Text, logo

Description automatically generated with medium confidence

Using a Generative Adversarial Network to Generate Kymograph Images of the Repressilator Gene Network

Chris Harvey (190266619)

MComp Computer Science

Supervisor: Dr Timothy J. Rudge

May 2022

Word Count: 16248

Declaration

I declare that this dissertation represents my own work except where otherwise stated.

Acknowledgements

I would like to thank my supervisor Dr Timothy J. Rudge for his advice and support throughout this project. I would also like to thank Gonzalo Vidal who provided support and assistance on using the Python library LOICA.

Abstract

This project aims to implement a Generative Adversarial Network (GAN) to generate kymograph images of the oscillations produced by a repressilator gene network. The first Generative Adversarial Network was proposed in a research paper in 2014 (Goodfellow, et al. , 2014). Generative Adversarial Networks are still on the cutting edge of machine learning research due to their wide range of applications. This project will make use of this machine learning strategy and explore its effectiveness at generating images of graph data. This network will use kymograph images as training data and attempt to generate more images like these as an alternative method to running a biological simulation to produce more data.

**Table of Contents**

[1 Introduction 11](#_Toc104546848)

[1.1 Motivation and Rationale 11](#_Toc104546849)

[1.1.1 The Context: 11](#_Toc104546850)

[1.1.2 The Problem: 12](#_Toc104546851)

[1.1.3 The Rationale: 12](#_Toc104546852)

[1.2 Aim 13](#_Toc104546853)

[1.3 Objectives 13](#_Toc104546854)

[1.4 Dissertation Structure 14](#_Toc104546855)

[1.5 Project Approach 15](#_Toc104546856)

[2 Background Research 16](#_Toc104546857)

[2.1 Generative Adversarial Networks 16](#_Toc104546858)

[2.1.1 Deep Convolutional Generative Adversarial Networks 17](#_Toc104546859)

[2.1.2 StyleGAN 19](#_Toc104546860)

[2.2 Repressilator Gene Network 21](#_Toc104546861)

[2.2.1 LOICA 22](#_Toc104546862)

[3 Design & Implementation 24](#_Toc104546863)

[3.1 System & Software Architecture 24](#_Toc104546864)

[3.1.1 Python 24](#_Toc104546865)

[3.1.2 Google Colab 24](#_Toc104546866)

[3.1.3 Keras 25](#_Toc104546867)

[3.1.4 LOICA 26](#_Toc104546868)

[3.1.5 Simulated Data 26](#_Toc104546869)

[3.1.6 Data Normalisation 30](#_Toc104546870)

[3.1.7 Data Preparation 32](#_Toc104546871)

[3.2 Test Strategy 33](#_Toc104546872)

[3.2.1 Manual Inspection 33](#_Toc104546873)

[3.2.2 Precision & Recall 34](#_Toc104546874)

[3.2.3 Fréchet Inception Distance & Inception Score 34](#_Toc104546875)

[3.2.4 Chosen Quantitative Test Strategy Justification 35](#_Toc104546876)

[3.2.5 Deployment of Fréchet Inception Score 36](#_Toc104546877)

[3.3 Rationale for Chosen GAN Approaches 37](#_Toc104546878)

[3.4 Fully Connected GAN 37](#_Toc104546879)

[3.4.1 Strengths and limitations of this approach 37](#_Toc104546880)

[3.4.2 Design of the Fully Connected GAN 38](#_Toc104546881)

[3.4.3 Creating the Generative Adversarial Network 40](#_Toc104546882)

[3.5 Deep Convolutional GAN 41](#_Toc104546883)

[3.5.1 Strengths and limitations of this approach 41](#_Toc104546884)

[3.5.2 Design of the DCGAN 42](#_Toc104546885)

[3.5.3 Creating the Generative Adversarial Network 43](#_Toc104546886)

[4 Results and Evaluation 46](#_Toc104546887)

[4.1 Challenges Evaluating Uncategorised Data 46](#_Toc104546888)

[4.2 Testing & Validation Approach 46](#_Toc104546889)

[4.3 Overview of the Results Obtained 48](#_Toc104546890)

[4.4 Evaluation of the Results Obtained 51](#_Toc104546891)

[4.4.1 Fully Connected GAN vs DCGAN 51](#_Toc104546892)

[4.4.2 GAN Train Times 51](#_Toc104546893)

[4.4.3 Time Required to Produce Kymograph Image 52](#_Toc104546894)

[4.4.4 Manual Inspection 53](#_Toc104546895)

[4.4.5 Fréchet Inception Distance Scores 53](#_Toc104546896)

[4.5 Evaluation of Project Approach 55](#_Toc104546897)

[4.5.1 Project Plan 55](#_Toc104546898)

[4.5.2 Python 55](#_Toc104546899)

[4.5.3 Keras 55](#_Toc104546900)

[4.5.4 Google Colab 55](#_Toc104546901)

[4.5.5 Scrum 56](#_Toc104546902)

[5 Conclusion 57](#_Toc104546903)

[5.1 The extent to which the original Aim & Objectives were met 57](#_Toc104546904)

[5.1.1 Aim 57](#_Toc104546905)

[5.1.2 Objectives 57](#_Toc104546906)

[5.2 Summary of what has been learned 58](#_Toc104546907)

[5.3 Personal Development 58](#_Toc104546908)

[5.4 Areas for Future Work 58](#_Toc104546909)

[References 60](#_Toc104546910)

[Appendix A – Simulated Kymograph Images 62](#_Toc104546911)

[Appendix B – Generated Kymograph Images 64](#_Toc104546912)

[Appendix C – Source Code for Simulating Kymograph Images 66](#_Toc104546913)

[Appendix D – Source Code to Transform Kymograph Images into NumPy Array 69](#_Toc104546914)

[Appendix E – Source Code for Fully Connected GAN for 28x28 Resolution Images 71](#_Toc104546915)

[Appendix F – Source Code for Fully Connected GAN for 72x72 Resolution Images 78](#_Toc104546916)

[Appendix G – Source Code for DCGAN for 28x28 Resolution Images 85](#_Toc104546917)

[Appendix H – Source Code for DCGAN for 72x72 Resolution Images 91](#_Toc104546918)

[Appendix I – Source Code for Calculcating Fréchet Inception Distance 98](#_Toc104546919)

Table of Figures

[Figure 1 GAN Diagram 11](#_Toc104546920)

[Figure 2 Repressilator Simple Genetic Oscillator Circuit Diagram (Feliú, et al. , 2020) 11](#_Toc104546921)

[Figure 3 Image of the Final Pattern from the Repressilator Simple Genetic Oscillator Circuit (Feliú, et al. , 2020) 12](#_Toc104546922)

[Figure 4 Example Kymograph Image of the Oscillations Produced by the Repressilator Gene Network Simulated Using LOICA (Vidal , Vidal-Céspedes & Rudge, 2021) 13](#_Toc104546923)

[Figure 5 Project Development Gantt Chart 15](#_Toc104546924)

[Figure 6 Algorithmic Representation of GAN (Goodfellow, et al. , 2014) 16](#_Toc104546925)

[Figure 7 Snippet from Previous Figure 16](#_Toc104546926)

[Figure 8 GAN Results a) MNIST b) TFD c) CIFAR-10 (fully connected model) d) CIFAR-10 (convolutional discriminator and "deconvolutional" generator) (Goodfellow, et al. , 2014) 17](#_Toc104546927)

[Figure 9 Fully Connected Layer and Convolutional Layer Comparison Diagram (Hope , Resheff & Lieder, 2017) 18](#_Toc104546928)

[Figure 10 Diagram of Convolutional Generator Model (Radford , Metz & Chintala, 2015) 18](#_Toc104546929)

[Figure 11 DCGAN Results Using MNIST Dataset (Radford , Metz & Chintala, 2015) 19](#_Toc104546930)

[Figure 12 Comparison Between Standard GAN Generator and Style-Based Generator (Karras , Laine & Aila, 2018) 20](#_Toc104546931)

[Figure 13 StyleGAN Generated Images Using Range of ψ Values to Demonstrate the Effect of Using Different Style Values (Karras , Laine & Aila, 2018) 21](#_Toc104546932)

[Figure 14 Diagrams and Graph of the Repressilator Genetic Oscillator Circuit (Feliú, et al. , 2020) 22](#_Toc104546933)

[Figure 15 Diagram to Show the Object-Oriented Nature of LOICA (Vidal , Vidal-Céspedes & Rudge, 2021) and its Interaction with Flapjack (Feliú, et al. , 2021) 23](#_Toc104546934)

[Figure 16 Programming Languages used for Machine Learning & Data Science (Hayes, 2020) 24](#_Toc104546935)

[Figure 17 Example Kymograph Simulated Using LOICA 26](#_Toc104546936)

[Figure 18 Sample MNIST Image 27](#_Toc104546937)

[Figure 19 Kymograph Image of 28x28 Resolution Simulated Using LOICA 28](#_Toc104546938)

[Figure 20 Kymograph Image of 72x72 Resolution Simulated Using LOICA 28](#_Toc104546939)

[Figure 21 Kymograph Simulated Using Alpha of 100 29](#_Toc104546940)

[Figure 22 Kymograph Simulated Using Alpha of 10,000,000 29](#_Toc104546941)

[Figure 23 Kymograph Simulated Using Degradation Rate of 1e-7 30](#_Toc104546942)

[Figure 24 Kymograph Simulated Using Degradation Rate of 10 30](#_Toc104546943)

[Figure 25 Graph of the tanh Activation Function 31](#_Toc104546944)

[Figure 26 Equation for Batch Normalisation 31](#_Toc104546945)

[Figure 27 Generated Images from Fully Connected GAN Using Batch Normalisation Layers After Each Convolution After 10 Epochs 32](#_Toc104546946)

[Figure 28 Generated Images from Fully Connected GAN After 10 Epochs 32](#_Toc104546947)

[Figure 29 Reading a 10 million data-point file from storage 33](#_Toc104546948)

[Figure 30 Precision and Recall for GANs Diagram (Kynkäänniemi, et al. , 2019) 34](#_Toc104546949)

[Figure 31 Graph to Show FID Scores Correlate with Better Quality Images (Heusel, et al. , 2017) 35](#_Toc104546950)

[Figure 32 FID Model Structure 36](#_Toc104546951)

[Figure 33 Fréchet Inception Distance and Perceptual Path Length Results from StyleGAN and StyleGAN2 on Four Datasets (Karras, et al. , 2019) 36](#_Toc104546952)

[Figure 34 Fully Connected GAN Discriminator Structure 38](#_Toc104546953)

[Figure 35 Comparison Between the ReLU Activation Function (left) and the LeakyReLU Activation Function (right) 39](#_Toc104546954)

[Figure 36 Fully Connected GAN Generator Structure 39](#_Toc104546955)

[Figure 37 Generated Images from Fully Connected GAN with One Dense Layer in Generator and Discriminator after 200 Epochs of Training 40](#_Toc104546956)

[Figure 38 Generated Images from Fully Connected GAN with Four Dense Layers in Generator and Discriminator after 200 Epochs of Training 41](#_Toc104546957)

[Figure 39 DCGAN Discriminator Structure 42](#_Toc104546958)

[Figure 40 Deep Convolutional GAN Generator Structure 43](#_Toc104546959)

[Figure 41 Generated Images from DCGAN Developed with Batch Normalisation Layers After Training for 10 Epochs 44](#_Toc104546960)

[Figure 42 Generated Images from DCGAN Without Batch Normalisation Layers After Training for 10 Epochs 44](#_Toc104546961)

[Figure 43 Generated Images from DCGAN Using ReLU Activation Function in Generator After Training for 200 Epochs 45](#_Toc104546962)

[Figure 44 Generated Images from DCGAN Using LeakyReLU Activation Function in Generator After Training for 200 Epochs 45](#_Toc104546963)

[Figure 45 Three Categories of Kymograph Waves (Feliú, et al. , 2020) 46](#_Toc104546964)

[Figure 46 Plot to Show Discriminator Accuracy Values Throughout Training for Real and Fake Data for the DCGAN for 28x28 Resolution Kymograph Images 47](#_Toc104546965)

[Figure 47 Plot to Show Discriminator Loss Values Throughout Training for Real and Fake Data for the DCGAN for 28x28 Resolution Kymograph Images 47](#_Toc104546966)

[Figure 48 Generated Images from fully connected GAN for 28x28 Resolution Kymograph Images Trained for 1,000 Epochs 48](#_Toc104546967)

[Figure 49 Generated Images from DCGAN for 28x28 Resolution Kymograph Images Trained for 1,000 Epochs 49](#_Toc104546968)

[Figure 50 Generated Images from Fully Connected GAN for 72x72 Resolution Kymograph Images Trained 1,000 Epochs 49](#_Toc104546969)

[Figure 51 Generated Images from DCGAN for 72x72 Resolution Kymograph Images Trained for 1,000 Epochs 50](#_Toc104546970)

[Figure 52 Generated Images from DCGAN for 72x72 Resolution Kymograph Images Trained for 150 Epochs 50](#_Toc104546971)

[Figure 53 Generated Images from Fully Connected GAN for 72x72 Resolution Kymograph Images Trained 800 Epochs 54](#_Toc104546972)

[Figure 54 Example 28x28 Resolution Kymograph Image Simulated Using Alpha=350000 and Degradation Rate=0.22 62](#_Toc104546973)

[Figure 55 Example 72x72 Resolution Kymograph Image Simulated Using Alpha=350000 and Degradation Rate=0.22 62](#_Toc104546974)

[Figure 56 Example 500x500 Resolution Kymograph Image Simulated Using Alpha=350000 and Degradation Rate=0.22 63](#_Toc104546975)

[Figure 57 Example Random 28x28 Resolution Kymograph Image Generated Using Fully Connected GAN 64](#_Toc104546976)

[Figure 58 Example Random 28x28 Resolution Kymograph Image Generated Using DCGAN 64](#_Toc104546977)

[Figure 59 Example Random 72x72 Resolution Kymograph Image Generated Using Fully Connected GAN 65](#_Toc104546978)

[Figure 60 Example Random 72x72 Resolution Kymograph Image Generated Using DCGAN 65](#_Toc104546979)

Glossary

Machine Learning Terms

**AI** – Artificial Intelligence is an area of Computer Science that focuses on using computers to perform tasks that mimic human intelligence.

**NN** – An Artificial Neural Network is composed of many highly interconnected artificial neurons. Each artificial neuron functions as a simple mathematical function, but by working in unison they can solve more complex problems.

**CNN** – A Convolutional Neural Network is a type of feed-forward artificial neural network that contains many convolutional layers. Convolutional layers filter the input of the layer and then apply a feature map to resize the input. Convolutional Neural Networks mirror the structure of the human visual cortex and are mainly used for image recognition.

**GAN** – A Generative Adversarial Network consists of two neural networks: the generator, and the discriminator. The generator is trained to produce fake data, and the discriminator is trained to identify the fake data generated by the generator from real data. Through backpropagation, the discriminator model provides feedback to the generator so that the weights of the generator can be updated to generate images more likely to fool the discriminator.

**Fully Connected GAN** – A type of generative adversarial network in which the generator and discriminator are both feedforward neural networks that use fully connected layers, also known as dense layers.

**DCGAN** – A Deep Convolutional Generative Adversarial Network is a version of a Generative Adversarial Network that uses deep convolutional networks instead of the fully-connected networks used in a normal Generative Adversarial Network (Radford , Metz & Chintala, 2015).

**Epoch** – Running all the training data through a machine learning model once.

**FID –** The Fréchet Inception Distance is a metric used to compare the quality and diversity of two different image sets. This metric is commonly used for assessing the quality of generated images.

Genetic Circuit Terms

**Synthetic Genetic Circuit** – A Synthetic Genetic Circuit is a man-made circuit designed out of DNA. A variety of circuits can be created using different genetic components, for example the simple logic gates inside computers can be recreated with a genetic circuit.

**Simple Genetic Oscillator Circuit** – A simple synthetic genetic circuit that produces oscillations. Oscillations can be produced by the expression and repression of the proteins produced by different DNA in the circuit.

**Repressilator** – An example of a simple genetic oscillator circuit, and the circuit studied by this project. Composed of three repressor proteins, each represses the expression of the next which creates a feedback loop and produces oscillations.

**Kymograph** – A Kymograph is a space-time plot which displays intensity values over time. The spatial information displayed by a Kymograph is only one-dimensional which is a compromise to displaying the time data.

**LOICA** – Logical Operators for Integrated Cell Algorithms is a Python package that allows for designing, modelling and characterising genetic networks.

# Introduction

## ****Motivation and Rationale****

### ****The Context:****

A GAN is a relatively new concept in machine learning. It was designed by Ian Goodfellow and his colleagues (Goodfellow, et al. , 2014).

Diagram

Description automatically generated

Figure 1 GAN Diagram

A GAN consists of two neural networks: the generator, and the discriminator. The generator learns to produce fake images from an input of random noise. The discriminator learns to distinguish the fake images from the real images. The generator and discriminator compete to become better at beating the other model. Through backpropagation the generator tweaks the weights in its model based on the loss value of the discriminator. This is how the generator learns to produce better quality images as it learns through training which weight values to use to have the best chance of fooling the discriminator. The weights of the discriminator are also updated based on how accurately it can classify real and fake images. In this architecture the generator and discriminator models both become better at their jobs over time during training. This approach has proven effective at creating a generator model that can generate high quality images, given enough training.

Diagram, schematic

Description automatically generated

Figure 2 Repressilator Simple Genetic Oscillator Circuit Diagram (Feliú, et al. , 2020)

A simple genetic oscillator circuit describes a circuit constructed of genes that produces oscillations. One example of a simple genetic oscillator circuit is the repressilator (Wyss Institute for Biologically Inspired Engineering at Harvard, 2019). The repressilator consists of three genes, each of which blocks the expression of one of the other proteins. These genes are linked in a negative feedback loop. If the concentration of one of the repressor proteins falls below a certain threshold value, the protein it has been repressing is expressed. The expression of this protein then represses the next and this process repeats in a cyclical fashion which is why the oscillations are produced. These oscillations can be represented in an image which is called a kymograph.

A picture containing vector graphics

Description automatically generated

Figure 3 Image of the Final Pattern from the Repressilator Simple Genetic Oscillator Circuit (Feliú, et al. , 2020)

### ****The Problem:****

The repressilator gene network is a type of simple genetic oscillator circuit and will produce oscillations which can be recorded and graphed on a kymograph. These kymographs show the concentration of each repressor protein over time. Depending on the genetic parameter values that are used to simulate the repressilator gene network, different kymograph images can be produced. These kymograph images can be studied using machine learning methods to conduct experiments to attempt to make further findings. However, this can require a lot of training data which is computationally expensive and slow to produce entirely by simulating the repressilator gene network. Machine learning based image generation is an alternative solution that could be used to generate more kymograph image data quicker than purely via simulation.

The repressilator’s ability to produce oscillations makes it useful as a genetic clock that can be used to construct more complex genetic circuits (Wyss Institute for Biologically Inspired Engineering at Harvard, 2019). To use the repressilator gene network for this application a complete understanding of how the repressilator functions is required.

