An AI Tool for Image Supersampling

Alex Jones
963906

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

September 14, 2022

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

I would like to dedicate this to my late grandfather.
His intelligence and drive will forever be a source of inspiration.

Abstract

Computer rendering of images has always been an expensive task in video games and image generation. In recent years we have seen an explosive increase in the default resolution of monitors and screens. Still, we have many lower quality images, which, when displayed on high-resolution monitors look lacklustre and jagged. Single Image Super-Resolution (SISR) has been described as a notoriously challenging ill-posed problem that aims to obtain high-resolution (HR) output from its low-resolution (LR) input [1]. Fundamentally, the idea of SISR is to remove the need to render pixels to a higher resolution and instead use the pre-computed resolution. Through learned intelligence this allows the computer to generate a higher resolution image, decreasing the total computational expense. The ideal solution is not to re-render these images from scratch but instead upscale the resolution to match the monitor *, i.e., 1920×1080 to 3840×2160 (4k).*

In the past few years, we have seen astronomical progress in generating super-resolution images with deep learning methods. In this document, we present the implementation of a novel deep learning approach to SISR instead of typical algorithmic or neural network implementations. The reason for utilising Artificial Intelligence (AI) in this matter is because, through Deep Learning, you can achieve state-of-the-art performance without the need to write long complex anti-aliasing algorithms. After implementation, we were able to see the comparison of network-generated images against original images and low-resolution images generically upscaled. This implementation represents an alternative approach to the problem and explores different concepts around this novel method.

Acknowledgements

I would like to thank my friendship group throughout University. Living alongside driven and like-minded individuals for the duration of my undergraduate degree has been nothing short of a pleasure. My family who offer continual support and love, I have endless gratitude towards. Finally, to the volleyball club that has allowed me to experience and fall in love with a new sport while at University, with a few exceptional individuals in the club who have become more like family.

Contents

Glossary **v**

1 Introduction **1**

1.1 Motivations 2

1.2 Project Aims and Objectives 3

1.3 Structure of the Document 3

2 Background Research **4**

2.1 Neural Networks: an overview 4

2.2 Generative Adversarial Networks 5

2.3 CycleGAN 8

2.4 Interpolation Based Methods 11

2.5 Convolutional Neural Networks 12

2.6 Nash Equilibrium 13

2.7 Machine Learned Supersampling 14

2.8 Summary 15

3 Methodology **16**

3.1 Tool choices 16

3.2 Implementation 17

3.3 Proposed Experimentation 20

3.4 Challenges 21

4 Results **23**

4.1 Altering batch size 23

4.2 Dataset alterations 24

4.3 Architectural adjustments 27

5 Conclusion and Future Work **28**

5.1 Result findings 28

5.2 Project Management 28

5.3 Future Work 29

5.4 Final Conclusion 30

Bibliography **31**

Appendices **35**

A Supplementary Data **36**

B Supplementary Tables **46**

Glossary

- **Single Image Super Resolution** - SISR, an image processing technique that aims to output a high resolution image from a low resolution input.
- **Machine Learning** - ML, is the computer science discipline of using algorithms to parse data, learn from that data, and make informed decisions based on what it has learned.
- **Deep Learning** - DL, is a branch of ML in which algorithms get structured in layers to create an "artificial neural network" so it can learn and make intelligent decisions on its own.
- **Natural Language Processing** - NLP, the process of using machine learning to analyse sentiment and meaning behind sentences or passages of text.
- **Interactive Development Environment** - IDE, a software application dedicated to the functional development and testing of code. Consisting of, at the very least: a code editor, build automation tools, and debugging features.
- **API** - Application Program Interface, is the overall accessible functionality of a program/piece of software.
- **Python** - An interpreted, high-level and general purpose language, mainly used for data manipulation and processing but is able to be used for anything.
- **Keras** - One of the leading neural network APIs that allows interface with the TensorFlow libraries.
- **TensorFlow** - Created by the Google Brain team, TensorFlow is an open source library for numerical computation and large-scale machine learning.

- **AI** - Artificial Intelligence, a discipline of computer science with the aim of building smart machines capable of performing tasks that typically require human intelligence
- **Markov Chain** - A Markov chain is a labelled transition system model describing a sequence of events where the probability of each event depends only on the state gotten from the previous event.
- **Zero-sum game** - a game where one player's gains are exactly equal to the other player's losses.
- **Mean Squared Error** - a common loss function used in neural networks that is calculated as the average of squared differences of the predicted outputs and ground truth of the network.
- **Peak Signal-To-Noise Ratio** - the ratio (measured in decibels) between a reconstructed image and original image where it describes the original values of the image versus the power of the noise corrupting the image or causing image artefacts. Higher PSNR means better quality of the reconstructed image is said to be.
- **GPU** - Graphical Processing Unit
- **CPU** - Central Processing Unit

Chapter 1

Introduction

> Generative Adversarial Networks is the most interesting idea in machine learning in the last ten years
>
> Yann LeCun (Facebook AI Director)

In recent years, AI has made headlines for good and bad due to the giant leaps and bounds in multiple fields towards which it has contributed. It is a concept of much uncertainty for those who do not understand computers and AI. However, for most professionals in the field, it is seen as a tool for elevating the ability of a computer in execution of increasingly complex tasks. In image processing, it has proven extremely useful for generation of images due to its ability to self-supervise quickly and efficiently while altering its algorithms as it learns for whatever is needed. Deep Learning (DL) is a branch of Machine Learning (ML) algorithms that aims to learn how to hierarchically represent data. It has shown superiority in fields of [1] [2] [3]:

- Computer Vision
- Speech Recognition
- Natural Language Processing (NLP)

Although it has found recent use in AI assisted video rendering and unpaired video translation, this does not detract from the difficulty of the problem of upscaling of single image resolution.

