primary approach in distri-

bution generation. In evaluating the effectiveness of the generated distribution, this study makes comparisons to MLE as its counterpart.

Training of the GAN model is highly reliant on the optimal combination of hyperparameters and model architechture, as such, there are infinitely many of such configurations, which is undoubtably impossible to go through during the course of completing this study. Hardware limitations include the kind of GPU and processor the training of the model is done on. Since the performance of the GAN is directly proportional to the training time of the model, amongst other factors, and that GANs are notorious for requiring extensive training time, the model possesses a conservative level of performance. This is supplemented by the software limitation of this research, which involve the use of a local integrated development environment, Spyder. While it is convenient and do not rely on internet connection, unlike the paid version of Google Colab, for which access is unavailable for this research, do not have the benefit of boosted GPUs, which can allow for faster training times and the ability to work on larger-scale projects. While the free version of Google Colab is available, this version reduces the available runtime and increases the cool down time for each subsequent use of the boosted GPU feature, on top of its disconnection feature upon inactivity even though a code cell is running, which can be highly inconvenient for the purpose of sufficient model training (*Google Colaboratory: Frequently Asked Questions*, n.d.).

CHAPTER 2

LITERATURE REVIEW

2.1 Probability Distribution Estimation

Maximum Likelihood Estimation (MLE) is a renown statistical approach to probability distribution estimation (Fisher, 1992). It involves finding a likelihood function with respect to a set of parameter values, then evaluating the estimated parameter values that create the most likely distribution of the target distribution. Since its inception in the 1900s, MLEs have been extensively implemented across multiple fields, and are still relevant today due to its established record of success, on top of its simplicity to implement and statistically sound fundamentals. MLEs are often used to estimate parameters for different models. For instance, Aït-Sahalia and Kimmel deployed MLE for stochastic volatility model estimation (Aït-Sahalia and Kimmel, 2007). Zhang et al. proposed a logistic regression prediction model for EHR phenotyping using MLE as an improved phenotype prevalence estimation method in terms of efficiency and accuracy (Zhang et al., 2019). However, MLE functions on a strong set assumptions that are not always applicable in practice. MLE is known to work poorly under small-sized data sample constraints (Jain and Wang, 2008), subject to a systematic error of assumption which produces biased and non-optimal outcomes. Further, MLE assumes that the observed data follows a specific distribution, and that deviations from this assumption would yield a less accurate model fit. In some cases, the likelihood function is not tractable due to complex dependencies of some data. Thus, an approximation method such as Monte Carlo methods and variational inference is adopted, where a prior distribution is adjusted to mimic its posterior distribution. Although the technique is simple to use and intuitive, the choice of the prior distribution can greatly affect the posterior distribution (Park and Haran, 2020).

2.2 Deep Generative Models

2.2.1 Variational Autoencoders (VAE)

Variational Autoencoders were introduced in 2013, by Kingma and Welling in their paper "Auto-Encoding Variational Bayes", which marries the strength of neural networks with the probabilistic framework of Variational Bayesian Inference. Variational Autoencoders are an extension of Autoencoders, which consist of three parts, the encoder, bottleneck and decoder. Visually, Autoencoders can be illustrated as below

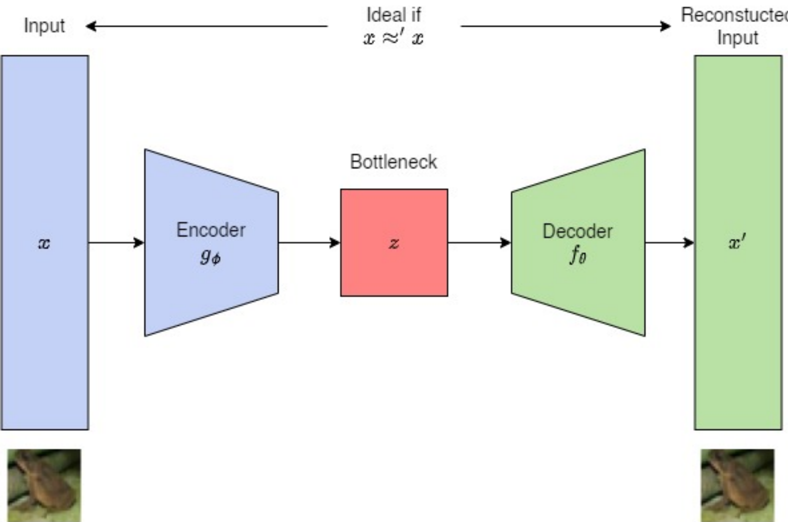


Figure 2.1: Illustration of autoencoder architechture.

The encoder maps its given input to a lower-dimension representation in the bottleneck, where a single encoding of the input is created, The bottle neck consists of the middle-most layers of the network, where the size of the bottleneck layers are proportionate to the amount of compression of vital information that gets stored in the latent space. The decoder recreates the original input by decoding its encoded variables. While there are benefits to reducing the dimensionality of high-dimensional data, generating new data samples of the same family as the input data would provide greater value than simply recreating the input data. The problem with Autoencoders lie in that the decoder does not possess the ability to validate encoded inputs to be of the same nature of the target data, in turn it produces random outputs that hold no value.

VAEs act as the solution to this problem by generating a range of possible encoded values for each latent variable from each input data in the form of probability distributions. The decoder then decodes a random sample from each probability distribution corresponding to each latent variable. The ability to sample from a set of probability distributions introduces variability in samples that may deviate from the input sample while holding the same types of unseen characteristics. Successfully decoding such samples implies that the VAE has the ability to generate new samples that are identical in distribution to the input data.

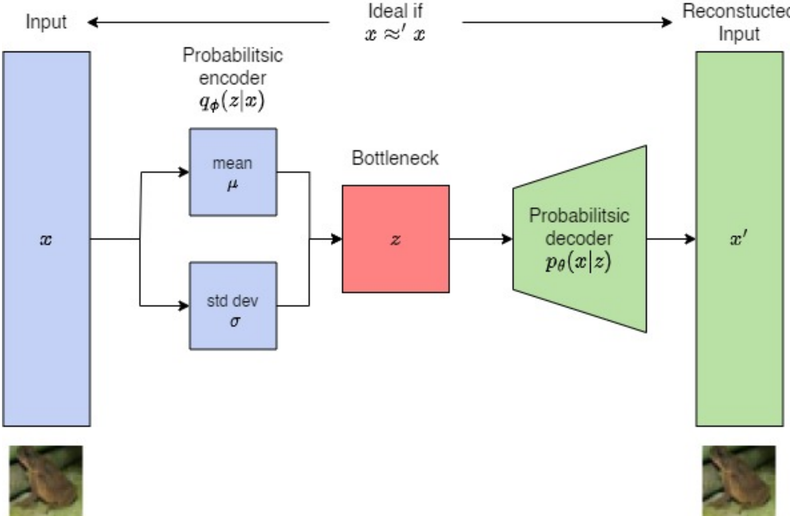


Figure 2.2: Illustration of variational autoencoder structure.

In a VAE, a probabilistic encoder and probabilistic decoder exists in place of an encoder and decoder. The goal of a VAE is to find a distribution $q_\phi(z|x)$ that maps inputs to latent variables, that by sampling from $z \sim q_\phi(z|x)$, new samples $x' \sim p_\theta(x|z)$ can be generated. The process of obtaining this $q_\phi(z|x)$, written as $P(z|x) = \frac{P(z|x)P(z)}{P(x)}$ is intractable in practice since the integral required to compute $P(x)$ is difficult to evaluate analytically. An alternative method is a variational inference approach that involves assuming a prior distribution (usually Gaussian) and minimizing the difference of the two distributions, molding the prior distribution to approximate the form of the target distribution. In doing so, a Kullback–Leibler (KL) divergence is utilized to calculate this difference.

$$L_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + \mathbb{KL}(q_\theta(z|x_i)||p(z)) \tag{2.1}$$

where

$p(z)$ - probability of generating latent code,

$_\phi(x_i|z)$ - probability of generating true data sample given the latent code (probabilistic decoder),

$q_\theta(z|x_i)$ - estimated posterior probability (probabilistic encoder),

However, a major pitfall of using KL-divergence surfaces when the prior distribution and the target distribution do not overlap, where in both the forward and reverse KL-divergence, this results in an infinite value of divergence.

Furthermore, since VAEs undergo variational inference which is an approximation of the posterior, it suffers from posterior collapse, where the posterior and prior distribution are equal, producing unmeaningful representations (Dai et al., 2019). An approach to handling this issue is by adding a noise term to the prior distribution so that the model can capture a greater measure of the the input data's complexity.

In contrast to VAE, a generative model introduced by Goodfellow et al. in 2014 - Generative Adversarial Networks (GANs) offer more flexible choices

in defining the objective function, such as Jensen-Shannon and f-divergences (Nowozin et al., 2016).

2.2.2 Generative Adversarial Network (GAN)

A Generative Adversarial Network (GAN) is a type of generative model consisting of two components with contrasting objectives, namely the generator and the discriminator. Through meeting their objectives, the GAN achieves its aim to learn the input data distribution in order to produce new samples that resemble it. More specifically, the generator takes a random noise vector and maps it to a point in the data space of the train dataset, producing outputs that resemble the data from the train dataset. Conversely, the discriminator outputs a value for the likelihood that the input sample comes from either the train (real) dataset or generated (fake) dataset, at a cut-off point at 0.5.

In the process of training, the generator learns a distribution, P_g that conforms to the train dataset distribution, P_r. The paper describes that the GAN objective is achieved when the Jensen-Shannon (JS) divergence between P_r and P_g is reduced to a minimum (Goodfellow et al., 2014).

Intuitively, the relationship between the generator and discriminator can be thought of as two components in a two-player minimax game. Let $D(x)$ be the probability that x is a real data sample, and $G(z)$ be the generator mapping of noise variables, z, to a data space. Then the minimax game exists such that the discriminator aims to maximimse the distance between real and generated samples, establishing a stark difference between the two, while the generator aims to close that distance, effectively confusing the discriminator into wrongly classifying fake data, which is to minimize $\log(1 - D(G(z)))$. The equation below is the minimax equation as described by Goodfellow (Goodfellow et al., 2014).

$$\min_G \max_D \mathbb{E}_{x \sim p_r(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[1 - \log D(G(z))] \tag{2.2}$$

where

$p_z(z)$ - simple prior noise distribution,

$p_r(z)$ - train data distribution,

$D(x)$ - discriminator output for the probability of x being real,

$G(z)$ - generated data sample

The first term in this equation gives the probability of the discriminator for correctly classifying real data, while the second term is the probability of the discriminator for correctly classifying fake data. The discriminator aims to maximise this value. In contrast, the generator aims to minimise the second term, reducing the discriminator's chances of correctly classifying fake data.

Goodfellow recommends against training the discriminator to optimality due to its high computational cost, and potential to overfit. When this happens, the discriminator becomes too advanced, and classifies fake data with high confidence. The JS divergence between the distribution for the real and fake data sample vanishes, leading to significantly slow generator learning, and training instability. Instead, by alternating the training of the discriminator and generator, the discriminator maintains a near optimal state, while the generator gradually improves in data generation quality. To balance the learning rate of the discriminator and generator, the algorithm suggested by Goodfellow is as follows

---

Algorithm 1: GAN Training Algorithm

1: **for** t number of epochs **do**
2:     **for** k number of steps **do**
3:         - sample a minibatch of m noise samples from a simple prior distribution
4:         - sample a minibatch m of generated samples from generator output
5:         - update the discriminator weights by backpropagation
6:     **end for**
7:     - sample a minibatch of m samples from a simple prior distribution
8:     - update the generator weights by backpropagation
9: **end for**

Conversely, when the generator is overtrained, and generates an output that fools the discriminator the 'most' (in the eye of the discriminator), the generator ignores the variability that comes from sampling from random noise, z,

thus collapsing its mode to a single point. This means the generator falls into the routine of generating the same data that the discriminator deems is true, exploiting the local minima of the discriminator. Therefore, maintaining a balance in training the generator and discriminator is a terribly tricky task, a direct factor to the difficulty for GANs to converge (Goodfellow et al., 2014).