### ****The Rationale:****

A GAN will be developed to generate kymograph images of the oscillation patterns produced by the repressilator gene network. This model will be trained using kymograph images produced by simulating the repressilator gene network with different genetic parameters for each image. The generator will attempt to generate images of similar quality to the training data to confuse the discriminator while the discriminator will be attempting to correctly determine which images are real or fake to beat the generator. After iterative training is complete there is no longer a need for the discriminator model as the generator model can be separated from the GAN and used to generate kymograph images. This is an alternative to running a biological simulation for each image which can be unfeasible for the large amounts of training data some machine learning applications require.

A picture containing colorful

Description automatically generated

Figure 4 Example Kymograph Image of the Oscillations Produced by the Repressilator Gene Network Simulated Using LOICA (Vidal , Vidal-Céspedes & Rudge, 2021)

## Aim

The aim of this project is to produce a GAN capable of generating kymograph images similar in quality to that of the kymograph images simulated using a biological model which will be used as training data.

## Objectives

1. **Simulate the repressilator gene network to produce the required training data for the GAN.**

The GANs being used will require a significant number of images to train the model. This training data will need to be simulated using LOICA (Vidal , Vidal-Céspedes & Rudge, 2021). Multiple datasets of different resolution kymograph images and different amounts of kymograph images will need to be simulated. A lower resolution dataset will be used for developing the GAN as it will allow for faster development/test cycle times. Once the development of the GAN is complete a higher resolution dataset can be used so that higher resolution images can be generated. For this objective the resolutions and number of images to simulate will need to be decided and these datasets will need to be produced. This objective will have been achieved when the training datasets have been produced.

1. **Process the training data to prepare it for the GAN.**

The training datasets of thousands of kymograph images will need to be processed before it can be used to train the GAN. Processing the training data is a vital step for machine learning models to improve performance. Processing the data before using it on the model helps to improve the quality of the data. If the data is high quality, then more useful information can be derived from it which will improve the model’s ability to learn. This processing may require changing how the images are stored and how the pixel values are stored. How the data is stored and what format it will be stored in will need to be decided to work best with the GAN. This objective will have been achieved when the training data has been processed in such a way that it is compatible with a GAN.

1. **Develop the GAN.**

The GAN will need to be developed to generate kymograph images. Ideally a simple GAN will first be developed as a proof of concept which will then allow experimentation to improve performance and for experience to be gained. This will then help inform the development of a more complex GAN that should be able to generate higher quality images. This objective will have been achieved when a GAN capable of producing images of similar quality to the training data has been produced.

1. **Test and train GAN.**

Once the GAN has been developed it will need to be tested to ensure that it will work with the different datasets that have been produced. This will involve testing to make sure that the GAN can train with higher resolution images and larger numbers of images. From this testing there are likely to be some changes that can be made to the GAN to improve the quality of the images produced. This step will also require the creation of some graphs to summarise the model’s performance. For example, graphing the loss values and accuracies from training. This objective will have been achieved when the GAN has been tested and trained on all the datasets and results from training have been produced.

1. **Quantitative evaluation of the quality of the generated images.**

To evaluate the quality of the images generated by the GAN a method for numerically comparing the fake images to the real images will need to be devised. GANs are widely known to be difficult to evaluate quantitatively and there are a wide number of possible methods that can be used. For this a method will need to be chosen and applied to the real and fake images to gauge how close the fake images are in terms of quality to the real images. This objective will have been achieved when a quantitative metric has been applied to the GAN and a result has been obtained.

## Dissertation Structure

This report is structured as follows:

**Introduction**: Firstly, there is the Introduction which contains the motivation and rationale as well as the aim and objectives.

**Background Research**: which contains all the research conducted before the implementation began. This focuses on the two main areas of GANs and Genetic Circuits.

**Design & Implementation**: This covers the design phase which involves more detail on the chosen approach and techniques used. The implementation phase is also covered which discusses how the solution was developed and any issues encountered at each stage.

**Results and Evaluation**: Covered in this section are a detailed explanation and evaluation of the results obtained. An overall evaluation of the whole project is also included which discusses the successfulness of this project.

**Conclusion**: This section discusses how successful the overall aim of this project was as well as each of the objectives. What has been learned is also summarised. Finally, this section covers any future work that should be completed if this project is to be built on and improved.

## Project Approach

Timeline

Description automatically generated with low confidence

Figure 5 Project Development Gantt Chart

The above image shows my development plan for this project which partitions the development into each of the main objectives that need to be accomplished and then splits up the time available among these objectives. The development strategy was to use the Scrum (Scrum.org, 2022) software development methodology and split the project into sections which can then be their own Scrum sprints. Efforts will be made to stick to this development schedule to keep the project on track.

# Background Research

## Generative Adversarial Networks

GANs are a recent invention in machine learning having been first proposed in 2014 (Goodfellow, et al. , 2014). GANs have become a very popular method for image generation due to their effectiveness. A GAN is composed of two machine learning models: the generator and the discriminator. The generator’s job is to produce images to fool the discriminator into thinking they are real images. The discriminator’s job is to correctly identify fake and real images. These two models are then connected to create the “adversarial” aspect of the GAN. The loss of the discriminator model is used to inform the generator model so that it can generate images more likely to fool the discriminator. The loss value of the discriminator is a measure of how accurate it is at determining whether an image is real or fake. Under ideal conditions and after enough training, the generator would produce images that were imperceptible to the discriminator from real images. This would result in an accuracy for the discriminator of 50% because the discriminator could not detect real or fake images correctly so must guess.

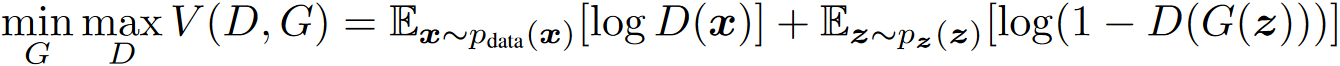


Figure 6 Algorithmic Representation of GAN (Goodfellow, et al. , 2014)

In the above equation the generator G is attempting to minimise the value of the function V, whilst the discriminator D is doing the opposite which is to maximise the value of the function V. The variable 𝑥 represents the training data of real images, while z represents the random noise used as input for the generator to generate fake images.

The discriminator D is trained to maximise the probability of correctly labelling both real and fake, so attempts to maximise the value of the function V.

Meanwhile the generator is trained to minimise the probability of the discriminator correctly labelling both real and fake images, so it is trained to minimise this part of the function V:

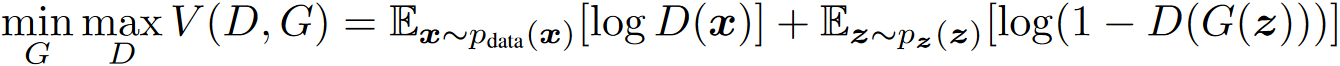


Figure 7 Snippet from Previous Figure

The generator does this by generating better fake images so that the discriminator’s probability of it being a real image increases and the value of the above snippet from the function V increases.

Graphical user interface, application

Description automatically generated

Figure 8 GAN Results a) MNIST b) TFD c) CIFAR-10 (fully connected model) d) CIFAR-10 (convolutional discriminator and "deconvolutional" generator) (Goodfellow, et al. , 2014)

The results in the above figure are for three different datasets to showcase the model’s effectiveness. For parts a), b), c) and d) in the above figure, twenty generated images are displayed. For each part in the rightmost column (highlighted in yellow) displays the closest matching images from the training data to the generated images in the adjacent column. This was to demonstrate that the generator is producing new images instead of memorising the training dataset and producing a random image from the training set as the output.

From the above figure the GAN has generated flawless looking images for a). This is because the images in the MNIST dataset are the simplest and the lowest resolution out of the three datasets. For b) the generated images are close to the real images in terms of quality. However, it is still possible to notice a difference, this is because these images are more complex than the MNIST images and a higher resolution so are harder for the generator to replicate. For c) and d) the CIFAR-10 dataset was used but on two different GAN architectures. For c) a standard GAN using fully connected layers was used, and for d) a GAN using convolutional layers was used. When comparing the results from these two architectures it is clear to see that the generated images from d) are closer to the real data than the generated images from c).

### Deep Convolutional Generative Adversarial Networks

After studying the above results, I conducted research into DCGANs as this looked to be a more effective architecture than a GAN using fully connected layers. I researched the paper Unsupervised Representation Learning with Deep Convolutional GANs (Radford , Metz & Chintala, 2015) where DCGANs were first proposed. The DCGAN models in this paper performed very well for image synthesis.

Diagram

Description automatically generated

Figure 9 Fully Connected Layer and Convolutional Layer Comparison Diagram (Hope , Resheff & Lieder, 2017)

The diagram above shows the difference between fully connected layers and convolutional layers. In a fully connected layer, each unit is connected to all units from the previous layer. In a convolutional layer, each unit is connected the same way by connecting to a set number of nearby units from the previous layer.

Diagram

Description automatically generated

Figure 10 Diagram of Convolutional Generator Model (Radford , Metz & Chintala, 2015)

The diagram above shows a convolutional generator model for generating images of 64x64 resolution. The model begins with the 100-dimensional uniform noise distribution z. From this the image is gradually expanded from 4x4 to 64x64 using four convolutional layers. This diagram also shows how the convolutional layers are connected by mapping an area of 1x1 from each layer to the next.

Shape, arrow

Description automatically generated

Figure 11 DCGAN Results Using MNIST Dataset (Radford , Metz & Chintala, 2015)

The above figure compares training data from the MNIST dataset to images generated by a standard GAN and then a DCGAN. The GAN images are easy to tell apart from the real MNIST images due to some random noise in the images. However, the images from the DCGAN are far better and could easily pass as actual images from the training data. This shows that the DCGAN is better for image generation than a regular GAN for this application at least.

### StyleGAN

In 2018 NVIDIA released a paper proposing an alternative generator architecture for GANs (Karras , Laine & Aila, 2018). This new architecture was a style-based generator which was capable of automatically learning high-level attributes from the images. A generator from a standard GAN begins with an input layer, this is usually a random noise vector. A style-based generator does away with the input layer and instead begins from a learned constant. The image is then informed with a style after each convolutional layer. This provides control over the images produced which a generator from a standard GAN cannot do. For example, when using a StyleGAN trained to generate images of faces, the user could specify the gender of the face, whether they are wearing glasses etc. A website (Wang, 2019) was created to demonstrate the capabilities of GANs to the public. This website shows a different 1024x1024 resolution image of someone’s face each time it is refreshed; however, these people do not exist as the images shown were all generated using NVIDIA’s StyleGAN2 (Karras, et al. , 2019). StyleGAN2 was created to fix the characteristic artefacts of the images produced by the original StyleGAN and to improve the overall quality of the images.

Diagram

Description automatically generated

Figure 12 Comparison Between Standard GAN Generator and Style-Based Generator (Karras , Laine & Aila, 2018)

The above image shows the generator from two different GAN architectures. Part (a) shows a standard layer structure from a DCGAN. Part (b) shows the generator structure from a style-based GAN which has similar layers to the DCGAN but incorporates styles after each convolution to inform the image generation. The style-based generator uses Adaptive Instance Normalisation (AdaIN) layers to normalise the image after each convolution. The AdaIN normalisation layers are controlled by a value “A” which controls the style of the images being generated. The noise vector “B” is added to introduce some randomness to the generated images.

StyleGAN uses a method for training called progressive growing (Karras, et al. , 2017). Progressive growing means that both the generator and discriminator models begin training with small images. In the case of StyleGAN a starting resolution of 4x4 is used. This is then doubled to 8x8 and so on until the desired output resolution is reached. For this to work the generator and discriminator models need to use different layers depending on the training resolution being used, so layers are added to each model with each increase in resolution. This approach speeds up the time to train a GAN but is significantly more complex to implement than standard training methods.

A collage of a person

Description automatically generated with low confidence

Figure 13 StyleGAN Generated Images Using Range of ψ Values to Demonstrate the Effect of Using Different Style Values (Karras , Laine & Aila, 2018)

The above image displays the effect that changing the style value has on the images generated. The closer the ψ value gets to zero the closer the faces produced are to the “mean” face. By using a negative ψ value the “anti-face” of any positive ψ value can be generated. Certain high-level attributes gradually become more prevalent as the ψ value changes. For example, different ψ values gradually change the gender, viewpoint, age and whether the face is wearing glasses. StyleGAN can generate images other than faces. In the paper they display generated images of bedrooms, cars and cats.

## Repressilator Gene Network

The repressilator gene network is an example of a simple genetic oscillator circuit. The repressilator consists of three genes each of which blocks the expression of one of the other proteins. These genes are linked in a negative feedback loop. If the concentration of one of the repressor proteins falls below a certain threshold value, the protein it has been repressing is expressed. The expression of this protein then represses the next and this process repeats in a cyclical fashion which is why the oscillations are produced.

Diagram

Description automatically generated

Figure 14 Diagrams and Graph of the Repressilator Genetic Oscillator Circuit (Feliú, et al. , 2020)

Part A of the above image is a diagram of the repressilator genetic oscillator circuit, this shows the three genes that make up the repressilator and how they interact with each other. The arrows signify the direction of the interaction, so 1 represses 2, 2 represses 3, and 3 represses 1. Part B shows a genetic circuit diagram of a plasmid encoding the repressilator. A plasmid is a small circular DNA molecule. Plasmid encoding is the process of encoding the genetic circuit onto a plasmid molecule. Part C shows the concentration of each repressor in the repressilator genetic oscillator circuit over time. The oscillations produced by the repressilator are rhythmic and uniform, so this should be something that a machine learning model can learn.

### LOICA

Logical Operators for Integrated Cell Algorithms (LOICA) is a Python package for designing, modelling and characterising genetic networks (Vidal , Vidal-Céspedes & Rudge, 2021). LOICA is an object-oriented library and makes use of classes to represent different biological and experimental components.

Diagram, schematic

Description automatically generated

Figure 15 Diagram to Show the Object-Oriented Nature of LOICA (Vidal , Vidal-Céspedes & Rudge, 2021) and its Interaction with Flapjack (Feliú, et al. , 2021)

In the above image the signal refers to the part of the genetic circuit that is being monitored and is where measurements would be taken. Vector describes the synthetic DNAs encoding of the genetic circuit. Supplement refers to any supplementary chemicals that interact with components of the genetic circuit. Sample corresponds to the basic unit that is subject to measurement. Finally, an Assay is the measurement experiments, including varying experimental conditions, performed to explore different aspects of the study (Feliú, et al. , 2021). LOICA uses Flapjack which is a data management system allowing the analysis and visualisation of data from biological experiments. LOICA also uses the Synthetic Biology Open Language (SBOL) which is a community-developed data standard that can be used to represent knowledge across multiple scales and throughout the entire synthetic biology workflow (McLaughlin, et al. , 2020).

# Design & Implementation

## System & Software Architecture

### Python

Python was chosen as the programming language because I had experience using Python for machine learning, so had an initial idea of how to execute the project and the libraries available to me.

In the 2019 Kaggle Machine Learning and Data Science Survey 87% of the 14,762 who answered the question said they used Python on a regular basis (Hayes, 2020). This made it by far the most popular language with SQL next at 44.3%. This was also shown by my early preliminary research into GANs where I found most online resources and tutorials used Python.

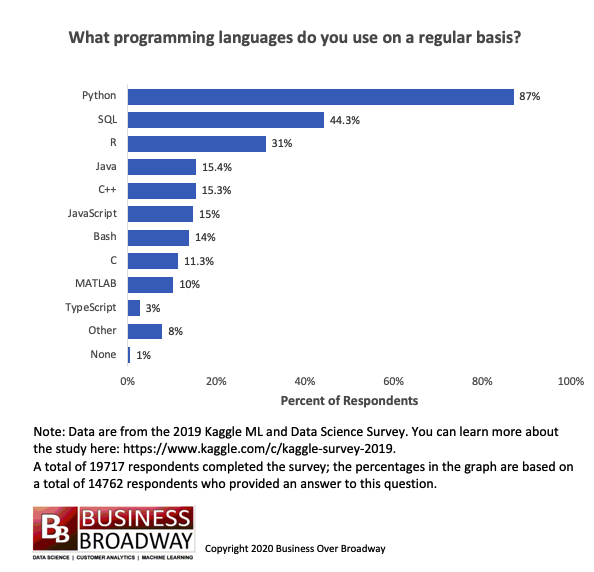


Figure 16 Programming Languages used for Machine Learning & Data Science (Hayes, 2020)

I did not want to slow down this project by learning a new language. The programming languages that I already know are Python, Java and C. Java was used regularly by 15.4% of programmers and C by 11.3% in the above survey. These languages are used much less often for data science and machine learning applications when compared to Python.

### Google Colab

I chose to use Google Colab (Google, 2022) for development. I had developed my previous machine learning project on Google Colab, so I had experience using the platform. Google Colab provides Graphics Processing Units (GPU) and Tensor Processing Units (TPU) to be used for machine learning. This was very important because using a GPU can vastly speed up the training time of a machine learning model when compared to just using a CPU. Using Google Colab would also require the project files to be stored on Google Drive, this would help reduce the chance of any data loss as my project data would be stored in the cloud as well as in regular local copies that I would make. Google Colab also auto saves my project files and allows me to view saved files from the last few days in case a rollback was needed.

The other alternative I explored was running my machine learning models locally on my PC which is equipped with a powerful enough GPU for machine learning applications. The main downside to this approach was that I would need to travel home from university during the development phase of the project. This would have been problematic as my laptop which I travelled home with does not have a GPU, so any machine learning would have been very slow to run. A possible solution would have been to remote into my PC from home, however this is a not a fool-proof solution because turning on a Windows PC remotely can be unreliable.

I purchased Google Colab Pro which gave access to faster GPUs which would speed up running machine learning models. There is also no maximum amount of time that you can use a GPU per day which there is with the free version. Google Colab Pro cuts off any program that has been running for 24 hours continuously, whereas the free version cuts off any program that has been running for 12 hours. This will allow more time to train the GANs which are notorious for taking long amounts of time to train.

### Keras

In my previous machine learning project, I used Keras for my machine learning framework. Keras worked well for my previous application, so I wanted to make use of it again. However, I decided to research alternative solutions to investigate if Keras was the best fit for this project. The two main alternatives were Tensorflow and PyTorch. These are both lower-level machine learning frameworks than Keras.

Keras is an open-source library built on top of Tensorflow. Tensorflow was developed by Google and is based on the Theano Python library. Keras provides a high-level abstraction of the Tensorflow library. This makes Keras easier to use than pure Tensorflow, but this comes at the cost of less control over the machine learning model. Google Colab comes with Keras built in, so this is an advantage when compared to the other two options.

Tensorflow could allow me to produce a better model that generates more accurate images. But this would most likely require more configuration to optimise the model due to Tensorflow being a lower-level API than Keras. As well as this I would be learning how to use Tensorflow having never used it before. My previous experience with Keras would probably help me here since Keras is an abstraction of Tensorflow. Using Tensorflow would require time to learn the Tensorflow API; a disadvantage given the already longer amount of time needed to produce a model using Tensorflow when compared to Keras.