In this project we explored and developed a DL implementation for SISR, trained using Keras and a custom Generative Adversarial Network (GAN) model. Initially, a generator and discriminator were trained to play a zero-sum game to output the highest possible resolution

images. Throughout this project we gained a deeper understanding of Machine Learning as a whole, and the surrounding concepts of AI and related technologies.

1.1 Motivations

> What I cannot create, I do not understand.
>
> Richard Feynman [4]

The first motivation for this project is that not all personal computers have top specification parts; this means that native rendering of higher resolutions proves very expensive. We hoped to develop a more concrete understanding of the positives and negatives of supersampling versus native rendering. The project aimed to efficiently use modern programming language and technologies to produce elegant, human-readable code that can abstract the complexities and intricacies of machine-learning systems and their implementation. GANs use unsupervised learning tasks to automatically discover regularities and patterns in input data and allow the network to generate new examples that can be from a combination of the sample data.

A great example to look at is `https://thispersondoesnotexist.com/`. [5]

GANs are a subset of Deep Learning (by extension, Machine Learning) and are a majorly popular field that focuses on their ability to exceed human performance by completing zero-sum games; until the discriminator is fooled over half of the time [6].

We not only had a vested interest in AI, which stems from the work completed on AI chat assistants during a recent work placement, but AI is also a diverse and ever-growing field that will shape the future of how we view technology. The benefits of single image super-resolution include, but are not limited to:

- Computation of 4k images requires higher pixel counts taken from a screen. Conversely, super sampling maintains the same pixel number as LR but takes larger samples per grid dot. This involves doing matrix calculations on each pixel per second to upscale the resolution.
- The time taken for Deep Learned Super Sampling (DLSS) has proven to be much faster than native rendering. [7]

1.2 Project Aims and Objectives

Primarily, the objectives for this project were to expand knowledge of Python and a new domain of Computer Science, namely Generative Adversarial Networks, while also making correct design choices to complete the project. With this goal completed, the project was centred around exploring SISR using a defined GAN architecture as laid out in *"Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network"* [8]. We also aimed to explore how slight changes and modifications to network architecture could change the training time and output, coupled with adjusting volume of training and testing data.

Objectives:

- Learn the TensorFlow library and implement a basic GAN [4] with a generator and a discriminator model, with low-resolution output (i.e., 256×256 output from 64×64 input). Then train this network to a point where the discriminator model is tricked over half of the time.
- Successfully generate HR image output from LR input.
- Train the network with enough data so that it does not diverge upon iterating: "Training generative adversarial networks (GAN) using too little data typically leads to discriminator overfitting, causing training to diverge" [9]
- Once a basic GAN is developed and prototyped, test it with a much larger dataset of higher quality images.

1.3 Structure of the Document

Chapter 2 presents extensive background research to understand the concepts behind the project as well as related work. Chapter 3 explains the implementation of the project, including tools used and challenges faced. Chapter 4 discusses and analyses the results of the trained GAN at different stages and varying architectures. Finally, chapter 5 summarizes the main contributions and critical points to take away from this project.

Chapter 2

Background Research

2.1 Neural Networks: an overview

Neural networks are algorithms that attempt to mimic the operations and neural pathways that occur in a human brain, focusing on recognising relationships in vast amounts of data. With applications ranging from financial services, personalization of online ads, and many uses in the computer industry, they are a widely used concept for tackling issues that the human brain would not manage as effectively. At the highest level, neural networks are multi-layer networks of neurons used for classification and prediction. The processing ability of neural networks is stored in the connection strength between units, or *weight* between *units or nodes* obtained from a process of adaptation to training patterns [10].

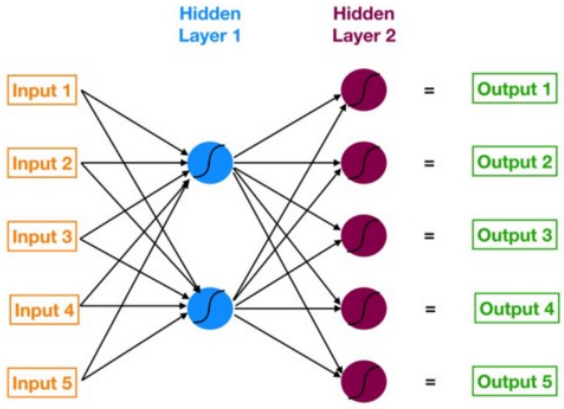


Figure 1: Simple neural network with 5 inputs, 5 outputs, 2 hidden layers of neurons [11]

As seen in figure 1, there is:

- The input layer, displayed in orange.
- A hidden layer of neurons, displayed in blue.
- The second hidden layer of neurons, displayed in purple.
- The output layer (the prediction) of the model, displayed in green.