While that is so, iterative inference calculation is not needed for GANs to learn the target distribution. The reduction of computational cost due to this fact is outweighed by the difficulty for convergence of GANs. However, the positive approach of GANs in learning a target distribution without assuming a prior distribution alone is worth exploring, so in an attempt to improve GANs, an evolved flavour of GAN was introduced 3 years later. (Arjovsky et al., 2017)

2.2.3 Wasserstein Generative Adversarial Network (WGAN)

In 2017, an improvement to the original GAN was introduced by Arjovsky at al., addressing the difficulty to maintain a balance between training of the subneural networks. It was found that the root of the issue related to the method in defining the closeness of the generator and target data distributions. In amending the divergence measure, a significant improvement in the nature of convergence was observed. Making this change effectively supposedly mitigated training instability in GANs. This meant that maintaining a balance in training in this enhanced GAN was no longer a prevalent issue. Training the discriminator, or rather, the critic, in this new flavour of GAN to its optimum without worry of training imbalance was now made possible, and was a more effective way to train the GAN. As a by-product of the updated divergence measure, its loss function was found to be highly correlated to the performance of the model, and could be plotted as a means to monitor the training process. The following section will go into more detail and technicality of this approach.

The paper by Arjovsky at al. make comparisons to Total Variation (TV) Distance, Kullback-leiber (KL) Divergence, Jensen-Shannon (JS) Divergence and the Earth-Mover (EM) or Wasserstein-1 Distance, which were distance measures for the distance between the distribution of target data and the learned distribution of the generator-equivalent of the predecessors of GANs and the

distance measure which later became that of WGANs.(Arjovsky et al., 2017).

TV distance calculates the distance between two probability distributions by summing the absolute difference between their density functions. The metric denoted by TV distance produces a 0 if the two distributions are identical and 1 if they are completely different Arjovsky et al. (2017). KL divergence is an asymmetrical calculation of the difference between probability distributions. It is also known as the relative entropy and measures the information lost when estimating one distribution from another. JS divergence on the other hand is the symmetrical version of the KL divergence. EM distance calculates the cost to transform one distribution to the other and the corresponding difference between the two probability distributions based on this minimal cost, effectively providing the amount of work required to apply the transformation.

It was found that EM distance was most suited distance among the 4, due to its dynamic yet stable properties. This is because the TV distance is nothing but a binary value for whether two probability distributions are identical or not, producing values 0 and 1 for the respective scenarios. KL divergence on the other hand is an asymmetrical calculation of the difference between probability distributions, which blows up at parts of the distribution where the two do not overlap. The third case is with JS divergence, which caps at log 2 when there is no overlap in distributions (Arjovsky et al., 2017). This is an improvement as compared to KL divergence. However, it is still not a true representation of distance, as increased distance between two non-overlapping distributions are represented by a constant rather than a variable that increases in proportion to their distance. The EM distance is superior in this regard, as it is directly proportional to the distance between probability distributions. Earth Mover's distance is explained as the cost of moving one pile of soil to another. By considering all transport paths and their respective costs, the WGAN aims to find the minimum cost to do so. This is possible for the EM distance is calculated horizontally, unlike the 3 previous distances that were calculated vertically (Arjovsky et al., 2017). The EM distance is the only continuous plot with usable gradients everywhere, which is key for gradient descent in the learning process (Arjovsky et al., 2017).

$$W(P_r, P_g) = \underset{\gamma \in \prod(P_r, p_g)}{inf} E_{(x,y)\sim\gamma}\left[||x - y||\right] \tag{2.3}$$

where

$\prod(p_r, p_g)$ = the set of all possible joint probability distributions between p_r and p_g,

γ = one of the transport plans,

p_r = distribution of the train data,

p_g = generator distribution of train data

The following example provides an illustration for the convergence and lack of as mentioned above.

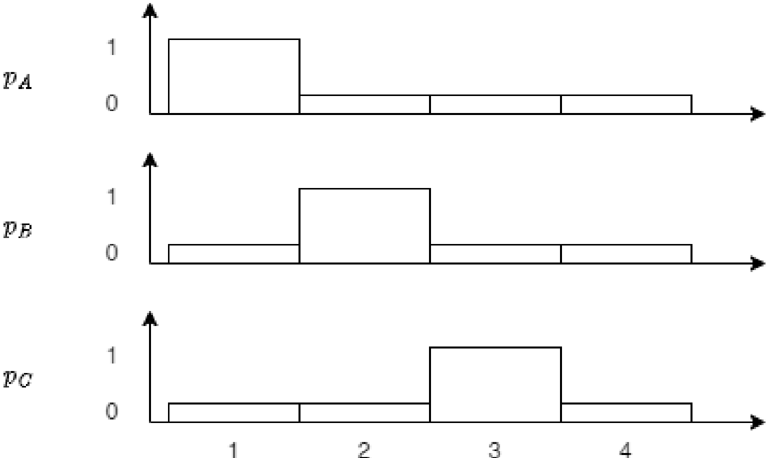


Figure 2.3: Three Distributions with Varying Distances

$$KL(p_A, p_B) = \sum_x p_A(x) \log \frac{p_A(x)}{p_B(x)} \tag{2.4}$$

$$= P(x = 1) \log \frac{p_A(x = 1)}{p_B(x = 1)} \tag{2.5}$$

$$= 1 \log \frac{1}{0} \tag{2.6}$$

$$= \infty \tag{2.7}$$

$$KL(p_A, p_C) = \sum_x p_A(x) \log \frac{p_A(x)}{p_C(x)} \tag{2.8}$$

$$= P(x=1) \log \frac{p_A(x=1)}{p_C(x=1)} \tag{2.9}$$

$$= 1 \log \frac{1}{0} \tag{2.10}$$

$$= \infty \tag{2.11}$$

$$JS(p_A, p_B) = \frac{1}{2} \left(KL(p_A, p_M) + KL(p_B, p_M) \right), p_M = \frac{p_A + p_B}{2} \tag{2.12}$$

$$= \frac{1}{2} \left(p_A(x=1) \log \frac{p_A(x=1)}{\frac{p_A(x=1) p_B(x=1)}{2}} + p_B(x=2) \log \frac{p_B(x=2)}{\frac{p_A(x=2) p_B(x=2)}{2}} \right) \tag{2.13}$$

$$= \frac{1}{2} \left(1 \log \frac{1}{\frac{1}{2}} + 1 \log \frac{1}{\frac{1}{2}} \right) \tag{2.14}$$

$$= \log 2 \tag{2.15}$$

$$JS(p_A, p_C) = \frac{1}{2} \left(KL(p_A, p_M) + KL(p_C, p_M) \right), p_M = \frac{p_A + p_C}{2} \tag{2.16}$$

$$= \frac{1}{2} \left(p_A(x=1) \log \frac{p_A(x=1)}{\frac{p_A(x=1) p_C(x=1)}{2}} + p_C(x=3) \log \frac{p_C(x=3)}{\frac{p_A(x=3) p_C(x=3)}{2}} \right) \tag{2.17}$$

$$= \frac{1}{2} \left(1 \log \frac{1}{\frac{1}{2}} + 1 \log \frac{1}{\frac{1}{2}} \right) \tag{2.18}$$

$$= \log 2 \tag{2.19}$$

$$EM(p_A, p_B) = 1 \tag{2.20}$$

$$\tag{2.21}$$

$$EM(p_A, p_C) = 2 \tag{2.22}$$

$$\tag{2.23}$$

Topologically, it was also found that KL divergence was strongest, followed by JS divergence, TV divergence, and lastly EM divergence. The weakness of EM divergence suggests that it would yield better properties as compared to JS divergence, since according to Arjovsky et al. (2017), weaker distance measures were shown to ease model convergence. Additionally, the differentiability of EM divergence imply that it can be used to train the critic to optimality, where the occurrence of vanishing gradients is mitigated. This phenomenon is prevalent for JS divergence as it is locally saturated, causing the gradients to converge to a true value of zero value as training continues. The experiment carried out in the paper by Arjovsky et al. (2017) confirmed this, where JS loss that is poorly correlated with the model performance, saturating at approximately log2, its maximum value, leading to zero discriminator loss. Conversely, it is not possible for the critic to saturate, for the constraint set on the weights limit the potential growth of the function to a linear function at most.

The ability to train the critic to optimality also means that mode collapse is no longer possible with increased training. There were no instance in any of the experiments carried out as described in the WGAN paper where mode collapse occurred for WGAN. All this pointed towards the fact that WGAN is superior to GAN.

The infimum in the EM distance is however intractable, as finding all possible joint distributions between p_0 and p_θ is too computationally costly. An equivalent formula was proposed by Arjovsky et al. based on the Kantorovich-Rubinstein duality, maximizing

$$W(p_r, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_r}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)] \tag{2.24}$$

where

$f(x)$ = the critic function,
p_r = distribution of the train data,
p_g = generator distribution of train data

where the optimal K-Lipschitz function $f(x)$ is one that finds the maximum difference between the expected critic values of the two distributions, and originates from a family of K-Lipschitz continuous functions (Arjovsky et al., 2017). This difference provides a tractable distance measure between distributions.

The loss function of the "discriminator" no longer acts as a classifier as with the original GAN, rather, it is trained by differentiation of this loss and back-propagation to learn w, to obtain the K-Lipschitz continuous function that is essential to computing the Wasserstein distance. The loss decreases as training occurs, which means the Wasserstein distance decreases as a result of the improving generator output(Arjovsky et al., 2017).

$$L(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))] \tag{2.25}$$

where
$g_\theta(z)$ = generated sample,
$f_w(x)$ = critic function with parameter w,
p_r = distribution of the train data,
p_g = generator distribution of train data

An essential condition for maintaining K-Lipschitz continuity involves ensuring that the weights w lie within a bounded, compact space. To do this, the weights were clamped to a fixed bound such as [-0.01, 0.01] after each gradient update. This was found not to be a good idea even though it maintained K-Lipschitz continuity, because a small clipping would lead to vanishing gradients, and an overly large clipping would slow down training immensely. In a paper published shortly after this, weight clipping was replaced by gradient penalty (Gulrajani et al., 2017) , which penalises the gradient norm for created

samples. This ensures that the objective function has a maximum norm value of 1 everywhere. The objective function for WGAN-GP is as follows

$$L(p_r, p_g) = \mathbb{E}_{\tilde{x}\sim\mathbb{P}_g}[f(\tilde{x})] - \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] + \lambda\mathbb{E}_{\hat{x}\sim\mathbb{P}_{\hat{x}}}[(||\nabla_{\hat{x}} f(\hat{x})||_2 - 1)^2] \quad (2.26)$$

where
$\tilde{x}$ = generated sample,
x = real sample,
$\hat{x}$ = linear interpolation between real and generated sample,
$f(x)$ = critic function of sample type x,
λ = penalty coefficient,
$||\nabla_{\tilde{x}} f(\tilde{x})||_2$ = gradient norm of the discriminator w.r.t. the generated sample

The first two terms being the original critic loss and the third term being the additional term for the gradient penalty. The linear interpolation term $\tilde{x}$ is used to enforce Lipschitz continuity by penalizing the norm of the gradient of the critic function with respect to interpolated samples and is obtained from uniformly sampling along the lines connecting sample pairs between the two distributions. The inception of the gradient penalty mitigated training difficulties previously faced, and aided in discriminator optimisation and thus, convergence.