Like Tensorflow, PyTorch is a low-level machine learning API. PyTorch uses the Python programming language as the software’s user interface and the Torch machine learning library for the machine learning back-end. Torch is one of the first machine learning libraries and was released in 2002. PyTorch is the software implementation of the Torch library made to use Python as the programming language (Terra, 2022). Like Tensorflow, PyTorch is a library I do not have previous experience with so I would need to learn how to use this library during the project.

After reviewing the best machine learning framework options, I researched GANs to find out which framework was the most used. I discovered that many of the online demonstration videos and articles using GANs used Keras. I suspect this is because Keras is a popular API and is easier to use than Tensorflow and PyTorch. Thus, more easily accessible resources exist to help beginners to machine learning who may find it more difficult to use Tensorflow or PyTorch. I see Keras being a higher-level API as an advantage and the main reason why I chose to use Keras. GANs were beyond the scope of what was taught in my machine learning modules so using a familiar machine learning framework would make researching and implementing a GAN more intuitive.

### LOICA

To simulate the repressilator genetic circuit and produce the kymograph images needed as training data for my model, I needed to use the Python package LOICA (Logical Operators for Integrated Cell Algorithms) (Vidal , Vidal-Céspedes & Rudge, 2021). This Python package, which was created for designing, modelling and characterising gene networks, is being developed by my dissertation supervisor and his peers. For this reason, I did not research alternative methods for simulating the training data I would need because I could get support from my supervisor and his peers if I had any difficulties using LOICA.

A picture containing surface chart

Description automatically generated

Figure 17 Example Kymograph Simulated Using LOICA

### Simulated Data

My model would be trained on images of kymographs from simulations of the repressilator genetic circuit. I would need to simulate these images and create my own datasets as these did not already exist. I knew that machine learning models can take a long time to train. GANs are no exception and can require even more training than simpler machine learning models. For this reason, I decided that I should use low-resolution images. This would allow me to use simpler models and would also reduce the training times when compared to using higher resolution images.

Training a GAN on a dataset with too few images can lead to GAN overfitting (GoodFellow , Bengio & Courville, 2016, pp. 110-116). This happens when the model learns the data too well. In the example of a GAN, if the model is overfitting, the generator will have learnt the training data and produce an exact copy of one of the images from the training data each time. This leads to the discriminator always returning the same value because the generated images are copies of the real images so it can’t differentiate them. A paper on training GANs with limited data (Karras, et al. , 2020) demonstrates that “good results are now possible using only a few thousand images”. This is a problem I would like to avoid, and it would be easier to simulate enough images for training rather than artificially producing more data to train with from a small dataset. Because of this the first datasets that I simulated for 28x28 resolution images and 72x72 resolution images were both 10,000 images in size. Another larger dataset of 65,536 (256²) 28x28 resolution images would also be simulated to test whether the results from the GAN would improve when using significantly more than 10,000 images for training.

GAN underfitting can also be a problem, but this is much easier to detect than overfitting. This is because an underfitting generator will generate poor quality images. The generated images are diverse, but don’t look very much like the training images. These images could be blurry or just random noise, in this case the discriminator has no problem detecting which images are fake and which are real.

My strategy was to develop a working model using low-resolution images and then to modify that model to work with higher resolution images. For the low-resolution images, I decided to simulate images of 28x28 pixels. In my previous machine learning project, I had used the MNIST dataset. This dataset contains 60,000 images of hand-written numbers which are 28x28 pixels in resolution but only have a single colour channel due to the images being monochromatic. The images I would be using would have the standard three colour channels for full colour images, so are more complex than the MNIST images and will require more training time.

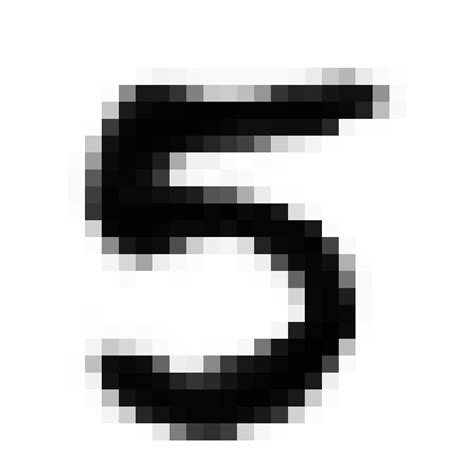


Figure 18 Sample MNIST Image

I had experience with machine learning on images of this size and did not want to use lower resolution images in case there was not adequate information in the images for the machine learning model to make predictions. For the higher resolution images, I decided to use images of 72x72 pixels. With this resolution the generated images from my model would have 6.6x the total pixels and use three colour channels instead of one. This is a lot more data than each image from the MNIST dataset has so this should allow for finer details in the images to be recognised by the model.

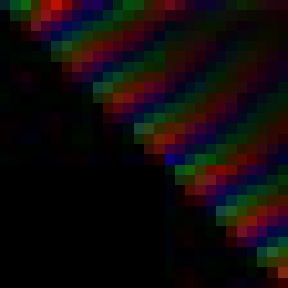


Figure 19 Kymograph Image of 28x28 Resolution Simulated Using LOICA

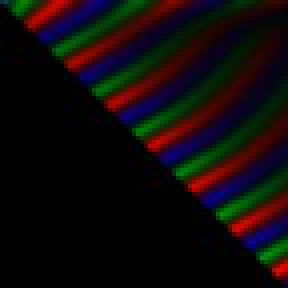


Figure 20 Kymograph Image of 72x72 Resolution Simulated Using LOICA

The training data for my GAN is the kymograph images produced by the repressilator gene network. The repressilator gene network is composed of three repressor genes. Each of these genes has two genetic parameters that can be set for the simulation. These two values are the alpha and degradation rate which can each be individually set for each repressor. After experimenting with the images produced, I came up with a range for each parameter for image simulation.

For the alpha value I used a range of 1,000 to 1,000,000. After experimentation I found this to be the best range to produce useful training data for my GAN. I discovered that alpha values too close to zero produced completely black images no matter what the degradation rate value was. These completely black images could confuse the machine learning model, so I picked 1,000 as the minimum value which I found produced images that would be more usable for training. There was a similar problem when picking the maximum value for the alpha. Beyond a certain value the images produced would be completely black. With a larger alpha value the less visible the green and blue pattern lines in the image would be. If the alpha value was large enough the image would just be red pattern lines with black voids in-between. I discovered 1,000,000 to be a good maximum value that did not produce this effect.



Figure 21 Kymograph Simulated Using Alpha of 100



Figure 22 Kymograph Simulated Using Alpha of 10,000,000

For the degradation rate, I decided to use a range of 1e-6 to 1 after experimenting with the images produced for different values. Like with the alpha value, a degradation rate value that is too high or too low would produce an entirely black image no matter what the alpha value was set to. Because of this I decided that 1e-6 was a good minimum value to use because the images produced using this value still had some discernible features for a machine learning model to learn. The value of 1 was chosen after reviewing the images produced in which I found that the larger the degradation rate value the more pattern lines would be in the image. I decided 1 was a good maximum value because you could still make out the pattern lines with the relatively low resolution of 28x28 that I was going to use for training my GAN.

A blurry image of a rainbow

Description automatically generated with low confidence

Figure 23 Kymograph Simulated Using Degradation Rate of 1e-7

Background pattern

Description automatically generated

Figure 24 Kymograph Simulated Using Degradation Rate of 10

### Data Normalisation

The image data needed to be normalised (Chollet, 2018, p. 101). This was because the generator uses the tanh activation function in its last convolutional layer which has a minimum value of -1 and a maximum value of 1. So, the pixel values of the training data need to be scaled down to fit this range.

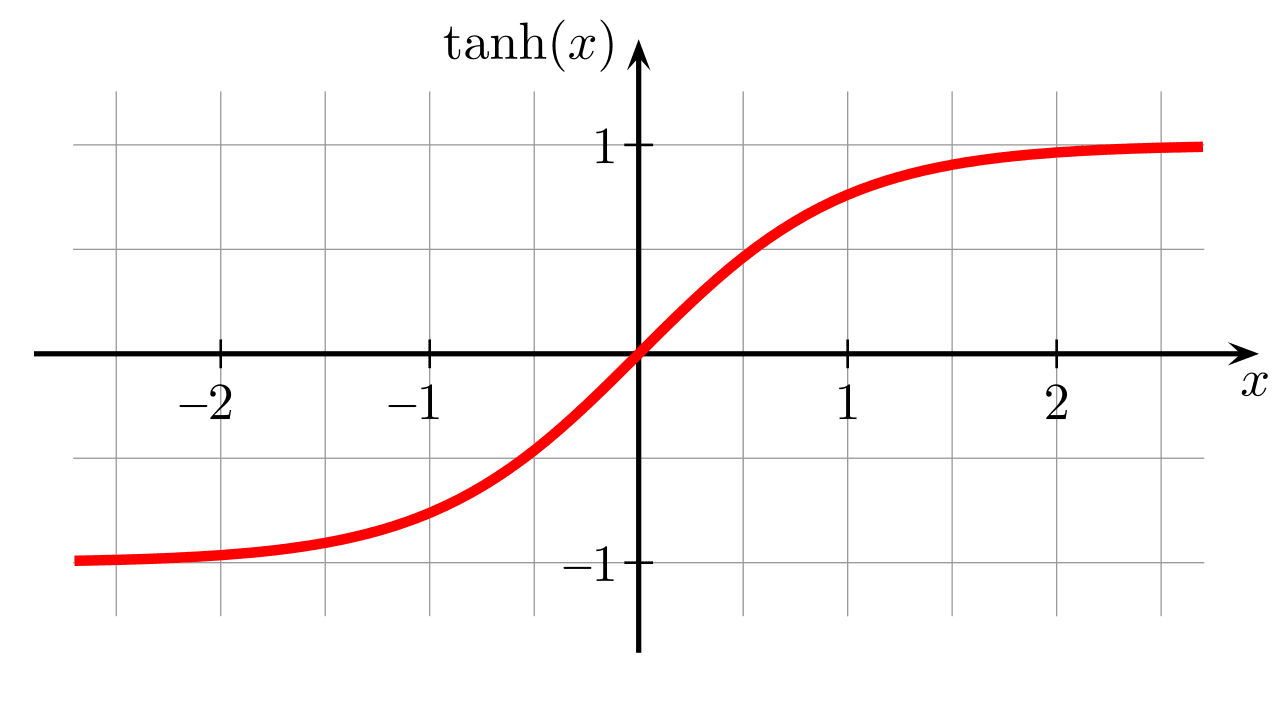


Figure 25 Graph of the tanh Activation Function

The pixel values of each image that is simulated using LOICA have the range of 0 – 255 and are stored as a double. A double requires eight bytes to store, but the image data only requires one byte to store each pixel value. Before the images are run through the GAN the pixel values are converted from doubles to float32’s. A float32 uses four bytes to store which is half of what a double requires. This is done to reduce the amount of data being run through the model and should help reduce train times. The pixel values are then scaled down to between -1 and 1. The double would allow for more significant figures to be stored of the now decimal pixel values which are between -1 and 1, but this would not provide any tangible benefit.

Another method of normalisation is to use normalisation layers inside the machine learning model. Batch normalisation layers normalise the output of a previous layer before it is used by the next layer. It does this by first calculating the mean and standard deviation of each “mini-batch”. First each value has the mean subtracted from it and is then divided by the square root of the variance (standard deviation squared) plus e which is a small value to avoid division by zero errors (Langr & Bok, 2019, p. 55).

Diagram

Description automatically generated

Figure 26 Equation for Batch Normalisation

Batch normalisation layers were trialled in my GAN but did not yield any improvement and were found to make the generated images completely unusable.

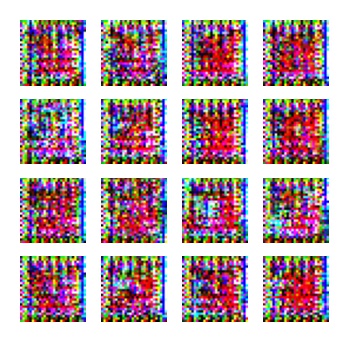


Figure 27 Generated Images from Fully Connected GAN Using Batch Normalisation Layers After Each Convolution After 10 Epochs

Background pattern

Description automatically generated

Figure 28 Generated Images from Fully Connected GAN After 10 Epochs

### Data Preparation

The simulated kymograph images are saved as jpg files. This is adequate for viewing individual images but is not ideal for loading and running through a machine learning model. This is because if the kymographs were all separate jpg files, then to run the model you would need to load every file and combine the image data into one array which takes a long time for thousands of images. This might not be such a big problem if the number of images being used was relatively small. However, the smallest datasets I would be using included 10,000 images. If it was required to load these images in every time and store each image in an array for the model to then process this would vastly increase the runtime of my model. Because of this I researched options for creating the array and then saving it as one file. With the image data all in one file it could then be loaded quickly and used in the model whenever needed. The image data that needed storing is a 4-dimensional array, so this is not something that can be efficiently stored as a .txt or .csv file. For example, for 10,000 images of 28x28 the array would have a shape of [10,000, 28, 28, 3], with 3 representing three colour channels per image.

I was planning on using the Python library NumPy to store the image data in a NumPy array for feeding into the GAN. I researched methods for saving a NumPy array and discovered there is a method built-in to NumPy that allows you to save the array as a single .npy file. This was the most convenient solution. However, I also wanted to make sure that it was a sufficiently good solution. I found an article online that listed the loading times for a 10,000,000 data-point file using .txt, .csv and .npy (Nistrup, 2019).

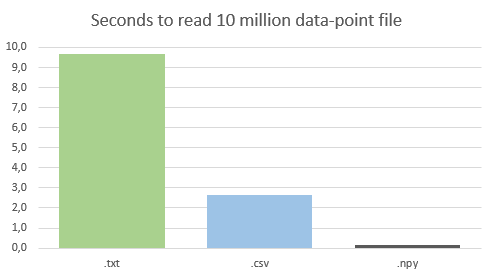


Figure 29 Reading a 10 million data-point file from storage

The loading times were 9.67, 2.66 and 0.13 seconds respectively. In this example the .npy file was almost two orders of magnitude faster than the .txt file. Even if a .npy file did not suit my application as well as the test example there would still be a large speed benefit over the alternatives. Another benefit of the .npy file compared to .txt and .csv is that the file size when saved can be quite a lot smaller. This will be useful as I am using Google Drive to store the training data so only have a limited amount of storage that I can use.

## Test Strategy

### Manual Inspection

Evaluating the performance of GANs can be quite challenging. This is because the data generated by a GAN is an image which can be difficult to assess quantitatively.

While several measures have been introduced, as of yet, there is no consensus as to which measure best captures strengths and limitations of models and should be used for fair model comparison. (Borji, 2018)

Visual examination of samples by humans is one of the common and most intuitive ways to evaluate GANs. (Borji, 2018)

An effective and easy method for assessing the images produced is manual inspection. This involves manually inspecting images at certain intervals during training and inspecting the generated images once training is complete. This method comes with its own drawbacks, for example assessing an image is subjective so two image inspectors cannot be expected to have the same criteria for what a “good” image is. I will be the only person assessing images during training, so this is not an issue. The main downside of manual inspection is that it is unfeasible to inspect all the images generated by the GAN. I will be saving 16 generated images from every tenth epoch as an image that can be reviewed later. This should be a sufficient number of images saved often enough to confirm that the model is improving. The images from the GAN will not change very much after each epoch so comparing images from one epoch to the next will make it difficult to spot any improvements. By comparing images from every tenth epoch, improvements in the images generated will be clearer to see. Another downside with manual inspection is that assessing the quality of the images requires some knowledge of what the images should look like. My GAN will generate kymograph images of the pattern formed by the repressilator genetic circuit, so being able to assess the images assessed will require experience with genetic circuits and pattern formation.

I will be using manual inspection during training and will then use a more quantitative method for assessing the images produced once training is complete. The process of inspection will help my GAN avoid mode collapse (Huang , Yu & Wang, 2018). Mode collapse happens when the generator works out how to best trick the discriminator with a very-limited number of images. This would result in the generated images from any input value of random noise being very similar. From reviewing the generated images, I can then pick the best generator model which is also saved every tenth epoch and use that for future image generation.

### Precision & Recall

Precision and Recall are two very popular metrics for assessing the accuracy of a machine learning model. These metrics are normally used to assess classification neural networks, but they have been adapted to work with GANs (Kynkäänniemi, et al. , 2019).

Scatter chart

Description automatically generated

Figure 30 Precision and Recall for GANs Diagram (Kynkäänniemi, et al. , 2019)

The above diagram shows what Precision and Recall signify in the context of GANs. Pg in red represents the distribution of generated images, while Pr in blue represents the distribution of real images. Precision is the probability that a random generated image falls within the distribution of real images. Recall is the probability that a random real image falls within the distribution of fake images. Computing this metric could be challenging as the distribution of real and fake images would need to be calculated.

### Fréchet Inception Distance & Inception Score

The Fréchet Inception Distance (FID) and inception score are two of the most popular ways to evaluate the quality of the images produced by a GAN. Both methods make use of the pre-trained InceptionV3 convolutional neural network which is trained to classify images from 1,000 different classes.

Inception Score was proposed as an alternative to using human annotators to judge the visual quality of samples (Salimans, et al. , 2016). This paper points out several flaws in using human participants to distinguish between generated images and real images which serves as the motivation for developing an automatic method for GAN evaluation. These include inherent biases that people have when assessing image quality, and after receiving feedback on their mistakes annotators behave differently for future image assessments which can skew the data recorded. The Inception Score metric involves using the InceptionV3 to classify many generated images. This model is used to predict the probability of each generated image belonging to each of the 1,000 classes that the model is trained to classify. The Inception Score aims to assess two properties about a collection of generated images which are the image quality and the image diversity. The Inception Score has a minimum value of 1 and a maximum value of 1,000, with higher values representing better quality images.

Inception Score is a good metric for assessing the quality and diversity of a collection of generated images, however it is unable to compare these generated images to the real images used in training. Due to this the Fréchet Inception Distance was conceived which captures the similarity of generated images to real images (Heusel, et al. , 2017). This metric again makes use of the InceptionV3 convolutional model. However, unlike the Inception Score the FID uses this model to classify both the generated images and the real images and then compares the similarity of the probabilities for each of the 1,000 classes in the InceptionV3 model. The lower the FID score the better the quality of the generated images are, a perfect FID score of 0 indicates that the generated images match the real images identically.

Chart, line chart

Description automatically generated

Figure 31 Graph to Show FID Scores Correlate with Better Quality Images (Heusel, et al. , 2017)

### Chosen Quantitative Test Strategy Justification

Of the quantitative GAN evaluation metrics considered, FID and Inception Score were the two that could be deployed most easily when compared to Precision and Recall. This was because all that was involved was using a pre-trained classification model and then performing some calculation on the output. FID was chosen over Inception Score because it is an evolution of the Inception Score and is a more effective metric for GAN evaluation. FID is also widely used across multiple papers that use GANs to summarise the model’s performance, so this would allow for easy comparisons to be made.