A simple example, with one input, one hidden layer and one output, would be single feature logistic regression with only one X variable. As seen above, the output calculated through logistic regression is carried out within the network. The hidden layers apply weight to the inputs as they travel through and assign them different routes through non-linear transformations.

Although this overview was brief, it bears weight concerning the project as GANs run on Convolutional Neural Networks; CNNs, to summarise, are a subclass of neural networks with at least one convolution layer (which is generally one part of a hidden layer). GANs run on CNNs as they are excellent at capturing local information (like neighbouring pixels in an image) and reducing the overall complexity (training time, samples needed, chance of overfitting) of a network.

2.2 Generative Adversarial Networks

> [...] we present a method that can learn to [capture] special characteristics of one image collection and figuring out how these characteristics could be translated into the other image collection, all in the absence of any paired training examples.
>
> Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017 [12]

GANs are a subset of machine learning introduced to Deep Learning (DL) by Ian Goodfellow as part of his PhD thesis; generally, DL models can divide into discriminant models and generative models [13].

A discriminator is a method of modelling the relationship between unknown data y and known data x. In contrast, a generator refers to a model that can generate random output, usually under the condition of given implicit parameters. The majority of generative models require the utilisation of Markov chains. GAN uses underlying codes to express latent dimensions, control data implicit relationships, and does not require Markov chain use [3].

For GANs to work, we must have a dataset of HR images in the discriminator. This network then plays a game against its "adversary" or the generator network, which takes one of the HR images and downsamples it into LR "noise". A notorious issue for SISR is that an LR image can map to many possible HR images in the training of generator models [2] [6].

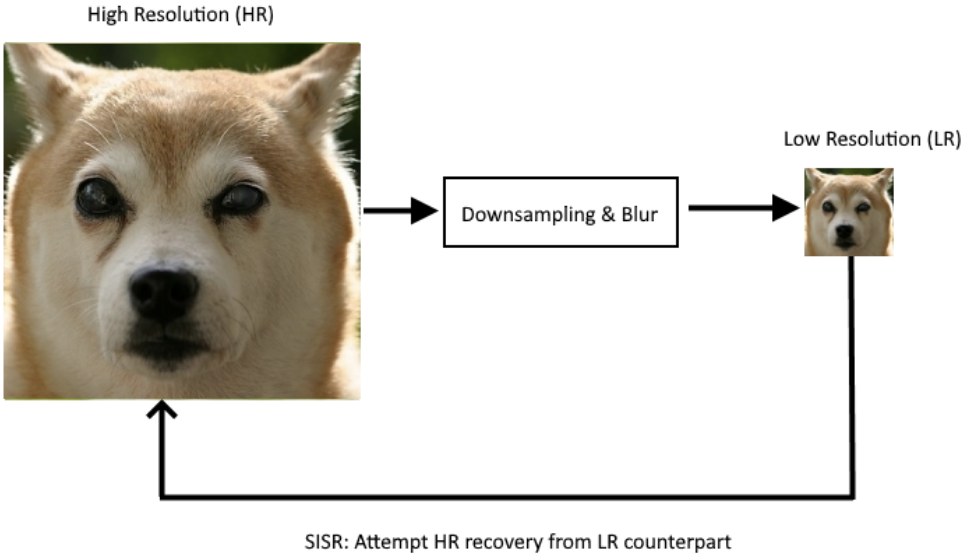


Figure 2: Sketch of the overall framework of SISR downsampling an HR image

There are three key steps to be followed for the working of GANs:

- Based on the input, the generator tries to create an authentic image while matching one of the samples from the original dataset.
- The discriminator receives the image generated in addition to the real data. After its evaluation, it will display the result deeming the image real or fake.
- The generator and discriminator networks are connected to produce a GAN.

The datasets in GANs must be large or the output can be prone to overfitting or underfitting; currently, there are a variety of datasets available for image super-resolution, which greatly

differ in image amounts, quality, and resolution [1]. To train a high-quality, high-resolution GAN requires a dataset of a variety of images in the magnitude of

$$\sim 10^5 - 10^6$$

images, often this proves costly in both matters of time and computational power [9].
The objective function of a basic GAN can be described using the mathematical notation as follows [3] [4] [13] [14] [15]:

$$min_G max_D V(D,G) = E_{x\sim Pdata(x)}[logD(x)] + E_{z\sim P_z(z)}[log(1-D(G(z)))]$$

- **G** - the generator network.
- **D** - the discriminator network.
- **x** - real data, the images put in the discriminator network.
- **z** - fake data, noise input into the generator network. Usually uses Gaussian noise, which is a random variable or a random variable in the potential space.
- **V** - the output of the *zero-sum* game played by the 2 networks.
- **P** - the fixed distribution for whatever follows, for x it usually assumes the maximum likelihood and for z the minimum.

$$Ex \sim p_{data}(x)[log(D(x))]$$

$$Ez \sim p_z(z)[log(1-D(G(z)))]$$

The equations above refer to the expectation of maximising/minimising the probability that the generated sample is seen as real/fake by the discriminator, respectively.

The function shows a zero-sum min-max game where the generator is trying to minimise V(D,G) and the discriminator attempts to maximise it. Ideally, the generator and discriminator achieve equilibrium:

$$P_{data}(x) = P_g(x)$$

Using a fixed value for G, a derivative can be taken from the min-max objective equation to produce the following to show the optimal D(x) value [3]:

$$D*(x) = \frac{P_g(x)}{P_g(x) + P_{data}(x)}$$

This shows us that the best value of D(x) would be 0.5 meaning, that it cannot discriminate between the real sample or the fake one. Figure A.1 is a basic outline of the architecture of this game [16].

