Low-Dimensional Generative Adversarial Networks

Matin Mahmood

Year 4 Project
School of Mathematics
University of Edinburgh
January 21, 2023

Abstract

Generative Adversarial Networks (GANs) are an active research area with a wide range of applications, but are plagued with training instability and lack of evaluation metrics. In this report, we investigate these nuances and prescribe possible solutions in two low-dimensional examples. Firstly, we propose a GAN to approximate the normal distribution and build a strong intuition of the inner workings of GANs. Using a rose figure, we visualize the failure modes of GANs including mode collapse. We successfully avoid mode collapse using one-sided label smoothing and prescribe a unique method to avoid mode collapse by sampling from the 2d-projection of samples on the surface of a sphere. During the course of the project, we comment on the limitations of our evaluation metrics and propose opportunities for future work.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Matin Mahmood*)

This report is dedicated to the memory of my late and beloved dog Chip.

Contents

Abstract ii

Contents vi

1 Introduction 1
- 1.1 Structure of this report . . . 1

2 Neural Networks 2
- 2.1 History . . . 2
- 2.2 Perceptrons and Neurons . . . 2
- 2.3 Structure of a Neural Network . . . 3
  - 2.3.1 Fully Connected Layer . . . 4
  - 2.3.2 Activation Functions . . . 4
- 2.4 Objective Function . . . 5
  - 2.4.1 Mean Squared Error . . . 5
  - 2.4.2 Cross Entropy . . . 5
- 2.5 Gradient Descent . . . 6
  - 2.5.1 Stochastic Gradient Descent . . . 7
  - 2.5.2 Adam . . . 7
- 2.6 Backpropagation . . . 8
- 2.7 Training a Neural Network . . . 8

3 Generative Adversarial Networks 10
- 3.1 Introduction . . . 10
  - 3.1.1 Applications of GANs . . . 10
- 3.2 A Min-Max Game . . . 11
- 3.3 Objective Function . . . 11
  - 3.3.1 Discriminator Loss . . . 11
  - 3.3.2 Generator Loss . . . 12
- 3.4 Tandem Training . . . 13
- 3.5 Generating the Normal Distribution . . . 14
  - 3.5.1 Varying the Noise Dimension . . . 15
  - 3.5.2 Reducing Generator Complexity . . . 16
  - 3.5.3 Reducing Generator & Discriminator Complexity . . . 18
  - 3.5.4 Varying Learning Rate . . . 19
- 3.6 Limitations of Evaluation . . . 21

4 Visualizing Failure Modes of GANs **22**
4.1 GAN Failures . . . 22
4.2 Generating Rose Curves . . . 23
4.2.1 Generator Complexity . . . 25
4.2.2 Discriminator Complexity . . . 25
4.2.3 Decreasing Generator Learning Rate . . . 26
4.2.4 One sided label smoothing . . . 27
4.2.5 Sampling from a hypersphere . . . 29
4.2.6 Training to ensure convergence . . . 30
4.3 Evaluation Limitations . . . 31

5 Conclusion **33**

A PyTorch GAN Implementations **34**
A.1 Normal Distribution GAN . . . 34
A.1.1 Generator Model . . . 34
A.1.2 Discriminator Model . . . 34
A.1.3 GAN Framework . . . 35
A.2 Rose GAN . . . 37
A.2.1 Rose Figure . . . 37
A.2.2 Generator Model . . . 37
A.2.3 Discriminator Model . . . 38
A.2.4 Generator Supervised Learning . . . 39
A.2.5 Discriminator Supervised Learning . . . 39
A.2.6 Hypersphere Sampling . . . 40

B Evaluation Metrics **41**
B.1 Kurtosis Test . . . 41
B.2 Skewness Test . . . 41
B.3 Shapiro Wilk Test . . . 41

Bibliography **44**

Chapter 1

Introduction

Over the recent years, the improvement in computational power and increase in the sheer amount of labeled datasets has given rise to deep neural networks. In this report, we will be investigating a framework known as Generative Adversarial Networks (GAN) that consists of two neural networks that compete against each other to generate samples from a data distribution. GANs are part of the family of generative models and are an active area of research. The GAN framework is highly adaptable and therefore has many applications ranging from astronomy to medical imaging. Their usefulness comes from being able to accurately generate high-dimensional data such as facial portraits and complex protein molecules [1]. Although GANs have proven to be state of the art generative models, they are still difficult to train and evaluate which limits their potential. We will be investigating these nuances with the help of low-dimensional examples and evaluate our models to prescribe possible solutions.

1.1 Structure of this report

In the next chapter, we will be reviewing concepts from neural networks that are relevant to the discussion of this report. In chapter 3 we introduce Generative Adversarial Networks mathematically and implement the generation of a 1-dimensional example to illustrate the various intricacies of the interplay between the adversarial subnetworks. We build on our understanding of Generative Adversarial Networks in chapter 4 by visualizing common failure modes of our implementation in a 2-dimensional example and report on possible solutions. In the last chapter, we present a conclusion of our work with Generative Adversarial Networks in low-dimensional scenarios.

Chapter 2

Neural Networks

2.1 History

The fundamental concept of neural networks dates to 1957 when Frank Rosenblatt published a paper called "The Perceptron" at the Cornell Aeronautical Laboratory [20]. However, it was not until the 1980s that Geoffrey Hinton David Rumelhart and Ronald Williams made several breakthroughs including the multilayer perceptron and backpropagation. All of them had a background in psychology, but many deep artificial neural networks that are popular today had prototypes in the 1980s.

2.2 Perceptrons and Neurons

The building blocks in a neural network are called neurons (a type of perceptron). A neuron can take a set of input values $(x_1, x_2, ..)$, pass them through a function, and produce an output value. Rosenblatt had proposed a simple rule to compute the output of a perceptron by introducing weights $(w_1, w_2, ..)$ that represent the relative significance of the inputs respectively. The neuron's binary output is determined by comparing the weighted sum to a pre-defined threshold.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{ threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{ threshold.} \end{cases} \tag{2.1}$$

We can simplify our expression by making three changes [16]:

i. writing $\sum_i w_i x_i$ as a dot product $w \cdot x$

ii. replacing the threshold with a bias variable θ

iii. rearranging the terms.

This yields,

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + \theta \leq 0 \\ 1 & \text{if } w \cdot x + \theta > 0. \end{cases} \tag{2.2}$$

It can be seen that a large value for the bias would result in the neuron's output to be 1 whereas a large negative value would most likely result in an output of 0.

We can produce different models by varying the weights and threshold. Moreover, by combining the neurons iteratively in multiple layers, we can influence subtle changes in the model, which can allow us to build models to carry out tasks such as handwritten digit recognition. However, if our network is built using the simple neurons, a small change in the parameters causes the neuron to invert its output value. In a network of neurons, this results in the complete network changing its learned parameters in a complex manner. Therefore, it is necessary that a small change in the input results in a small incremental change in the output of a neuron. To achieve this, we use a neuron that can output real values between 0 and 1. A commonly used function in the machine learning community is the **logistic function**:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{2.3}$$

Using the function of the simple neuron as an input to the logistic function, we can define the output of the logistic neuron as

$$\text{output} = \frac{1}{1+e^{-(w\cdot x+\theta)}}. \tag{2.4}$$

If $w\cdot x+\theta$ takes a value that is close to 0, then the resulting output will be close to 0.5, which which is analogous to the threshold in the perceptron. Moreover, small deviations from the 0 value result in small changes in the same direction for the output. However, when the value of $w \cdot x + \theta$ is largely negative then the output ≈ 0 as $\sigma(z) \to 0$ when $e^{-z} \to \infty$. Similarly, when the value of $w \cdot x + \theta$ is largely positive then the output ≈ 1 as $\sigma(z) \to 1$ when $e^{-z} \to -\infty$.

The logistic function is also very useful as its derivative w.r.t z is $\sigma(z)(1-\sigma(z))$ which makes it particularly useful as shown in the next section.

2.3 Structure of a Neural Network

Neural networks are layers of interconnected neurons. The first layer in a neural network is called the input layer and the final layer is called the output layer. There are also layers in between these two layers known as hidden layers. Neural networks with a large number of hidden layers are called Deep Neural Networks [6]. In this chapter, the discussion has been limited to feed-forward neural networks where the output of the previous layer is used as an input to the layer that follows it (except the input layer). The number of layers in a neural network defines the depth of the network while the number of neurons in a layer defines the width of the network. Practically, neural network implementations require the use of different types of layers.

2.3.1 Fully Connected Layer

The simplest type of layer is called a fully connected layer, linear layer, or dense layer. A fully connected layer is a linear transformation of the form:

$$f_1(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{\theta}$$

where W is a matrix of size *input-features* $\times$ *output-features* that consist of the weights w while $\boldsymbol{\theta}$ is a column vector of biases. The output of a network with two stacked fully connected layers f_1 and f_2 can be written as

$$f_2(f_1) = \boldsymbol{W_2} f_1 + \boldsymbol{\theta_2}$$

which is only capable of describing a linear transformation.

2.3.2 Activation Functions

To learn nonlinear functions, a fully connected layer is followed by an activation function $a(x)$. The activation function is applied element-wise to a set of inputs in order to transform them non-linearly.

Sigmoid

One such activation commonly used in binary classifiers is the sigmoid activation function as described by equation 2.3 in section 2.2. One drawback of the sigmoid function is the saturation of the output for large positive and negative values. This result in zero gradients for the parameters and thereby a lack of parameter updates.

Rectified Linear Activation Unit (ReLU)

In order to remedy the saturation problem of sigmoid functions, Rectified Linear Activation Units (ReLU) [15] are more popularly used in deep neural networks. The ReLU function is not continuously differentiable as it is described by the following equation:

$$a_{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0. \end{cases} \tag{2.5}$$

The left derivative of $a_{ReLU}(x)$ at $x = 0$ is 0 and the right derivative is 1, however in practice this does not pose a problem as this scenario is unlikely due to numerical precision. The *PyTorch* software implementation uses either the right or left derivative for non-differentiable functions. For large negative values, the gradient is zero and hence neurons can get stuck at an irrecoverable state of 0 known as the *dying ReLU problem*. For a deep neural network, a significant number of neurons can be affected by dying ReLU activation, rendering regions of a neural network useless.

Leaky ReLU (LReL or LReLU)

The Leaky ReLU (LReLU or LReL) [14] alleviates the dying ReLU problem by using a small negative value for inputs less than 0. The function is modified as follows,

$$a_{LReL}(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0. \end{cases} \tag{2.6}$$

Using a small gradient (commonly 0.01) for negative values prevent the gradient from saturating to a value of zero.

2.4 Objective Function

The goal of our neural network is to find a set of parameters, in particular, weights w and biases θ that results in the closest approximation of a target function $y(x)$ that we want the neural network to learn. Typically a dataset $\mathcal{D} = \{x^{(i)}, y^{(i)}\}$ of size n is employed for the task of approximating target values $y^{(i)}$. In order to quantify the proximity of the neural network's approximation $\hat{y}$ to the target function, we define an objective function L also referred to as a cost function or loss function which is minimized. The selection of an objective function is crucial for the accuracy of a network.

2.4.1 Mean Squared Error

A commonly used objective function is the Mean Squared Error (MSE), which is the average distance between the target values and the predicted values:

$$\text{MSE}(y, \hat{y}) = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}. \tag{2.7}$$

The squared difference ensures that the MSE is always positive and heavily penalizes any outlier predictions. This is useful for tasks where minimizing error is crucial, but can be counterproductive when outliers are irrelevant to the task at hand.

2.4.2 Cross Entropy

Cross Entropy (CE) is a loss metric that measures the divergence between target values and predicted values [11]. The CE for multiple classes is given by

$$\text{CE}(y, \hat{y}) = -\frac{\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i)}{n}. \tag{2.8}$$

In a neural network, the cross-entropy is independent of the layers and non-linearity functions that are used to output the target distribution. The cross-entropy increases exponentially as the difference between true and predicted probability distributions increases. For binary classes, the loss function is called the Binary Cross Entropy (BCE):

$$\text{BCE}(y, \hat{y}) = -(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})) \tag{2.9}$$

where $y \in \{0, 1\}$ is the true value and $\hat{y}$ is the predicted value such that $0 < \hat{y} < 1$. The binary cross-entropy is also called log-loss and is extensively used as the objective function for binary classifiers.