### Deployment of Fréchet Inception Score

Firstly, to calculate this metric I needed a dataset of generated images to be used alongside the real images. For this a pre-trained generator model was used that was saved from the last time I had trained my GAN. This had been run for 1,000 epochs on images of 28x28 resolution so this model was as well trained as it could be.

Diagram

Description automatically generated

Figure 32 FID Model Structure

The above diagram shows the structure of the machine learning model used to classify the fake and real images. The InceptionV3 model which is the final layer in the above diagram was trained with images of 299x299 resolution. Because of this, to classify images from my GAN they would first need to be resized to 299x299. This proved to not be the easiest thing to do as attempting to resize 10,000 images from 28x28 to 299x299 was causing Google Colab to run out of memory. Even when using the high RAM option in Google Colab the program still ran out of memory. This was a problem I had encountered previously in my machine learning project, so I knew how to solve it. I made use of a Lambda layer which allows you to insert a simple operation into a layer in the model. In this Lambda layer was a function call to resize the image. With this change the program no longer ran out of memory and could run through the model.

After adding the code to calculate the FID from the model’s output, I had an FID value for the DCGAN that generates 28x28 resolution images. This value was 0.002. An FID value of zero would signify that the generated images are exact copies of the real images, so I knew something was not correct.

Table

Description automatically generated

Figure 33 Fréchet Inception Distance and Perceptual Path Length Results from StyleGAN and StyleGAN2 on Four Datasets (Karras, et al. , 2019)

The above image shows the FID results from StyleGAN and StyleGAN2 on four different image datasets. The best FID value obtained was 2.32 which is over three orders of magnitude higher than the FID value for the images generated by my GAN. I then computed the FID for images from the CIFAR10 dataset (Keras, n.d.). This was to assess whether it was my code to calculate the FID that wasn’t working or if it was something else. I used 10,000 images from this dataset for the real images and another 10,000 for the fake images. With both these datasets using images from the same source the FID between them should be very low. Using my code an FID value of 5.348 was calculated. From this I knew that my code was working correctly, so the problem must be with my GAN. This was when I realised that the InceptionV3 model which is trained on the ImageNet image database is only trained on real things. Of the 1,000 image classes most of them are images of different objects and different animals. The images my GAN is trained to generate are images of kymographs which don’t resemble any of the image classes that InceptionV3 is trained on. Because of this the probability the InceptionV3 model gives any kymograph image, whether it is real or generated is almost zero for all 1,000 of the image classes. Therefore, the FID values for my GAN are so close to zero making it not ideal as a quantitative metric for assessing my GANs.

## Rationale for Chosen GAN Approaches

From my research into GANs, I found several different GAN architectures that could be used. By researching the original GAN paper (Goodfellow, et al. , 2014) I found a comparison between a GAN using fully connected layers and a GAN using convolutional layers, see Figure 8. This comparison showed that the GAN using convolutional layers could generate better quality images than the GAN using fully connected layers. This followed my previous experience of machine learning models which was that convolutional models generally performed better than fully connected models. However, convolutional models often contain more layers which are also more complex than the dense layers in a fully connected model. This leads to convolutional models often requiring more time to train than fully connected models for the same number of epochs. Due to this I have found from experience that it is quicker to deploy a fully connected model as opposed to a convolutional model. In addition, I did not have any previous experience using GANs so decided that developing a fully connected GAN first would provide valuable experience for then developing more complex GANs. I could then use the findings from developing this model to inform the development of the DCGAN which should produce higher quality images.

From researching the original DCGAN paper (Radford , Metz & Chintala, 2015) it was very likely better-quality images could be generated using this approach than with a fully connected GAN, see Figure 11. This would therefore be a logical improvement to make after obtaining results with the fully connected GAN.

I also conducted research into NVIDIA’s StyleGAN model (Karras , Laine & Aila, 2018) which can generate high detail images of up to 1024x1024 resolution. They have also released StyleGAN2 (Karras, et al. , 2019) which made some changes to the original model to improve the image quality and remove artefacts. StyleGAN is a more complex version of a convolutional GAN which is built to incorporate styles to inform the image generation process, see Figure 12. This architecture was not chosen due to the complexity of developing a style-based GAN from scratch and the time constraints of this project. However, this may be worth investigating in a follow-on project.

## Fully Connected GAN

### Strengths and limitations of this approach

This approach allowed me to develop a working GAN in a short amount of time and then quickly test improvements. It is easy to test the model on different image resolutions because the layers in the model remain the same for different resolutions and still produce images of reasonable quality. Because of this I can use my dataset of 28x28 resolution images and 72x72 images on the same model with minimal changes. I could test a feature with the lower resolution images and run the model quickly, and then run it on the higher resolution images with relatively few changes to the code needed. The limitations of this approach were that the images produced weren’t perfect quality and still contained quite a lot of noise even after many epochs.

### Design of the Fully Connected GAN

Diagram

Description automatically generated

Figure 34 Fully Connected GAN Discriminator Structure

The above image shows the structure of the final design for the fully connected GAN discriminator. This model used four dense fully connected layers, each separated by a LeakyReLU layer. Leaky Rectified Linear Unit, or LeakyReLU is a type of activation function. Activation functions determine whether a node in a layer should be activated or not. They are used to bind the value of the net input from one layer to the next. The ReLU activation function only applies to positive values, whereas the LeakyReLU activation function has a slight gradient for negative values. LeakyReLU is used in models where sparse gradients can be a problem which is common in GANs. These LeakyReLU layers all use an alpha value of 0.2, as recommended (Radford , Metz & Chintala, 2015) for DCGANs, but I found this worked well with fully connected GANs as well. The same paper also recommended using a learning rate for the Adam optimiser of 0.0002 and a β1 value of 0.5 which I found worked well with this model as well. The model made use of four dense fully connected layers, the first three of these were used for gradually reducing the number of nodes in the layers. The final layer was used for classification, it only has one node which represents whether the model thinks an image is fake or real. I found four dense layers was the best compromise between better quality images from using more layers and longer train times from using more layers.

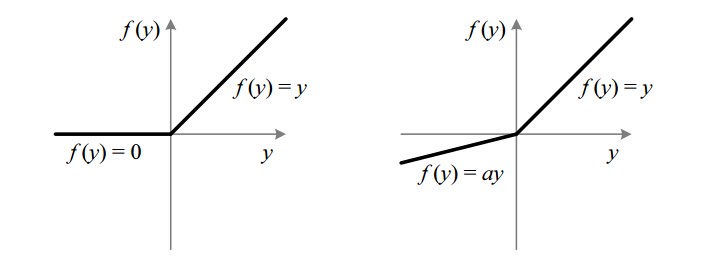


Figure 35 Comparison Between the ReLU Activation Function (left) and the LeakyReLU Activation Function (right)

Diagram

Description automatically generated

Figure 36 Fully Connected GAN Generator Structure

The above image shows the structure of the final design for the fully connected GAN generator. The layers for this model are largely the same as the discriminator model above but are in reverse, see Figure 34. This is because instead of down sampling an image until there is only a single node, we are up sampling from a random point in latent space. Latent space in this case is a 100-element vector of Gaussian random numbers. This model, like the fully connected GAN generator, uses LeakyReLU layers with alpha values of 0.2 as the activation layers.

### Creating the Generative Adversarial Network

The original design of this model only included one convolutional layer in the generator and discriminator models. This was to test that the GAN was functioning correctly and could at least replicate some characteristics of the images from the training dataset. With this very simple generator and discriminator architecture the images produced showed the black triangle in the bottom left corner which showed the GAN was functioning correctly. However, it struggled to generate the patterns in the top right of the kymograph images. For the majority of training the top right corner is mainly noise, but some images do show signs of forming the pattern lines.

Background pattern

Description automatically generated

Figure 37 Generated Images from Fully Connected GAN with One Dense Layer in Generator and Discriminator after 200 Epochs of Training

After the results were obtained for the simple fully connected GAN, it was clear that a fully connected GAN with a more complex architecture should be assessed to see if it could improve the image quality of the generated images. After some experimenting it was found that a generator and discriminator with four dense layers could produce better quality images, as shown in the figure below.



Figure 38 Generated Images from Fully Connected GAN with Four Dense Layers in Generator and Discriminator after 200 Epochs of Training

## Deep Convolutional GAN

### Strengths and limitations of this approach

This approach allowed for better quality images to be produced when compared to the images produced by the fully connected GAN. One downside of this model compared to the fully connected GAN is that the layers of the model need to be changed if different resolutions images need to be used. Because of this I needed to develop two separate DCGAN models to be able to use the 28x28 image resolution dataset and the 72x72 image resolution dataset. This was because the convolutional layers for down sampling the images in the discriminator and up sampling the images in the generator were resolution specific. So, the amount of these layers and the resolution of these layers will change depending on the resolution being used.

### Design of the DCGAN

Diagram, table

Description automatically generated

Figure 39 DCGAN Discriminator Structure

The above image shows the final design for the DCGAN discriminator model. This model makes use of three convolutional layers, the first of these layers is a normal convolutional layer with no stride so the image size is not changed. The second and third of these convolutional layers both have a stride of 2x2 which down samples the image size to have a height and width of half what it was in the previous layer. These convolutional layers down sample the image size from the starting 28x28 to 7x7 which is as low as you can go with convolutional layers with a stride of 2x2. The convolutional layers are each followed by a LeakyReLU activation layer with an alpha value of 0.2, which was used in the original DCGAN paper (Radford , Metz & Chintala, 2015). After the final convolutional layer there is a flatten layer which flattens the input into one channel so that the input shape is compatible with the final dense layer which only accepts inputs with one channel. This is then followed by a dropout layer. Dropout layers randomly set neurons to be ignored during training. This is done to reduce the chance of the model overfitting to the training data.

Diagram

Description automatically generated

Figure 40 Deep Convolutional GAN Generator Structure

The generator architecture in the above figure takes the 100-dimensional uniform noise distribution as the input for generating the images. This is then inputted into a dense layer with 12,544 nodes. 12,544 is a significant number as it represents the size of the starting image tensor. This tensor is constructed by reshaping the output of the dense layer into a 7x7x256 image tensor. 7x7 was chosen as the starting resolution for the image as it is the lowest whole number you can obtain from repeatedly halving the desired image resolution of 28x28. Unlike the DCGAN discriminator which uses Conv2D layers, this generator model uses Conv2DTranspose layers. These transposed convolution layers which are sometimes called deconvolution layers are used to perform a transformation going in the opposite direction of a normal convolution. Two Conv2DTranspose layers are used which up sample the image, first to 14x14 and then to 28x28. Each of these transposed convolution layers are followed by a LeakyReLU activation layer with an alpha value of 0.2. Finally, the DCGAN generator model uses a Conv2D layer as the output layer to resize the image tensor from 28x28x128 to 28x28x3. This Conv2D layer used the tanh activation function, so the pixel values outputted will be in the range of -1 to 1.

### Creating the Generative Adversarial Network

The development of my DCGAN was straightforward due to the experience I had gained creating the fully connected GAN and with information from the original DCGAN paper (Radford , Metz & Chintala, 2015). This paper made the following recommendations which I experimented with:

#### Use batch normalisation layers in the generator and discriminator.

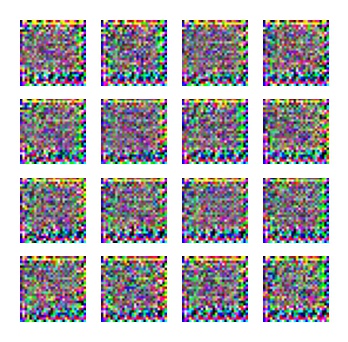


Figure 41 Generated Images from DCGAN Developed with Batch Normalisation Layers After Training for 10 Epochs

Background pattern

Description automatically generated with medium confidence

Figure 42 Generated Images from DCGAN Without Batch Normalisation Layers After Training for 10 Epochs

Figure 41 shows generated images very similar to the images generated when I had previously used batch normalisation in the fully connected GAN Figure 27. It was not clear why the generated images are this bad when using batch normalisation layers after each convolution. The reported accuracies when using batch normalisation layers are 0% for real and fake images. I can only think that this is because the discriminator model is not functioning properly, so does not make any prediction when given an image, which would give the accuracy of 0%. Figure 42 shows the DCGAN can generate authentic kymograph images without batch normalisation layers. Even after only ten epochs the generated images are beginning to resemble a kymograph image, because of this I chose to remove the batch normalisation layers.

#### Use the ReLU activation function for the generator and the LeakyReLU activation function for the discriminator.

Background pattern

Description automatically generated

Figure 43 Generated Images from DCGAN Using ReLU Activation Function in Generator After Training for 200 Epochs

A picture containing background pattern

Description automatically generated

Figure 44 Generated Images from DCGAN Using LeakyReLU Activation Function in Generator After Training for 200 Epochs

For my fully connected GAN I had used the LeakyReLU activation function for both the generator and the discriminator. Because of this I conducted an experiment to check which approach produced better quality images. From the above two figures I decided that using LeakyReLU for both models managed to generate the better-quality images, so I decided to stick with this approach from my fully connected GAN.

# Results and Evaluation

I have included some results in the previous section as this project was developed iteratively through the design and implementation phases. This section dives deeper into the final outputs from the project and evaluates their effectiveness at meeting the projects objectives.

## Challenges Evaluating Uncategorised Data

Evaluating the effectiveness of the GANs used proved to be challenging. One of the main reasons for this difficulty was that the training data being used was uncategorised. An example categorised image dataset would be the CIFAR10 image dataset (Keras, n.d.). This dataset includes 6,000 images from ten different image categories.

Chart, funnel chart, surface chart

Description automatically generated

Figure 45 Three Categories of Kymograph Waves (Feliú, et al. , 2020)

The above figure shows three different categories of waves that can be formed by the repressilator gene network. However, once the kymograph image datasets were produced, I could not see any way to separate the images into three distinct classes. To simulate the kymograph image datasets, I used the full ranges of both the alpha and degradation rate values that would still produce useful images. Because of this the images in the dataset would gradually transform as the parameters were changed, so I found that my image datasets did not include any distinct classes that could easily be separated from the rest of the data.

My original idea for assessing the accuracy of my GAN involved separating the kymograph images into these three distinct classes. I was then planning to develop a convolutional neural network (CNN) that could classify these images into one of the three categories. The certainty value from the CNN that a generated image belonged to a certain class would then be used to judge the quality of the image. But because the images could not be categorised into distinct classes other methods for GAN assessment would need to be used that did not rely on categorised data. Due to this I researched alternative GAN assessment. I investigated using the Fréchet inception distance metric which (Borji, 2018) was found to be “consistent with human judgments and more robust to noise” which looked to be the most suitable for my application.

## Testing & Validation Approach

The testing approach used was to first trial any changes to the GAN using the 28x28 low resolution images as this would be quicker to run than using the higher resolution images. The generated images from the GAN which are saved every ten epochs could then be manually inspected as the model continued to train. This helped reduce the amount of time that was spent waiting for the model to complete training and increased the number of models I could prototype. Manual inspection proved to be a very useful method for analysing the model’s performance during testing. From the generated images after only fifty epochs, I could tell by comparison whether a change had improved the generated images. Running fifty epochs only required ten minutes, and with some changes it was clear whether they had worked or not before this. For example, adding batch normalisation layers which broke the GAN, see Figure 27 and Figure 41. It was only after the GAN models were fully developed that I needed some way of validating the accuracy of the generated images.

To validate the effectiveness of my GANs I first created a plot that showed the discriminator training accuracy for both real and fake images during training. Another plot showed the discriminator loss values for both real and fake images during training.

Shape, arrow

Description automatically generated

Figure 46 Plot to Show Discriminator Accuracy Values Throughout Training for Real and Fake Data for the DCGAN for 28x28 Resolution Kymograph Images

The discriminator accuracy on fake images hardly changes during training, but the discriminator accuracy for real data increases steadily over time. So, you may expect the best quality images to be generated when the discriminator accuracy for real data stops increasing. This looks to happen at around epoch 800. If discriminator accuracy for real data was correlated with generated image quality, then you would expect the best quality images to be generated around epoch 800. This assumption does not fit well with manual inspection which found that the best quality images were generated around epoch 150 for the DCGAN models. After this image quality remained constant but image diversity got worse over time.

A picture containing shape

Description automatically generated

Figure 47 Plot to Show Discriminator Loss Values Throughout Training for Real and Fake Data for the DCGAN for 28x28 Resolution Kymograph Images

The loss plot shows a steady decline in the loss value during training which shows that the model is improving. But this does not necessarily correspond to the quality of the generated images. The improvements in loss value are very slight after 500 epochs and look to level off around 800 epochs. Through manual inspection I had discovered that the maximum image quality was reached after 150 epochs for the DCGAN models, and then after this the image diversity became less and less. This graph does not fit with those findings because if the loss value gradient was correlated with image quality, then the images would be improving in quality until at least epoch 800.

These plots proved to not be very useful for making meaningful observations about the quality and diversity of the generated images. As well as this these plots are a very crude metric of GAN performance, and they may not be fully reliable for inferring the quality of the generated images.

Because of this I decided to implement the Fréchet inception distance (FID) (Heusel, et al. , 2017) for each of the GAN models. This metric would compare the real images to the generated images and then summarise how far the distribution of generated images is from the distribution of real images with a single value. This metric worked well for assessing how each of my GAN models compared to each other. However, the FID could not be compared to distance values from other GAN applications. This was because a kymograph image does not resemble any image from any of the 1,000 datasets the InceptionV3 model is trained on, which is used when calculating FID. Because of this an incredibly low FID value is calculated for my GAN models which is multiple orders of magnitude lower than the values from other GAN applications.

## Overview of the Results Obtained

The following four figures show the resulting generated images from each of the four GANs developed after 1,000 epochs of training. 1,000 epochs of training were excessive as the generated images did not improve perceptibly to the human eye after a maximum of a few hundred epochs. However, the image quality also did not get worse with more training after the peak image quality was reached. So, each of the four models were trained for 1,000 epochs to give them more than enough training to produce the best quality images they are capable of.

Background pattern

Description automatically generated with medium confidence

Figure 48 Generated Images from fully connected GAN for 28x28 Resolution Kymograph Images Trained for 1,000 Epochs

Background pattern

Description automatically generated with medium confidence

Figure 49 Generated Images from DCGAN for 28x28 Resolution Kymograph Images Trained for 1,000 Epochs

For generating 28x28 resolution kymograph images it is clear to see that the DCGAN manages to generate better quality images. The kymograph images generated by the fully connected GAN still contain significant noise which harms the image quality. After 1,000 epochs of training the maximum generated image quality should have been reached, so this is likely a limitation of this GAN architecture. However, the images from the fully connected GAN do look to be more diverse than the images generated by the DCGAN. I think this happens because the fully connected GAN takes many more epochs to train fully than compared to the DCGAN. The images from the fully connected GAN are constantly improving for at least the first 800 epochs; after this it becomes harder to tell if the images are improving in quality. However, with the DCGAN the image quality of the generated images after 150 epochs is comparable to that of the generated images after 1,000 epochs. The DCGAN generated images from epoch 150 are also more diverse than the generated images epoch 1,000. This is likely because of mode collapse (Huang , Yu & Wang, 2018), so this extra training is unnecessary and harms the diversity of the generated images.