2.3 Applications of Deep Generative Models for Non-Image Data

Generative Adversarial Nets have shown its competence in image-based data (Kang et al., 2022) (Xiang et al., 2023) (Huang et al., 2018). However, few papers expound on the use of GANs on non-image data (Hu et al., 2021) (Phillips et al., 2022), and even fewer on distributive modeling. One of the uses of GANs that is similar to the use case of this research is that of utilizing GANs as a multiple output regression model, named Multiple Output Regression GAN (MORGAN). The proposed model aims to learn the distribution of the underlying re-

Algorithm 2: WGAN-GP Training Algorithm

Require: λ, n_{critic}, m, a^l, b_1, b_2.
Require: starting critic and generator parameters ω_0, θ_0.
1: **while** θ is not converged **do**
2: **for** $k = 1, ..., n_{critic}$ **do**
3: **for** $i = 1, ..., m$ **do**
4: Get a real data sample $x \sim \mathbb{P}_r$, latent var $z \sim p(z)$, a rand num $\epsilon \sim \mathrm{U}[0,1]$
5: $\tilde{x} \leftarrow G_\theta(z)$
6: $\hat{x} \leftarrow \epsilon x + (1-\epsilon)\tilde{x}$
7: $L^{(i)} \leftarrow D_\omega(\tilde{x}) - D_\omega(x) + \lambda(||\nabla_{\hat{x}} D_\omega(\hat{x})||_2 - 1)^2$
8: **end for**
9: $\omega \leftarrow \mathrm{Adam}(\nabla_\omega \frac{1}{m}\sum_{i=1}^{m} L^{(i)}, \omega, a^l, b_1, b_2)$
10: **end for**
11: Get a batch of latent variable samples $\{\mathrm{z}^{(i)}\}_{i=1}^{m} \sim p(z)$
12: $\theta \leftarrow \mathrm{Adam}(\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} -D_\omega(G_\theta(z)), \theta, a^l, b_1, b_2)$
13: **end while**

gression problem, as well as extending the trained GAN as a method to predict outputs of specified independent variables.

Comparisons were made to equivalent outputs by a popular machine learning technique for regression problems called Gaussian Process Regression, based on Bayesian theory and statistical learning. A python library was even created (Hack, n.d.) for the use of this technique, momentous towards its adoption in dealing with computationally costly statistics problems.

First take to consideration a simple regression problem with input and output x and y respectively. By nature of GAN, the generator outputs both x and y, that seem to originate from the same distribution of real samples. To specify the independent variable x, a prediction algorithm was introduced. Starting with a trained WGAN, latent variables are first set to random numbers. The trained WGAN generator produces an output from this random valued input, say (x, y). If the intended independent variable is x_p, then the latent variables are updated via back-propagation so as to reduce the difference between x and x_p. By doing so, the expected outcome is assumed to be (x_p, y_p). Since the generator is a non-linear function of latent variables, starting with different random states mean that different y_p values can be obtained from the same x_p value. Thus, a distribution

of y_p values can be obtained from this process, which can reflect the uncertainty or probability of the y_p.

A matrix $[01]$ is introduced and acts to isolate and project the generated outputs in the form $(x, y)^T$ onto a space for constrained variables. Thus, the WGAN in this paper can be used either with randomly generated independent variables or constrained ones (specified x values). In the first case, there is no control over the independent variables generated by the WGAN, however, the distribution of the input data can be learned naturally by sampling from generated outputs. Conversely, to find the distribution of output values, given a specified input value, the algorithm as described above, using a trained WGAN, would yield a set of constrained tuples, from which the distribution of y_p values can be obtained.

As a method of statistical evaluation in this paper, the Kolomogorov-Smirnov (KS) test was applied, which is a goodness of fit test between two distributions. The distributions are created based on x-values from both real and generated datasets of a given range, and their corresponding y-values. On the other hand, the Mann-Whitney U test is used to determine whether real and generated samples are likely to be derived from the same population, or whether one sample tends to have larger values than the other.

In the context of distribution generation, by constraining tuples, one x input value can yield more than one y value, leading to the possibility of distribution construction of types of data where two data points may share the same x value but different y value, as with annulus data and other multi-modal datasets. The author also makes use of the critic value to determine confidence in outputs and locations in data where extra training is required. By normalizing the critic values at each point on a grid of the distribution, and plotting a heatmap based on those values, regions where critic value is lower represent points of lesser generator confidence. Unrealistic solutions can also be removed to improve results. Alternatively, it can be an indicator of the location of training data that should be added so that training in that area can be improved. The figure below shows an example of this, where areas such as (0,0) has a lower critic value, which indicates that more of such training data should be passed into the model

for additional training.

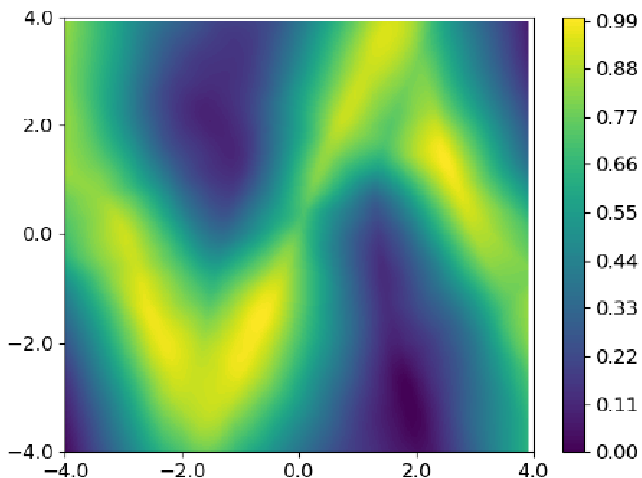


Figure 2.4: Illustration of heatmap of critic values (Phillips et al., 2022).

The 3D distribution was created by first getting a 3D spiral dataset from a given formulation for x,y and z, and parameters chosen randomly. The input of the generator are the x and y values of the train dataset. The generated output is also in the form of x's and y's, but since the spiral is stacked in 3D, after training the WGAN, 10 of the x,y values are selected as starting points, however the next 10 x,y values are evaluated from the prediction algorithm by constraining the x values to get the different y values of the given x values, as a result we are left with a new y value for each x value, that is used to plot the next 10 points in the dataset(Phillips et al., 2022).

2.4 Summary

In summary, generative models in Machine Learning are still, as of writing this report, an actively studied field. New use cases of generative models, specifically, GANs are continually on the rise. Throughout this literature review, the importance of the distance measure was highlighted, where the improvement of distance measure brought about positive evolutions to the mentioned generative models. Key differences and improvements of the different generative models are illustrated below:

Table 2.1: Comparison between generative models

Generative Model	Contributions/Advantages	Disadvantages
VAE (2013)	• An extention of Autoencoders • Adds variation to latent variables, allowing for generation of new data resembling train data	• Uses KL-divergence, goes to ∞ when distributions do not completely overlap • KL divergence is asymmetrical
Vanilla GAN (2014)	• Learns target distribution through simulation • Able to handle complex and scarce data • JS-divergence used is symmetrical	• Difficult to maintain training balance (mode collapse, diminishing gradients) • Difficult to converge • Prone to training instability
WGAN (2017)	• Improves training instability of GAN • Uses EM distance that has continuous gradients everywhere • EM distance as good indicator of training performance	• Too much or too little weight clipping cause negative impact to model performance • Slower training compared to GAN
WGAN-GP (2017)	• Additional gradient penalty term maintains Lipschitz continuity without weight clipping • More stable training compared to WGAN	• Increased computational resource due to additional gradient penalty term

The evolution of GANs point to the direction that WGAN-GP should be the model framework to deploy.

Further, the application of WGAN-GP as a regression model (MOR-GANs) displays the possibility for continuous coordinate-like data of multivariate nature to be learned, however it is not explicitly used on standard probability distribution shapes. Thus, by proving that it is possible, this project serves to contribute to the literature joining statistics and machine learning.

CHAPTER 3

METHODOLOGY AND WORK PLAN

3.1 Introduction

The end product of of this project is a learned generative model of data of known statistical distributions or shapes with added noise. This model aims to display robustness towards uncertainty in data and capability in reproducing uni-variate and multi-variate data, in this way, mirroring the stochasticity of real-world data. This section discusses the theoretical reasoning for the intended generative model used, its proposed structure and algorithm.

3.2 Overview of Procedure

As an overview, due to the commendable generative ability of WGAN-GP, this project aims to deploy a WGAN-GP model, testing the model starting with simple training data such as normally distributed data and uniformly distributed data, and progressively increase the complexity of data distribution, with added noise. The generated data samples and train data samples are collected and sampled to form a plot on a Cartesian plane, indicated by distinctive colors. The plots are converted to a distribution function by means of Universal Approximation Theorem, and the performance of the model is evaluated using the Earth Movers' distance measure and Kolmogorov-Smirnov test.

3.3 Software Library and Integrated Development Environment

The intended software library for deployment is TensorFlow. TensorFlow is an open-source software library that is commonly used for the development, training and deployment of deep learning models for various applications (Abadi et al., 2016). As such, it is a flexible and scalable platform for machine learning algorithm implementation, and it supports multiple programming languages, including Python, which is the intended programming language of use. Pytorch was selected due to its homogeneity across updates, preventing cases where newer functions are not compatible with older modules, as prone in Tensorflow,

as well as its widely available learning resources. The Integrated Development Environment (IDE) deployed is Spyder, an easy-to-use and popular platform containing integrations with multiple Python libraries that are essential for machine learning tasks, making it suitable for deep learning model deployment tasks such as this.

3.4 Data and Training Progression

Training data is randomly generated using existing mathematical libraries in Python, following the list of known probability distributions

Table 3.1: Types of Dataset

Dataset	Noise	Type
Uniform distribution	yes	uni-modal
Normal distribution	yes	uni-modal
Sine distribution	yes	uni-modal
Annulus data	yes	multi-modal

3.5 WGAN-GP Architecture and Algorithm

The WGAN-GP model framework makes use of the Wasserstein distance, that has proven to be a better representation of the dissimilarity between distributions, successfully side-stepping calculation limitations as with JS and KL divergence, discussed above (Arjovsky et al., 2017). Since the loss function of WGAN-GP is entirely continuous, in theory, this means that the calculation of gradients for back-propagation in training is a smoother process, and one that leads to quicker convergence and eliminates training instability. For these reasons, the WGAN-GP architecture is deployed instead of the Original GAN.

A typical WGAN-GP consists of a neural network for a generator and a critic, respectively. To implement these networks, a set of hyperparameters and the layering structure of the neural net are required. Values for the required hyperparameters are either tuned from hyperparameter optimisation, or replicated from learned literature. The prior set of hyperparameters are adapted from that mentioned in the MOR-GAN paper by Phillips et al. (2022), and where the value

for gradient penalty critic iterations per generator iteration is taken as a scientific consensus as those values have proven to be well functioning across a multitude of architectures (Gulrajani et al., 2017). On the other hand, the number of layers and nodes in those layers should be adjusted based on the complexity of input data, although completely arbitrary. Prior work on similar datasets show that three to four layers containing around 30-40 neural network nodes for generator and critic networks provided sufficient learnable parameters for reliable results (Phillips et al., 2022).

Table 3.2: Initial WGAN-GP Hyperparameters

Hyperparameters	**Value**
Learning rate, a^l	10^{-3}
Number of Critic iterations per Generator iterations, n_{critic}	5
Batch Size, m	100
Latent Space Dimension, z	3
Adam optimiser hyperparameters, b_1, b_2	0.5, 0.9
Gradient penalty hyperparameter, λ	10

The training algorithm of the WGAN-GP model involves training the critic for n_{critic} steps followed by 1 step of generator training per epoch. This alternative training method of the generator and critic is supported in various literature (Goodfellow et al., 2014) (Arjovsky et al., 2017) and (Phillips et al., 2022) with the intention of optimizing both the generator and critic at each epoch. The gradient penalty tern in WGAN-GP encourages the critic to produce smooth gradients throughout the input space, which prevents it from becoming too powerful and dominating the training process.

A real and generated sample is collected and an interpolation of the two samples are selected for the calculation of the critic's loss, from which the gradients are calculated and critic weights are updated. The training and weight updating process is repeated for the generator. As training progresses, the Wasserstein distance decreases and in principle, the generator is able to produce counterfeit output samples of high characteristic similarity.