2.5 Gradient Descent

Gradient Descent is a technique used to minimize the objective function which is analogous to rolling down a ball from the top of a valley to find the minimum point [16]. The objective function is parameterized by parameters w_k. To find the change in the objective function ΔL, we compute the partial derivatives of the cost function w.r.t all parameters,

$$\Delta L \approx \sum_k \frac{\partial L}{\partial w_k} \Delta w_k \tag{2.10}$$

where Δw_k is a small change in the direction of w_k. Since we want to minimize L we move w_k in the opposite direction so that,

$$\Delta w_k = -\eta \frac{\partial L}{\partial w_k} \tag{2.11}$$

where η is a fixed step size known as the **learning rate**. This learning rate is analogous to the horizontal distance moved by the ball at each step. Substituting equation 2.11 in equation 2.10, our change in the objective function now becomes

$$\Delta L \approx \sum_k \frac{\partial L}{\partial w_k}(-\eta \frac{\partial L}{\partial w_k}) = -\eta \sum_k (\frac{\partial L}{\partial w_k})^2. \tag{2.12}$$

Since $(\frac{\partial L}{\partial w_k})^2 \geq 0$, we have $\Delta L \leq 0$ so the change in objective function will be 0 at the minimum point. We update w_k after each step in order to move closer to this minimum point. The **update step** is given by:

$$w_k \leftarrow w_k - \eta \frac{\partial L}{\partial w_k}.$$

The learning rate η must be chosen so that the optimization continues at a reasonable rate and manages to converge to the minimum point. If we choose a value for η that is too small, training might occur at a very slow rate. However, if we choose a large value η, we might not be able to find the minimum point. Moreover, for a large dataset $\mathcal{D}$ of size n, it can become computationally inefficient to calculate the derivative of the objective function as it is calculated over all input points.

2.5.1 Stochastic Gradient Descent

In order to make gradient descent more computationally efficient, Stochastic Gradient Descent (SGD) is commonly used to minimize an objective function. The idea of SGD is to perform gradient descent on a randomly chosen subset of $\mathcal{D}$ with size $m << n$ (known as a *minibatch*) for each update step [20]. SGD relies on the assumption that the estimated gradient over a minibatch is a close approximation of the gradient computed over the entire dataset.

2.5.2 Adam

Another popularly used technique to minimize the objective function is Adaptive Moment Estimation (ADAM), which uses different learning rates for each parameter [12]. ADAM is a combination of two gradient descent techniques and uses the exponential moving averages of the weight parameters at time t $(w_1^t, w_2^t, ..., w_k^t)$. The exponential moving average (EMA) is calculated as

$$EMA(u^t) = \begin{cases} u^t & t = 0 \\ u^t(1-\beta) + EMA(u^{t-1})\beta & t > 0 \end{cases} \tag{2.13}$$

The first gradient descent algorithm is *momentum* where updates to the weights are calculated using an EMA of the derivative of the objective function w.r.t parameter w_k^t so that

$$EMA_m^t = EMA\left(\frac{\partial L}{\partial w_k^t}\right)$$

where β_1 is the smoothing parameter used by momentum. The second gradient descent algorithm is *RMSProp* where updates to the weights are calculated using an EMA of the squared derivative of the objective function w.r.t parameter w_k^t as

$$EMA_v^t = EMA\left(\left(\frac{\partial L}{\partial w_k^t}\right)^2\right)$$

where β_2 is the smoothing parameter used by RMSProp. For ADAM, The initial values at $t = 0$ are initialized to 0 and as β_1 and β_2 are close to 1, the EMA values are biased towards 0. Therefore, bias correction is performed such that

$$\widehat{EMA_m^t} = \frac{EMA_m^t}{1-\beta_1} \qquad \widehat{EMA_v^t} = \frac{EMA_v^t}{1-\beta_2}$$

Finally, the update step for ADAM is given by

$$w_{k+1}^t \leftarrow w_k^{t-1} - \widehat{EMA_m^t}\left(\frac{\eta}{\sqrt{\widehat{EMA_v^t}} + \epsilon}\right)$$

where ϵ is a small constant with default value 10^{-8} to prevent division by zero. The parameters β_1 and β_2 are known as weight decay constants and have default

values 0.9 and 0.999, respectively.

2.6 Backpropagation

Backpropagation (also known as *backprop*) allows us to calculate gradients more efficiently by reusing calculated gradients in the chain rule [6]. This is important as we have seen in section 2.5 that the gradients of the the objective function have to be calculated repeatedly. In order to illustrate the algorithm, we define a simple neural network with layers l consisting of an input layer l_1, a hidden layer $l_2 = H$ and an output layer $l_3 = O$. Each layer l, except the input layer, has a corresponding weight matrix $W^{(l)}$ and a bias vector $\theta^{(l)}$ that is followed by an activation function with output $a^{(l)}$. To simplify calculations, we define weighted outputs of a layer as $z^{(l)} = W^{(l)}a^{(l-1)} + \theta^{(l)}$ such that $a^{(l)} = a(z^{(l)})$ for an activation function $a(x)$. The output of the input layer is equivalent to $a^{l_1}(x)$. For backpropagation, we start by calculating the derivatives of the objective function with respect to the weight parameters $w^{(O)}$ in the output layer by applying the chain rule:

$$\frac{\partial L}{\partial w^{(O)}} = \frac{\partial L}{\partial a^{(O)}} \frac{\partial a^{(O)}}{\partial z^{(O)}} \frac{\partial z^{(O)}}{\partial w^{(O)}}. \tag{2.14}$$

For the hidden layer H, we calculate the partial derivatives with respect to the weight parameters $w^{(H)}$ as

$$\frac{\partial L}{\partial w^{(H)}} = \frac{\partial L}{\partial a^{(O)}} \frac{\partial a^{(O)}}{\partial z^{(O)}} \frac{\partial z^{(O)}}{\partial a^{(H)}} \frac{\partial a^{(H)}}{\partial z^{(H)}} \frac{\partial z^{(H)}}{\partial w^{(H)}} \tag{2.15}$$

where we can reuse the partial derivatives w.r.t $a^{(O)}$ and $z^{(O)}$ from equation 2.14. We can also calculate the derivatives of the objective function with respect to the bias parameter by replacing w with θ. The re-use of the local gradients makes backprop, a highly efficient method for calculating gradients in a deep neural network.

2.7 Training a Neural Network

Training a neural network can be broken down into the following steps:

1. Prepare a dataset $\mathcal{D}$ of sufficient size n.

2. Define a neural network's structure (section 2.3) including the width and depth of the network with non-linearity (section 2.3.2) in order to introduce non-linearity and a final objective function (section 2.4) that is to be minimized.

3. Initialize weight matrices W to small random values and biases θ to 0.

4. Forward-pass input samples through the network to calculate the activation outputs $a^{(l)}$.

5. Backpropagate (section 2.6) through the network to compute gradients for all learnable parameters.

6. Update weights w and biases θ by descending the gradients using an optimizer (section 2.5) with a suitable choice of learning rate η.

7. Repeat step 6 until the change in gradient is less than a predefined threshold or the maximum number of iterations (also known as **epochs**) is reached.

Chapter 3

Generative Adversarial Networks

3.1 Introduction

Generative Adversarial Networks (GANs) were first proposed by Goodfellow et.al in June 2014 [7]. GANs are a framework consisting of a generator model and a discriminator model that are usually implemented as a multilayer neural network. The generator model learns the mapping from a latent space to a sample space that represents the data distribution. The discriminator model is usually a binary classifier that measures the probability of the input sample being from the true data distribution as opposed to the generated data distribution. GANs are incredibly useful and can be applied to a variety of problems where traditional maximum likelihood models are unable to produce satisfactory results.

3.1.1 Applications of GANs

GANs have been applied to a wide range of fields ranging from remote-sensing to medical imaging. We will explore three applications of GANs in the following sections.

Semi-supervised Learning

Semi-supervised learning is a combination of supervised learning (where training data is labeled) and unsupervised learning (where the training data is unlabeled). GANs can be used for the task of semi-supervised learning by modifying the discriminator to also act as a classifier that has access to labeled, unlabelled and generated samples as implemented in SSGAN [17]. This makes the application of GANs for supervised learning particularly useful where a majority of the dataset is unlabeled.

Data Augmentation and Generation

Traditional data augmentation involves applying a fixed number of operations (such as rotation, scaling and reflecting) to an existing dataset with the aim to train a network with improved generalization. With the application of GANs for data augmentation and generation, a dataset can be artificially increased with

new samples that also improves the network's ability to extract more features [2]. GANs have been applied in the field of remote-sensing to improve object detection in satellite imagery by augmenting datasets consisting of remote-images using models like MARTA GAN [13].

Image to Image Translation

The task of transferring the characteristic styles of a set of images to a separate set of images is called image to image translation. GANs can perform this task by feeding the generator the first set of images to generate look-alike images from the second set. Models such as Pix2Pix [10] and CycleGAN [31] have been successfully used on medical imagery to transform CT-scan imagery to MRI imagery.

3.2 A Min-Max Game

In a simplistic view, the two submodels are set against each other in a finite zero-sum game in order to reach a Nash Equilibrium. In this case, the generator and the discriminator are trained in tandem in order to converge to a saddle point, where the gain in loss of the generator is equal to the decrease in loss of the discriminator [5]. The loss function of the generator L_G is bound to the loss function of the discriminator L_D as

$$L_G = -L_D.$$

Once the saddle point of the min-max game is reached, the two submodels are said to be in an equilibrium. However, the entanglement of the equations is problematic because the generator is maximizing the same loss function that the discriminator is minimizing. During the initial phase of training, the samples produced by the generator are dissimilar to the real samples, which allows the discriminator to classify the generated samples correctly and causes vanishing gradients (section 4.1) for the generator [8]. Although impractical, the min-max equation is useful for theoretical analysis and is an active area of research [4].

3.3 Objective Function

In practice, the discriminator and generator have cost functions that are optimized separately, but are dependant on each other's parameters. The discriminator's output is defined by the function $D(x)$ while the output of the generator is defined by $G(z)$.

3.3.1 Discriminator Loss

Given a training sample x as input, the discriminator predicts $D(x)$ which is given a true label of 1. Therefore, the binary cross entropy (section 2.9) is given

by

$$\begin{aligned} \mathrm{BCE}(1, D(x)) &= -(1 \cdot \log D(x) + (1-1) \cdot \log(1 - D(x))) \\ &= -\log D(x). \end{aligned} \tag{3.1}$$

The discriminator should maximize $\log D(x)$. As log is a monotonic increasing function, the discriminator must maximize $D(x)$ for true samples. Similarly when a generated sample $G(z)$ is input to the discriminator, the predicted value $D(G(z))$ has a label of 0. The binary cross entropy is calculated as

$$\begin{aligned} \mathrm{BCE}(0, D(G(z))) &= -(0 \cdot \log D(G(z)) + (1-0) \cdot \log(1 - D(G(z))) \\ &= -\log(1 - D(G(z)). \end{aligned} \tag{3.2}$$

The discriminator should maximize $\log(1 - D(G(z))$. Therefore, the discriminator must minimize $D(G(z))$ for generated samples. Since the discriminator is fed true samples and generated samples in an alternating fashion, the optimization problem for one sample can be written as

$$\mathrm{Max}[\log D(x) + \log(1 - D(G(z)))].$$

For multiple samples in a mini batch, we can average the required cost values to give us following cost function for the discriminator:

$$L_D = -\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}\,(\boldsymbol{x})}[\log D(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{3.3}$$

where $p(x)$ is the probability distribution of real training data and $p(z)$ is the probability distribution of the noise distribution. We sample z from the noise distribution $p(z)$ which is typically a gaussian or uniform distribution. It can be viewed as a source of randomness that allows the generator to output a wide variety of different vectors.