A picture containing background pattern

Description automatically generated

Figure 50 Generated Images from Fully Connected GAN for 72x72 Resolution Kymograph Images Trained 1,000 Epochs

Background pattern

Description automatically generated with medium confidence

Figure 51 Generated Images from DCGAN for 72x72 Resolution Kymograph Images Trained for 1,000 Epochs

Much like with generating 28x28 resolution kymograph images, the DCGAN can clearly generate the better-quality images at the higher 72x72 resolution as well. The generated images from the fully connected GAN again included noticeable image noise, but this is reduced when compared to the 28x28 resolution generated images from the fully connected GAN. With the larger resolution fully connected GAN generated images, the bottom left black corner is entirely black and does not include any image noise. Both image resolutions use the same layers for the fully connected GAN model, so either the larger images work better with this model or this difference is due to machine learning models being inherently nondeterministic. Much like with the lower resolution kymograph images the DCGAN can generate much better-quality images than the fully connected GAN, but these images lack diversity. Again, it was found through manual inspection that the fully connected GAN images required 800 epochs to reach maximum image quality, while the DCGAN only required 150 epochs. Mode collapse is to blame for the lack of diversity in the DCGAN images above as the model was overtrained for hundreds of epochs to produce these comparison images.

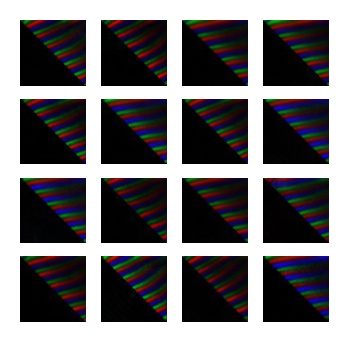


Figure 52 Generated Images from DCGAN for 72x72 Resolution Kymograph Images Trained for 150 Epochs

The above figure shows the clearest case of mode collapse when compared to Figure 51. The generated images from epoch 1,000 all look like the exact same image just with some being darker than others. This has happened because the generator has learnt that it achieves the best results when the same image is produced for the discriminator every time. Whereas with the generated images from epoch 150 there is clear diversity among the generated images, so these are a much more accurate representation of the training dataset.

## Evaluation of the Results Obtained

### Fully Connected GAN vs DCGAN

The two GAN models used both proved to be successful at generating kymograph images. The fully connected GANs could generate images that resembled kymographs, but these images contained a significant amount of image noise which made them poor representations of the training dataset. The DCGANs however proved capable at generating high-quality kymograph images that contained no visible image noise. The fully connected GANs were much simpler models than the DCGANs so the development and testing time required was significantly less. Because of this the fully connected GANs required less knowledge of GANs to deploy than the DCGANs, this was an advantage as I did not have any previous experience using GANs. However, the significantly better results obtained by the DCGAN made the extra research and development worthwhile.

There were other GAN approaches that were beyond the scope of this project but could be used in a future project. The most promising alternative solution would be to use NVIDIA’s StyleGAN2 model (Karras, et al. , 2019). StyleGAN2 can generate images up to 1,024x1,024 in resolution which is much larger than the image resolutions used in this project. It can be trained to generate any kind of images so could work well to generate much higher resolution kymograph images. The main reason this wasn’t chosen was that to generate the higher resolution images the train time required is extremely long. In the original paper proposing StyleGAN they say, “Our training time is approximately one week on an NVIDIA DGX-1 with 8 Tesla V100 GPUs”, so they are using eight GPUs that are each twice as powerful as the single NVIDIA Tesla P100 that Google Colab provides.

Another promising approach that could be used in a future project is the Fully Connected and Convolutional Net Architecture for GANs (FCC-GAN) (Barua , Erfani & Bailey, 2019). This GAN architecture makes use of both fully connected and convolutional layers in both the generator and discriminator models. This paper suggests that this approach can be more effective than a traditional DCGAN. It would be interesting to evaluate this approach against the DCGANs in a future project to see if better quality kymograph images could be generated.

### GAN Train Times

The time spent training varied for each GAN used. The following are the results for how long one epoch took for each of the GANs:

|  |  |
| --- | --- |
| **GAN Used** | **Time for One Epoch (Seconds)** |
| Fully Connected GAN for 28x28 Resolution Kymograph Images | 11 |
| DCGAN for 28x28 Resolution Kymograph Images | 7 |
| Fully Connected GAN for 72x72 Resolution Kymograph Images: | 14 |
| DCGAN for 72x72 Resolution Kymograph Images | 18 |

From the above results I found that train times were similar, and the gap between train times was four seconds for both image resolutions. What was unexpected however was that for the lower resolution images the DCGAN was faster per epoch, but the opposite was true for the higher resolution images. I am not exactly sure why this is, but it is most likely because the fully connected GAN models use the same layers for both image resolutions. This could mean that the layers for the fully connected GAN model for 72x72 resolution images were sub-optimal. The lower resolution fully connected GAN used 2,352 nodes to represent the image (28\*28\*3), whereas the higher resolution fully connected GAN used 15,552 nodes to represent the image (72\*72\*3). The highest number of units used in a dense layer in the two fully connected GANs is 1,024 which is much nearer to 2,352 than 15,552. More layers could have improved the performance of the higher resolution fully connected GAN. But since the two fully connected GANs were used as a learning experience for the DCGANs this was not a priority as it was known that better results could be obtained using the DCGAN approach.

From the train times per epoch, you would think that the fully connected GAN would be fastest for 28x28 resolution images, and the DCGAN would be fastest for the 72x72 resolution images. However, the time per epoch proved to be insignificant compared to the number of epochs the different GAN approaches required to reach maximum image quality. Through manual inspection I found that for both fully connected GANs the image quality was gradually improving for the first 800 epochs of training, after this it was more difficult to notice any improvement. Using the same method for the DCGANs I found that the image quality was improving for the first 150 epochs and then improvements became harder to notice. Using these epoch numbers, we can calculate the total train time to reach maximum image quality with each model:

|  |  |
| --- | --- |
| **GAN Used** | **Total Train Time** |
| Fully Connected GAN for 28x28 Resolution Kymograph Images | 2 Hours and 26 Minutes |
| DCGAN for 28x28 Resolution Kymograph Images | 17 Minutes |
| Fully Connected GAN for 72x72 Resolution Kymograph Images | 3 Hours and 6 Minutes |
| DCGAN for 72x72 Resolution Kymograph Images | 45 Minutes |

From these results the DCGAN has a massive advantage in terms of train time. This coupled with the fact that the DCGAN produced better quality images makes it the much better choice for this application.

### Time Required to Produce Kymograph Image

One of the main goals of this project was to assess whether a GAN could be a better method for producing more kymograph image data. The following table shows the time to produce 1,000 kymograph images for each of the GAN models, and then for the biological simulation which was the existing solution for this problem.

|  |  |
| --- | --- |
| **Approach Used** | **Time to Produce 1,000 Kymograph Images** |
| Fully Connected GAN for 28x28 Resolution Kymograph Images | 0.75 Seconds |
| DCGAN for 28x28 Resolution Kymograph Images | 10.34 Seconds |
| Fully Connected GAN for 72x72 Resolution Kymograph Images | 1.77 Seconds |
| DCGAN for 72x72 Resolution Kymograph Images | 31.30 Seconds |
| Simulating Biological Model for 28x28 Resolution Kymograph Images | 1 Hour and 24 Minutes |
| Simulating Biological Model for 72x72 Resolution Kymograph Images | 1 Hour and 27 Minutes |

These results show that all the GANs are more than two orders of magnitude faster than the biological simulation at producing 1,000 images. This is a very large advantage for the machine learning approach and would allow for many thousands of images to be generated relatively quickly. However, for the fully connected GANs and the DCGANs the percentage increase in time to produce 1,000 images from the lower resolution to the higher resolution images is much greater than the biological simulation. With the amount of time it takes to generate 1,000 images with any of the GANs this is insignificant. But if this is an exponential increase then there may be an image resolution where the biological model is faster at producing one image than any of the GANs.

The benefit in time to produce images would also need to be weighed against the time to design, develop, and train the GAN if a new image resolution was to be used. With the GANs developed it would be relatively easy to adapt one to generate images of higher resolutions. The most time-consuming part would be simulating the training dataset for this higher resolution GAN, which may need to be more than the 10,000 images used to train the GANs in this project. This is because for larger resolutions there is more for the generator to learn so larger datasets give the generator more data to learn from.

### Manual Inspection

The quantitative evaluation of GANs is still a current problem with the state of the technology today. Manual inspection was used extensively for assessing the effectiveness of the different GAN approaches. This was due to its easy deployment of this approach, but also because it is an effective method for obtaining a well-trained GAN.

The generator model was saved every ten epochs as well as sixteen kymograph images generated by that generator. This allowed me to easily assess which epoch had the best quality images been generated. The generator model could then easily be deployed to generate more of these images judged to be the best quality. This served as a suitable replacement for an objective measure that other non-GAN machine learning models would have. However, manual inspection is not a perfect solution as it comes with its own disadvantages. The main downside is that evaluating the quality of an image is subjective due to this method’s reliance on human assessors. This was mostly mitigated because I was the only person inspecting the generated images. However, there are still biases I will have an effect, but this would at least be consistent across all models because the same assessor is assessing all the images. With only one assessor the number of generated images that could be viewed was limited. For this reason, sixteen generated images were saved every ten epochs for each model which could be reviewed relatively quickly.

### Fréchet Inception Distance Scores

The following table shows the FID scores that were calculated from each of the GAN models. The FID scores are calculated for two different epochs for each model. The FID for epoch 1,000 is calculated for all GANs as it was the maximum epoch that was trained to. For the DCGANs the FID is calculated for epoch 150 as this was when it became difficult to notice an increase in image quality when manually inspecting generated images. For the same reason the FID for epoch 800 is calculated for the fully connected GANs.

|  |  |  |
| --- | --- | --- |
| **GAN Used** | **Training Epochs** | **Fréchet Inception Distance Score** |
| Fully Connected GAN for 28x28 Resolution Kymograph Images | 1,000 | 0.032 |
| Fully Connected GAN for 28x28 Resolution Kymograph Images | 800 | 0.027 |
| DCGAN for 28x28 Resolution Kymograph Images | 1,000 | 0.002 |
| DCGAN for 28x28 Resolution Kymograph Images | 150 | 0.001 |
| Fully Connected GAN for 72x72 Resolution Kymograph Images | 1,000 | 0.105 |
| Fully Connected GAN for 72x72 Resolution Kymograph Images | 800 | 0.166 |
| DCGAN for 72x72 Resolution Kymograph Images | 1,000 | 0.035 |
| DCGAN for 72x72 Resolution Kymograph Images | 150 | 0.011 |

The FID scores calculated did not fit with the scores I had seen in multiple research papers where the FID scores were never lower than single digits numbers. The FID scores obtained for the GANs used were however much lower than single digit numbers. This was found to be because the image dataset used did not match with any of the 1,000 image classes that the InceptionV3 pre-trained classifier is trained on. This is the model used to calculate the FID hence the low scores. Unfortunately, due to how low the FID values were, comparisons with other GAN applications could not effectively be made.

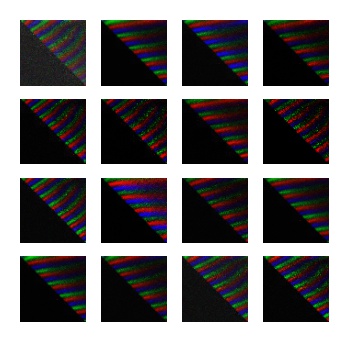


Figure 53 Generated Images from Fully Connected GAN for 72x72 Resolution Kymograph Images Trained 800 Epochs

However, the FID was found to match the manual inspection findings for all but one result. The one result where there was a discrepancy was for the fully connected GAN for 72x72 resolution images. From manual inspection the best quality images were judged to be at epoch 800. However, the FID value for this model for epoch 800 was higher than for epoch 1,000 meaning that the images were of worse quality. The generated images from training were then reviewed again and it was determined that the images from epoch 1,000 (Figure 50) were indeed slightly better quality than those from epoch 800 shown in the above figure. However, this difference was difficult to see which demonstrates why manual inspection is not a perfect solution to GAN evaluation due to the subjective nature of it.

I found that the FID values correlated well with image quality. For both image resolutions the DCGAN had the lower inception score by a large margin. This corresponded well with manual inspection so I knew this was a quantitative metric I could use to compare the GANs with each other.

The FID measures the quality and diversity of a set of generated images. Through manual inspection I had found that for the fully connected GAN the generated images at epoch 800 were of the same quality as those from epoch 1,000 but were more diverse. The same was found for the DCGAN but at epoch 150 instead of 800. Because of this I expected a lower FID score to be calculated for these lower epochs, and this was backed up by the results which showed that this was correct. This proved that the FID scores correlated well with image quality and diversity for the GANs used.

## Evaluation of Project Approach

### Project Plan

The project plan (Figure 5) originally involved developing the GAN generator and then the GAN discriminator. This was found to be an incorrect method for developing the GAN as it would be difficult to test either model until they were linked together to complete the GAN. Because of this the approach was changed to first develop a simple GAN that could be used to perform testing which would then inform the development of a more complex GAN architecture. For this same reason the testing approach was changed as well to test the completed GAN model rather than the individual generator and discriminator models.

### Python

Python was chosen as the programming language to use. On reflection I think this was the best choice due to the previous experience I had with this language and the machine learning library Keras that can be used in Python. With Python being one of the most popular languages for data science and machine learning (see Figure 16) there were a wide array of online resources related to this project.

### Keras

Keras was chosen because I lacked previous experience with GANs so decided a machine learning framework that I was familiar with would help when learning how to develop a GAN. Keras proved to be a good choice for this because a lot of the extra control over the model that you get with lower-level machine learning frameworks like Tensorflow and PyTorch was not needed. Because of the lower-level nature of Tensorflow and PyTorch optimising models can take longer due to the tweaking of more attributes than with Keras. This would have been difficult to accomplish due to my initial lack of GAN knowledge.

### Google Colab

Google Colab (Google, 2022) was the chosen platform for the development. Google Colab worked well as it allowed me to make use of a powerful GPU wherever I happened to be. This made development possible when I was at home using my laptop which has no dedicated GPU. Without the use of Google Colab, it would have been unfeasible to train GANs on my laptop due to the very long train times. Google Colab also allowed me to store the code and training images on Google Drive which was more secure than any local storage method and allowed for the seamless switching between devices which helped to increase my productivity.

### Scrum

The software development methodology Scrum (Scrum.org, 2022) was used for this project. For this each stage of the project was split into sprints that could be completed in a week or two. Overall, this methodology worked well. Using sprints with early delivery of partially working systems allowed identification of changes needed to the aims of the project once a deeper understanding of GANs was gained. This project included delays which affected the development due to there being very little slack in-between each of the sprints. However, this was mainly due to the ambitiousness of this project rather than the software development methodology chosen. The Scrum software development methodology benefits from customer feedback from one sprint to inform the development of the next. As I was a one-man team this aspect of Scrum was not fully utilised.

# Conclusion

## The extent to which the original Aim & Objectives were met

### Aim

The aim of this project was to develop a GAN that could generate kymograph images similar in quality to those simulated by a biological model. This aim was met because the DCGAN models developed can generate high quality kymograph images. This is however subjective as assessing the image quality of generated images proved to be challenging. However, I think the images generated are very high quality when compared to simulated kymograph images of the same resolution.

This differs from the aim of this project at the proposal stage, which was to develop a generative adversarial network to generate kymograph images based on an input of genetic parameters. This was found to not be an appropriate aim because the generator models in GANs rely on random noise as the input. For this to work a method to alter the generator to be compatible with the genetic parameter values would be required, but no such method could be found. If a solution was developed to fix this problem, then this would be a new GAN architecture that.

### Objectives

1. **Simulate the repressilator gene network to produce the required training data for the GAN.**

For this objective the repressilator gene network was first simulated using LOICA and a kymograph was produced. Code was then added to automate the process of simulating the kymograph many times but with different genetic parameters each time to produce a different image. Using this code multiple different kymograph image datasets were then produced for training. This objective was met fully by the final project.

1. **Process the training data to prepare it for the GAN.**

This objective involved processing and then normalising the kymograph image datasets. The image datasets were all transformed from folders of thousands of images to one file each which held all the image data and could be loaded much faster. Normalisation was the next step which changed the image data to be compatible with the GAN. This objective was met fully by the final project.

1. **Develop the GAN.**

For this objective four GANs were developed instead of just one. The development of the fully connected GANs helped inform the development of the DCGANs which proved to be far more effective. This objective was met fully by the final project.

1. **Test and train GAN.**

This objective required the testing and training of the four separate GANs that had been developed. The testing and training of the two fully connected GANs was very similar with the only difference being the training dataset that was used. The same is true for the two DCGAN models. This objective was met fully by the final project.

1. **Quantitative evaluation of the quality of the generated images.**

For this objective the Fréchet inception distance was chosen as the quantitative metric to use to assess the GANs developed. This metric was deployed successfully, but it was found that this project was an imperfect application for this metric. The FID was useful for its intended purpose of comparing the different GANs, but due to this limitation comparisons to GANs from various research papers could not be made. This objective was met but the solution could be improved.

## Summary of what has been learned

One of the main learning experiences from this project was how much training data was required to train the GANs. After producing smaller datasets, I found that 10,000 kymograph images worked well with the GAN models. A larger dataset of 65,536 images was produced and trialled with the GAN but no noticeable improvement could be observed. The image datasets used in this project contained quite low-resolution images, for generating higher resolution images it is likely that more than 10,000 training images would be required.

The advantages and disadvantages of a convolutional model over a fully connected model was also a learning point of this project. The convolutional approach produced much better results, but this came at the cost of model complexity and increased development time compared when compared to the fully connected approach.

Evaluation of the GAN models was problematic due to the uniqueness of kymograph images when compared to other popular machine learning image datasets. Throughout training, manual inspection was used to evaluate the GAN models which worked successfully. The problems were with calculating a quantitative metric for assessing the GANs. The main learning point from this was that GANs are still a very new concept, and the quantitative evaluation methods are imperfect due to them still being developed.

## Personal Development

The development of this project has allowed me to gain the knowledge and skills from researching and developing different image generation techniques using machine learning. My previous experience with machine learning was only with image classification models, which were all much simpler models than GANs. These could also easily be assessed by the accuracy value from the classification model. Due to this my knowledge of metrics to assess machine learning models other than the accuracy and loss values provided was limited. Developing multiple GAN models not only required a greater understanding of how the model worked, but also an understanding of how to assess the results from the GANs.

My ability to research complex topics has also improved, researching GANs was challenging because of how new this machine learning architecture is. There are multiple different architectural versions of the GAN all with their own advantages and disadvantages which need to be assessed to pick the correct approach. For this project numerous research papers needed to be reviewed so that useful findings from those papers could be used to improve the effectiveness of the GANs in this project. I had not needed to make use of research papers for my past projects as publicly available websites and tutorials had sufficed. However, this project involved a much greater amount of research. For this research papers needed to be reviewed to determine how useful they would be as a source of information.