Algorithm 3: Proposed WGAN-GP Training Algorithm

Require: λ, n_{critic}, m, a^l, b_1, b_2.
Require: starting critic and generator params w_0, q_0.
1: **while** θ is not converged **do**
2: **for** $t = 1, ..., n_{critic}$ **do**
3: real data $x \sim \mathbb{P}_r$, latent variable $a \sim p(a)$, a random number $\epsilon \sim U[0, 1]$
4: $\tilde{x} \leftarrow G_\theta(a)$
5: $\hat{x} \leftarrow \epsilon x + (1 - \epsilon)\tilde{x}$
6: $L^{(i)} \leftarrow D_w(\tilde{x}) - D_w(x) + \lambda(||\nabla_{\hat{x}} D_w(\hat{x})||_2 - 1)^2$
7: **end for**
8: $w \leftarrow$ Adam($\nabla_w \frac{1}{m} \sum_{i=1}^{m} L(i), w, a^{(i)}, b_1, b_2$)
9:
10: Get a sampled batch of latent variables $a^{(i)}{}_{i=1}^{m}(a)$
11: $\theta \leftarrow$ Adam($\nabla_w \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(a))\theta, a^{(l)}, b_1, b_2$)
12: **end while**

3.6 Experimental Implementation

The execution process of the proposed model require coding particulars which can be imitated through past work by other authors. The skeletal flow of this implementation is as follows:

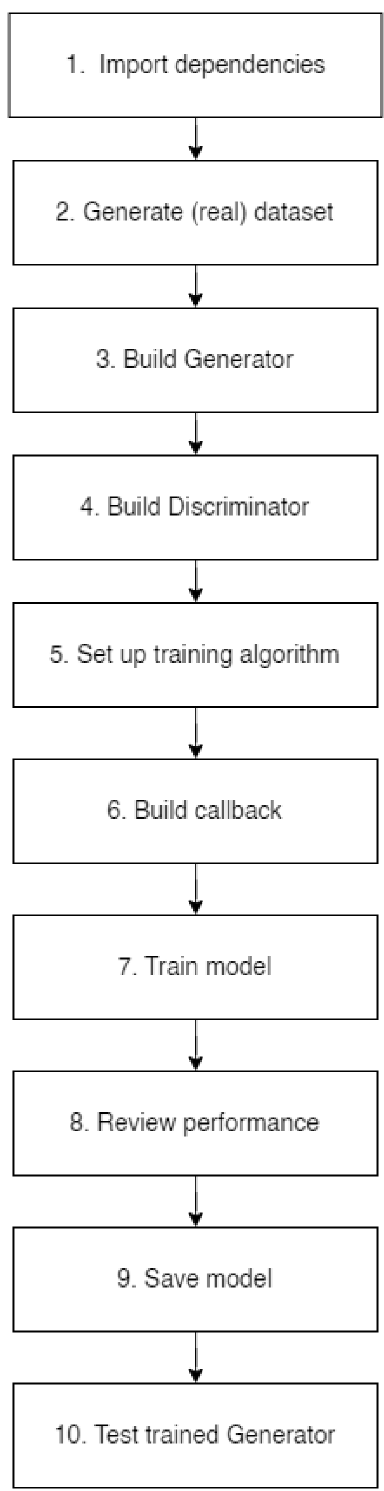


Figure 3.1: Flow chart of proposed coding structure of WGAN-GP.

First, the dependencies required for the code to function must be imported, this generally includes libraries such as pytorch, matplotlib, numpy and

os. When generating the dataset for the different distributions, np.rand is used as parameters for the different built-in distribution methods. The generator and discriminator networks consist of multiple neural network layers, where the output shape of the generator should coincide with the input shape of the critic. This shape should also be the same as the real data samples. In this case, when dealing with coordinates, this value should be that of two values for x and y-coordinates. Batch normalisation should be used in the generator neural net so as to normalise the activations of each layer to improve the training stability and performance of the model. Leaky rectified linear activation functions (LeakyReLU) should also be used to introduce non-linearities to the neural networks. A dropout should also be utilised in the critic neural net structure. The paper by Phillips et al. (2022) changed batch normalisation with layer normalisation on the critic so as to maintain the functionality of the gradient penalty term in the loss function. Next, the training algorithm is the process that occurs for each training epoch. This is described in the above section, and it proposes to use Adam optimizer (Kingma and Ba, 2017) as it is a reputable optimizer commonly used for similar tasks.

Step 6 in the flowchart involves creating a callback, which in this case could be used to store generator outputs and sampling them to create a plot using the generated coordinates. Once all structures are in place, the model can be trained over a set number of epochs. The number of epochs needed to sufficiently train a model is highly dependent on the complexity of the data itself. Training the model takes time but models can be partially trained and saved to train upon the partially trained model.

The performance of the model can be viewd from the graphical plot of the progression of the Wasserstein loss. The Wasserstein loss was found to be very accurately representative of the performance of the model (Arjovsky et al., 2017). The model can be saved in intervals or manually after reviewing the performance of the model. Finally, the product of the well trained WGAN-GP model is the generator that has learnt the characterstics of the train data, so much as to replicate such data.

The proposed method to obtain distributions from generated data is to

first obtain the cumulative plot of datapoints, train and generate data on this cumulative set of datapoints, and then rearrange the generated datapoints in descending order, deducting and storing each intermediate value of differnece as the raw form of data. To test the model performance, the Earth Movers' distance is proposed as the evaluation medium. The data samples taken for testing can be based on a specified range of x-values, and y-values from the generated samples and the real samples.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction

Following through with the steps mentioned in Chapter 3, this section aims to relay the detailed actual procedure, results and discussion on the results obtained.

4.2 Training Dataset

In this study, a number of train datasets were used. The used datasets are one-dimensional and followed standard known statistical distributions that were obtained through the random generation of data samples using Python's random number generator libraries either in tensor or numpy form, depending on the situation. The purpose of using these datasets is to test the flexibility of the model in learning distributions of data with varying properties.

Table 4.1: List of Training Dataset

Dataset	Distribution	Type	Section
Sinusoidal		unimodal	5.4.1
Linear	Uniform	unimodal	5.4.2
Sigmoid	Normal	unimodal	5.4.3
Annular		multimodal	5.4.3

4.3 Model Setup

The generative model used follows a WGAN-GP framework. The generator and critic were built separately with differing neural network architecture. The generator utilised a leaky RELU activation function and batch normalisation, which aids in reducing the chances of no learning in the neural network weights as well as stabilisation of training respectively. A hyperbolic tangent activation function, tanh() was used to output values in the range [-1,1] so as to obtain a greater coverage over the dataspace Radford et al. (2016). The set of hyperparameters used for the model are listed below, of which the number of critic iter-

ations per generator iteration and gradient penalty coefficient used for training followed values that proved to be effective via empirical derivation from previously demonstrated successes by Arjovsky et al. (2017) and Gulrajani et al. (2017).

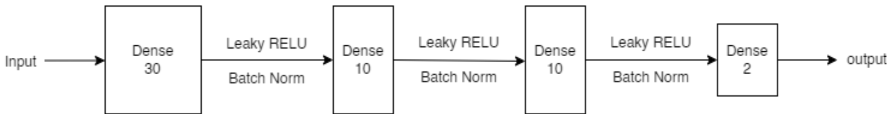


Figure 4.1: General WGAN-GP Generator Structure.

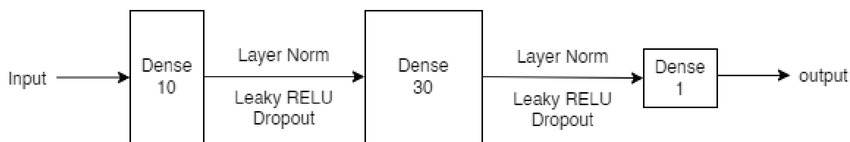


Figure 4.2: General WGAN-GP Critic Structure.

Table 4.2: Initial WGAN-GP Hyperparameters

Hyperparameter	Value
Learning Rate	0.001
z Dimension	3
Sample Size	1000
Batch Size	100
Epochs	1000
Adam Optimiser	0.5, 0.9
Critic iterations	5
Gradient Penalty	10

4.4 Training Results

The algorithm used follows that of Algorithm 3 in Chapter 3, with initial hyperparameters set according to Table 3.2. A significant amount of time and effort was placed in the trial and error of hyperparamter and model architechture tweaking. From past experimentations, the number of epochs required to train a GAN for imagerial data ranged from 200 to 100,000 (Goodfellow et al., 2014) and (Gulrajani et al., 2017). This wide range implied that there was no baseline to start with, since the type of data used in this study is far simpler compared

to large image data, coupled with other factors like more complex neural network structure and number of training samples. Thus, it was decided that initial training started off with 100 epoch increments.

4.4.1 Sinusoidal Data

The first dataset used to train the model was using the sinusoidal dataset. Let $X \sim U[-1, 1]$, $\phi \sim Normal(0, 1)$ and $Y = \sin(4x) + 0.2\phi$. Using the initial hyperparameters as in Table 3.2, the following is the results over 100 epoch increments, naming this version SGR01 (short for Sinusiodal Generated Results 01) for ease of future reference:

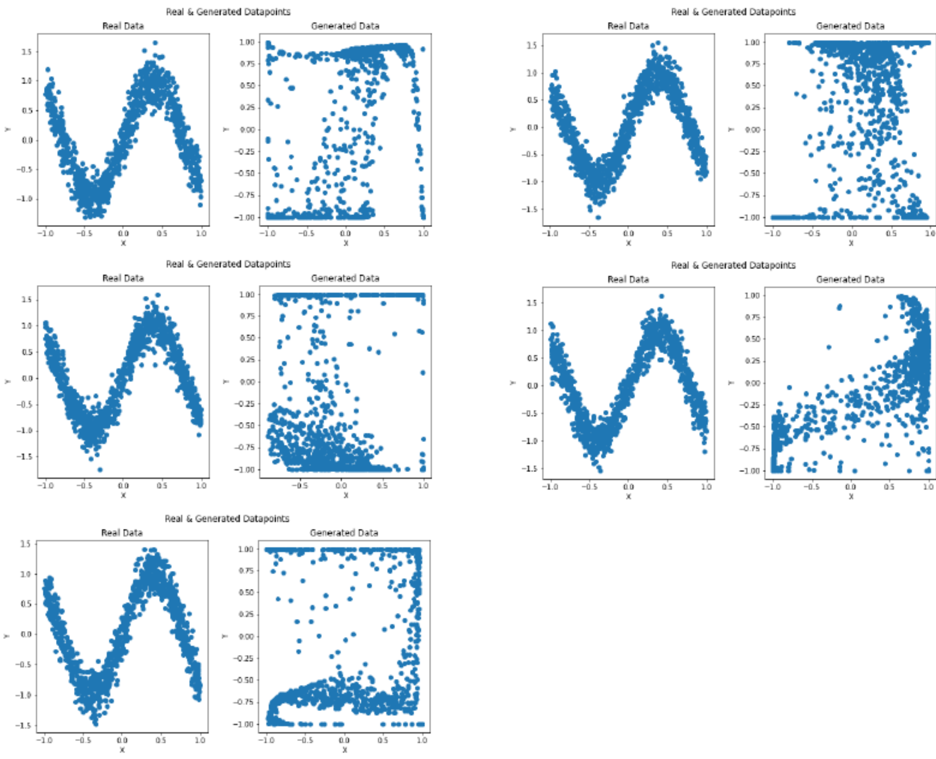


Figure 4.3: 500 Epochs of SGR01 Ordered from Left-to-right, Top-to-bottom

Figure 4.4: SGR01 Loss

From the results, it shows that there is a growing disparity in convergence. That the generator loss climbs higher and higher with each iteration. Interestingly, it was found that when the generator neural net is freed from the constraint of producing values ranging from -1 to 1, that is, while it does not use a hyperbolic tangent function, without changing any other parameter, it seemed to produce more meaningful outputs and loss function, as can be seen in Figure 4.5. However, it is noticeable that the loss of both the generator and critic show a repeating rise and fall pattern, which is presumably due to the learning rate being too large, which causes the model to approach and overstep its minimum point, restarting the path to its minimum. Multiple attempts were applied to find the optimum learning rate, which can be viewed in the appendix. At the 500th epoch of training the sine model 3 with learning rate 0.0002, critic and generator losses converged momentarily, where there was a shadow of the similar sine graph, which can be viewed in Figure 4.6, before it degraded in output quality upon further training.