3.3.2 Generator Loss

The generator wants to fool the discriminator, so the prediction of the discriminator on the generated sample $D(G(z))$ is given a label of 1. The binary cross entropy gives

$$\begin{aligned} \mathrm{BCE}(1, D(G(z))) &= -(1 \cdot \log D(G(z)) + (1-1) \cdot \log(1 - D(G(z)))) \\ &= -\log D(G(z)). \end{aligned} \tag{3.4}$$

The generator should maximize $\log(D(G(z)))$. Therefore the generator will try to minimize $\log(1 - D(G(z)))$. For multiple samples, the generator loss is given by

$$L_G = -\mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})} \log D(G(\boldsymbol{z})) \tag{3.5}$$

With a separate loss function for the generator network (equation 3.5) and the discriminator network (equation 3.3), we can optimize the adversarial cost functions more reliably [7].

3.4 Tandem Training

The training of the submodels in the adversarial network is done in tandem. The training process for the discriminator consists of calculating the loss function (section 3.3) by averaging the loss on real samples and fake samples. The discriminator can be discarded after the training is done as it is only used as a guide to train the generator. A summary of the tandem training process is given in Algorithm 1. In this case, we begin by sampling a minibatch of the latent samples z and use it as an input to the generator. Similar to the discriminator function, the generator function $G(z)$ must be a differentiable function that has parameters that can be learned by gradient descent (section 2.5). It is important to note that we usually represent the generator as a deep neural network, however it could also be any other type of model that satisfies the differentiability property. For both submodels, gradients are computed using backpropagation (section 2.6). We have applied an update of weights to the discriminator for each update of the generator. However, it is also possible to apply multiple training steps of the discriminator for each step in the generator [7].

Algorithm 1 Batch Tandem Training for Discriminator and Generator for E Epochs and B number of batches with batch size b

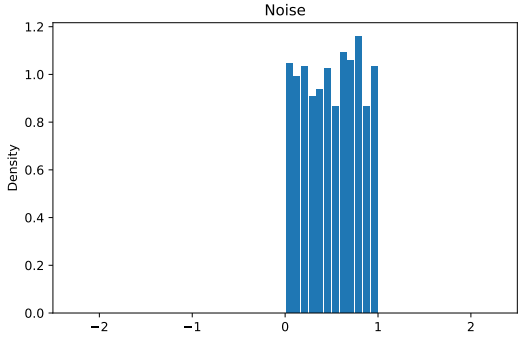
Require: $\boldsymbol{D}$ and $\boldsymbol{G}$ to be Differential Functions
  `target_ones` $\leftarrow (1^{(1)}, \ldots, 1^{(b)})$
  `target_zeros` $\leftarrow (0^{(1)}, \ldots, 0^{(b)})$
  for 1 to E **do**
    for 1 to B **do** ▷ Discriminator Training Step
      `real_samples` $\leftarrow$ `Sample Data Distribution`
      `pred_real` $\leftarrow \boldsymbol{D}$(`real_samples`)
      `loss_real` $\leftarrow \boldsymbol{BCE}$(`target_ones, pred_real`)
      `latent_vec` $\leftarrow$ `Sample Noise Distribution`
      `fake_samples` $\leftarrow \boldsymbol{G}$(`latent_vec`)
      `pred_fake` $\leftarrow \boldsymbol{D}$(`fake_samples`)
      `loss_fake` $\leftarrow \boldsymbol{BCE}$(`target_zeros, pred_fake`)
      `loss_d` $\leftarrow \frac{\texttt{loss\_real}+\texttt{loss\_fake}}{2}$
      Backpropagate Gradients for Discriminator
      Update Weights for Discriminator ▷ Generator Training Step
      `latent_vec` $\leftarrow$ `Sample Noise Distribution`
      `generated_samples` $\leftarrow \boldsymbol{G}$(`latent_vec`)
      `classifications` $\leftarrow \boldsymbol{D}$(`generated_samples`)
      `loss_g` $\leftarrow \boldsymbol{BCE}$(`target_ones, classifications`)
      Backpropagate Gradients for Generator
      Update Weights for Generators
    end for
  end for

3.5 Generating the Normal Distribution

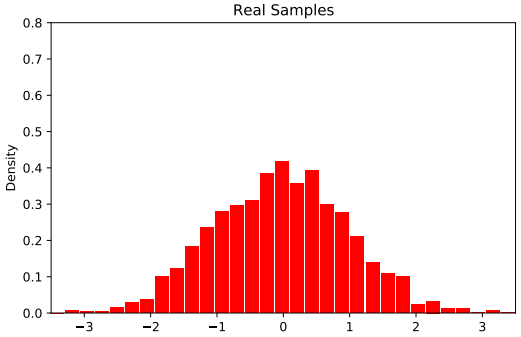
To illustrate the generative adversarial network, we reproduce results of the simplest possible implementation given in the original GAN paper [7] by approximating the normal distribution. The map that we try to learn is

$$f : X \sim U(0,1) \to Y \sim \mathcal{N}(0,1).$$

The generator will sample noise from the standard uniform distribution and learn to produce samples that approximate the standard normal distribution. In other words, the generator will be learning the inverse cumulative distribution function of the normal distribution, also called the quantile function.



(a) Noise Distribution $p(z)$



(b) Real Data Distribution

The source of unstructured noise for the GAN is the uniform distribution $U(0,1)$ shown in figure 3.1a. An example of the normal distribution is shown in figure 3.1b. Our implementation for the task uses the Python programming language and PyTorch machine learning framework. The code given in appendix A.1 is split into three parts:

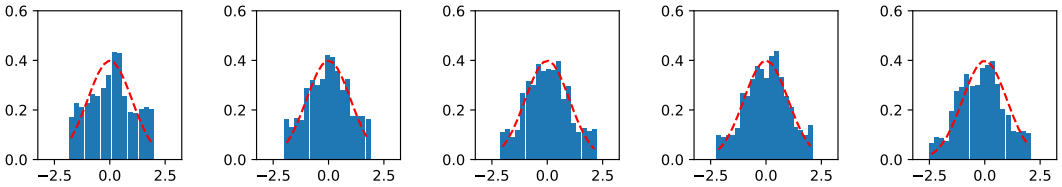
i. Generator Model (A.1.1)

ii. Discriminator Model (A.1.2)

iii. GAN Framework (A.1.3)

For the remainder of the section, we will be using a generator and discriminator network with varying number of hidden layers. For both submodels, the input layer and hidden linear layers are followed by a LeakyReLU non-linearity (section 2.3.2). The final activation used in the discriminator is the sigmoid function (equation 2.3) as it outputs a confidence score in the inclusive unit interval. In contrast, the output layer for the generator is a fully connected layer (section 2.3.1) with an output width of 1. It should be noted that a sigmoid output activation can not be used for the generator as it would restrict the range of the generated sample. Instead of using a precomputed training set, a sampling function is used to create the data distribution in real-time. The Adam optimizer (section 2.5.2) is used with a specified learning rate and beta coefficients 0.5 and 0.999. Unless specified otherwise, training is carried out for 100 epochs over 100 batches of size 32 each.

For the following experiment, the generator and discriminator network have two hidden layers of width 64 and 32, respectively. The optimizer for the generator is initialized with a learning rate of 0.0002, while the discriminator optimizer is initialized with a learning rate of 0.001. To illustrate the progress of the training process, we generate samples and plot their histograms at intervals of 20 epochs as shown in Figure 3.2.



(a) Epoch 20 (b) Epoch 40 (c) Epoch 60 (d) Epoch 80 (e) Epoch 100

Figure 3.2: Generated probability density distribution during the training progress. The red line show the real data distribution.

Compared to the original data distribution (figure 3.1b), the range of values in the generated distribution does not encompass the full range of the true data distribution. We suspect that the generator is unable to traverse the sample space of the true data distribution as it can only sample values in 1 dimension. This results in samples being generated very close to each other. As a consequence, the discriminator does not receive a broad range of generated samples and therefore, does not provide meaningful weight changes to the generator during the back-propagation step. This results in generated samples that do not span the range of true distribution. It is generally considered sufficient to have a latent dimension of at least the same dimension as that of the real data [5].

3.5.1 Varying the Noise Dimension

We train different GAN models by sampling z from a k-dimensional uniform distribution. For example, we sample from the closed unit square when $k = 2$. We use the same parameters from our previous experiment except the generator has an input layer width of k now.

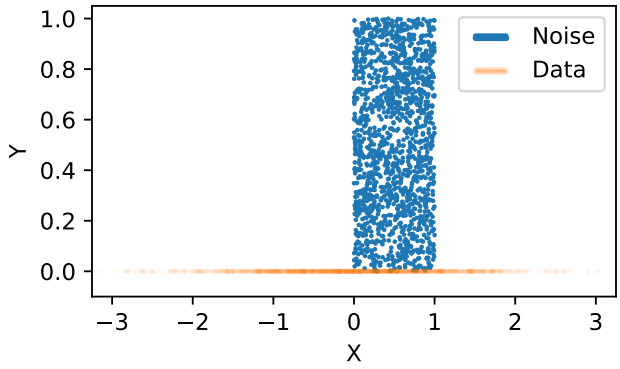
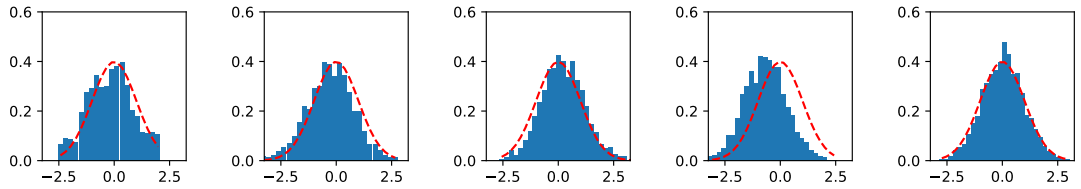


Figure 3.3: 1500 Samples from the 2-dimensional standard uniform distribution (blue) and 1-dimensional standard normal distribution (orange)

The following plot illustrates the generated distribution after 100 training iterations:



(a) $k = 1$ (b) $k = 2$ (c) $k = 4$ (d) $k = 10$ (e) $k = 64$

Figure 3.4: Generated probability density distribution during the training progress. The red line show the real data distribution.

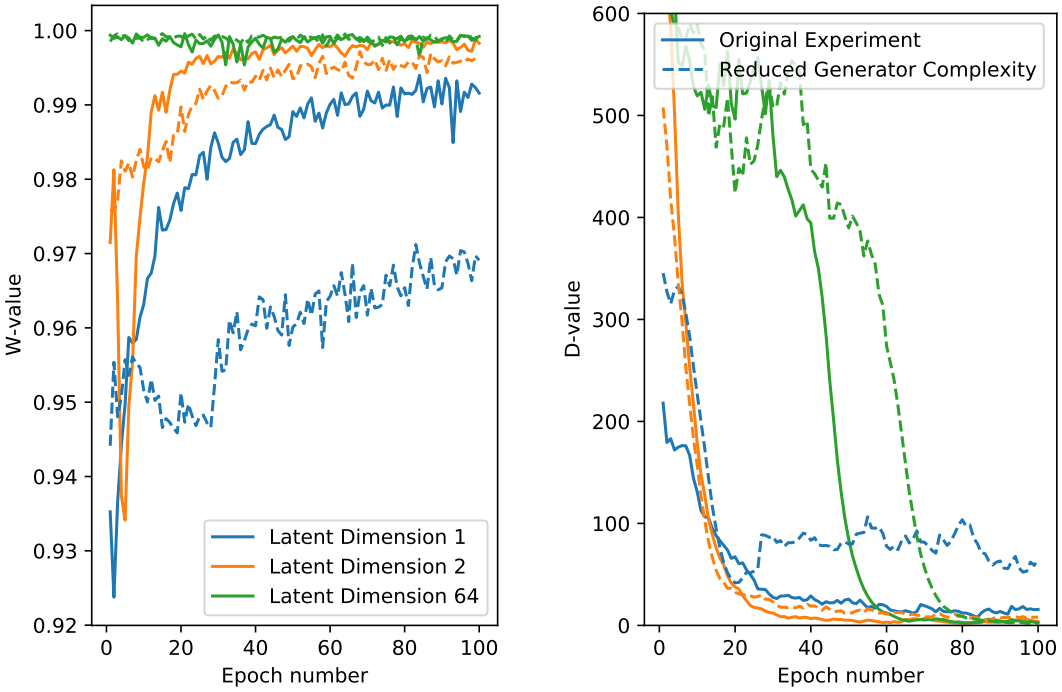
Figure 3.4 shows an improvement in generated distribution as the latent dimension is increased. It should be noted that the normalized sum of a sufficiently large number of samples from the uniform distribution tends to be normally distributed as given by the Central Limit Theorem [19]. We also suspect that the improvement in the generated samples might be due to an increase in total number of parameters in the generator's input layer as we increase the noise dimension. The total number of optimizable parameters for the model with $k = 64$ is 6273 compared to 2241 parameters for the network with $k = 1$.