## Areas for Future Work

The GAN models developed met the project aim as well as all the objectives. However, there are some things that could have been done better that would help achieve better results. A dataset of higher resolution kymograph images could be produced and then tested with both GAN architectures used. Higher resolution images will be able to include more details from the image, so the generated images should be more representative of the actual kymograph images. This would require new GAN models to be created to work with the higher resolution and the train times would be higher than with lower resolution images.

Another area of improvement would be to explore alternatives to the two GAN architectures used. The most promising option is NVIDIA’s StyleGAN2 (Karras, et al. , 2019) which can generate very high-quality images of up to 1,024x1,024 resolution. This method requires a long time to train the model and many images to train it but should be able to produce high-quality high-resolution images given enough training.

Another possible machine learning architecture that could be used to produce the kymograph images would be an autoencoder (GoodFellow , Bengio & Courville, 2016, pp. 502-525). An autoencoder is a neural network that consists of two different parts: an encoder and a decoder. The encoder uses the image to produce a code that represents it. The decoder then takes this code and reconstructs the image. The input image and output image are compared and if the model is working perfectly then the output image will be an exact copy of the input image. If the decoder is trained sufficiently then it can be separated from the encoder and generated codes could be fed into it to generate images.

One of the main areas of this project that was not as effective as it could be was the quantitative metric that was implemented to evaluate the effectiveness of the GANs. The FID values that were calculated for the GANs were found to be very low when compared to other papers using GANs so no meaningful comparisons could be made. Because of this another quantitative metric should be used that would allow for more meaningful comparisons to be made.

References

Barua, S. Erfani, S. M. & Bailey, J. (2019) *FCC-GAN: A Fully Connected and Convolutional Net* Available at : https://arxiv.org/abs/1905.02417 (Accessed : 23/04/2022).

Borji, A. (2018) *Pros and Cons of GAN Evaluation Measures* Available at : https://arxiv.org/abs/1802.03446 (Accessed : 25/04/2022).

Chollet, F. (2018) *Deep Learning with Python* 1st ed. Shelter Island: Manning.

FeliúG. Y. et al.(2021) *Flapjack: Data Management and Analysis for Genetic Circuit Characterization* Available at : https://doi.org/10.1021/acssynbio.0c00554 (Accessed : 25/03/2022).

Feliú, G. Y.Vidal, G. Silva, M. M. & Rudge, T. J.(2020) *Novel Tunable Spatio-Temporal Patterns From a Simple Genetic Oscillator Circuit* Available at : https://doi.org/10.3389/fbioe.2020.00893 (Accessed : 05/03/2022).

GoodFellow, I. Bengio, Y. & Courville, A. (2016) *Deep Learning* 1st ed. London: MIT Press.

Goodfellow, I. J.Pouget-Abadie, J. Mirza, M. Xu, B. Warde-Farley, D. Ozair, S. Courville, A. & Bengio, Y. (2014) *Generative Adversarial Networks* Available at : https://arxiv.org/abs/1406.2661 (Accessed : 28/02/2022).

Google, (2022) *Google Colab* Available at : https://colab.research.google.com/?utm\_source=scs-index (Accessed : 21/04/2022).

Hayes, B. (2020) *Usage of Programming Languages by Data Scientists: Python Grows while R Weakens* Available at : http://businessoverbroadway.com/2020/06/29/usage-of-programming-languages-by-data-scientists-python-grows-while-r-weakens/ (Accessed : 28/04/2022).

Heusel, M. Ramsauer, H. Unterthiner, T. Nessler, B. & Hochreiter, S. (2017) *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium* Available at : https://arxiv.org/abs/1706.08500 (Accessed : 29/04/2022).

Hope, T. Resheff, Y. S. & Lieder, I. (2017) Chapter 4. Convolutional Neural Networks In: *Learning TensorFlow* s.l.:O'Reilly Media, Inc., p. Chapter 4.

Huang, H. Yu, P. S. & Wang, C. (2018) *An Introduction to Image Synthesis with Generative Adversarial Nets* Available at : https://arxiv.org/abs/1803.04469 (Accessed : 11/04/2022).

Karras, T. Aila, T. Laine, S. & Lehtinen, J. (2017) *Progressive Growing of GANs for Improved Quality, Stability, and Variation* Available at : https://arxiv.org/abs/1710.10196 (Accessed : 19/03/2022).

Karras, T. Aittala, M. Hellsten, J. Laine, S. Lehtinen, J. & Aila, T. (2020) *Training Generative Adversarial Networks with Limited Data* Available at : https://arxiv.org/abs/2006.06676 (Accessed : 12/04/2022).

Karras, T. Laine, S. & Aila, T. (2018) *A Style-Based Generator Architecture for Generative Adversarial Networks* Available at : https://arxiv.org/abs/1812.04948 (Accessed : 17/04/2022).

Karras, T. Laine, S. Aittala, M. Hellsten, J. Lehtinen, J. & Aila, T. (2019) *Analyzing and Improving the Image Quality of StyleGAN* Available at : https://arxiv.org/abs/1912.04958 (Accessed : 15/04/2022).

Keras, (n.d.) *CIFAR10 small images classification dataset* Available at : https://keras.io/api/datasets/cifar10/ (Accessed : 02/05/2022).

Kynkäänniemi, T. Karras, T. Laine, S. Lehtinen, J. & Aila, T. (2019) *Improved Precision and Recall Metric for Assessing Generative Models* Available at : https://arxiv.org/abs/1904.06991 (Accessed : 23/04/2022).

Langr, J. & Bok, V. (2019) *GANs in Action* 1st ed. Shelter Island: Manning Publications.

McLaughlinJ. A. et al.(2020) *The Synthetic Biology Open Language (SBOL) Version 3: Simplified Data Exchange for Bioengineering* Available at : https://doi.org/10.3389/fbioe.2020.01009 (Accessed : 21/03/2022).

Nistrup, P. (2019) *What is .npy files and why you should use them…* Available at : https://towardsdatascience.com/what-is-npy-files-and-why-you-should-use-them-603373c78883 (Accessed : 03/04/2022).

Radford, A. Metz, L. & Chintala, S. (2015) *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* Available at : https://arxiv.org/abs/1511.06434v2 (Accessed : 13/03/2022).

Salimans, T. Goodfellow, I. Zaremba, W. Cheung, V. Radford, A. & Chen, X. (2016) *Improved Techniques for Training GANs* Available at : https://arxiv.org/abs/1606.03498 (Accessed : 20/04/2022).

Scrum.org, (2022) *A Better Way Of Building Products* Available at : https://www.scrum.org/resources/what-is-scrum (Accessed : 18/03/2022).

Terra, J. (2022) *Keras vs Tensorflow vs Pytorch: Key Differences Among the Deep Learning Framework* Available at : https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article (Accessed : 09/04/2022).

Vidal, G. Vidal-Céspedes, C. & Rudge, T. J.(2021) *LOICA: Logical Operators for Integrated Cell Algorithms* Available at : https://doi.org/10.1101/2021.09.21.460548 (Accessed : 05/03/2022).

Wang, P. (2019) *This Person Does Not Exist* Available at : https://thispersondoesnotexist.com/ (Accessed : 14/04/2022).

Wyss Institute for Biologically Inspired Engineering at Harvard, (2019) *Oscillating Genetic Circuit: A Reliable Clock for Your Microbiome* Available at : https://scitechdaily.com/oscillating-genetic-circuit-a-reliable-clock-for-your-microbiome/ (Accessed : 16/03/2022).

Appendix A – Simulated Kymograph Images

Below are representative images from the LOICA simulator:

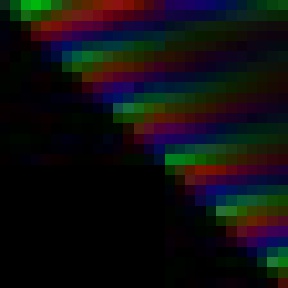
****

Figure 54 Example 28x28 Resolution Kymograph Image Simulated Using Alpha=350000 and Degradation Rate=0.22

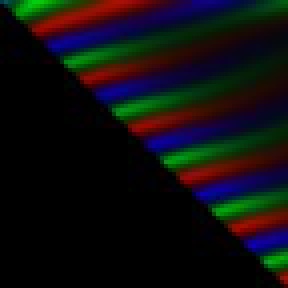


Figure 55 Example 72x72 Resolution Kymograph Image Simulated Using Alpha=350000 and Degradation Rate=0.22

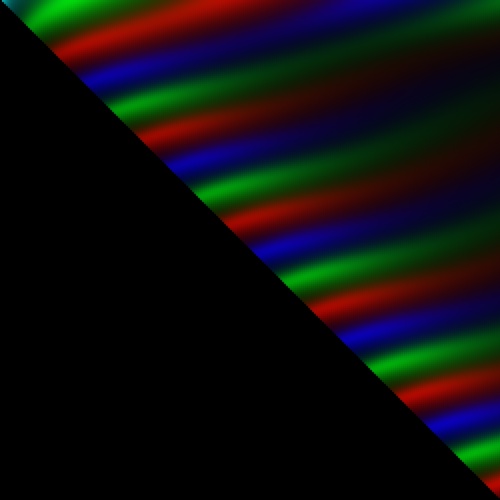


Figure 56 Example 500x500 Resolution Kymograph Image Simulated Using Alpha=350000 and Degradation Rate=0.22

Appendix B – Generated Kymograph Images

Below are representative images from the GANs created:

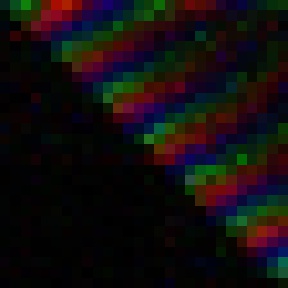
****

Figure 57 Example Random 28x28 Resolution Kymograph Image Generated Using Fully Connected GAN



Figure 58 Example Random 28x28 Resolution Kymograph Image Generated Using DCGAN

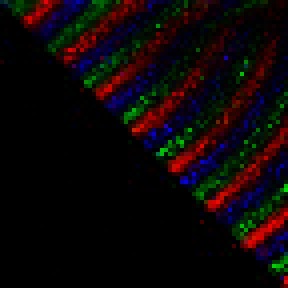


Figure 59 Example Random 72x72 Resolution Kymograph Image Generated Using Fully Connected GAN

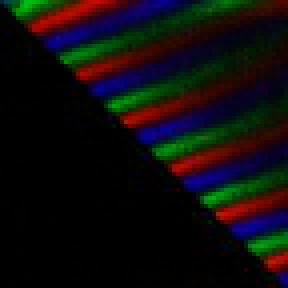


Figure 60 Example Random 72x72 Resolution Kymograph Image Generated Using DCGAN

Appendix C – Source Code for Simulating Kymograph Images

!pip install loica

from loica import \*

import matplotlib.pyplot as plt

import numpy as np

import time

from google.colab import drive

drive.mount('/content/drive')

# Simulate repressilator gene network

def gen\_image(genetic\_parameters, count):

N = 3

ring = GeneticNetwork()

regs = [Regulator(name=f'Rep{i}', degradation\_rate=genetic\_parameters[i][0]) for i in range(N)]

ring.add\_regulator(regs)

reps = [Reporter(name=f'SFP{i}', color='blue', degradation\_rate=genetic\_parameters[i][0]) for i in range(N)]

ring.add\_reporter(reps)

nots = [Hill1(input=regs[i], output=regs[(i+1)%N], alpha=genetic\_parameters[i][1], K=1, n=2) for i in range(N)]

ring.add\_operator(nots)

nots = [Hill1(input=regs[i], output=reps[i], alpha=genetic\_parameters[i][1], K=1, n=2) for i in range(N)]

ring.add\_operator(nots)

def growth\_rate(t):

return 1

def biomass(t):

return 1

metab = SimulatedMetabolism("name", biomass, growth\_rate)

regs[0].init\_concentration = 5

reps[0].init\_concentration = 5

col = Colony(ring, 1, 1)

kymo = col.kymograph(nx=250, t0=0, tmax=48)

# Figsize controls simulated image resolution with 1, 1 being 72x72

fig, ax = plt.subplots(figsize=(1, 1))

# Configure and save plot

plt.imshow(col.map\_kymo(col.norm\_kymo(kymo)), aspect='auto')

ax.set\_position([0, 0, 1, 1])

plt.axis("off")

plt.savefig("drive/MyDrive/Colab Notebooks/Raw Data 10000x72/image" + str(count) + ".jpg")

plt.close()

# Number of images produced = (n+1)^2

n = 99

alpha\_max = 1000000

alpha\_min = 1000

deg\_max = 1

deg\_min = 1e-6

deg\_step = (deg\_max - deg\_min) / n

alpha\_step = (alpha\_max - alpha\_min) / n

genetic\_parameters = []

# Construct genetic parameters array

for i in range(n + 1):

for j in range(n + 1):

deg\_val = deg\_min + (i \* deg\_step)

alpha\_val = alpha\_min + (j \* alpha\_step)

genetic\_parameters.append([deg\_val, alpha\_val])

j += 1

i += 1

genetic\_parameters = np.asarray(genetic\_parameters)

np.save("drive/MyDrive/Colab Notebooks/Raw Data 10000x72/genetic\_parameters.npy", genetic\_parameters)

# Image range to produce, distance between min and max > 250 can cause memory error

min = 0

max = 250

# Simulate kymograph images

start\_time = time.perf\_counter()

for i in range(genetic\_parameters.shape[0]):

if min <= i <= max:

gen\_image([[genetic\_parameters[i][0], [genetic\_parameters[i][1], 0]]]\*3, i)

print(i, genetic\_parameters[i][0], genetic\_parameters[i][1])

elif i > max:

break

end\_time = time.perf\_counter()

print(f"Execution Time : {end\_time - start\_time:0.6f}")

Appendix D – Source Code to Transform Kymograph Images into NumPy Array

import numpy as np

from PIL import Image

import matplotlib.pyplot as plt

import random

from google.colab import drive

drive.mount('/content/drive')

# Define empty arrays

train\_x = np.empty((10000, 72, 72, 3))

train\_y = np.empty((10000, 2))

# Array of random indexes to shuffle simulated images

random\_order = []

for i in range(10000):

random\_order.append(i)

random.shuffle(random\_order)

# Open all simulated images and add to numpy array

def img\_to\_array(array, indexes):

i = 0

j = 0

for index in indexes:

img = Image.open("drive/MyDrive/Colab Notebooks/Raw Data 10000x72/image" + str(index) + ".jpg")

array[i] = np.asarray(img)

i += 1

percentage = (i / array.shape[0]) \* 100

if (percentage > j):

print(str(j) + "% complete.")

j += 1

return array

# Shuffle and save genetic parameter array

def construct\_genetic\_parameters\_array(array, indexes):

genetic\_parameters = np.load("drive/MyDrive/Colab Notebooks/Raw Data 10000x72/genetic\_parameters.npy", allow\_pickle=True)

i = 0

for index in indexes:

array[i] = genetic\_parameters[index]

i += 1

return array

# Construct arrays

train\_x = img\_to\_array(train\_x, random\_order[:train\_x.shape[0]])

train\_y = construct\_genetic\_parameters\_array(train\_y, random\_order[:train\_y.shape[0]])

# Save numpy arrays

np.save("drive/MyDrive/Colab Notebooks/train\_x10000x72.npy", train\_x)

np.save("drive/MyDrive/Colab Notebooks/train\_y10000x72.npy", train\_y)

Appendix E – Source Code for Fully Connected GAN for 28x28 Resolution Images

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

from tqdm import tqdm

from keras.datasets import mnist

from keras.layers import Dense, Flatten, Reshape, Conv2D, Conv2DTranspose, Dropout, Reshape

from keras.layers.advanced\_activations import LeakyReLU

from keras.models import Sequential

from tensorflow.keras.optimizers import Adam

from keras.utils.vis\_utils import plot\_model

from google.colab import drive

drive.mount('/content/drive')

path = '/content/drive/MyDrive/Colab Notebooks/'

# Building the discriminator model

def build\_discriminator(img\_shape=(28, 28, 3)):

model = Sequential()

model.add(Dense(1024, input\_dim=28\*28\*3))

model.add(LeakyReLU(0.2))

model.add(Dense(512))

model.add(LeakyReLU(0.2))

model.add(Dense(256))

model.add(LeakyReLU(0.2))

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

model.compile(loss='binary\_crossentropy', optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5), metrics=['accuracy'])

return model

# Build, compile and display a summary of the discriminator

discriminator = build\_discriminator()

plot\_model(discriminator, show\_shapes=True, show\_layer\_names=True)

# Building the generator model

def build\_generator(latent\_dim, img\_shape=(28, 28, 3)):

model = Sequential()

model.add(Dense(256, input\_dim=latent\_dim))

model.add(LeakyReLU(0.2))

model.add(Dense(512))

model.add(LeakyReLU(0.2))

model.add(Dense(1024))

model.add(LeakyReLU(0.2))

model.add(Dense(28\*28\*3, activation='tanh'))

return model

# Size of the latent space

latent\_dim = 100

# Build and display a summary of the discriminator

generator = build\_generator(latent\_dim)

plot\_model(generator, show\_shapes=True, show\_layer\_names=True)

# Building the GAN model which combines the discriminator and generator models

def define\_gan(generator, discriminator):

# Set the weights in the discriminator to not trainable

discriminator.trainable = False

model = Sequential()

model.add(generator)

model.add(discriminator)

# Compile combined GAN model

model.compile(loss='binary\_crossentropy',

optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5))

return model

# Build and display a summary of the GAN

gan\_model = define\_gan(generator, discriminator)

gan\_model.summary()

# Load and return the training data

def load\_training\_data():

train\_x = np.load(path + 'train\_x10000x28.npy', allow\_pickle=True)

# Converting image data from doubles to floats

train\_x = train\_x.astype('float32')

# Scale from [0, 255] to [-1, 1]

train\_x = (train\_x - 127.5) / 127.5

return train\_x

# Select random images from training data

def select\_real\_samples(train\_x, n\_samples):

# Choose n\_samples amount of random indexes

random\_indexes = np.random.randint(0, train\_x.shape[0], n\_samples)

# Select the images from the training data

real\_samples\_x = train\_x[random\_indexes]

# Label '1' for each image to signify that it is real

real\_samples\_y = np.ones((n\_samples, 1))

return real\_samples\_x, real\_samples\_y

# Generate random points in the latent space

def generate\_latent\_points(latent\_dim, n\_samples):

# Choose n\_samples amount of random points in the latent space

fake\_x\_input = np.random.randn(latent\_dim \* n\_samples)

# Reshape into batches of inputs for the model

fake\_x\_input = fake\_x\_input.reshape(n\_samples, latent\_dim)

return fake\_x\_input

# Generate fake images using random points in the latent space

def generate\_fake\_samples(generator, latent\_dim, n\_samples):

# Generate random points in the latent space

fake\_x\_input = generate\_latent\_points(latent\_dim, n\_samples)