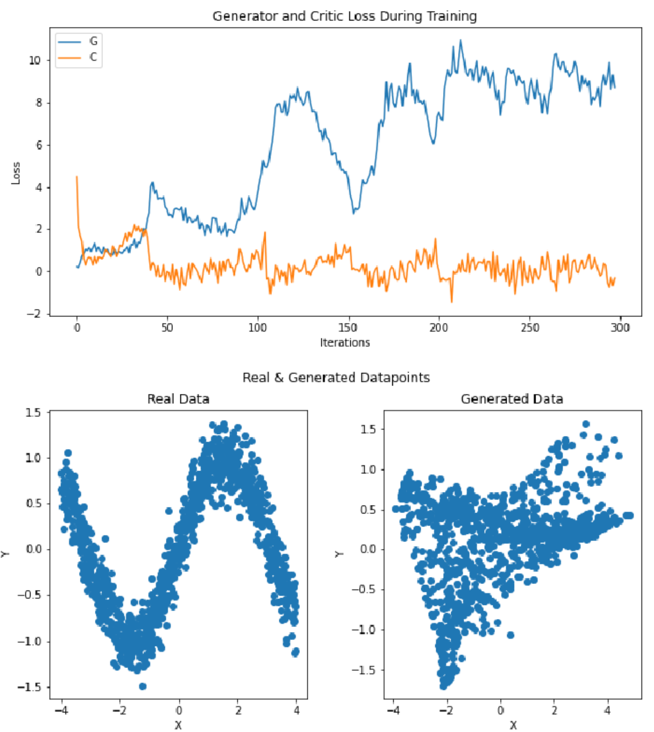


Figure 4.5: SGR02 Loss and Results at 300 Epochs

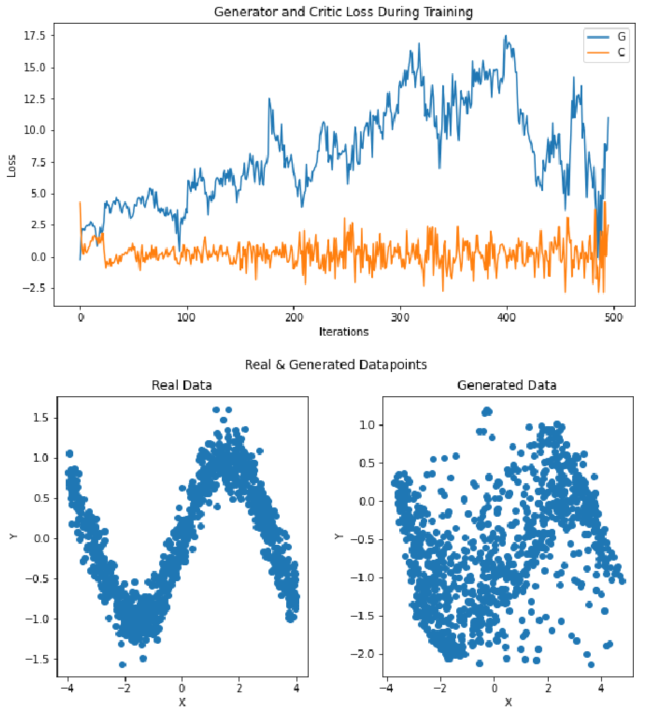


Figure 4.6: SGR03 Loss and Results at 500 Epochs

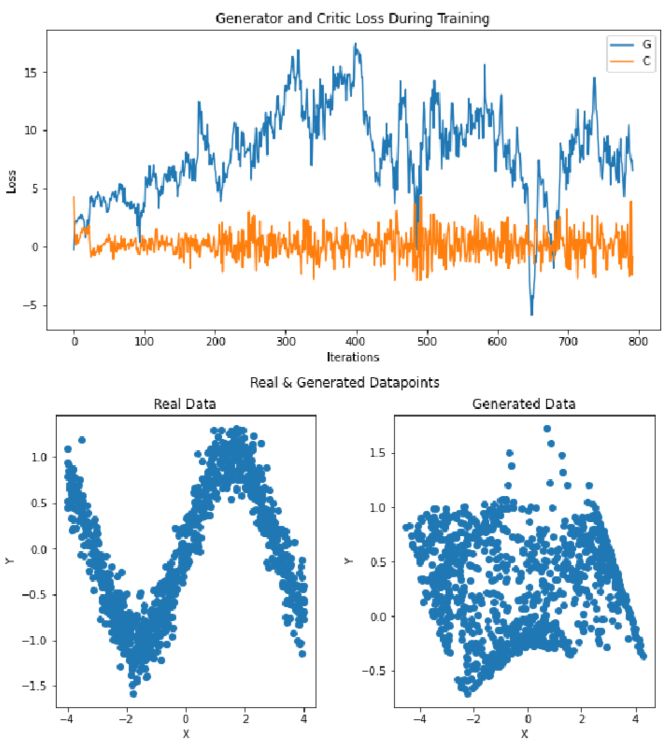


Figure 4.7: SGR03 Loss and Results Beyond 500 Epochs

It can also be observed that throughout the previous figures, the learning rate of the critic would flatten out from the very start, which led to the query on whether the gradient penalty value was too strong for this particular set up. According to the paper by Arjovsky et al, the gradient penalty coefficient functions to penalise the critic for producing gradients that are high in magnitude. Thus, reducing this value should subsequently prevent the excessive flattening of gradients. The following figures show the results repeating the above with a smaller gradient penalty value 9.5, 9.0, and 8.5, comparing outputs at 500 epochs.

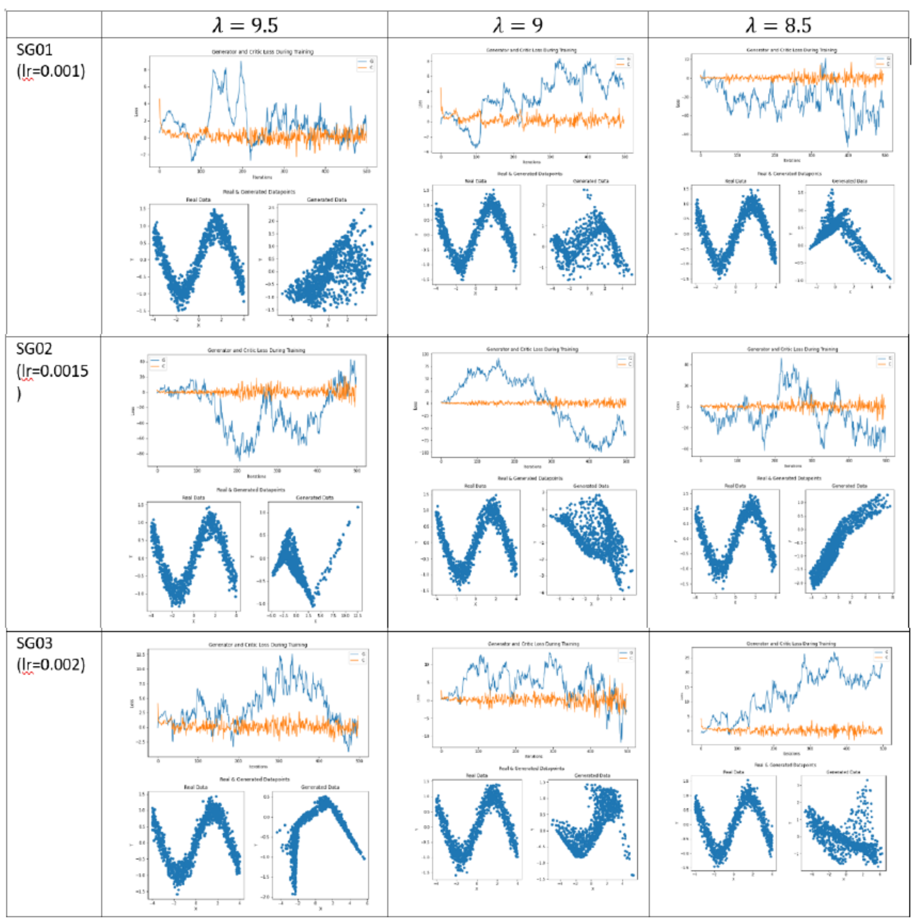


Figure 4.8: Comparison Between Varying Learning Rates and Gradient Penalty Values at 500 Epochs

From this comparison, the non-existence in learning rate patterns can be viewed clearly. While the individual effects of increasing and decreasing learning rates and gradient penalty values are known, this proving that optimality in training the model is not solely achieved by maximizing or minimizing individual parameters, but rather by determining the optimal configuration of these parameters.

Nevertheless, upon visual inspection, the top two most ideal shapes produced by the 9 combinations above are SG01 with $\lambda = 9$ and SG03 with $\lambda = 9$. Narrowing it down, in terms of range of generated data, SG03 trumps SG01 due to the outliers in SG01 with y values beyond $y = 2$. This instability in loss could arise from the lack of diverse samples in each batch Brock et al. (2019). Therefore, tweaking from the SG03 configuration with $\lambda = 9$, attempted runs were made with increased batch sizes from 100 to 200. At a batch size of 200,

after 800 epochs, the generated plot produced resembled that of the real data, which still has room for improvement, but it is significantly better than all past tries. In terms of training speed, training was 15.16 seconds per 100 epochs, summing to about 121.03 seconds for 800 epochs.

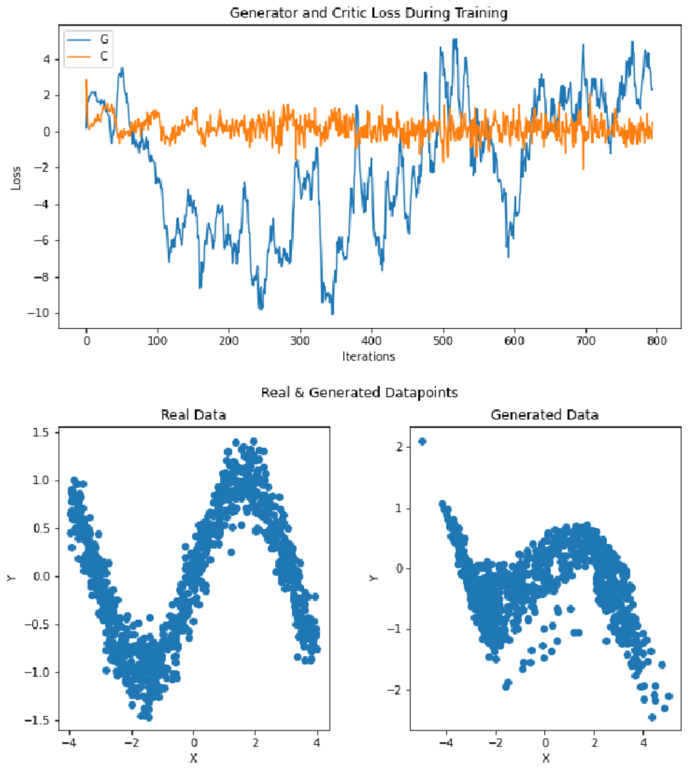


Figure 4.9: SGR03 Loss and Results with $\lambda = 9$ and batch size = 200 at 800 Epochs

4.4.2 Linear Data

With this same model architecture and hyperparameters, a linear dataset is used. The purpose of maintaining the use of this model architecture is to portray the dynamic ability and flexibility of a single model. For this section, real data is created in the form $(X = x, Y = y)$ with $X \sim U[-1, 1], \phi \sim Normal(0, 1)$ and $Y = X + 0.2\phi$

At the current training configuration, the results using a linear dataset forms generated data with a shape closely resembling that of real data, The relatively quick convergence implies that the linear dataset can be considered a simple task for this configuration of WGAN. For ease of naming, this configu-

ration for the linear dataset is named L01.