3.5.2 Reducing Generator Complexity

The aim is now to investigate whether the total number of optimizable parameters in a model has a significant effect on the generated distribution. Therefore, we reduce the complexity of the generator by removing the first hidden layer of width 64. Our new generator network consists of: an input layer of width k, a hidden layer of width 32, and an output layer of width 1.

To provide a better comparison between the performance of the different generator models, we use statistical hypothesis testing for the null hypothesis that the generated samples are drawn from a normal distribution. The first normality

test that we use is an omnibus test, known as **D'Agostino's K-squared test**, that combines kurtosis k (appendix B.1) and skew s (appendix B.2). *Skewness* is the measure of asymmetry of a given probability distribution while *kurtosis* is the measure of combined probability mass in the tail relative to the center of the distribution. A standard normal distribution has a skew of 0 and kurtosis of 3. A narrow bell shape with a sharp peak has a higher kurtosis (> 3) than a wide bell shape with a broad peak. We define the d-value as $s^2 + k^2$, where s and k are the z-scores from the skew-test and kurtosis-test. A small d-value means there is less deviation from the standard normal distribution. However, it is possible for a distribution to be non-normally distributed but have the same kurtosis and skewness as the standard normal distribution. Therefore, we also calculate a w-value from the **Shapiro-Wilk normality test** [24]. A limitation of the Shapiro-Wilk test is that for a large sample size, it is more likely to result in a high w-value. A small w-value would reject our null hypothesis. Due the the limitations of both test, we will draw conclusions for our experiments by using information from both tests in conjunction.



(a) W-values from Shapiro-Wilk Test as training progresses

(b) Exponential Moving Average of D-values from D'Agostino's K-squared Test

Figure 3.5: Statistical Tests for the Original Experiment and Reduced Generator Experiment as training progressed.

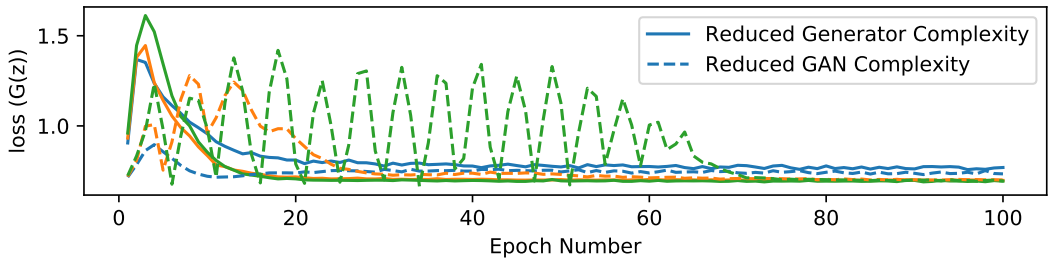
Figure 3.5b shows that all networks, except the low complexity generator with $k = 1$ latent dimension, generate samples that have the required kurtosis and skewness. Moreover, all networks improved their D-values as training progressed, although the D-value for the models with $k = 64$ dimensions oscillates for the first 40 epochs, which suggests an instability during the initial training phase

for high dimensional noise distributions. The w-values from the Shapiro-Wilk test in figure 3.5a suggest that the samples produced by the networks in our original experiment are more similar to the standard normal distribution than those produced by the lower complexity networks. However, for $k = 64$ latent dimensions, the w-values are very high even in the initial training phase. This is because the produced samples are heavily centred around the value 0 for the first 50 epochs and results in a high w-value as can be deduced from equation B.3. The network, quickly learns to produce higher quality samples as suggested by the sharp decrease in d-values for both experiments where the reduced complexity model follows a similar pattern but is delayed by 20 epochs.

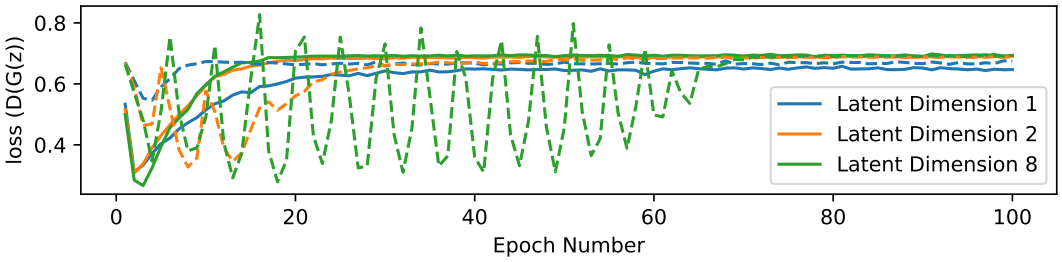
For $k = 64$ our reduced generator network has 2113 learnable parameters while our original experiment generator network for $k = 1$ has 2241 parameters. Comparing these two similar sized networks, we can conclude that our very high dimensional network with reduced complexity outperforms the 1 dimensional network with higher complexity on both tests. However, we can also see that a very high latent dimension is not necessary to produce accurate samples as the network with $k = 2$ already significantly improves sample generation in both experiments.

3.5.3 Reducing Generator & Discriminator Complexity

Given the results from our previous experiments, we also reduce the discriminator complexity and report our results.



(a) Generator Loss



(b) Discriminator Loss for Generated Samples

Figure 3.6: Generator and discriminator loss plots comparing the reduced generator experiment and reduced GAN experiment for $k = 1, 2, 8$ noise dimensions

In most other machine learning tasks, the loss is regarded as a measure of model performance, however, this does not hold true for networks in the GAN

framework. Nevertheless, the loss of the sub-network when viewed in conjunction can reveal if one of the four scenarios occurs:

i. **Low Discriminator Loss & Low Generator Loss**
The generator and discriminator are close to perfect. The discriminator is able to reject most samples generated by the generator network. The networks do not improve further.

ii. **Low Discriminator Loss & High Generator Loss**
This scenario occurs when the discriminator is significantly better than the generator. The generator produces samples that are rejected by the discriminator with a high confidence. The propagated loss does not provide any useful information to the generator. This results in a GAN failure known as Vanishing Gradients where the generator is unable to recover from a high loss value.

iii. **High Discriminator Loss & Low Generator Loss**
This scenario occurs when the generator is significantly better than the discriminator. The generator learns to produces samples that the discriminator is unable to reject in the majority of cases. The generator would not learn much because the error would be absorbed by the discriminator. It is possible that the generated samples are of very low quality and very similar to each other as the the GAN framework encounters another failure known as Mode Collapse (section 4.1). The discriminator loss on generated samples will be very high in this case, while the discriminator loss on real samples might still be within a reasonable range.

iv. **High Discriminator Loss & High Generator Loss**
The discriminator and generator are balanced in this case. In an ideal scenario, the discriminator would reach an equilibrium where the average confidence score is 0.5. The loss is loss is shared between both sub-networks as both networks improve with successive training iterations.

For $k = 8$ noise dimensions, the new reduced complexity network shows instability during training as can be deduced from the oscillating loss values in figure 3.6. We suspect that this instability is due to an imbalance between the subnetworks.

3.5.4 Varying Learning Rate

It is also possible that our reduced complexity GAN does not perform as well due to the difference in learning rate between training the generator and discriminator. The learning rate for optimizing the generator loss function was 0.0002, while for the discriminator a learning rate of 0.001 was used. Therefore, we decrease the learning rate for the discriminator in an attempt to balance the two networks.

We use $k = 2$ noise dimensions in the reduced GAN complexity network for the following experiments.

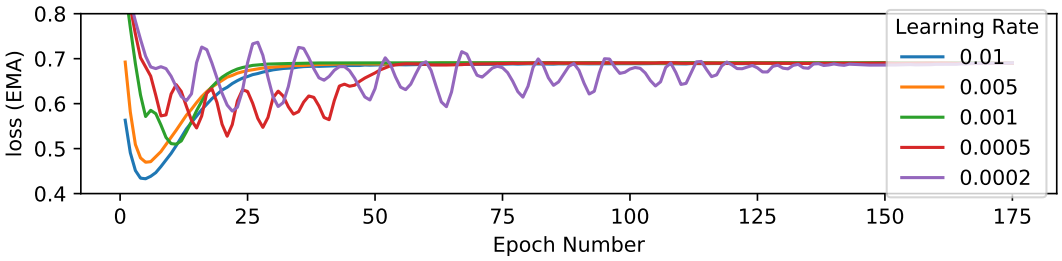
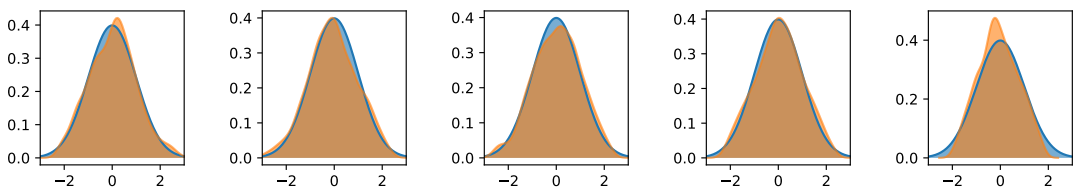


Figure 3.7: Exponential moving average discriminator loss values for $k = 2$ noise dimensions comparing networks trained with different learning rates.

By **decreasing the learning rate**, we would expect improved training stability but longer convergence time. However, from figure 3.7 it can be observed that decreasing the discriminator learning rate from 0.001 to 0.005 and then subsequently to 0.0002 results in increase training instability. This suggest that other factors, in particular, the imbalance between the subnetworks are likely to be at play.

By **increasing the learning rate**, we would expect a decrease in training stability due to the possible nonconvergence of the discriminator loss. For a learning rate of 0.005 and 0.01 we can observe a sharp decrease in loss values for the first 15 epochs followed by a convergence to a loss value of 0.690. Although this convergence is achieved in fewer epochs with a learning rate of 0.005 compared to 0.01, the difference is negligible. It can be concluded that an increased learning rate (up-to 0.01) will result in stable training of the discriminator network, however we must also check the quality of the samples generated. To illustrate the quality of generated samples after 200 training iterations, we interpolate the height of the histograms using kernel density estimation as shown in figure 3.8.



(a) 0.01 (b) 0.005 (c) 0.001 (d) 0.0005 (e) 0.0002

Figure 3.8: Generated probability density distribution (orange) overlaid on the standard normal distribution (blue) for varying discriminator learning rates (displayed under their respective graphs) after 200 epochs.

The generated probability distributions of the samples shown in figure 3.8 are very close to the standard normal distribution despite training instabilities for the small learning rates. For larger learning rates, it is possible that the generator or discriminator might have overfit after 200 epochs as the losses already converge at epoch 50.

3.6 Limitations of Evaluation

The shape of our histograms shown in figure 3.2 & 3.4 are sensitive to the number of bins and the bin width of the histogram. This makes it difficult to compare the network performance by visual inspection. Similarly, the curve produced by kernel density estimation (figure 3.8) depends on the kernel and bandwidth used to calculate the approximation. Furthermore, in section 3.5.2, we introduced the D'Agostino's K-squared test and the Shapiro-Wilk normality test. However, we have seen that some of our non-normally distributed samples were able to pass the D'Agostino's K-squared test as it is only relies on the kurtosis and skewness of the distribution. Likewise, the Shapiro-Wilk test has also shown to be unreliable as it reported a higher w-value for large sample sizes. Overall, despite the use of different evaluation metrics, we found it difficult to compare the accuracy of our networks and had to rely on manual verification as the most reliable non-quantitative evaluation method.