There are a plethora of GANs used today; popular image processing networks are CycleGANs (transferring characteristics of one image to another), CartoonGANs (transforming real-time photos into cartoon-style images), CariGANs (photo-to-caricature translation), and StyleGANs (for generation of photorealistic high-quality images of faces) with the last three belonging to conditional GANs [12] [17] [16] [18].

2.3 CycleGAN

SISR with a CycleGAN would be a novel approach when tackling the problem; previous attempts have been made at super-resolution, but the results were not ideal; they were also not the main focus [18] [19]. Yielding mainly macro-effects (where the background is blurred) and lacking the crisp upscaled resolution as seen in other papers [15] [19].

CycleGANs are a specific type of conditional GAN as described by Goodwell [13] for image-to-image translation, covering anything from [12]:

- Translating summer photos to winter photos.
- Changing a photograph to a painting in a particular style.
- Changing a horse to a zebra, or Donald Trump to Barack Obama [6], or anything along these lines.
- Transferring lower quality image to higher quality with a shallower depth of field. [19]

For several years, training an image-to-image model for translation required a large dataset made of paired examples of identical images. However, for each pair there had to be the ground truth and what the model was attempting to translate to.

This usually had to be a vast dataset of an extreme amount of example input images X and desired style change for the expected output Y. Presently, there is a desire for image-to-image

translation without paired datasets; being that these are notoriously hard to find. Commonly referred to as *the problem of unpaired image-to-image translation*, is where CycleGANs have proved extremely successful. Instead of paired image collections, CycleGANs use any 2 unordered image collections *X* and *Y*, where the algorithm will learn to automatically "translate" the image from one to the other and vice versa. [19]
CycleGAN architecture is an extension of GAN, it involves the simultaneous training of 2 generator models and discriminator models as seen in Figures A.2 and A.3.

One can see that a ResNet architecture of *n* layers is generally used in CycleGANs with consistent use of more than 5 layers of ResNet blocks. [20]
As the dataset needs to be two separate unpaired image collections, there will be one collection of HR images and another collection made up of images at a much lower resolution.

The CycleGAN uses an additional extension called cycle consistency; this is the idea that if we transform from source to target and then back again to the source distribution, we should get the same result as from our source distribution. [12] [19] [20]

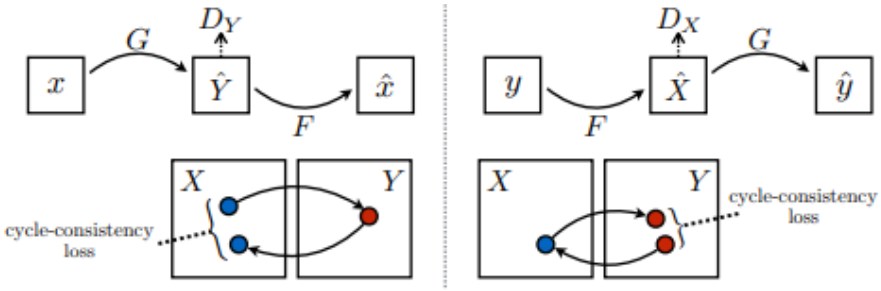


Figure 3: Forward and backward cycle-consistency loss [19]

From the objective equation, as shown in 2.2, we have some slight adjustments to consider, first we will change the mathematical notations [19]:

- **G** - one of the generator models.
- **F** - the other generator model.
- **x** - input data for generator *G*.
- **y** - input data for generator F.
- $\mathbf{D_X}$ - the discriminator model to distinguish between input *x* and generated F(y).
- $\mathbf{D_Y}$ - the discriminator model to distinguish between input *y* and generated G(x).

- $\mathbf{L_{GAN}}$ - the output of the *zero-sum* "game" played by the 2 networks, the adversarial losses for both mapping functions.
- $\mathbf{L_{cyc}}$ - cycle consistency loss
- $$En \sim p_{data}(n)[log(D(n))]$$

  Is the expectation of maximising the probability that the generated n is seen as real.

As seen above, there is firstly no fake data/noise input to the generator network. This is because the two generators are trying to generate the same image as the original image, depending on the input point (see fig 3 and 4).

$$G : X \rightarrow Y$$

$$F : Y \rightarrow X$$

Where G and F are the 2 generator models and we have, in addition, 2 adversarial discriminator models D_x and D_Y. The objective equation differs slightly from a normal GAN, this is in part due to the fact that GANs objective function effectively addresses adversarial loss (discrimination between generated data and ground truth). In the case of CycleGANs the adversarial loss equation changes slightly, for **G: X→Y** it is [19]:

$$\begin{aligned} min_G max_{D_Y} L_{GAN}(G, D_Y, X, Y) = & E_{y \sim Pdata(y)}[log D_Y(y)] + \\ & E_{x \sim P_{data}(x)}[log(1 - D_Y(G(x)))] \end{aligned}$$