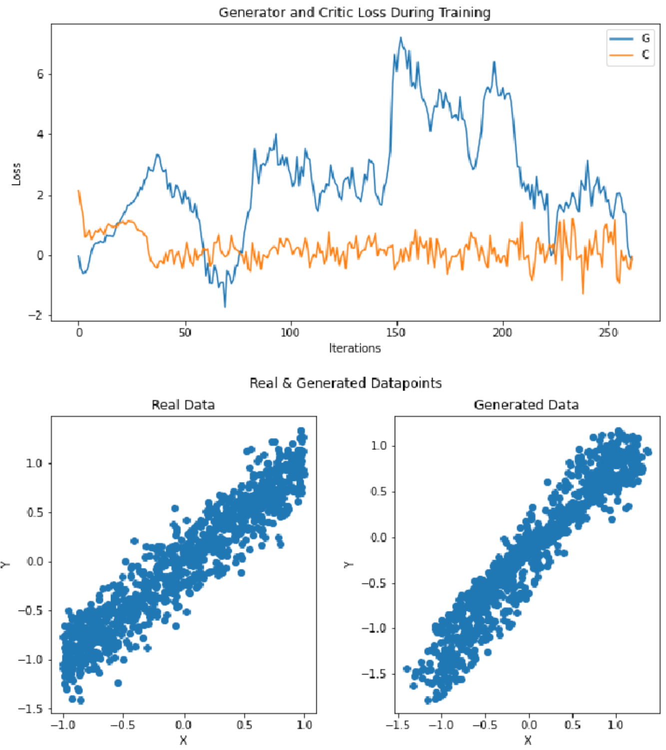


Figure 4.10: L01 Loss and Results with $\lambda = 9$ and batch size = 200 at 100 Epochs

With that established, it is also worth noting that overtraining the model leads to inconsistent outputs, as overtraining causes the model to overstep the minimum of the descent in training. Below are the following outputs at 300, 350, and 400 epochs. Training speed for Linear data took an average of 15.24 seconds per 100 epochs.

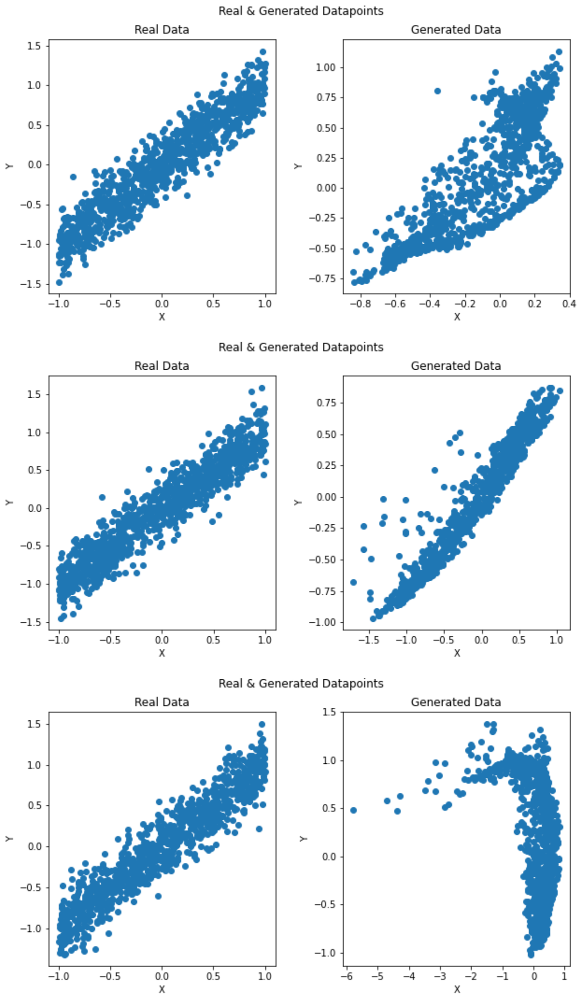


Figure 4.11: L01 Loss and Results with $\lambda = 9$ and batch size = 200 at 300, 350 and 400 Epochs

4.4.3 Sigmoidal Data

The sigmoidal dataset is representative of a cumulative normally distributed data, and is in the form $(X = x, Y = y)$ with $X \sim U[-1, 1], \phi \sim Normal(0, 1)$ and $Y = expit(X) + 0.2\phi$, with expit() being a logistic sigmoid function from the scipy library in Python. The difficulty in learning the distribution of this dataset was evident in contrast to the linear dataset. Upon visual inspection, the generator attempts to wrap the shape of the generated data points in two directions, accommodating for the linear section of the sigmoidal curve and the horizontal ends of the curve. At certain times, the shadow of the sigmoidal curve makes an appearance but is nothing worth contending against real data. Training

speed of sigmoidal data took an average of 18.01 seconds per 100 epochs.

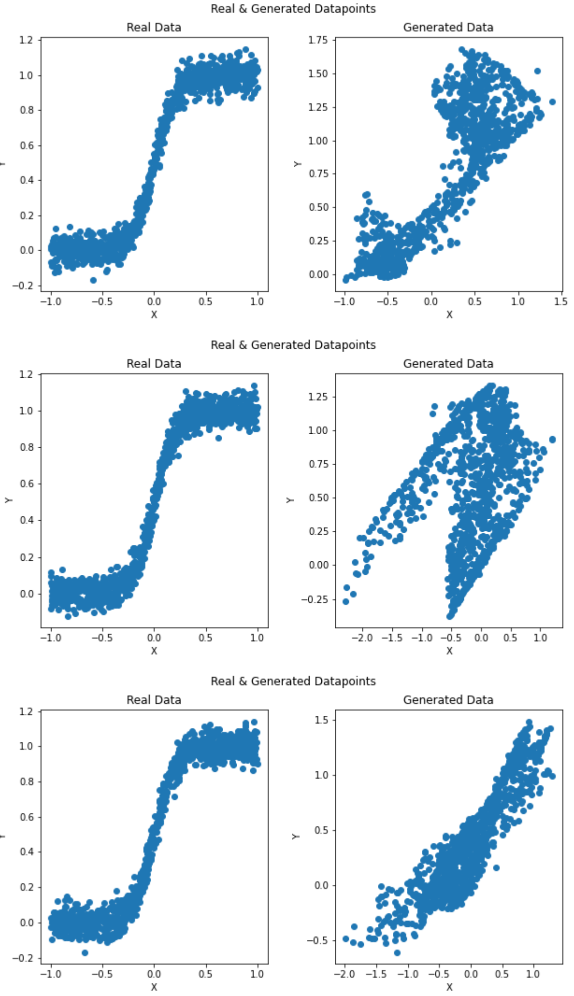


Figure 4.12: Sigmoid Loss and Results with $\lambda = 9$ and batch size = 200 at 300, 700 and 1100 Epochs

4.4.4 Annular Data

Increasing the difficulty of the input data to one that is multi-modal in nature, the previously sufficient framework lacks in dynamic ability for this scenario. The annulus dataset is formed as $(X = x, Y = y)$ with θ = 1000 values of equal spacing between 0 and 2π, $V \sim$ U(0,1), $X = \sin\theta + 0.75V$, and $Y = \cos\theta + 0.75V$, the model tries to produce annular data as per its train data but fails to do so. In the following figure, it can be observed that the generated data tries to morph its shape by being projected across different directions but do not quite make it, even at 1300 Epochs. Training speed for annular data averages at

15.10 seconds for 100 epochs.

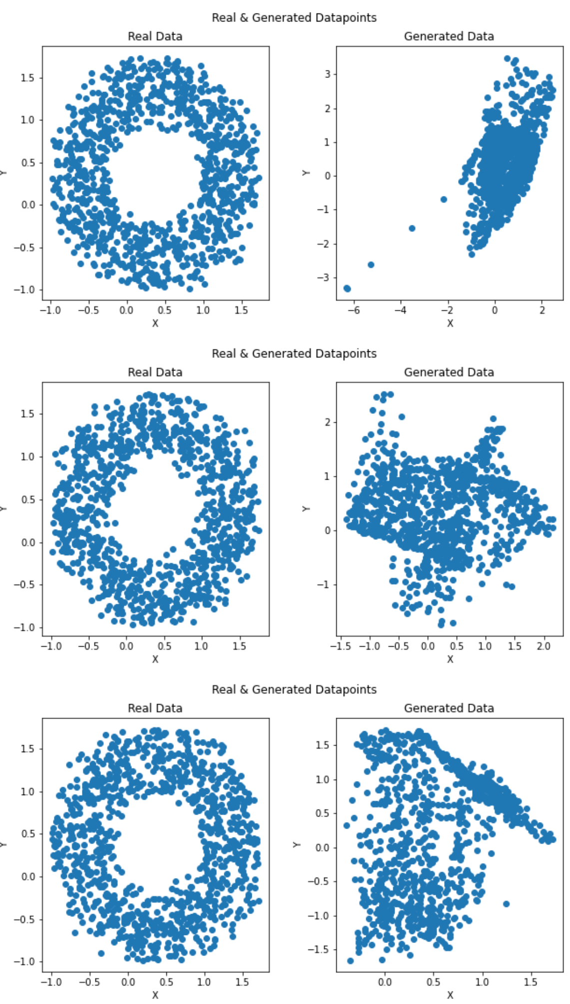
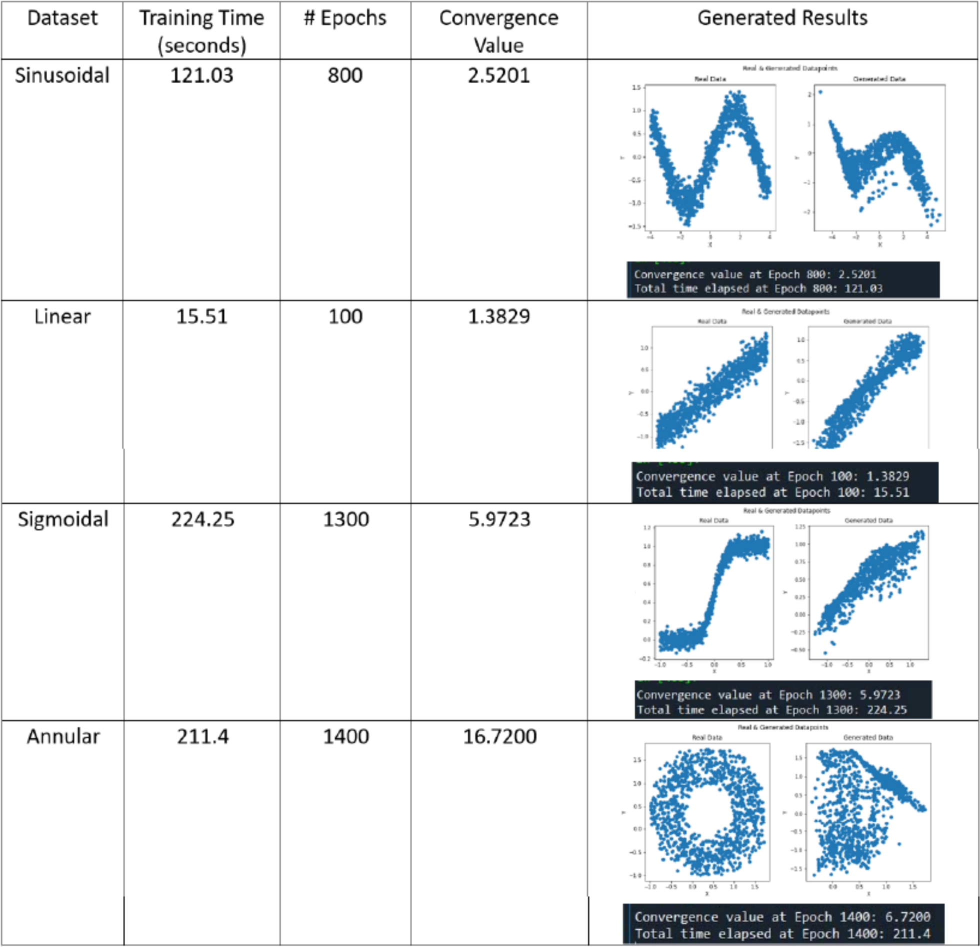


Figure 4.13: Annular Data WGAN-GP Loss and Results with $\lambda = 9$ and batch size = 200 at 400, 900 and 1300 Epochs from Left to Right

4.5 Summary

In short, the generated data from linear data produced the best result among the four different datatypes. According to the comparison figure shown below, the convergence value of annular data was the worst at 16.720, as compared to linear data at 1.383 units, where convergence is taken as the norm of the difference between generator and discriminator losses, since convergence occur when both distributions overlap and converge with each other.