Chapter 4

Visualizing Failure Modes of GANs

4.1 GAN Failures

GANs are difficult to train due to the complex interplay between the subnetworks. We explore the three most common GAN failures in the following sections.

Vanishing Gradients

For a perfect discriminator, $D(x) = 1\ \forall x$ and $D(G(z)) = 0\ \forall z$ [30] and therefore $L_D = 0$ according to equation 3.3. This results in a *vanishing gradient problem* as there is no update to the weights after each training iteration. In practice, this can happen when the discriminator overpowers the generator as seen in scenario **ii** of section 3.5.3. Vanishing gradients were also seen in the min-max objective function (section 3.2) and alleviated by using separate loss functions for the generator and discriminator [7].

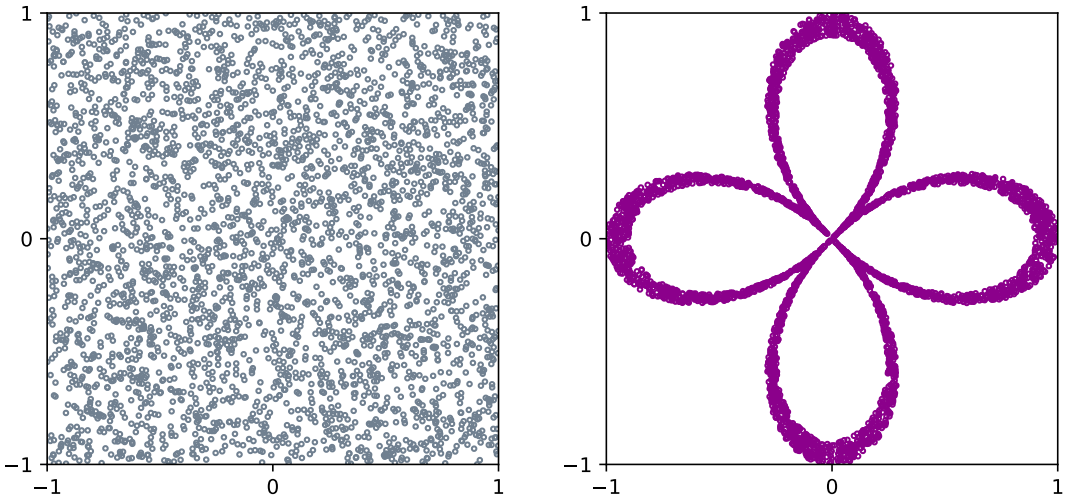
Mode Collapse

Mode collapse is a common GAN failure that occurs when the generator produces samples that do not span the entire range of the target function [3]. This usually happens when the generator produces the same sample when given different inputs from a noise distribution. In essence, the generator is always producing samples with the goal to maximize $D(G(z))$, but the discriminator also learns that the samples are fake and updates its weights. However, if the discriminator gets stuck in a local minimum, the same set of samples is able to fool the discriminator every time, resulting in the generation of repetitive samples. To identify mode collapse, one can visually inspect a range of generated samples to identify a lack of diversity amongst the samples. A more common type of GAN failure is *partial mode collapse* [5] where the generator learns to generate only a subset of the target samples.

Non-Convergence

Although the min-max equation for GANs is guaranteed to converge to a minimum point [7], this is not the case for separate loss functions for the generator and discriminator. Training GANs can be very unstable with losses oscillating and unable to converge to a minimum point [22].

4.2 Generating Rose Curves



(a) Noise Distribution (b) Target Data Distribution

To visualize mode collapse, we generate a 2-dimensional dataset using polar curves. Our target function in this case is a region based on a rose curve (also known as rhodonea curve) which is a polar equation with a phase shift of zero. The radial coordinates of the rose for a polar angle θ and an amplitude of a is generally given by:

$$r = a\cos(n\theta)$$

For $n \in \mathbb{Z}$, the rose has n petals if n is odd and $2n$ petals if n is even. We use $n = 2$ to produce a rose with four petals also know as a **quadrifolium**. In order to generate a region, we use the polar equation:

$$r = 0.1(\frac{\delta + 1}{2}) + 0.9|\cos(2\theta)|$$

where $\theta = \alpha\pi$ and $\alpha, \delta \in [-1, 1]$. We sample δ and α from from the horizontal and vertical components of the noise distribution, respectively. Furthermore, we modify the general equation by:

i. using the absolute value of the cosine function with a weightage of 0.9

ii. adding $\frac{\delta+1}{2}$ with a weightage of 0.1

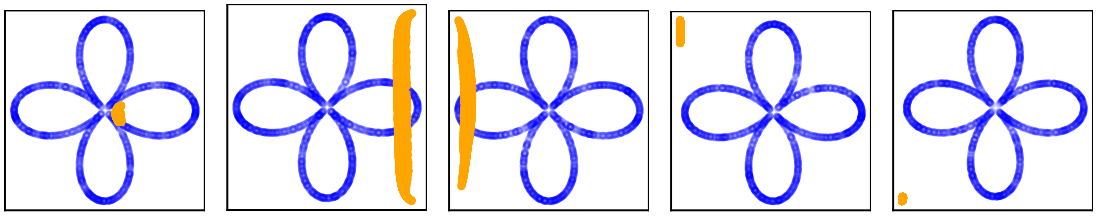
This results in positive values for r in the range [0,1] that produce a rose region of 0.1 maximal width. As the polar angle $\theta = \alpha\pi$ ranges from $-\pi$ to π, the target

distribution shown in figure 4.1b is produced. The code implementation for the sampling function is given in appendix A.2.1.

Based on our experiments in section 3.5, we sample the noise distribution from a bi-variate uniform distribution over the square $(-1, 1) \times (-1, 1)$ as shown in figure 4.1a.

Experimental Setup

We train the generator as part of the GAN framework to generate samples from the target distribution when given samples from the noise distribution. Similar to our experimental setup from section 3.5, we use the same GAN framework implementation from appendix A.1.3, but with higher layer widths for the generator and discriminator. The implementation of the generator and discriminator in the PyTorch framework is given in appendix A.2.2 and appendix A.2.3, respectively. For both submodels, the input layer and hidden linear layers are followed by a LeakyReLU non-linearity (section 2.3.2). The final activation for the discriminator remains to be the sigmoid function (equation 2.3) as it acts as a binary classifier. However, we now use an output layer of width 2 for the generator network as we require a 2 dimensional output. Due to the increased dimensionality of the output, we add two hidden layers of width 512 for the generator and discriminator. The **optimizer** used is ADAM (section 2.5.2) with default beta parameters. We initialize the optimizer for the generator loss to 0.0003 and the optimizer for the discriminator loss to a learning rate of 0.001. **Training** is carried out over 600 training iterations for 100 batches with a batch size of 128. At the end of every training iteration, we generate 4000 samples from our current generator and display a scatter plot at specified intervals as shown in figure 4.2.



(a) Epoch 10 (b) Epoch 20 (c) Epoch 40 (d) Epoch 60 (e) Epoch 80

Figure 4.2: Generated samples (orange) overlaid on the maximal rose curve (blue) as training progresses with the epoch number displayed under their respective graphs

Figure 4.2 illustrates **mode collapse** as the the generator fails to produce samples that span the range of the rose figure. During the first 10 epochs, the generator produces samples in close proximity of each other, but the discriminator learns to reject the samples. Therefore, the generator is forced to explore the sample space as shown in the scatter plot for epoches 20 and 40. Since the samples only cover a small part of the sample space, the discriminator learns that samples within this region are produced by the generator, while samples in the remaining sample space are from the true data distribution. This is particularly noticeable in epoch 60, as the generator is more likely to produce samples in the extreme regions

of sample space where the discriminator has never observed samples. Ultimately, the discriminator learns to reject these samples, but now assigns a high real sample confidence score to the remaining sample space where fake samples were being generated earlier. Epoch 80 shows that the generator produces samples in another extreme region of the sample space. If we allow training to continue for further epochs, the generator will cycle back to producing samples in a region that has now been unlearned by the discriminator - the generator's sample mode has collapsed to a small region of the sample space.

4.2.1 Generator Complexity

It is also possible that the generator might not be complex enough to learn the target function mapping. Therefore, we use supervised learning to check if the generator is complex enough to learn the mapping. The PyTorch implementation is given in appendix A.2.4. We train the generator on a data set consisting of samples from the noise distribution and their corresponding values from the target data distribution. Training is carried out over 40 epochs for 100 batches with a batch size of 32. One notable difference in the loss criteria is the use of the mean square error (section 2.4.1) instead of cross entropy.

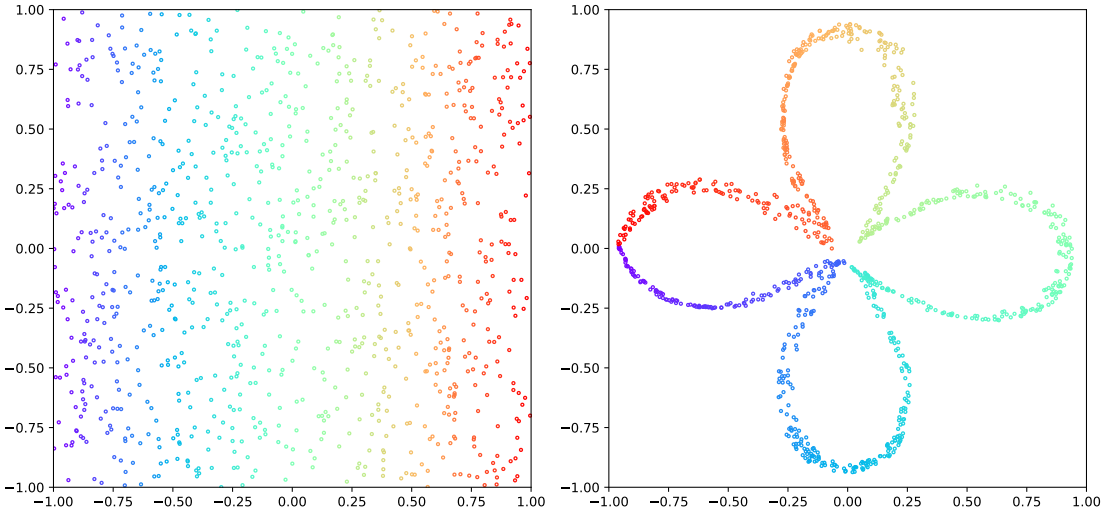


Figure 4.3: Generator mapping of the noise samples (left) to the rose figure (right)

Figure 4.3 shows that the generator is capable of learning the mapping from our noise distribution to the target distribution. The samples from the noise distribution are color coded so that we can visualize how the latent space transforms into the sample space.

4.2.2 Discriminator Complexity

Similarly, it can also be suspected that the discriminator is not complex enough to learn the target data distribution. Therefore, we use supervised learning to learn the decision boundary between real and fake samples. The dataset consists 50% of samples from the noise distribution and 50% of samples from the target

distribution. The samples from the noise distribution are given a label of 0 to signify that they are fake and the samples from the target distribution are given a label of 1 as they are from the real data distribution. Figure 4.4 shows the decision boundary learned by the discriminator model.

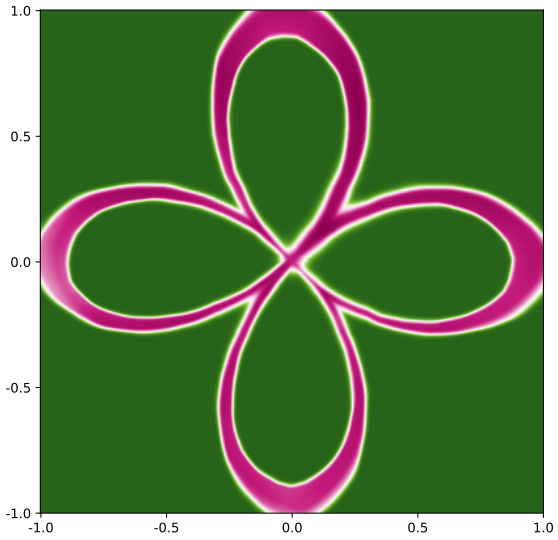


Figure 4.4: Discriminator's decision boundary between real (pink) and fake (green) samples using supervised learning. A darker colour corresponds to a higher confidence of a sample being real.