Conversely, for **F: Y→X** the equation yields:

$$\begin{aligned} min_F max_{D_X} L_{GAN}(F, D_X, Y, X) = & E_{x \sim Pdata(x)}[log D_X(x)] + \\ & E_{y \sim P_{data}(y)}[log(1 - D_X(F(y)))] \end{aligned}$$

However, CycleGANs have a pair of generator models and discriminator models, meaning we must consider the cycle-consistency loss. We must address both *forward cycle-consistency* and *backward cycle-consistency*, respectively represented as:

$$x \rightarrow G(x) \rightarrow F(G(x)) \approx x$$

$$y \rightarrow F(y) \rightarrow G(F(y)) \approx y$$

The incentivisation of this behaviour for forward cycle-consistency is using the loss equation:

$$L_{cyc}(G,F) = E_{x \sim Pdata(x)}[|F(G(x)) - x|_1] + E_{y \sim P_{data}(y)}[|G(F(y)) - y|_1]$$

$L_{cyc}(F,G) \equiv L_{cyc}(G,F)$ as the components have merely changed places, therefore only the one equation needs to be considered, either forward or backward. With CycleGANs the objective aim is to solve:

$$G*, F* = min_{G,F} max_{D_X, D_Y} V(G, F, D_X, D_Y)$$

2.4 Interpolation Based Methods

Image interpolation is a term often used synonymously with image processing [21]. Generally, it refers to anything related to replicating images at different sizes, with efforts to maintain the image's properties and qualities. Before introducing machine learning, these algorithms were often used for upscaling (interpolation) of images.

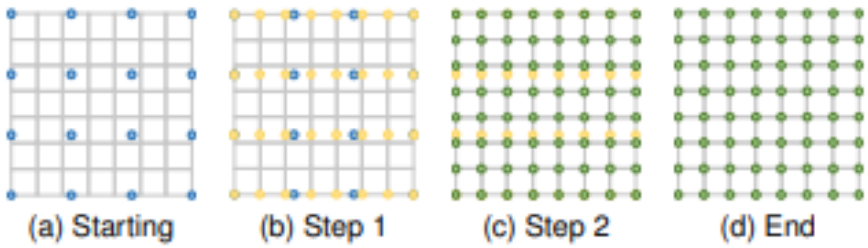


Figure 4: Interpolation based upsampling methods [1]

Figure 4 shows a grey board denoting the coordinates of pixels on an image. Respectively, the *blue*, *yellow*, and *green* points represent the initial, intermediate, and output pixels. Discussed below are some prominent and efficient early interpolation algorithms [1].

All of the below methods are easy to read and implement. None take into account the content of the image (artifacts, patterns, features) but instead focus on manipulating pixels directly. Some are still used in super-resolution models implemented with convolutional neural networks.

2.4.1 Nearest Neighbour

Nearest neighbour interpolation selects the nearest pixel from the original (source) image by rounding the relative coordinates of interpolation. With this, the closest corresponding pixel in the source to match the destination image is found. New pixels (at the destination) are made up of the pixels nearby that hold the same value as the source pixel. As destination images gain size, the pixilation and jaggedness of edges increases. This high-speed method produces low quality and blocky results. [21]

2.4.2 Bilinear

Bilinear interpolation takes an average of the neighbouring pixels from the source of the coordinates of interpolation. This makes up the colour value for the destination image, often yielding a smoothing effect from the source image. It results in a quadratic interpolation whilst taking 2×2 pixels into account and has much better output than nearest-neighbour yet remains quite fast. [22]

2.4.3 Bicubic

Bicubic interpolation considers the closest 4×4 pixels into account. When speed and efficiency is no issue this will be chosen over Nearest Neighbour and Bilinear. As it takes more pixels into account, the nearer are given higher weightings. Bicubic interpolation results in smoother results with fewer artifacts but is much slower. In terms of sharpness and output quality (smoothness), it is arguably the most efficient. [23]

2.5 Convolutional Neural Networks

CNNs have been around since the 1980s; however, recently have seen much popularity due to success in image classification [24]. Before the widespread adoption of CNNs the techniques were slower and required exact loss function design; CNNs group image super-resolution techniques into prediction-based methods (bilinear, bicubic, Lanczos) [18]. Many solutions to SISR have involved CNNs as they provide fast and accurate reconstructions of HR images [1] [2] [18] [25].

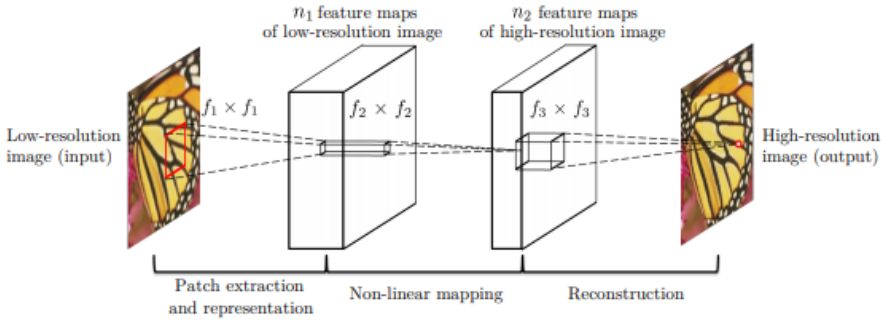


Figure 5: The architecture of a CNN [24]

Figure 5 shows that given a low-resolution image, the first convolutional layer extracts a set of feature maps. These feed into the second layer which maps this extracted set nonlinearly to high-resolution patch representations (a kernel of a fixed size, seen to be 3x3, 9x9, etc.). The last layer then produces the final HR image. The basic steps to follow for CNNs are [24]:

- **Patch extraction** - patches (or kernels) of a fixed size are extracted from the low resolution image, then represented as a high-dimensional vector.
- **Non-linear mapping** - this operation maps each high-dimensional vector onto another high-dimensional vector.
- **Reconstruction** - combining the above high-resolution patch representations, the final HR image is generated.