# Generate fake images

fake\_x = generator.predict(fake\_x\_input)

# Label '0' for each image to signify that it is fake

fake\_y = np.zeros((n\_samples, 1))

return fake\_x, fake\_y

# Display and save a plot of n^2 generated images

def plot\_gen\_imgs(generated\_imgs, epoch, n=4):

# Scale from [-1, 1] to [0, 1] for displaying images

generated\_imgs = (generated\_imgs + 1) / 2.0

# Plot generated images in a grid

fig, ax = plt.subplots(n, n, figsize=(2, 2))

count = 0

for i in range(n):

for j in range(n):

ax[i, j].imshow(generated\_imgs[count])

ax[i, j].axis('off')

count += 1

# Save the plot of generated images

plt.savefig(path + '/FCGAN for 28x28 Output/generated\_images\_epoch\_%d.jpg' %

(epoch+1), bbox\_inches='tight', dpi = 200)

plt.close()

# Save the generator model

def save\_generator(generator, epoch):

generator.save(path + '/FCGAN for 28x28 Output/generator\_model\_epoch\_%d.h5' %

(epoch+1))

losses = []

accuracies = []

epochs = []

# Periodic evaluation of the model

def evaluate\_gan(epoch, generator, discriminator, train\_x, latent\_dim, n\_samples=150):

# Evaluate discriminator on real images

real\_x, real\_y = select\_real\_samples(train\_x, n\_samples)

real\_x = real\_x.reshape(n\_samples, 2352)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

# Evaluate discriminator on fake images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, n\_samples)

fake\_x = fake\_x.reshape(n\_samples, 2352)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

print('\nDiscriminator Accuracy (Real Data): %.0f%%, Discriminator Accuracy (Fake Data): %.0f%%\n' %

(d\_acc\_real\*100, d\_acc\_fake\*100))

fake\_x = fake\_x.reshape(n\_samples, 28, 28, 3)

plot\_gen\_imgs(fake\_x, epoch)

save\_generator(generator, epoch)

# Display and save plots to show discriminator loss and accuracy throughout training

def plot\_loss\_and\_accuracy(losses, accuracies, epochs):

losses = np.array(losses)

accuracies = np.array(accuracies)

epochs = np.array(epochs)

# Plot training losses for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, losses.T[0], label='Discriminator Loss (Real Data)')

plt.plot(epochs, losses.T[1], label='Discriminator Loss (Fake Data)')

# Calculate line of best fit

a0, a1 = np.polyfit(epochs, losses.T[0], 1)

b0, b1 = np.polyfit(epochs, losses.T[1], 1)

# Plot line of best fit

plt.plot(epochs, a0\*epochs + a1, color='blue', linestyle='dashed',

label='Line of best fit Discriminator Loss (Real Data)')

plt.plot(epochs, b0\*epochs + b1, color='red', linestyle='dashed',

label='Line of best fit Discriminator Loss (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss Values')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim()

plt.legend()

plt.savefig(path + '/FCGAN for 28x28 Output/discriminator\_loss\_plot.jpg',

bbox\_inches='tight')

# Plot training accuracies for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, accuracies.T[0], label='Discriminator Accuracy (Real Data)')

plt.plot(epochs, accuracies.T[1], label='Discriminator Accuracy (Fake Data)')

# Calculate line of best fit

d0, d1 = np.polyfit(epochs, accuracies.T[0], 1)

e0, e1 = np.polyfit(epochs, accuracies.T[1], 1)

# Plot line of best fit

plt.plot(epochs, d0\*epochs + d1, color='blue', linestyle='dashed',

label='Line of best fit (Real Data)')

plt.plot(epochs, e0\*epochs + e1, color='red', linestyle='dashed',

label='Line of best fit (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy Values (%)')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim([0, 110])

plt.legend()

plt.savefig(path + '/FCGAN for 28x28 Output/discriminator\_accuracy\_plot.jpg',

bbox\_inches='tight')

# Train GAN

def train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim, number\_of\_epochs=1000, batch\_size=128):

batches\_per\_epoch = int(train\_x.shape[0] / batch\_size)

# Half of batch for real images and half for fake images

half\_batch\_size = int(batch\_size / 2)

for i in range(number\_of\_epochs):

# Loading bar for each batch in an epoch

for j in tqdm(range(batches\_per\_epoch), desc='Epoch %d/%d' % (i+1, number\_of\_epochs)):

# Select half\_batch\_size amount of random real samples from training data

real\_x, real\_y = select\_real\_samples(train\_x, half\_batch\_size)

# Train discriminator on real images

real\_x = real\_x.reshape(64, 2352)

d\_loss\_real, \_ = discriminator.train\_on\_batch(real\_x, real\_y)

# Select half\_batch\_size amount of random fake samples from generated images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, half\_batch\_size)

# Train discriminator on fake images

fake\_x = fake\_x.reshape(64, 2352)

d\_loss\_fake, \_ = discriminator.train\_on\_batch(fake\_x, fake\_y)

# Select batch\_size amount of random points in the latent space

gan\_x = generate\_latent\_points(latent\_dim, batch\_size)

# Inverted label for each image

gan\_y = np.ones((batch\_size, 1))

# Update the generator via the discriminators erorr

g\_loss = gan\_model.train\_on\_batch(gan\_x, gan\_y)

print('Discriminator Loss (Real Data): %.3f, Discriminator Loss (Fake Data): %.3f, Generator Loss: %.3f' %

(d\_loss\_real, d\_loss\_fake, g\_loss))

# Add data from each epoch to these arrays for plotting

real\_x, real\_y = select\_real\_samples(train\_x, 150)

real\_x = real\_x.reshape(150, 2352)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, 150)

fake\_x = fake\_x.reshape(150, 2352)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

losses.append((d\_loss\_real, d\_loss\_fake))

accuracies.append((100.0 \* d\_acc\_real, 100.0 \* d\_acc\_fake))

epochs.append(i + 1)

# Evaluate GAN every 10 epochs

if (i + 1) % 10 == 0:

save\_generator(generator, i)

evaluate\_gan(i, generator, discriminator, train\_x, latent\_dim)

train\_x = load\_training\_data()

train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim)

plot\_loss\_and\_accuracy(losses, accuracies, epochs)

Appendix F – Source Code for Fully Connected GAN for 72x72 Resolution Images

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

from tqdm import tqdm

from keras.datasets import mnist

from keras.layers import Dense, Flatten, Reshape, Conv2D, Conv2DTranspose, Dropout, Reshape

from keras.layers.advanced\_activations import LeakyReLU

from keras.models import Sequential

from tensorflow.keras.optimizers import Adam

from keras.utils.vis\_utils import plot\_model

from google.colab import drive

drive.mount('/content/drive')

path = '/content/drive/MyDrive/Colab Notebooks/'

# Building the discriminator model

def build\_discriminator(img\_shape=(72, 72, 3)):

model = Sequential()

model.add(Dense(1024, input\_dim=72\*72\*3))

model.add(LeakyReLU(0.2))

model.add(Dense(512))

model.add(LeakyReLU(0.2))

model.add(Dense(256))

model.add(LeakyReLU(0.2))

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

model.compile(loss='binary\_crossentropy', optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5), metrics=['accuracy'])

return model

# Build, compile and display a summary of the discriminator

discriminator = build\_discriminator()

plot\_model(discriminator, show\_shapes=True, show\_layer\_names=True)

# Building the generator model

def build\_generator(latent\_dim, img\_shape=(72, 72, 3)):

model = Sequential()

model.add(Dense(256, input\_dim=latent\_dim))

model.add(LeakyReLU(0.2))

model.add(Dense(512))

model.add(LeakyReLU(0.2))

model.add(Dense(1024))

model.add(LeakyReLU(0.2))

model.add(Dense(72\*72\*3, activation='tanh'))

return model

# Size of the latent space

latent\_dim = 100

# Build and display a summary of the discriminator

generator = build\_generator(latent\_dim)

plot\_model(generator, show\_shapes=True, show\_layer\_names=True)

# Building the GAN model which combines the discriminator and generator models

def define\_gan(generator, discriminator):

# Set the weights in the discriminator to not trainable

discriminator.trainable = False

model = Sequential()

model.add(generator)

model.add(discriminator)

# Compile combined GAN model

model.compile(loss='binary\_crossentropy',

optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5))

return model

# Build and display a summary of the GAN

gan\_model = define\_gan(generator, discriminator)

gan\_model.summary()

# Load and return the training data

def load\_training\_data():

train\_x = np.load(path + 'train\_x10000x72.npy', allow\_pickle=True)

# Converting image data from doubles to floats

train\_x = train\_x.astype('float32')

# Scale from [0, 255] to [-1, 1]

train\_x = (train\_x - 127.5) / 127.5

return train\_x

# Select random images from training data

def select\_real\_samples(train\_x, n\_samples):

# Choose n\_samples amount of random indexes

random\_indexes = np.random.randint(0, train\_x.shape[0], n\_samples)

# Select the images from the training data

real\_samples\_x = train\_x[random\_indexes]

# Label '1' for each image to signify that it is real

real\_samples\_y = np.ones((n\_samples, 1))

return real\_samples\_x, real\_samples\_y

# Generate random points in the latent space

def generate\_latent\_points(latent\_dim, n\_samples):

# Choose n\_samples amount of random points in the latent space

fake\_x\_input = np.random.randn(latent\_dim \* n\_samples)

# Reshape into batches of inputs for the model

fake\_x\_input = fake\_x\_input.reshape(n\_samples, latent\_dim)

return fake\_x\_input

# Generate fake images using random points in the latent space

def generate\_fake\_samples(generator, latent\_dim, n\_samples):

# Generate random points in the latent space

fake\_x\_input = generate\_latent\_points(latent\_dim, n\_samples)

# Generate fake images

fake\_x = generator.predict(fake\_x\_input)

# Label '0' for each image to signify that it is fake

fake\_y = np.zeros((n\_samples, 1))

return fake\_x, fake\_y

# Display and save a plot of n^2 generated images

def plot\_gen\_imgs(generated\_imgs, epoch, n=4):

# Scale from [-1, 1] to [0, 1] for displaying images

generated\_imgs = (generated\_imgs + 1) / 2.0

# Plot generated images in a grid

fig, ax = plt.subplots(n, n, figsize=(2, 2))

count = 0

for i in range(n):

for j in range(n):

ax[i, j].imshow(generated\_imgs[count])

ax[i, j].axis('off')

count += 1

# Save the plot of generated images

plt.savefig(path + '/FCGAN for 72x72 Output/generated\_images\_epoch\_%d.jpg' %

(epoch+1), bbox\_inches='tight', dpi = 200)

plt.close()

# Save the generator model

def save\_generator(generator, epoch):

generator.save(path + '/FCGAN for 72x72 Output/generator\_model\_epoch\_%d.h5' %

(epoch+1))

losses = []

accuracies = []

epochs = []

# Periodic evaluation of the model

def evaluate\_gan(epoch, generator, discriminator, train\_x, latent\_dim, n\_samples=150):

# Evaluate discriminator on real images

real\_x, real\_y = select\_real\_samples(train\_x, n\_samples)

real\_x = real\_x.reshape(n\_samples, 15552)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

# Evaluate discriminator on fake images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, n\_samples)

fake\_x = fake\_x.reshape(n\_samples, 15552)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

print('\nDiscriminator Accuracy (Real Data): %.0f%%, Discriminator Accuracy (Fake Data): %.0f%%\n' %

(d\_acc\_real\*100, d\_acc\_fake\*100))

fake\_x = fake\_x.reshape(n\_samples, 72, 72, 3)

plot\_gen\_imgs(fake\_x, epoch)

save\_generator(generator, epoch)

# Display and save plots to show discriminator loss and accuracy throughout training

def plot\_loss\_and\_accuracy(losses, accuracies, epochs):

losses = np.array(losses)

accuracies = np.array(accuracies)

epochs = np.array(epochs)

# Plot training losses for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, losses.T[0], label='Discriminator Loss (Real Data)')

plt.plot(epochs, losses.T[1], label='Discriminator Loss (Fake Data)')

# Calculate line of best fit

a0, a1 = np.polyfit(epochs, losses.T[0], 1)

b0, b1 = np.polyfit(epochs, losses.T[1], 1)

# Plot line of best fit

plt.plot(epochs, a0\*epochs + a1, color='blue', linestyle='dashed',

label='Line of best fit Discriminator Loss (Real Data)')

plt.plot(epochs, b0\*epochs + b1, color='red', linestyle='dashed',

label='Line of best fit Discriminator Loss (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss Values')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim()

plt.legend()

plt.savefig(path + '/FCGAN for 72x72 Output/discriminator\_loss\_plot.jpg',

bbox\_inches='tight')

# Plot training accuracies for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, accuracies.T[0], label='Discriminator Accuracy (Real Data)')

plt.plot(epochs, accuracies.T[1], label='Discriminator Accuracy (Fake Data)')

# Calculate line of best fit

d0, d1 = np.polyfit(epochs, accuracies.T[0], 1)

e0, e1 = np.polyfit(epochs, accuracies.T[1], 1)

# Plot line of best fit

plt.plot(epochs, d0\*epochs + d1, color='blue', linestyle='dashed',

label='Line of best fit (Real Data)')

plt.plot(epochs, e0\*epochs + e1, color='red', linestyle='dashed',

label='Line of best fit (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy Values (%)')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim([0, 110])

plt.legend()

plt.savefig(path + '/FCGAN for 72x72 Output/discriminator\_accuracy\_plot.jpg',

bbox\_inches='tight')

# Train GAN

def train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim, number\_of\_epochs=1000, batch\_size=128):

batches\_per\_epoch = int(train\_x.shape[0] / batch\_size)

# Half of batch for real images and half for fake images

half\_batch\_size = int(batch\_size / 2)

for i in range(number\_of\_epochs):

# Loading bar for each batch in an epoch

for j in tqdm(range(batches\_per\_epoch), desc='Epoch %d/%d' % (i+1, number\_of\_epochs)):

# Select half\_batch\_size amount of random real samples from training data

real\_x, real\_y = select\_real\_samples(train\_x, half\_batch\_size)

# Train discriminator on real images

real\_x = real\_x.reshape(64, 15552)

d\_loss\_real, \_ = discriminator.train\_on\_batch(real\_x, real\_y)

# Select half\_batch\_size amount of random fake samples from generated images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, half\_batch\_size)

# Train discriminator on fake images

fake\_x = fake\_x.reshape(64, 15552)

d\_loss\_fake, \_ = discriminator.train\_on\_batch(fake\_x, fake\_y)

# Select batch\_size amount of random points in the latent space

gan\_x = generate\_latent\_points(latent\_dim, batch\_size)

# Inverted label for each image

gan\_y = np.ones((batch\_size, 1))

# Update the generator via the discriminators erorr

g\_loss = gan\_model.train\_on\_batch(gan\_x, gan\_y)

print('Discriminator Loss (Real Data): %.3f, Discriminator Loss (Fake Data): %.3f, Generator Loss: %.3f' %

(d\_loss\_real, d\_loss\_fake, g\_loss))

# Add data from each epoch to these arrays for plotting

real\_x, real\_y = select\_real\_samples(train\_x, 150)

real\_x = real\_x.reshape(150, 15552)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, 150)

fake\_x = fake\_x.reshape(150, 15552)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

losses.append((d\_loss\_real, d\_loss\_fake))

accuracies.append((100.0 \* d\_acc\_real, 100.0 \* d\_acc\_fake))

epochs.append(i + 1)

# Evaluate GAN every 10 epochs

if (i + 1) % 10 == 0:

save\_generator(generator, i)

evaluate\_gan(i, generator, discriminator, train\_x, latent\_dim)

train\_x = load\_training\_data()

train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim)

plot\_loss\_and\_accuracy(losses, accuracies, epochs)

Appendix G – Source Code for DCGAN for 28x28 Resolution Images

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

from tqdm import tqdm

from keras.datasets import mnist

from keras.layers import Dense, Flatten, Reshape, Conv2D, Conv2DTranspose, LeakyReLU, Dropout

from keras.models import Sequential

from keras.utils.vis\_utils import plot\_model

from tensorflow.keras.optimizers import Adam

from google.colab import drive

drive.mount('/content/drive')

path = '/content/drive/MyDrive/Colab Notebooks/'

# Building the discriminator model

def build\_discriminator(img\_shape=(28, 28, 3)):

model = Sequential()

# Start with 28x28 image

model.add(Conv2D(64, (3,3), padding='same', input\_shape=img\_shape))

model.add(LeakyReLU(alpha=0.2))

# Downsample to 14x14

model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Downsample to 7x7

model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Binary classifier

model.add(Flatten())

model.add(Dropout(0.1))

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

# Compile discriminator model

model.compile(loss='binary\_crossentropy', optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5), metrics=['accuracy'])

return model

# Build, compile and display a summary of the discriminator

discriminator = build\_discriminator()

plot\_model(discriminator, show\_shapes=True, show\_layer\_names=True)

# Building the generator model

def build\_generator(latent\_dim):

model = Sequential()

# Start with 7x7 image

model.add(Dense(256\*7\*7, input\_dim=latent\_dim))

model.add(LeakyReLU(alpha=0.2))

model.add(Reshape((7, 7, 256)))

# Upsample to 14x14

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Upsample to 28x28

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Output layer

model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))

return model

# Size of the latent space

latent\_dim = 100

# Build and display a summary of the discriminator

generator = build\_generator(latent\_dim)

plot\_model(generator, show\_shapes=True, show\_layer\_names=True)

# Building the GAN model which combines the discriminator and generator models

def define\_gan(generator, discriminator):

# Set the weights in the discriminator to not trainable

discriminator.trainable = False

model = Sequential()

model.add(generator)

model.add(discriminator)

# Compile combined GAN model

model.compile(loss='binary\_crossentropy',

optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5))

return model

# Build and display a summary of the GAN

gan\_model = define\_gan(generator, discriminator)

gan\_model.summary()

# Load and return the training data

def load\_training\_data():

train\_x = np.load(path + 'train\_x10000x28.npy', allow\_pickle=True)

# Converting image data from doubles to floats

train\_x = train\_x.astype('float32')

# Scale from [0, 255] to [-1, 1]

train\_x = (train\_x - 127.5) / 127.5

return train\_x

# Select random images from training data

def select\_real\_samples(train\_x, n\_samples):

# Choose n\_samples amount of random indexes

random\_indexes = np.random.randint(0, train\_x.shape[0], n\_samples)

# Select the images from the training data

real\_samples\_x = train\_x[random\_indexes]

# Label '1' for each image to signify that it is real

real\_samples\_y = np.ones((n\_samples, 1))

return real\_samples\_x, real\_samples\_y

# Generate random points in the latent space

def generate\_latent\_points(latent\_dim, n\_samples):

# Choose n\_samples amount of random points in the latent space

fake\_x\_input = np.random.randn(latent\_dim \* n\_samples)

# Reshape into batches of inputs for the model

fake\_x\_input = fake\_x\_input.reshape(n\_samples, latent\_dim)

return fake\_x\_input

# Generate fake images using random points in the latent space

def generate\_fake\_samples(generator, latent\_dim, n\_samples):