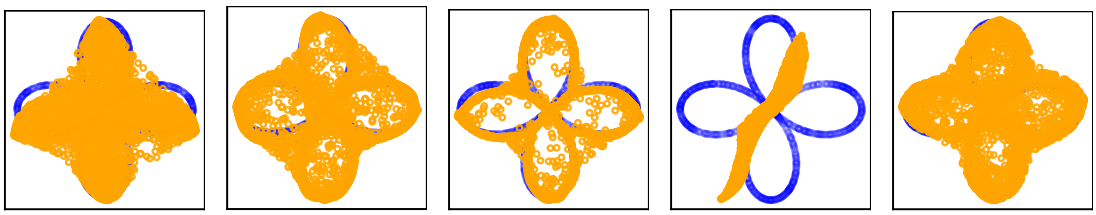
We conclude that the discriminator has sufficient complexity to learn the decision boundary between the target data distribution and noise samples.

4.2.3 Decreasing Generator Learning Rate

Our experiment from section 3.5.4 illustrated that a higher learning rate resulted in increased instability during training of the GAN. Based on our results, we decrease the learning rate of the generator loss optimizer from 0.0003 to 0.0002. Experiments carried out by Radford in 2016, suggest that using a β_1 parameters of 0.5 instead of 0.9 for the ADAM optimizer (2.5.2) also results in improved training stability [19]. We incorporate both changes and train our GAN model for 600 epochs with 100 batches of batch size 128.

It can be observed from figure 4.5 that the GAN results in **partial mode collapse**, although this only happens after more than 580 epochs. In fact, the generator network was converging to the rose figure at epoch 560, but then resulted in collapsing to a smaller region on the rose figure at epoch 580. Unlike our previous experiments, the networks recover from the partial mode collapse and continue to generate samples that span the entire rose figure as seen at epoch 600.

Before we carry out further experiments to avoid mode collapse, we would like to find an optimal set of learning rates for the remainder of the section. Thus we train multiple GAN models with varying learning rate to optimize the discriminator and generator loss function.



(a) Epoch 50 (b) Epoch 450 (c) Epoch 560 (d) Epoch 580 (e) Epoch 600

Figure 4.5: Samples generated (orange) by a generator with reduce learning rate overlaid on the maximal rose curve (blue) as training progresses with the epoch number displayed under their respective graphs

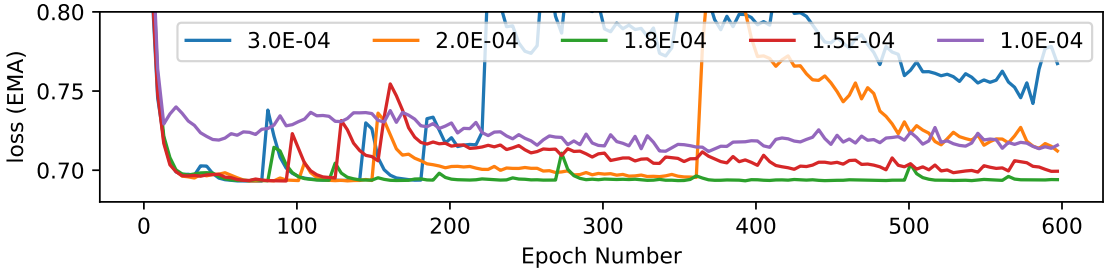


Figure 4.6: Exponential moving average of generator loss values comparing generator networks trained with varying learning rates.

Figure 4.6 shows that the generator network learning rates that resulted in mode collapse (3.0×10^{-4}) and partial mode collapse (2.0×10^{-4}) have heavily oscillating loss values as training progresses. The loss value for the partial mode collapse network is lower than the mode collapse networks which coincides with better sample generation in figure 4.5 compared to figure 4.2. The most stable training progress with the lowest loss value is achieved using a generator trained with a learning rate of 1.8×10^{-4}. Between epoches 100 and 200, an even lower learning rate of 1.5×10^{-4} shows higher fluctuations of loss values than the smallest learning rate of 1.0×10^{-4}. Therefore we conclude that an appropriate learning rate for the generator optimizer is 1.8×10^{-4} which we will use in all future experiments.

4.2.4 One sided label smoothing

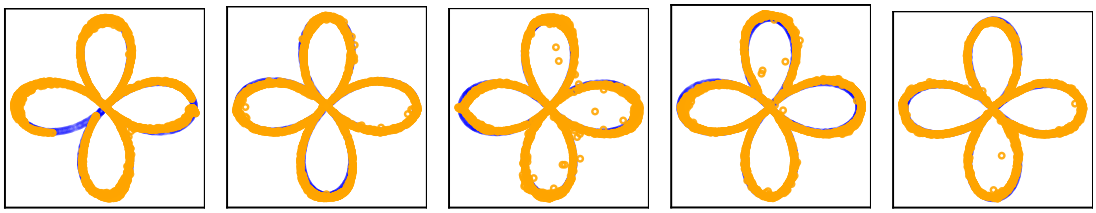
One sided label smoothing [22, 5] is a technique used to improve GAN training stability by reducing the confidence score of the discriminator on real samples. This is carried out by using a soft-label $\lambda < 1$ as the target label for calculating binary cross entropy loss on samples from the data distribution. We implement one-sided label smoothing by modifying our training algorithm (see algorithm 1 in section 3.4) by calculating the discriminator loss on real samples as follows:

$$\texttt{loss\_real} \leftarrow \boldsymbol{BCE}(\lambda \cdot \texttt{target\_ones}, \texttt{ pred\_real}).$$

This has a regularizing effect on the discriminator loss as the discriminator is punished for being too overconfident i.e predicting a confidence score greater

than λ. The label smoothing is strictly one-sided as regularizing the loss function on fake samples discourages the generator to produce better samples [5].

We report the impact of one-sided label smoothing by training GANs with $\lambda \in \{0.95, 0.9, 0.8, 0.7, 0.6\}$ using a learning rate of 1.8×10^{-4} for optimizing the generator loss in figure 4.7.



(a) $\lambda = 0.95$ (b) $\lambda = 0.90$ (c) $\lambda = 0.80$ (d) $\lambda = 0.70$ (e) $\lambda = 0.60$

Figure 4.7: Generated Samples (orange) overlaid on the rose figure (blue) at epoch 600 for soft-labels $\lambda \in \{0.95, 0.9, 0.8, 0.7, 0.6\}$.

In order to provide a quantitative evaluation of our generated samples, we use our discriminator trained using supervised learning (section 4.2.2) to output the probability of a sample belonging to the true distribution. The accuracy metric is calculated as the ratio of samples that have a confidence probability above a threshold of 0.4 and the total number of samples tested.

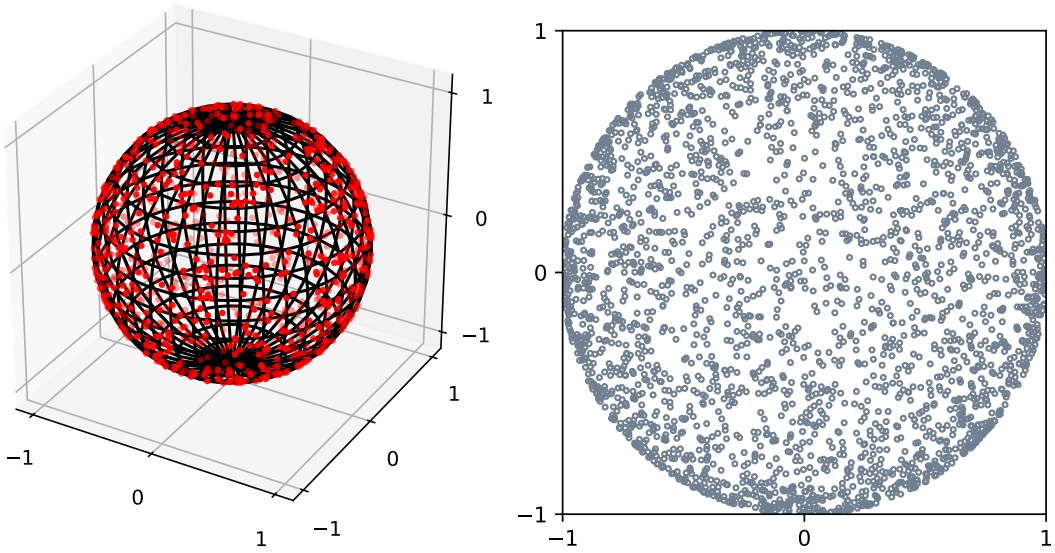
λ	loss $D(x)$	loss $D(G(z))$	loss $G(z)$	Accuracy
1.00	0.6920	0.6944	0.6928	92.95
0.95	0.7292	0.6197	0.6197	96.58
0.90	0.7784	0.5979	0.5979	90.45
0.80	0.8266	0.4976	0.4976	90.42
0.70	0.8639	0.4310	0.4310	93.05
0.60	0.8654	0.3565	0.3565	93.42
target distribution	-	-	-	98.45

Table 4.1: One-sided label smoothing experiment results for discriminator loss on real samples (loss $D(x)$), discriminator loss on fake samples (loss $D(G(z))$), generator loss (loss $G(z)$) and accuracy (%) for generated samples at epoch 600. The accuracy on the target distribution is also provided for comparison.

The result of our experiment is summarized in table 4.1. The discriminator loss on real samples increases as we decrease λ due to the increased regularization of the discriminator. In contrast, the discriminator loss on real samples and generator loss decreases as we increase the regularization of the discriminator. It is surprising to see that the heavily regularized discriminator ($\lambda = 0.60$) is more accurate than the non-regularized discriminator ($\lambda = 1.00$) by 0.47%. It should be noted that the least-regularized discriminator ($\lambda = 0.0.95$) results in a very weak partial mode collapse as seen in figure 4.7a, but scores the highest accuracy by a significant margin of 3.16% compared to the next best accuracy scored by the heavily regularized discriminator.

4.2.5 Sampling from a hypersphere

Now, we sample our noise distribution from the surface of a 3 dimensional hypersphere (figure 4.8a), in order to encourage the generator to produce samples from the greater circle and avoid mode collapse. Hyperspheres have previously been used in CircleGAN [25] and SphereGAN [18] to improve the diversity of generated samples, although the use of such spheres has been as an embedding space to measure the divergence between real and fake samples. The concepts are primarily based on the principle that latent spaces are high dimensional and consist of subregions that are not mapped to a sub-region in the sample space. These dead-zones in the latent space have been successfully avoided during interpolation by using hyperspheres as the noise distribution to produce samples of a wide range. [29].



(a) Sampling points from the surface of a 3d-sphere

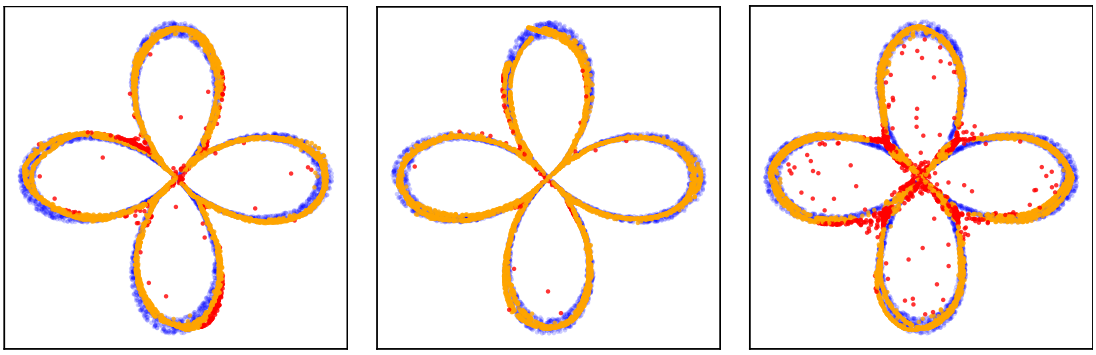
(b) Sampling points projected in 2 dimensional space

Figure 4.8: Hypersphere as noise distribution

Figure 4.8b shows that samples from the noise distribution have a higher density in the outermost regions and are more sparse towards the center as the larger circles in the 3d-sphere have a bigger surface area while circles at the pole occupy a smaller surface area.