Convolutional models were key to consider in regards to the project as GANs run 2 models, both of which are made of CNNs; the first generating a HR output to attempt to trick the discriminator, the second to detect which of the two inputs are real.

2.6 Nash Equilibrium

In any game played, the default strategy is to maximise the reward in the worst-case scenario. As GANs are two models playing a zero-sum game, then this is no different. Nash equilibrium describes and models this effectively through derivation of a compromise whereby each player will follow a specific path/algorithm to obtain this maximum reward; there should be no need for deviation from this strategy. As the game size increases, so does the search space or possible number of algorithms for these strategies. Therefore there needs to be a derivation of specific

and specialised algorithms to find the equilibria [26]. The training of GANs creates a necessity in finding the Nash equilibrium to a game that is adversarial in the manner that each side wishes to minimise its cost function [27] [28]. The generator and discriminator are playing a game where Nash equilibrium is achieved when the distribution of the generator becomes the same as the desired distribution; as seen in above sections the ideal is a value of *0.5* [20]. GANs are two player games; Wimpee stated that finding the equilibria of games of more than two players proves extremely difficult and far more complex [26]. Due to the innate complexity and difficulty of finding Nash equilibria for normal two player games, finding a method for minimising loss effectively for the discriminator and generator proves difficult even now.

2.7 Machine Learned Supersampling

Image interpolation has always been an expensive task in video games and image processing which, until recently, relied on algorithmic processing (see 2.4). However, the last ten years has seen an exponential increase in the use of deep neural networks and, more recently GANs for heightened results and more accurate image upscaling.

2.7.1 Super-Resolution with Convolutional Networks

SRCNN [24] was the first deep CNN to surpass previous image super-resolution algorithmic standards within a few iterations. Despite these methods being very fast, they usually yield solutions with overly smooth textures [8]. This method was also able to outperform sparse encoding methods, namely [29] and [30], with more training – this approach by Dong et al. [24] [31] uses bicubic interpolation for the interpolation of the LR image input and then passes this through a three-layer deep CNN for fast and accurate SISR performance (as mentioned in 2.5). While following the three layer lightweight architecture of:

- Patch Extraction and representation
- Non-linear mapping
- Reconstruction

This CNN was able to output state-of-the-art restoration quality while achieving fast speed for practical use online [24]. This method, however, uses MSE as a loss function which when described in [8] is shown as less effective as it tends to underestimate the complexity of image feature retrieval and thereby over smooths output images.

2.7.2 Super-Resolution using Generative Adversarial Networks

SRGAN, as proposed by Ledig et al. [8] (2016), tackles the novel approach of using a GAN for SISR as opposed to fast and accurate deep CNNs. This novel approach aims to tackle the recovery of finer texture details at much larger upscaling factors. As CNN solutions use MSE for loss functions, they begin to over smooth image textures, which worsen with larger interpolation factors. Proposed therein was a custom-defined loss function called perceptual loss and is made up of the adversarial loss and content loss (or the loss of the GAN and loss between images during training). It goes on to describe how although common minimisation of MSE maximises PSNR, which is often a metric for evaluation of super-resolution algorithms [8], this metric fails to capture perceptual differences for humans (this is backed by Zhu et al. [32]).

Interestingly, Ledig et al. base the loss function on MSE and refer to it as an improvement that changes relative to perceptual characteristics instead of pixel-wise losses and builds upon the VGG network described in [33].

2.8 Summary

From the background research conducted, it becomes apparent that deep networks provide more accurate results on larger image SR problems when compared with their algorithmic interpolation predecessors. While it may not prove the most efficient method for small image datasets, we decided that a GAN approach would prove novel compared with CNN approaches. The architecture laid out by SRGAN was followed closely in our implementation, and adjustments were made to residual block number and network layout to compare time versus performance. We decided to use fewer images than described in [8] [24] [32] and opted for an approach as defined by Karras et al. [9] due to resource limitations caused by training on a laptop CPU.

Chapter 3

Methodology

As a result of the literature review and background research conducted concerning super-resolution we have researched algorithmic and machine learning-based techniques. We have found that although there is speed and efficiency for interpolation-based methods, true accuracy and quality come from the implementation of machine learning-based approaches. These approaches are particularly effective at generating images closer to the ground truth at larger up-scaling ratios. The majority of these networks focus on a CNN approach with various implementations on minimising the loss functions to achieve competitive PSNR. The most novel approach seemed to avoid using MSE as a loss function and instead used perceptual loss functions which are based on feature extraction of pre-trained networks (namely VGG).