Dataset	Training Time (seconds)	# Epochs	Convergence Value	Generated Results
Sinusoidal	121.03	800	2.5201	Convergence value at Epoch 800: 2.5201 Total time elapsed at Epoch 800: 121.03
Linear	15.51	100	1.3829	Convergence value at Epoch 100: 1.3829 Total time elapsed at Epoch 100: 15.51
Sigmoidal	224.25	1300	5.9723	Convergence value at Epoch 1300: 5.9723 Total time elapsed at Epoch 1300: 224.25
Annular	211.4	1400	16.7200	Convergence value at Epoch 1400: 6.7200 Total time elapsed at Epoch 1400: 211.4

Figure 4.14: Summary between Best Outputs of Different Datatypes

4.6 Discussion

Various inconsistencies in literature and practice was found throughout this study regarding the WGAN-GP. A noticeable pattern throughout all trials prevailed, where the generator loss had more drastic range of movement while the critic loss hovered around 0. Interestingly, this was sufficient to cause an improvement in generated data over time, given the hyperparameters set were suitable. This arguably opposes that written by Arjovsky et al, that states that the Earth Movers' distance is a good measure for the distance between distributions, better than that of JS divergence and KL divergence, where more is discussed in Chapter 2.

Furthermore, it is also possible to overtrain the GAN in this study, which again contradicts what was discussed earlier regarding how WGAN-GP has mitigated the worry on overtraining the generator or discriminator, and how training

the critic to completion is made possible. In the above experiments, it was found that overtraining the model could lead to regressed performance of the generator, as in Figure 4.12.

CHAPTER 5

CONCLUSION

5.1 Conclusion

To conclude, a WGAN can be deployed as a probability distribution construction tool, with all toy distributions overcomed by the model under 1000 iterations, using less than 25 minutes, aside from the annulus dataset. The search for the optimal hyperparameter combination together with the optimal neural network architecture does not hold a maximal or minimal relationship, that is, the increase in value of a hyperparameter does not imply a better model performance. Rather, it is the combined influence of all hyperparameters that achieve the model's optimal performance, which is a challenging task. A paper by Google Brain emphasises the inconsistency in performance of the different flavous of GANs against the original GAN (Lucic et al., 2018). This challenges the integrity of the GAN performance aross different datatypes. Proper hyperparameter setting is critical, such that theory backing the improvements of later versions of GAN is not valid when improper hyperparameters are used. The biggest difficulty lies in finding the optimal hyperparameter and model architechture configuration, as it is undoubtedly infeasible to test all combinations. Thus, further experimentation is required on this matter to provide more robust evidence for a more easily attainable and fool-proof model.

In most cases, the convergence of the WGAN-GP model reflects the quality of generated samples in their likeness to train data. The use of the earth mover's distance

5.2 Recommendation for Future Improvements

Upon completeing this study within the circumstances and limitations provided, a set of improvements was found could be implemented in future research regarding this matter.

Firstly, the dynamics of GANs should be explored and studied for the solid foundation and mathematical reasoning for the correlation between fluc-

tuations of model loss and generated data, especially in the area of using 2-dimensional data as train data. This will allow a mathematically backed logical deduction which will aid in the sense of direction in hyperparameter tuning.

Second, a more systematic method of searching for the optimal hyperparameters should be used, which was lacking in this study. This is due to the nature of GANs that is highly difficult to train, coupled with the many hyperparameter and model architecture configurations that need to be tuned in co-dependence with each other.

In regards to the statistical bridging between this approach and what could be done better, to reiterate, the transfer between generation of cumulative data to raw data is a significant loophole. The use of a better method to do so could be employed in future research, so that the dimensionality of the end data can be reduced and compared with well known statistical methods such as the Mann-Whitney U test and Kolmogorov Smirnov test for 1D data.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X., 2016. 'Tensorflow: Large-scale machine learning on heterogeneous distributed systems'.

Arjovsky, M. and Bottou, L., 2017. 'Towards principled methods for training generative adversarial networks'.

Arjovsky, M., Chintala, S. and Bottou, L., 2017. 'Wasserstein gan'.

Aït-Sahalia, Y. and Kimmel, R., 2007. 'Maximum likelihood estimation of stochastic volatility models', *Journal of Financial Economics* **83**(2), 413–452.

Brock, A., Donahue, J. and Simonyan, K., 2019. 'Large scale gan training for high fidelity natural image synthesis'.

Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W. and Robins, J., 2018. 'Double/debiased machine learning for treatment and structural parameters', *The Econometrics Journal* **21**(1), C1–C68.

Dai, B., Wang, Z. and Wipf, D. P., 2019. 'The usual suspects? reassessing blame for VAE posterior collapse', *CoRR* **abs/1912.10702**.

Dalal, S., Fowlkes, E. and Hoadley, B., 1989. 'Risk analysis of the space shuttle: Pre-challenger prediction of failure', *Journal of The American Statistical Association - J AMER STATIST ASSN* **84**, 945–957.

Di Paola, G., Bertani, A., De Monte, L. and Tuzzolino, F., 2018. 'A brief introduction to probability', *J. Thorac. Dis.* **10**(2), 1129–1132.

Fisher, R. A., 1992. On the mathematical foundations of theoretical statistics, *in* 'Springer Series in Statistics', Springer New York, pp. 11–44.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial nets, *in* 'Advances in neural information processing systems', pp. 2672–2680.

Google Colaboratory: Frequently Asked Questions, n.d.. `https://research.google.com/colaboratory/faq.html`. Accessed: 2023-04-07.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. and Courville, A. C., 2017. 'Improved training of wasserstein gans', *CoRR* **abs/1704.00028**.

Hack, J., n.d.. 'Pybn documentation', `https://hackl.science/pybn/`. Accessed: 2023-04-07.

Hu, A., Xie, R., Lu, Z., Hu, A. and Xue, M., 2021. 'Tablegan-mca: Evaluating membership collisions of gan-synthesized tabular data releasing', *CoRR* **abs/2107.13190**.

Huang, H., Yu, P. S. and Wang, C., 2018. 'An introduction to image synthesis with generative adversarial nets'.

Jain, R. B. and Wang, R. Y., 2008. 'Limitations of maximum likelihood estimation procedures when a majority of the observations are below the limit of detection', *Analytical Chemistry* **80**(12), 4767–4772.

Kang, H.-H. and Liu, S.-B., 2014. 'The impact of the 2008 financial crisis on housing prices in china and taiwan: A quantile regression analysis', *Economic Modelling* **42**, 356–362.

Kang, M., Shin, J. and Park, J., 2022. 'Studiogan: A taxonomy and benchmark of gans for image synthesis'.

Kingma, D. P. and Ba, J., 2017. 'Adam: A method for stochastic optimization'.

Lucic, M., Kurach, K., Michalski, M., Gelly, S. and Bousquet, O., 2018. 'Are gans created equal? a large-scale study'.

Nowozin, S., Cseke, B. and Tomioka, R., 2016. ‘f-gan: Training generative neural samplers using variational divergence minimization’.

Park, J. and Haran, M., 2020. ‘Reduced-dimensional monte carlo maximum likelihood for latent gaussian random field models’, *Journal of Computational and Graphical Statistics* **30**(2), 269–283.

Phillips, T. R. F., Heaney, C. E., Benmoufok, E., Li, Q., Hua, L., Porter, A. E., Chung, K. F. and Pain, C. C., 2022. ‘Multi-output regression with generative adversarial networks (MOR-GANs)’, *Applied Sciences* **12**(18), 9209.

Radford, A., Metz, L. and Chintala, S., 2016. ‘Unsupervised representation learning with deep convolutional generative adversarial networks’.

Rao, C. R., 1957. ‘Maximum likelihood estimation for the multinomial distribution’, *Sankhya: The Indian Journal of Statistics (1933-1960)* **18**(1/2), 139–148.

Sadeh, I., Abdalla, F. B. and Lahav, O., 2016. ‘Annz2: Photometric redshift and probability distribution function estimation using machine learning’, *Publications of the Astronomical Society of the Pacific* **128**(968), 104502.

Wei, Y. and Jiang, Z., 2022. ‘Estimating parameters of structural models using neural networks’, *USC Marshall School of Business Research Paper* .

Xiang, P., Xiang, S. and Zhao, Y., 2023. ‘Texturize a gan using a single image’.

Zhang, L., Ding, X., Ma, Y., Muthu, N., Ajmal, I., Moore, J. H., Herman, D. S. and Chen, J., 2019. ‘A maximum likelihood approach to electronic health record phenotyping using positive and unlabeled patients’, *Journal of the American Medical Informatics Association* **27**(1), 119–126.

APPENDICES

APPENDIX A: Prelimenary Analysis Code

```
import torch.nn as nn
from torch.utils.data import Dataset


# create uniform distribution dataset class
class RandomUniformDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        return self.data[index]

# create critic class
class Critic(nn.Module):
    def __init__(self, in_features):
        super().__init__()
        self.crit = nn.Sequential(
            nn.Linear(in_features, 10), # reduces the output from 1 to 10
            nn.LeakyReLU(0.2), # apply leaky relu with gradient 0.2
            nn.LayerNorm(10), # apply layer normalisation on input of size 10
            nn.Dropout(0.2),
            nn.Linear(10, 30), # reduces the output from 1 to 10
            nn.LeakyReLU(0.2), # apply leaky relu with gradient 0.2
            nn.LayerNorm(30), # apply layer normalisation on input of size 10
            nn.Dropout(0.2),
            nn.Flatten(),
            nn.Linear(30, 1),
            )

    def forward(self, x): # forward pass of nn
        return self.crit(x)
```

Figure 5.1: Code Part 1

```
# create generator class
class Generator(nn.Module):
    def __init__(self, z_dim, in_features):
        super().__init__()
        self.gen = nn.Sequential(
            nn.Linear(z_dim, 10),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(10),
            nn.Linear(10, 6),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(6),
            nn.Linear(6, 6),
            nn.LeakyReLU(0.2),
            nn.BatchNorm1d(6),
            nn.Linear(6, in_features),
            #nn.LeakyReLU(0.2),
            #nn.BatchNorm1d(3),
            #nn.Linear(3, in_features),
            #nn.Tanh() # ensures output is btw -1, 1
            )
    def forward(self, x): # x dimension should be batch_size x z_dim
        return self.gen(x)
```

Figure 5.2: Code Part 2

```
fixed_noise = torch.rand((batch_size, z_dim)).to(device)

start_time = time.time()
train_step(gen, crit, num_epochs, dataloader, fixed_noise, opt_critic, opt_gen, \
           g_losses, c_losses, start_time=0)
elapsed_time = elapsed_time = time.time() - start_time
print(f"total time elapsed: {elapsed_time:.2f} seconds")
```

Figure 5.3: Code Part 3

```
# initialise the critic & generator
crit = Critic(in_features).to(device)
gen = Generator(z_dim, in_features).to(device)

# create noise matrix
noise = torch.randn((batch_size, z_dim)).to(device)

# Real Data
x = np.random.uniform(-1, 1, sample_size)
phi = np.random.normal(0, 1, sample_size)
def func(x):
    return x + 0.2*phi
y = func(x)
data = torch.Tensor(list(zip(x,y)))

dataloader = DataLoader(data, batch_size=batch_size, shuffle=True)

# set optimiser for discriminator and generator
opt_critic = optim.Adam(crit.parameters(), lr=lr, betas=(beta1, beta2))
opt_gen = optim.Adam(gen.parameters(), lr=lr , betas=(beta1, beta2))

# create lists to record progress
g_losses = []
c_losses = []

# load model
# if file exists, then load checkpoint
fname = "./training_checkpoints/final_wgan_gp_linear_5_lr02_p90_bs300.pth.tar"
p = pathlib.Path(fname)
if p.exists():
    load_checkpoint(torch.load(fname), gen, crit, opt_gen, opt_critic, g_losses, c_losses)
```