We train three different GANs with the same learning parameters from section 4.2.4 but with $\lambda \in \{1, 0.95, 0.6\}$. The generation of 4000 samples is shown in figure 4.9 with points in red classed as fake by our external discriminator trained using supervised learning in section 4.2.2. Our non-regularized GAN scored an accuracy of 93.58% with a spherical noise distribution compared to an accuracy of 92.95% (table 4.1) with a uniform noise distribution. Figure 4.9a shows that samples from the non-regularized GAN produced a coherent rose-figure with some erroneous samples compared to the lightly-regularized GAN samples (figure 4.9b) that form a less coherent rose figure but results in significantly fewer erroneous samples with a significantly higher accuracy score of 98.15%. Finally, samples

produced by the heavily regularized GAN (figure 4.9c) formed a coherent rose figure, but resulted in a significant number of erroneous samples, especially in the center of the rose. In contrast, our heavily regularized GAN sampled from the uniform distribution produced samples with an accuracy of 93.42% (table 4.1) compared to a much lower accuracy of 78.92% for the GAN trained with a spherical noise distribution.



(a) non-regularized $\lambda = 1.00$ (b) lightly-regularized $\lambda = 0.95$ (c) heavily-regularized $\lambda = 0.60$

Figure 4.9: Generated Samples (orange) overlaid on the rose figure (blue) at epoch 600 for labels $\lambda \in \{1, 0.95, 0.6\}$.

The positive impact of using a spherical noise distribution remains inconclusive as the first two networks showed a significant improvement in accuracy, but the third network reported degraded performance. It is possible that the networks were not trained for sufficient number of epochs and have not converged close enough to an optimal solution.

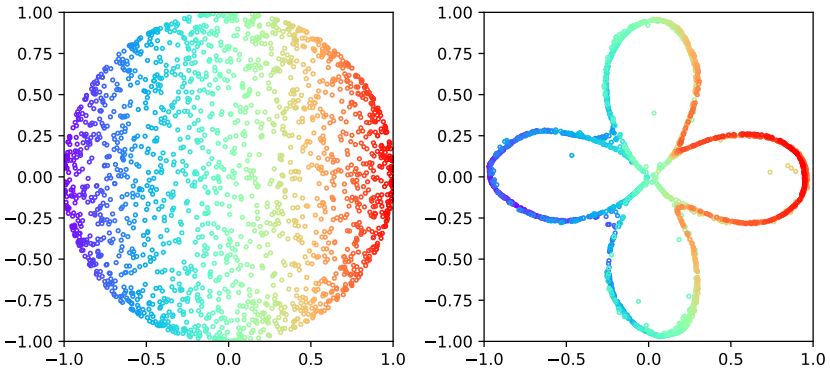
4.2.6 Training to ensure convergence

We train two of our GAN networks from the previous two experiments (section 4.2.4 & 4.2.5) for 1200 epochs with 100 batches of size 128 each in order improve convergence of the networks. The two GAN networks we chose are:

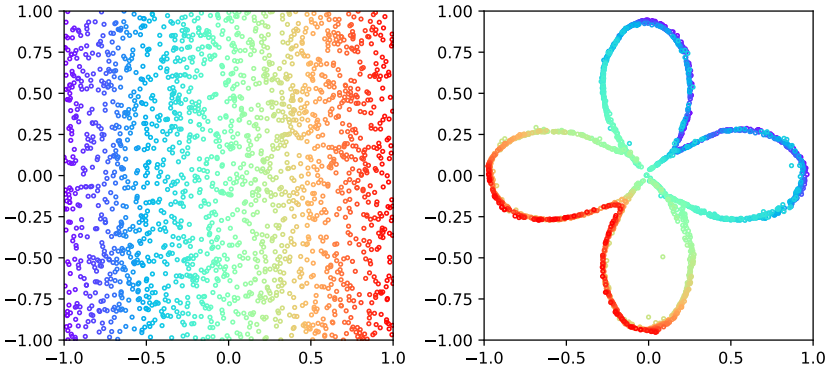
A. Non-regularized Discriminator with spherical noise distribution (4.9a)

B. Heavily-regularized Discriminator with uniform noise distribution (4.7e)

Figure 4.10 visually describes a mapping of the latent space to the sample space learned by both generator networks. Comparing figure 4.3 and figure 4.10, we observe that the mapping learned by the generator trained using supervised learning follows an ordered radial pattern in the samples space as we move horizontally through the latent space. In contrast, the mappings learned by the two GAN networks follow a linear pattern in the sample space as well. The spherical noise distribution (figure 4.10a) in network **A** is mapped horizontally in the sample space, while the uniform distribution (figure 4.10b) is mapped diagonally by network **B**. By visual inspection, the rose figure produced by GAN network **B** is a closer approximation to the target rose figure than the one produced by



(a) Linear interpolation of GAN network **A**



(b) Linear interpolation of GAN network **B**

Figure 4.10: Generated samples (right) from the noise distribution (left) for GAN networks **A** & **B**. Interpolation is signified by the horizontal colour gradient flowing through the points.

GAN network **A**. Therefore, we conclude that GAN network **B** has the best performance in our experimental setup.

4.3 Evaluation Limitations

The accuracy score introduced in section 4.2.4 is calculated using an external discriminator with a threshold of 0.4 and has various limitations for evaluating GANs. A high accuracy score has shown to correlate with better sample generation, however the score does not incorporate a measure of diversity of the samples. For example, for the GAN network with $\lambda = 0.95$ the accuracy is 96.58% (table 4.1), however on visual inspection of figure 4.7a, one can see that there is a gap of sample points on the left petal of the rose figure. Moreover, a GAN network with mode collapse would produce samples with an accuracy of 100% even though all samples would be centred at the same location. The choice of a threshold value of 0.4 was made based on visual inspection of generated samples. A better approach would be to view a Receiver Operating Characteristic (ROC) curve of the discriminator and choose the value that maximizes true positives and minimizes false negatives. Furthermore, in most practical use cases, a target function is not available to train an external discriminator, which limits our evaluation method

to theoretical research use cases only.

Chapter 5

Conclusion

In this project, we explored Generative Adversarial Networks in a low-dimensional setting and visualized the generation process to gain a better understanding of the complex interplay between the subnetworks. We have seen that balancing the power of the generator and discriminator has been largely attributed to an exploration of suitable training hyperparameters. In particular, we found that a higher learning rate for the discriminator relative to the generator resulted in stable training for both low-dimensional examples. This coincides with the findings made by Radford for the implementation of the stable DCGAN [19]. This shows that our results in low-dimensional settings could be extrapolated to higher dimensions as well.

We were able to successfully avoid mode collapse by regularizing the discriminator using one-sided label smoothing and engineering the latent space using a hypersphere. It has previously been reported that regularizing the discriminator by periodically adding random noise to the discriminator layers has shown improved training stability [21]. Moreover, encoding the latent space has also shown significant improvements in the diversity of sample generation [27].

Throughout this project, we have attempted to use different evaluation metrics to compare our models and have reported their limitations. In recent years, the Inception Score (IS) [23] and Frechet Inception Distance (FID) [9] have been proposed as a possible quantitative evaluation metric. However, the Inception Score is unable to quantify diversity amongst samples, while the FID has limited ability to quantify smaller details [26]. Although we explored different evaluation metrics, the most reliable method was visual inspection. Manually inspecting generated samples is time consuming, therefore, we believe that finding a reliable evaluation metric will speed up GAN research.

Appendix A

PyTorch GAN Implementations

The code implementations for the networks are implemented in the PyTorch framework which is available at url: https://github.com/mnm-matin/LowDim-GAN.

A.1 Normal Distribution GAN

The following is the PyTorch framework implementation for the GAN used to approximate the normal distribution in chapter 3.

A.1.1 Generator Model

```
class Normal_Generator(nn.Module):
    def __init__(self, latent_dim):
        """A generator for the normal distribution with
           variable latent dimensions.
        Args:
            latent_dim (int): latent dimension
        """
        super(Normal_Generator, self).__init__()
        self.latent_dim = latent_dim

        self.map1 = nn.Linear(latent_dim, 64)
        self.map2 = nn.Linear(64, 32)
        self.map3 = nn.Linear(32, 1)
        self.a = nn.LeakyReLU()

    def forward(self, x):
        """Forward pass to map noize z to target y"""
        x = self.map1(x)
        x = self.a(x)
        x = self.map2(x)
        x = self.a(x)
        x = self.map3(x)
        return x
```

A.1.2 Discriminator Model

```
class Normal_Discriminator(nn.Module):
    def __init__(self, input_dim):
```

```
        """A discriminator for discerning real from generated
            samples.
        Args:
            input_dim (int): width of the input (output of
                generator)
        """
        super(Normal_Discriminator, self).__init__()
        self.input_dim = input_dim

        self.map1 = nn.Linear(input_dim, 64)
        self.map2 = nn.Linear(64, 32)
        self.map3 = nn.Linear(32, 1)
        self.a = nn.LeakyReLU()
        self.f = nn.Sigmoid()


    def forward(self, input_tensor):
        """Forward pass; output confidence probability of
            sample being real"""
        x = self.map1(x)
        x = self.a(x)
        x = self.map2(x)
        x = self.a(x)
        x = self.map3(x)
        x = self.f(x)
        return x
```

A.1.3 GAN Framework

```
def Normal_GAN(
    generator=Normal_Generator(1),
    discriminator=Normal_Discriminator(1),
    noise_fn=lambda x: torch.rand((x, 1), device='cpu'),
    data_fn=lambda x: torch.randn((x, 1), device='cpu'),
    plot_every = 0,
    lr_d=1e-3,
    lr_g=2e-4,
    epochs=100,
    batches=100,
    batch_size=32,
    device='cpu'):

    generator = generator.to(device)
    discriminator = discriminator.to(device)
    criterion = nn.BCELoss()
    optim_d = optim.Adam(discriminator.parameters(),
                                lr=lr_d, betas=(0.5, 0.999))
    optim_g = optim.Adam(generator.parameters(),
                                lr=lr_g, betas=(0.5, 0.999))
    target_ones = torch.ones((batch_size, 1)).to(device)
    target_zeros = torch.zeros((batch_size, 1)).to(device)
    loss_g, loss_d_real, loss_d_fake = [], [], []
    gen_samples = []
    fake_mean, fake_std = [], []
    start = time()
    for epoch in range(epochs):
```

```
        loss_g_running, loss_d_real_running,
            loss_d_fake_running = 0, 0, 0
        fake_mean_running, fake_std_running = 0, 0
        for batch in range(batches):

            # Train the generator one step
            generator.zero_grad()
            latent_vec = noise_fn(batch_size)
            generated = generator(latent_vec)
            classifications = discriminator(generated)
            loss = criterion(classifications, target_ones)
            loss.backward()
            optim_g.step()
            lg_ = loss.item()

            # Train the discriminator one step
            discriminator.zero_grad()

            # real samples
            real_samples = data_fn(batch_size)
            pred_real = discriminator(real_samples)
            loss_real = criterion(pred_real, target_ones)

            # generated samples
            latent_vec = noise_fn(batch_size)
            fake_samples = generator(latent_vec).detach()
            pred_fake = discriminator(fake_samples)
            loss_fake = criterion(pred_fake, target_zeros)

            # combine
            loss = (loss_real + loss_fake) / 2
            loss.backward()
            optim_d.step()

            ldr_ = loss_real.item()
            ldf_ = loss_fake.item()

            loss_g_running += lg_
            loss_d_real_running += ldr_
            loss_d_fake_running += ldf_

            fake_mean_running += fake_samples.mean()
            fake_std_running += fake_samples.std()

        loss_g.append(loss_g_running / batches)
        loss_d_real.append(loss_d_real_running / batches)
        loss_d_fake.append(loss_d_fake_running / batches)
        fake_mean.append(fake_mean_running / batches)
        fake_std.append(fake_std_running / batches)

        print(f"Epoch {epoch+1}/{epochs} ({int(time() - start)}
            s):"
                f" G={loss_g[-1]:.3f},"
                f" D(x)={loss_d_real[-1]:.3f},"
                f" D(G(z))={loss_d_fake[-1]:.3f}",
                f" Mean(G(z))={fake_mean[-1]:.3f}",
```

```
                f" Std(G(z))={fake_std[-1]:.3f}")

        def generate_sample(test_size=2000):
            test_latent_vec = noise_fn(test_size)
            test_gen = generator(test_latent_vec).detach().cpu
                ().numpy().flatten()
            return test_gen, test_latent_vec

        test_gen, test_latent_vec = generate_sample(2000)
        gen_samples.append(test_gen)

        if plot_every != 0 and (epoch+1) % plot_every == 0:
            plot_his(test_gen, plot_normal=True, epoch="at
                Epoch "+str(epoch+1))

    model_dict ={
        "gen_samples": gen_samples,
        "losses": (loss_g, loss_d_real, loss_d_fake),
        "stats": (fake_mean, fake_std)
    }

    return model_dict
```

A.2 Rose GAN

The following is the PyTorch framework implementation for the GAN used to generate the rose figure in chapter 4.