# Generate random points in the latent space

fake\_x\_input = generate\_latent\_points(latent\_dim, n\_samples)

# Generate fake images

fake\_x = generator.predict(fake\_x\_input)

# Label '0' for each image to signify that it is fake

fake\_y = np.zeros((n\_samples, 1))

return fake\_x, fake\_y

# Display and save a plot of n^2 generated images

def plot\_gen\_imgs(generated\_imgs, epoch, n=4):

# Scale from [-1, 1] to [0, 1] for displaying images

generated\_imgs = (generated\_imgs + 1) / 2.0

# Plot generated images in a grid

fig, ax = plt.subplots(n, n, figsize=(2, 2))

count = 0

for i in range(n):

for j in range(n):

ax[i, j].imshow(generated\_imgs[count])

ax[i, j].axis('off')

count += 1

# Save the plot of generated images

plt.savefig(path + '/GAN for 28x28 Output/generated\_images\_epoch\_%d.jpg' %

(epoch+1), bbox\_inches='tight', dpi = 200)

plt.close()

# Save the generator model

def save\_generator(generator, epoch):

generator.save(path + '/GAN for 28x28 Output/generator\_model\_epoch\_%d.h5' %

(epoch+1))

losses = []

accuracies = []

epochs = []

# Periodic evaluation of the model

def evaluate\_gan(epoch, generator, discriminator, train\_x, latent\_dim, n\_samples=150):

# Evaluate discriminator on real images

real\_x, real\_y = select\_real\_samples(train\_x, n\_samples)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

# Evaluate discriminator on fake images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, n\_samples)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

print('\nDiscriminator Accuracy (Real Data): %.0f%%, Discriminator Accuracy (Fake Data): %.0f%%\n' %

(d\_acc\_real\*100, d\_acc\_fake\*100))

plot\_gen\_imgs(fake\_x, epoch)

save\_generator(generator, epoch)

# Display and save plots to show discriminator loss and accuracy throughout training

def plot\_loss\_and\_accuracy(losses, accuracies, epochs):

losses = np.array(losses)

accuracies = np.array(accuracies)

epochs = np.array(epochs)

# Plot training losses for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, losses.T[0], label='Discriminator Loss (Real Data)')

plt.plot(epochs, losses.T[1], label='Discriminator Loss (Fake Data)')

# Calculate line of best fit

a0, a1 = np.polyfit(epochs, losses.T[0], 2)

b0, b1 = np.polyfit(epochs, losses.T[1], 2)

# Plot line of best fit

plt.plot(epochs, a0\*epochs + a1, color='blue', linestyle='dashed',

label='Line of best fit Discriminator Loss (Real Data)')

plt.plot(epochs, b0\*epochs + b1, color='red', linestyle='dashed',

label='Line of best fit Discriminator Loss (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss Values')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim()

plt.legend()

plt.savefig(path + '/GAN for 28x28 Output/discriminator\_loss\_plot.jpg',

bbox\_inches='tight')

# Plot training accuracies for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, accuracies.T[0], label='Discriminator Accuracy (Real Data)')

plt.plot(epochs, accuracies.T[1], label='Discriminator Accuracy (Fake Data)')

# Calculate line of best fit

c0, c1 = np.polyfit(epochs, accuracies.T[0], 1)

d0, d1 = np.polyfit(epochs, accuracies.T[1], 1)

# Plot line of best fit

plt.plot(epochs, c0\*epochs + c1, color='blue', linestyle='dashed',

label='Line of best fit (Real Data)')

plt.plot(epochs, d0\*epochs + d1, color='red', linestyle='dashed',

label='Line of best fit (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy Values (%)')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim([0, 110])

plt.legend()

plt.savefig(path + '/GAN for 28x28 Output/discriminator\_accuracy\_plot.jpg',

bbox\_inches='tight')

# Train GAN

def train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim, number\_of\_epochs=1000, batch\_size=128):

batches\_per\_epoch = int(train\_x.shape[0] / batch\_size)

# Half of batch for real images and half for fake images

half\_batch\_size = int(batch\_size / 2)

for i in range(number\_of\_epochs):

# Loading bar for each batch in an epoch

for j in tqdm(range(batches\_per\_epoch), desc='Epoch %d/%d' % (i+1, number\_of\_epochs)):

# Select half\_batch\_size amount of random real samples from training data

real\_x, real\_y = select\_real\_samples(train\_x, half\_batch\_size)

# Train discriminator on real images

d\_loss\_real, \_ = discriminator.train\_on\_batch(real\_x, real\_y)

# Select half\_batch\_size amount of random fake samples from generated images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, half\_batch\_size)

# Train discriminator on fake images

d\_loss\_fake, \_ = discriminator.train\_on\_batch(fake\_x, fake\_y)

# Select batch\_size amount of random points in the latent space

gan\_x = generate\_latent\_points(latent\_dim, batch\_size)

# Inverted label for each image

gan\_y = np.ones((batch\_size, 1))

# Update the generator via the discriminators erorr

g\_loss = gan\_model.train\_on\_batch(gan\_x, gan\_y)

print('Discriminator Loss (Real Data): %.3f, Discriminator Loss (Fake Data): %.3f, Generator Loss: %.3f' %

(d\_loss\_real, d\_loss\_fake, g\_loss))

# Add data from each epoch to these arrays for plotting

real\_x, real\_y = select\_real\_samples(train\_x, 150)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, 150)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

losses.append((d\_loss\_real, d\_loss\_fake))

accuracies.append((100.0 \* d\_acc\_real, 100.0 \* d\_acc\_fake))

epochs.append(i + 1)

# Evaluate GAN every 10 epochs

if (i + 1) % 10 == 0:

save\_generator(generator, i)

evaluate\_gan(i, generator, discriminator, train\_x, latent\_dim)

train\_x = load\_training\_data()

train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim)

plot\_loss\_and\_accuracy(losses, accuracies, epochs)

Appendix H – Source Code for DCGAN for 72x72 Resolution Images

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

from tqdm import tqdm

from keras.datasets import mnist

from keras.layers import Dense, Flatten, Reshape, Conv2D, Conv2DTranspose, LeakyReLU, Dropout

from keras.models import Sequential

from keras.utils.vis\_utils import plot\_model

from tensorflow.keras.optimizers import Adam

from google.colab import drive

drive.mount('/content/drive')

path = '/content/drive/MyDrive/Colab Notebooks/'

# Building the discriminator model

def build\_discriminator(img\_shape=(72, 72, 3)):

model = Sequential()

# Start with 72x72 image

model.add(Conv2D(64, (3,3), padding='same', input\_shape=img\_shape))

model.add(LeakyReLU(alpha=0.2))

# Downsample to 36x36

model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Downsample to 18x18

model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Downsample to 9x9

model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Binary classifier

model.add(Flatten())

model.add(Dropout(0.1))

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

# Compile discriminator model

model.compile(loss='binary\_crossentropy', optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5), metrics=['accuracy'])

return model

# Build, compile and display a summary of the discriminator

discriminator = build\_discriminator()

plot\_model(discriminator, show\_shapes=True, show\_layer\_names=True)

# Building the generator model

def build\_generator(latent\_dim):

model = Sequential()

# Start with 9x9 image

model.add(Dense(256\*9\*9, input\_dim=latent\_dim))

model.add(LeakyReLU(alpha=0.2))

model.add(Reshape((9, 9, 256)))

# Upsample to 18x18

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Upsample to 36x36

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Upsample to 72x72

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# Output layer

model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))

return model

# Size of the latent space

latent\_dim = 100

# Build and display a summary of the discriminator

generator = build\_generator(latent\_dim)

plot\_model(generator, show\_shapes=True, show\_layer\_names=True)

# Building the GAN model which combines the discriminator and generator models

def define\_gan(generator, discriminator):

# Set the weights in the discriminator to not trainable

discriminator.trainable = False

model = Sequential()

model.add(generator)

model.add(discriminator)

# Compile combined GAN model

model.compile(loss='binary\_crossentropy',

optimizer=Adam(learning\_rate=0.0002, beta\_1=0.5))

return model

# Build and display a summary of the GAN

gan\_model = define\_gan(generator, discriminator)

gan\_model.summary()

# Load and return the training data

def load\_training\_data():

train\_x = np.load(path + 'train\_x10000x72.npy', allow\_pickle=True)

# Converting image data from doubles to floats

train\_x = train\_x.astype('float32')

# Scale from [0, 255] to [-1, 1]

train\_x = (train\_x - 127.5) / 127.5

return train\_x

# Select random images from training data

def select\_real\_samples(train\_x, n\_samples):

# Choose n\_samples amount of random indexes

random\_indexes = np.random.randint(0, train\_x.shape[0], n\_samples)

# Select the images from the training data

real\_samples\_x = train\_x[random\_indexes]

# Label '1' for each image to signify that it is real

real\_samples\_y = np.ones((n\_samples, 1))

return real\_samples\_x, real\_samples\_y

# Generate random points in the latent space

def generate\_latent\_points(latent\_dim, n\_samples):

# Choose n\_samples amount of random points in the latent space

fake\_x\_input = np.random.randn(latent\_dim \* n\_samples)

# Reshape into batches of inputs for the model

fake\_x\_input = fake\_x\_input.reshape(n\_samples, latent\_dim)

return fake\_x\_input

# Generate fake images using random points in the latent space

def generate\_fake\_samples(generator, latent\_dim, n\_samples):

# Generate random points in the latent space

fake\_x\_input = generate\_latent\_points(latent\_dim, n\_samples)

# Generate fake images

fake\_x = generator.predict(fake\_x\_input)

# Label '0' for each image to signify that it is fake

fake\_y = np.zeros((n\_samples, 1))

return fake\_x, fake\_y

# Display and save a plot of n^2 generated images

def plot\_gen\_imgs(generated\_imgs, epoch, n=4):

# Scale from [-1, 1] to [0, 1] for displaying images

generated\_imgs = (generated\_imgs + 1) / 2.0

# Plot generated images in a grid

fig, ax = plt.subplots(n, n, figsize=(2, 2))

count = 0

for i in range(n):

for j in range(n):

ax[i, j].imshow(generated\_imgs[count])

ax[i, j].axis('off')

count += 1

# Save the plot of generated images

plt.savefig(path + '/GAN for 72x72 Output/generated\_images\_epoch\_%d.jpg' %

(epoch+1), bbox\_inches='tight', dpi = 200)

plt.close()

# Save the generator model

def save\_generator(generator, epoch):

generator.save(path + '/GAN for 72x72 Output/generator\_model\_epoch\_%d.h5' %

(epoch+1))

losses = []

accuracies = []

epochs = []

# Periodic evaluation of the model

def evaluate\_gan(epoch, generator, discriminator, train\_x, latent\_dim, n\_samples=150):

# Evaluate discriminator on real images

real\_x, real\_y = select\_real\_samples(train\_x, n\_samples)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

# Evaluate discriminator on fake images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, n\_samples)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

print('\nDiscriminator Accuracy (Real Data): %.0f%%, Discriminator Accuracy (Fake Data): %.0f%%\n' %

(d\_acc\_real\*100, d\_acc\_fake\*100))

plot\_gen\_imgs(fake\_x, epoch)

save\_generator(generator, epoch)

# Display and save plots to show discriminator loss and accuracy throughout training

def plot\_loss\_and\_accuracy(losses, accuracies, epochs):

losses = np.array(losses)

accuracies = np.array(accuracies)

epochs = np.array(epochs)

# Plot training losses for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, losses.T[0], label='Discriminator Loss (Real Data)')

plt.plot(epochs, losses.T[1], label='Discriminator Loss (Fake Data)')

# Calculate line of best fit

a0, a1 = np.polyfit(epochs, losses.T[0], 2)

b0, b1 = np.polyfit(epochs, losses.T[1], 2)

# Plot line of best fit

plt.plot(epochs, a0\*epochs + a1, color='blue', linestyle='dashed',

label='Line of best fit Discriminator Loss (Real Data)')

plt.plot(epochs, b0\*epochs + b1, color='red', linestyle='dashed',

label='Line of best fit Discriminator Loss (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss Values')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim()

plt.legend()

plt.savefig(path + '/GAN for 28x28 Output/discriminator\_loss\_plot.jpg',

bbox\_inches='tight')

# Plot training accuracies for Discriminator

plt.figure(figsize=(15, 5))

plt.plot(epochs, accuracies.T[0], label='Discriminator Accuracy (Real Data)')

plt.plot(epochs, accuracies.T[1], label='Discriminator Accuracy (Fake Data)')

# Calculate line of best fit

c0, d1 = np.polyfit(epochs, accuracies.T[0], 1)

d0, d1 = np.polyfit(epochs, accuracies.T[1], 1)

# Plot line of best fit

plt.plot(epochs, d0\*epochs + d1, color='blue', linestyle='dashed',

label='Line of best fit (Real Data)')

plt.plot(epochs, e0\*epochs + e1, color='red', linestyle='dashed',

label='Line of best fit (Fake Data)')

# Configure and save plot

plt.xticks(np.arange(0, np.max(epochs) + 1, epochs.shape[0] / 20))

plt.title('Discriminator Training Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy Values (%)')

plt.xlim([0, np.max(epochs) + 1])

plt.ylim([0, 110])

plt.legend()

plt.savefig(path + '/GAN for 28x28 Output/discriminator\_accuracy\_plot.jpg',

bbox\_inches='tight')

# Train GAN

def train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim, number\_of\_epochs=1000, batch\_size=128):

batches\_per\_epoch = int(train\_x.shape[0] / batch\_size)

# Half of batch for real images and half for fake images

half\_batch\_size = int(batch\_size / 2)

for i in range(number\_of\_epochs):

# Loading bar for each batch in an epoch

for j in tqdm(range(batches\_per\_epoch), desc='Epoch %d/%d' % (i+1, number\_of\_epochs)):

# Select half\_batch\_size amount of random real samples from training data

real\_x, real\_y = select\_real\_samples(train\_x, half\_batch\_size)

# Train discriminator on real images

d\_loss\_real, \_ = discriminator.train\_on\_batch(real\_x, real\_y)

# Select half\_batch\_size amount of random fake samples from generated images

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, half\_batch\_size)

# Train discriminator on fake images

d\_loss\_fake, \_ = discriminator.train\_on\_batch(fake\_x, fake\_y)

# Select batch\_size amount of random points in the latent space

gan\_x = generate\_latent\_points(latent\_dim, batch\_size)

# Inverted label for each image

gan\_y = np.ones((batch\_size, 1))

# Update the generator via the discriminators erorr

g\_loss = gan\_model.train\_on\_batch(gan\_x, gan\_y)

print('Discriminator Loss (Real Data): %.3f, Discriminator Loss (Fake Data): %.3f, Generator Loss: %.3f' %

(d\_loss\_real, d\_loss\_fake, g\_loss))

# Add data from each epoch to these arrays for plotting

real\_x, real\_y = select\_real\_samples(train\_x, 150)

\_, d\_acc\_real = discriminator.evaluate(real\_x, real\_y, verbose=0)

fake\_x, fake\_y = generate\_fake\_samples(generator, latent\_dim, 150)

\_, d\_acc\_fake = discriminator.evaluate(fake\_x, fake\_y, verbose=0)

losses.append((d\_loss\_real, d\_loss\_fake))

accuracies.append((100.0 \* d\_acc\_real, 100.0 \* d\_acc\_fake))

epochs.append(i + 1)

# Evaluate GAN every 10 epochs

if (i + 1) % 10 == 0:

save\_generator(generator, i)

evaluate\_gan(i, generator, discriminator, train\_x, latent\_dim)

train\_x = load\_training\_data()

train\_gan(generator, discriminator, gan\_model, train\_x, latent\_dim)

plot\_loss\_and\_accuracy(losses, accuracies, epochs)

Appendix I – Source Code for Calculcating Fréchet Inception Distance

import numpy as np

import tensorflow as tf

from keras.models import Sequential

from keras.models import load\_model

from keras.layers import InputLayer, Lambda

from keras.applications.inception\_v3 import InceptionV3

from keras.applications.inception\_v3 import preprocess\_input

from keras.utils.vis\_utils import plot\_model

from scipy.linalg import sqrtm

from google.colab import drive

drive.mount('/content/drive')

path = '/content/drive/MyDrive/Colab Notebooks/'

# Image shape being used

img\_shape = (72, 72, 3)

# Load and return the training data

def load\_training\_data():

real\_imgs = np.load(path + 'train\_x10000x72.npy', allow\_pickle=True)

# Converting image data from doubles to floats

real\_imgs = real\_imgs.astype('float32')

# Scale from [0, 255] to [0, 1]

real\_imgs = real\_imgs / 255

return real\_imgs

# Generate random points in the latent space

def generate\_latent\_points(latent\_dim, n\_samples):

# Choose n\_samples amount of random points in the latent space

fake\_x\_input = np.random.randn(latent\_dim \* n\_samples)

# Reshape into batches of inputs for the model

fake\_x\_input = fake\_x\_input.reshape(n\_samples, latent\_dim)

return fake\_x\_input

def generate\_images(n\_samples):

# Load generator model

model = load\_model(path + '/FCGAN for 72x72 Output/generator\_model\_epoch\_900.h5')

# Generate images

latent\_points = generate\_latent\_points(100, n\_samples)

gen\_imgs = model.predict(latent\_points)

# Scale pixel values from [-1,1] to [0,1]

gen\_imgs = (gen\_imgs + 1) / 2.0

# Reshape images to correct shape

gen\_imgs = gen\_imgs.reshape(gen\_imgs.shape[0], img\_shape[0], img\_shape[1], img\_shape[2])

return gen\_imgs

# calculate frechet inception distance

def calculate\_fid(model, real\_imgs, fake\_imgs):

# calculate activations

act1 = model.predict(real\_imgs)

act2 = model.predict(fake\_imgs)

# calculate mean and covariance statistics

mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=False)

mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=False)

# calculate sum squared difference between means

ssdiff = np.sum((mu1 - mu2)\*\*2.0)

# calculate sqrt of product between cov

covmean = sqrtm(sigma1.dot(sigma2))

# check and correct imaginary numbers from sqrt

if np.iscomplexobj(covmean):

covmean = covmean.real

# calculate score

fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 \* covmean)

return fid

# Define InceptionV3 model with layers to resize images

model = Sequential()

model.add(InputLayer(img\_shape))

model.add(Lambda(lambda image: tf.image.resize(image, (299, 299))))

model.add(InceptionV3(include\_top=False, pooling='avg', input\_shape=(299, 299, 3)))

plot\_model(model, show\_shapes=True, show\_layer\_names=True)

# Load real data and generate fake data

gen\_imgsreal\_imgs = load\_training\_data()

fake\_imgs = generate\_images(10000)

# Process images

real\_imgs = real\_imgs.astype('float32')

fake\_imgs = fake\_imgs.astype('float32')

real\_imgs = preprocess\_input(real\_imgs)

fake\_imgs = preprocess\_input(fake\_imgs)

# Calculate FID Score

fid = calculate\_fid(model, real\_imgs, fake\_imgs)

print('FID: %.3f' % fid)