Figure 5.4: Code Part 4

```
# initialise the critic & generator
crit = Critic(in_features).to(device)
gen = Generator(z_dim, in_features).to(device)

# create noise matrix
noise = torch.randn((batch_size, z_dim)).to(device)

# Real Data
x = np.random.uniform(-1, 1, sample_size)
phi = np.random.normal(0, 1, sample_size)
def func(x):
    return x + 0.2*phi
y = func(x)
data = torch.Tensor(list(zip(x,y)))

dataloader = DataLoader(data, batch_size=batch_size, shuffle=True)

# set optimiser for discriminator and generator
opt_critic = optim.Adam(crit.parameters(), lr=lr, betas=(beta1, beta2))
opt_gen = optim.Adam(gen.parameters(), lr=lr , betas=(beta1, beta2))

# create lists to record progress
g_losses = []
c_losses = []

# load model
# if file exists, then load checkpoint
fname = "./training_checkpoints/final_wgan_gp_linear_5_lr02_p90_bs300.pth.tar"
p = pathlib.Path(fname)
if p.exists():
    load_checkpoint(torch.load(fname), gen, crit, opt_gen, opt_critic, g_losses, c_losses)
```

Figure 5.5: Code Part 5

```
fixed_noise = torch.rand((batch_size, z_dim)).to(device)

start_time = time.time()
train_step(gen, crit, num_epochs, dataloader, fixed_noise, opt_critic, opt_gen, \
           g_losses, c_losses, start_time=0)
elapsed_time = elapsed_time = time.time() - start_time
print(f"total time elapsed: {elapsed_time:.2f} seconds")
```

Figure 5.6: Code Part 6

```
# Test trained generator
noise_test = torch.rand(1000, z_dim).to(device)
plot_fake = gen(noise_test).detach().numpy()
plot_real = data[:1000,:]
b_fake, m_fake = polyfit(plot_fake[:,0], plot_fake[:,1], 1)
b_real, m_real = polyfit(plot_real[:,0], plot_real[:,1], 1)

# get results of losses as
plt.figure(figsize=(10,5))
plt.title("Generator and Critic Loss During Training")
plt.plot(g_losses,label="G")
plt.plot(c_losses,label="C")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.savefig(f'./final_results_linear_5_lr02_p90_bs300/loss{len(g_losses)}.png')
plt.show()


# Plot real and fake into plane
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
fig.suptitle('Real & Generated Datapoints')
ax1.scatter(plot_real[:,0], plot_real[:,1])
#ax1.plot(plot_real[:,0], m_real*plot_real[:,0]+b_real)
ax1.set_title('Real Data')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')

ax2.scatter(plot_fake[:,0], plot_fake[:,1]) #scatter(p_fake[:,0], p_fake[:,1])
#ax2.plot(plot_fake[:,0], m_fake*plot_fake[:,0]+b_fake)
ax2.set_title('Generated Data')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
#ax2.set_xlim([0, 1.0])
#ax2.set_ylim([0, 0.5])
```

Figure 5.7: Code Part 7

```
plt.subplots_adjust(wspace=0.3)
plt.savefig(f'./final_results_linear_5_lr02_p90_bs300/plot{len(g_losses)}.png')
plt.show()

# Plot random variables
real_randvar = get_rv(plot_real)
fake_randvar = get_rv(plot_fake)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
fig.suptitle('Histogram of Random Variable')
ax1.hist(real_randvar)
ax1.set_title('Real Data')
ax1.set_xlabel('X')
ax1.set_ylabel('Frequency')

ax2.hist(fake_randvar)
ax2.set_title('Generated Data')
ax2.set_xlabel('X')
ax2.set_ylabel('Frequency')

plt.subplots_adjust(wspace=0.3)
fig.savefig(f'./final_results_linear_5_lr02_p90_bs300/rv{len(g_losses)}.png')
plt.show()
```

Figure 5.8: Code Part 8

```
# Plot convergence
conv = []
for i in range(len(g_losses)):
    c = abs(g_losses[i]-c_losses[i])
    conv.append(c)
plt.figure(figsize=(10,5))
plt.title("Generator and Critic Convergence")
plt.plot(conv)
plt.xlabel("Iterations")
plt.ylabel("Convergence")
plt.savefig(f'./final_results_linear_5_lr02_p90_bs300/conv{len(g_losses)}.png')
plt.show()
```

Figure 5.9: Code Part 9

```python
import torch
import numpy as np
from scipy.stats import pearsonr, ks_2samp, kstwobign
import time
from matplotlib import pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel
from numpy import random

device = "cuda" if torch.cuda.is_available() else "cpu"
penalty=9
critic_iterations = 5
batch_size = 200
g_losses = []
c_losses = []

# create gradient penalty function (NO ISSUE)
def gradient_penalty(critic, real, fake, device="cpu"):
    BATCH_SIZE, cols = real.shape
    epsilon = torch.rand((BATCH_SIZE, 1)).repeat(1, cols).to(device)
    interpolated_values = real * epsilon + fake * (1 - epsilon) # WGAN-GP interpolated values

    # calculate critic scores for WGAN-GP loss
    mixed_scores = critic(interpolated_values)

    # evaluate gradient of mixed_scores wrt interpolated_values
    gradient = torch.autograd.grad(
        inputs = interpolated_values,
        outputs = mixed_scores,
        grad_outputs = torch.ones_like(mixed_scores),
        create_graph = True,
        retain_graph = True,
        only_inputs = True, # excludes unnecessary tensors from the gradient computation
    )[0]
```

Figure 5.10: Code Part 10

```python
    # flatten gradient values
    gradient = gradient.view(gradient.size(0), -1)
    # calculate gradient norm
    gradient_norm = gradient.norm(2, dim=1)
    gradient_penalty = torch.mean((gradient_norm - 1)**2) #(gradient of Crit(Interpolated Value) - 1)**
    return gradient_penalty

# def save_checkpoint
def save_checkpoint(state, filename= "./training_checkpoints/final_wgan_gp_linear_5_lr02_p90_bs300.pth.
    print("=> Saving checkpoint")
    torch.save(state, filename)

# def load_checkpoint
def load_checkpoint(checkpoint, gen, crit, opt_gen, opt_critic, g_losses, c_losses):
    print("=> Loading checkpoint")
    gen.load_state_dict(checkpoint['generator_state_dict'])
    crit.load_state_dict(checkpoint['critic_state_dict'])
    opt_gen.load_state_dict(checkpoint['optimizerG_state_dict'])
    opt_critic.load_state_dict(checkpoint['optimizerC_state_dict'])
    g_losses.extend(checkpoint['g_losses'])
    c_losses.extend(checkpoint['c_losses'])

def proj(x): # x is a 1x2 tensor
    m1 = torch.tensor([1.0,0.0])
    result = torch.matmul(m1, x)
    return result

# def prediction algorithm
def prediction_function(desired, z, gen, num_iterations):
    for i in range(num_iterations):
        output = gen(z) # 2x3
        loss = sum(abs(proj(output.t() - desired.t()))) # sum of each entry of output - desired
        loss.backward()
        z.data.sub_(0.015*z.grad)
        z.grad.zero_()
```

Figure 5.11: Code Part 11

```
def prediction_function(desired, z, gen, num_iterations):
    for i in range(num_iterations):
        output = gen(z) # 2x3
        loss = sum(abs(proj(output.t() - desired.t()))) # sum of each entry of output - desired
        loss.backward()
        z.data.sub_(0.015*z.grad)
        z.grad.zero_()

        # plot gen(z) x values, and plot compared to desired x values for each 5 iterations
        if (i + 1) % 5 == 0:
            plt.scatter(desired.detach().numpy()[:,0], output.detach().numpy()[:,0], alpha=0.3)
            plt.xlabel('xp')
            plt.ylabel('x')
            plt.savefig(f'./final_results_sin/prediction_plot{i}.png')
            plt.show()

        gradient = (output.detach().numpy()[49,0]-output.detach().numpy()[2,0])/(gen(z).detach().numpy(
        print("gradient = ", gradient)

    return z


def print_properties(plot_real_np, plot_fake_np):
    p_value, ks_statistic = ks2d2s(plot_real_np[:, 0], plot_real_np[:, 1], plot_fake_np[:, 0], plot_fak

    print("KS Test P-Value:", p_value)
    print("KS Statistic:", ks_statistic)
```

Figure 5.12: Code Part 12

```
def get_rv(gen_data):
    # arrange data according to decreasing x order
    arranged_data = np.vstack(sorted(gen_data, key=lambda row: (row[0], row[1]), reverse=True))
    # for each data, get the difference between nth value and n-1th
    # value and append them to a list storing random variables
    rv = []
    for i in range(arranged_data.shape[0]):
        if i < arranged_data.shape[0]-1:
            rv.append(arranged_data[i][1]-arranged_data[i+1][1])
    return rv

def train_step(gen, crit, num_epochs, dataloader, fixed_noise, opt_critic, opt_gen, g_losses, c_losses,
    step = 0

    # set generator and critic to train mode
    gen.train()
    crit.train()

    # Training loop
    start_epoch = 0

    # Create loop to eihter load or initialise

    for epoch in range(start_epoch, num_epochs):

        # save checkpoint at every 3 epochs
        checkpoint = {
            'generator_state_dict': gen.state_dict(),
            'critic_state_dict': crit.state_dict(),
            'optimizerG_state_dict': opt_gen.state_dict(),
            'optimizerC_state_dict': opt_critic.state_dict(),
            'g_losses': g_losses,
            'c_losses': c_losses,
            #'epoch': epoch
            }
        save_checkpoint(checkpoint)
```

Figure 5.13: Code Part 13

```
# Get current time
    start_time = time.time()

    for batch_idx, real in enumerate(dataloader):
        real = real.to(device)
        # real = real.view(-1,2).to(device) # view used to reshape to value of in_features
        for _ in range(critic_iterations):

            # Train critic
            fake = gen(fixed_noise)
            crit_real = crit(real).view(-1) # flatten everything
            crit_fake = crit(fake).view(-1)

            gp = gradient_penalty(crit, real, fake, device = device)
            # WGAN-GP loss function
            lossC = -(torch.mean(crit_real) - torch.mean(crit_fake)) + penalty * gp
            crit.zero_grad()
            # Compute gradients of model's parameters wrt loss function
            lossC.backward(retain_graph = True)
            # Update parameters
            opt_critic.step()

            #opt_gen.zero_grad()

            #if batch_idx % critic_iterations == 0:

        # Train generator
        #fake_samples = gen(fixed_noise)
        output = crit(fake).view(-1) # flatten, fake reused made possible with retain_grap
        lossG = -torch.mean(output) # min -E(crit(fake)) (WGAN)
        gen.zero_grad()
        lossG.backward(retain_graph = True) # only update generator weights
        opt_gen.step()

        # Output training stats
        if batch_idx == 0:
```

Figure 5.14: Code Part 14

```
        opt_gen.step()

        # Output training stats
        if batch_idx == 0:
            elapsed_time = time.time() - start_time

            print(
                f"Epoch [{epoch}/{num_epochs}] Batch {batch_idx}/{len(dataloader)} \
                    Loss C: {lossC:.4f}, loss G: {lossG:.4f}, Elapsed Time: {elapsed_time:.2f} seco
                )

            # Save Losses for plotting over ALL iterations
            g_losses.append(lossG.item())
            c_losses.append(lossC.item())

        step  += 1
```

Figure 5.15: Code Part 15