A.2.1 Rose Figure

```
def target_function(Z):
    '''
    Map Z ([-1,1], [-1,1]) to a rose figure
    '''
    X = Z[:, 0]
    Y = Z[:, 1]
    theta = X * np.pi
    r = 0.05*(Y+1) + 0.90*abs(np.cos(2*theta))
    polar = np.zeros((Z.shape[0], 2))
    polar[:, 0] = r * np.cos(theta)
    polar[:, 1] = r * np.sin(theta)
    return polar
```

A.2.2 Generator Model

```
class Generator(nn.Module):
    def __init__(self, layer_size=[2,512,512,512,2],
        layer_activation=nn.LeakyReLU(0.1), output_activation=
        nn.Tanh()):
        """A generator for mapping a latent space to a sample
            space.
        Args:
            layers_size (List[int]): A list of layer widths,
                e.g [input_size, hidden_size_1, ...,
                    hidden_size_3, output_size]
```

```
            layer_activation: torch activation function to
                follow all layers except the output layer
            output_activation: torch activation function or
                None
        Example:
            layer_size=[2,512,512,512,2] produces the following
                 4 layers
            (0): Linear(in_features=2, out_features=512, bias=
                True)
            (1): Linear(in_features=512, out_features=512, bias
                =True)
            (2): Linear(in_features=512, out_features=512, bias
                =True)
            (3): Linear(in_features=512, out_features=2, bias=
                True)
        """
        super(Generator, self).__init__()
        self.input_size = layer_size[0]
        self.layer_size = layer_size[1:]
        self.layers = nn.ModuleList()
        self.a = layer_activation #LeakyReLU
        self.o = output_activation #Tanh or SELU


        current_dim = self.input_size
        for layer_dim in self.layer_size:
            self.layers.append(nn.Linear(current_dim, layer_dim
                )) # Add all layers
            current_dim = layer_dim

        print(self.layers)

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = self.a(layer(x))
        x = self.layers[-1](x)
        return self.o(x) if self.o is not None else x
```

A.2.3 Discriminator Model

```
class Discriminator(nn.Module):
    def __init__(self, layer_size=[2,512,512,512,1],
        layer_activation=nn.LeakyReLU(0.1), output_activation=
        nn.Sigmoid()):
        """A discriminator for mapping a latent space to a
            sample space.
        Args:
            layers_size (List[int]): A list of layer widths,
                e.g [input_size, hidden_size_1, ...,
                    hidden_size_3, output_size]
            layer_activation: torch activation function to
                follow all layers except the output layer
            output_activation: torch activation function or
                None
        """
        super(Discriminator, self).__init__()
```

```
        self.input_size = layer_size[0]
        self.layer_size = layer_size[1:]
        self.layers = nn.ModuleList()
        self.a = layer_activation #LeakyReLU
        self.o = output_activation #Sigmoid


        current_dim = self.input_size
        for layer_dim in self.layer_size:
            self.layers.append(nn.Linear(current_dim, layer_dim
                )) # Add all layers
            current_dim = layer_dim

        print(self.layers)

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = self.a(layer(x))
        x = self.layers[-1](x)
        return self.o(x)
```

A.2.4 Generator Supervised Learning

```
generator=Generator(layer_size=[2,512,512,512,2],
    layer_activation=nn.LeakyReLU(0.1))
generator = generator.to('cpu')
criterion = nn.MSELoss()
noise_fn=sample_noise
data_fn=lambda Z: torch.from_numpy(target_function(Z).astype('
    float32'))
optim_g = optim.Adam(generator.parameters(),lr=2e-4)
batch_size=32
for epoch in range(40):
        for batch in range(100):
            generator.zero_grad()
            gen_latent_vec = noise_fn(batch_size)
            generated = generator(gen_latent_vec)
            target = data_fn(gen_latent_vec)
            loss = criterion(generated, target)
            loss.backward()
            optim_g.step()
            lg_ = loss.item()
            print(f"epoch={epoch}",
            f" G={lg_:.3f}")
```

A.2.5 Discriminator Supervised Learning

```
eval_dis=Discriminator(layer_size=[2,512,512,512,1],
    layer_activation=nn.LeakyReLU(0.1))
eval_dis = eval_dis.to('cpu')
criterion = nn.MSELoss()
noise_fn=generate_noise
data_fn=sample_from_target_function
optim_d = optim.Adam(eval_dis.parameters(),lr=1e-3)
batch_size=64
for epoch in range(400):
```

```
9           b_loss = []
10          for batch in range(100):
11              eval_dis.zero_grad()
12              gen_latent_vec = noise_fn(batch_size // 2) # Half
                    Noise Samples
13              target = sample_from_target_function(batch_size //
                    2) # Half Real Samples
14              samples = torch.from_numpy(np.concatenate((
                    gen_latent_vec, target), axis=0).astype('
                    float32')) # Combine Noise and Real
15              labels = torch.from_numpy(np.concatenate((np.zeros
                    ((batch_size//2, 1)), np.ones((batch_size//2,
                    1))), axis=0).astype('float32')) # Create
                    Labels
16              conf = eval_dis(samples)
17              loss = criterion(conf, labels)
18              loss.backward()
19              optim_d.step()
20              ld_ = loss.item()
21              b_loss.append(ld_)
22          print(f"epoch={epoch}",
23          f" D={np.mean(b_loss):.3f}")
24  # torch.save(eval_dis.state_dict(), './eval-dis.pt')
```

The weights of our model are available in the git repository and can be loaded as shown in the following code snippet.

```
1  eval_dis = Discriminator(layer_size=[2,512,512,512,1],
       layer_activation=nn.LeakyReLU(0.1))
2  eval_dis.load_state_dict(torch.load('./eval-dis.pt'))
```

A.2.6 Hypersphere Sampling

```
1  def sample_spherical(samples, hypersphere_dim=3):
2      sphere_sample = np.random.randn(hypersphere_dim, samples)
3      sphere_sample /= np.linalg.norm(sphere_sample, axis=0)
4      return sphere_sample
```

Appendix B

Evaluation Metrics

B.1 Kurtosis Test

The kurtosis of a distribution [28] is given by

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2\right)^2}-3 \tag{B.1}$$

where x_i is the random sample value and $\bar{x}$ is the sample mean for n samples.

B.2 Skewness Test

The skewness [28] of a distribution is given by

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2\right)^{3/2}} \tag{B.2}$$

where x_i is the random sample value and $\bar{x}$ is the sample mean for n samples.

B.3 Shapiro Wilk Test

The Shapiro Wilk metric [24] of a distribution is given by

$$W=\frac{(\sum_{i=1}^{n}\alpha_i x_i)^2}{\sum_{i=1}^{n}(x_i-\bar{x})^2} \tag{B.3}$$

where x_i is the random sample value and α_i is a constant calculated from the co-variance and mean of n samples from the standard normal distribution.

Bibliography

[1] D. S. Berman, C. Howser, T. Mehoke, and J. D. Evans, *Mutagan: A seq2seq gan framework to predict mutations of evolving protein populations*, arXiv preprint arXiv:2008.11790, (2020).

[2] A. Dash, J. Ye, and G. Wang, *A review of generative adversarial networks (gans) and its applications in a wide variety of disciplines - from medical to remote sensing*, ArXiv, abs/2110.01442 (2021).

[3] R. Durall, A. Chatzimichailidis, P. Labus, and J. Keuper, *Combating mode collapse in gan training: An empirical analysis using hessian eigenvalues*, in VISIGRAPP, 2021.

[4] F. Farnia and A. Ozdaglar, *Gans may have no nash equilibria*, 2020.

[5] I. Goodfellow, *Nips 2016 tutorial: Generative adversarial networks*, 2017.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016. http://www.deeplearningbook.org.

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

[8] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, *A review on generative adversarial networks: Algorithms, theory, and applications*, 2020.

[9] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, *Gans trained by a two time-scale update rule converge to a local nash equilibrium*, Advances in neural information processing systems, 30 (2017).

[10] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, *Image-to-image translation with conditional adversarial networks*, CVPR, (2017).

[11] J. M. Joyce, *Kullback-Leibler Divergence*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 720–722.

[12] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[13] D. Lin, K. Fu, Y. Wang, G. Xu, and X. Sun, *Marta gans: Unsupervised representation learning for remote sensing image classification*, IEEE Geoscience and Remote Sensing Letters, 14 (2017), p. 2092–2096.

[14] A. L. Maas, *Rectifier nonlinearities improve neural network acoustic models*, 2013.

[15] V. Nair and G. E. Hinton, *Rectified linear units improve restricted boltzmann machines*, ICML'10, Omnipress, 2010, p. 807–814.

[16] M. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.

[17] A. Odena, *Semi-supervised learning with generative adversarial networks*, arXiv preprint arXiv:1606.01583, (2016).

[18] S. W. Park and J. Kwon, *Spheregan: Sphere generative adversarial network based on geometric moment matching and its applications*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 44 (2022), pp. 1566–1580.

[19] A. Radford, L. Metz, and S. Chintala, *Unsupervised representation learning with deep convolutional generative adversarial networks*, 2016.

[20] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain*, Psychological Review Vol. 65, No. 6, (1958).

[21] K. Roth, A. Lucchi, S. Nowozin, and T. Hofmann, *Stabilizing training of generative adversarial networks through regularization*, Advances in neural information processing systems, 30 (2017).

[22] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, *Improved techniques for training gans*, 2016.

[23] T. Salimans, A. Karpathy, X. Chen, and D. P. Kingma, *Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications*, arXiv preprint arXiv:1701.05517, (2017).

[24] S. S. Shapiro and M. B. Wilk, *An analysis of variance test for normality (complete samples)*, Biometrika, 52 (1965), pp. 591–611.

[25] W. Shim and M. Cho, *Circlegan: Generative adversarial learning across spherical circles*, in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds., vol. 33, Curran Associates, Inc., 2020, pp. 21081–21091.

[26] K. Shmelkov, C. Schmid, and K. Alahari, *How good is my gan?*, 2018.

[27] A. Srivastava, L. Valkov, C. Russell, M. U. Gutmann, and C. Sutton, *Veegan: Reducing mode collapse in gans using implicit variational learning*, Advances in neural information processing systems, 30 (2017).

[28] A. Takemura, M. Matsui, and S. Kuriki, *Skewness and kurtosis as locally best invariant tests of normality*, 2006.

[29] T. White, *Sampling generative networks*, 2016.

[30] M. Wiatrak and S. V. Albrecht, *Stabilizing generative adversarial network training: A survey*, ArXiv, abs/1910.00927 (2019).

[31] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, *Unpaired image-to-image translation using cycle-consistent adversarial networks*, in Computer Vision (ICCV), 2017 IEEE International Conference on, 2017.