3.1 Tool choices

3.1.1 Jupyter Notebooks

Jupyter Notebooks [34] is an open source web application run as a virtual machine from the user's computer that is often coupled with data visualization and machine learning. The only pitfall realised late on in project development is that with weaker hardware (even in newer machines) one is restricted on other processes while training. We used this instead of an IDE with Python due to a desire for faster debugging and testing of code, and with Jupyter breaking code into small cells it was easier to find and fix problems. However, if we ran this project again, Google Colab would be used to allow the running and training of Jupyter Notebooks on hardware abstracted from a user's machine provided they have an internet connection.

3.1.2 Keras

Keras [35] is a modular and high-level API that wraps common deep learning layers and operations into easy to understand blocks, it abstracts the complexities of deep learning away from the developer. It performs all of this by running on top of TensorFlow 2.0. We used this API for the custom definition of our network using the sequential model, see: `https://keras.io/guides/sequential_model/`, which allowed the modification and de-cluttering of the layers of the models.

3.1.3 Tensorflow

TensorFlow [36] is an open-source machine learning framework, packaged as a Python library, developed by Google with a particular focus on the training of deep neural networks, and this was used to help train the generator and discriminator models. The documentation for TensorFlow is concise and easy to understand with many great examples and helped the debugging process and comprehension of certain imports.

3.1.4 Matplotlib

Matplotlib [37] is a library with the core focus of data visualization for python. We used this for outputting graphical representations of the network and its various loss results recorded and images generated during the testing phase.

3.1.5 NumPy

NumPy [38] is a commonly used Python library for array manipulation. Array manipulation bears huge importance to our project as images are just large 3 dimensional arrays. Anything regarding the manipulation or transformation of values inside them made use of NumPy's high-level mathematical operations to ease programming.

3.2 Implementation

We have implemented a basic GAN that tries to super resolve images to the standard of their respective HR counterparts.

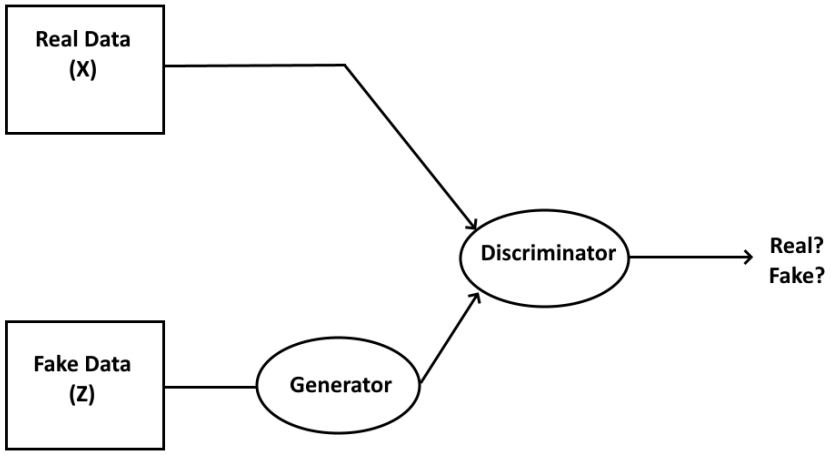


Figure 6: Our view of the basic architecture of a GAN (simplified from A.1)

We have included a loss function as described in [8] that allows the backpropagation to update weightings between neurons according to how well or poorly the Generator performs when outputting an SR image. This was done through the loss function called Perceptual Loss (as defined by Ledig et al.)

$$l^{SR} = \underbrace{\underbrace{l_{\mathrm{X}}^{SR}}_{\text{content loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

Perceptual Loss

$$l_{Gen}^{SR} = \sum_{n=1}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR}))$$

Adversarial Loss

Figure 7: Loss functions defined in [8]

Perceptual loss is the core loss function as mentioned above and comprises Content loss and Adversarial loss. Content loss is also known as Reconstruction loss and is calculated based on feature maps from the pre-defined VGG network described by Johnson et al. [18] In their paper (Perceptual Losses for Real-Time Style Transfer and Super-Resolution), VGG loss is defined as the *"Euclidean distance between feature representations"*, in this case, the

feature representations being compared are those from the super-resolved image and the ground truth image. The Adversarial loss moves our solution towards the ground truth by updating weightings in our Discriminator network that differentiates between the original photos and the generated images.

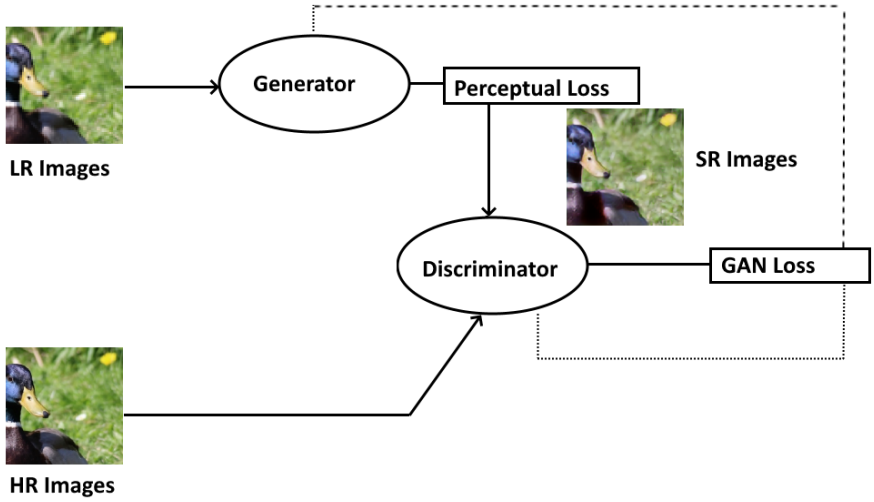


Figure 8: Updated GAN architecture showing where the loss functions come in

As described by Ledig et al. [8] we implemented our GAN following their architecture and the design pointers for stable and deep GAN training set out by Radford et al., 2016 [39], with key points being that:

- Replace all pooling layers with strided convolutions in the discriminator and generator (these being fractionally-strided or pixel shuffling layers as described [40] [41])
- Make use of Batch Normalization in both the generator and discriminator
- Using ReLU activation in the generator (in our case we made use of Parametric ReLU) for all layers except the output, which uses Tanh
- Using LeakyReLU activation throughout the discriminator

These design choices were reinforced by [6] [8] [13] [27] [42] [43] [44] and are reflected in Figures A.4, A.5.

3.3 Proposed Experimentation

We have successfully developed and implemented a GAN that takes in an LR image and feeds forward an SR image to a discriminator network that compares it to its HR counterpart. It uses backpropagation to assign the relevant losses and update weightings across the network and improve the quality of the SR image. The losses fed backward are calculated using a Perceptual Loss Function based on the paper by Ledig et al. [8]. This loss function is used rather than Mean Squared Error (MSE) due to its ability to not over smooth textures when upsampling to higher resolutions while maintaining a desirable PSNR.

3.3.1 Metric Variations

From looking at graphical plots of loss changes (described in Section 4) over various epochs and the resulting SR images at each designated output stage, we analysed the networks ability to learn and update its weightings based on the variation of different metrics:

- **Training set size** - we varied the dataset used for training to see the resulting bias. We used 40, 200, and 1000 images for training with 8, 50, and 200 images for testing, respectively.
- **Number of Residual Blocks** - by varying this number between 2, 4, 8, and 16 we aimed to see the differences of learning of the network and how the time necessary per epoch changed.
- **Number of epochs** - we varied this just to show how the network changed its output and which epoch number it started to show signs of accuracy versus plateau.
- **Batch size** - for an unknown reason, when training on batch sizes of larger than 1, our network struggled to learn and output undesirable results (see 4)

3.3.2 Training and Testing

The only pre-processing applied for downscaling the images was normalization into the range of [-1,1] as is required to fit the range of the tanh activation function used in the generator [39] to output the SR image. For the VGG loss function implemented, we had to remove the default 'top' values, as for VGG19 and VGG16 the image inputs have to be certain dimensions (for VGG19, which we used, they must be 224x224). We trained the network on the Linnaeus

dataset, see: `http://chaladze.com/l5/`, using the (256x256) images. Instead of using all 6000 training images, we shrunk this randomly down to one of the set sizes outlined above to save time as we knew this was limited. Epochs were varied from 1 to 50, to view the change in quality and save time when attempting to optimise the code. When varying residual blocks, the number of trainable parameters changed which also meant that the time per epoch changed significantly (Figure A.6 and Table B.1).

Testing was done on a subset of images the network had never seen before which was 20% of the size of the training set. After all epochs had run or at intervals of every third epoch, the testing was conducted and outputted eight resulting images with their LR and HR counterparts. We displayed the results underneath the cell for training in Jupyter Notebooks. These outputs are outlined in the Results section (4) and display good and bad results, the majority of outputs can be found in the Appendix.

3.4 Challenges

3.4.1 Documentation

Python [45], Keras [35], and TensorFlow [36] were the core components of the implementation of our GAN. Despite this, the novel approach to SISR was not very well documented and did not have many research papers backing it up or describing it in thorough enough detail. Therefore, our development mainly stemmed from reviewing architecture and open source code and seeing what could be learned and used for our implementation.

We set out this project with the goals of:

- Learning a new language, Python [45]
- Learning how to integrate Keras and custom define a Neural Network
- Use this knowledge to implement a GAN, 2 networks competing against each other
- From the back of this, try and implement a novel approach to SISR

These were all monumental goals in hindsight and hindered the ability to do things quickly and iteratively. Also, the fact that GANs as a topic are relatively new (first coming to light in 2014 [13]) meant that help was limited, although a few GitHub links proved extremely useful in providing guidance.

- https://github.com/eriklindernoren/Keras-GAN
- https://github.com/tensorlayer/srgan
- https://github.com/deepak112/Keras-SRGAN
- https://github.com/aitorzip/PyTorch-SRGAN

3.4.2 Computational resources

Before the start of this project, we were aware of how intensive the training of Neural Networks can be and procured a new machine with a dedicated GPU in the hopes we could train on it to vastly reduce the time required. However, after searching through TensorFlow documentation and downloading various Cuda drivers for NVIDIA, we found that the graphics card we had was one of the few that was unsupported for TensorFlow GPU despite being a newer GeForce model. The issue outline forced the project to be trained purely on a laptop CPU which undoubtedly vastly increased training times and limited us on what we could and could not train for. Arbitrarily, we calculated if training times scaled linearly, then 1000 epochs with 350,000 images would take over 50 years.

3.4.3 Debugging

The process of debugging neural networks is extremely difficult because of their intricate layers, especially when using high-level APIs like TensorFlow [36] or Keras [35], as unlike PyTorch there is more of a 'black-box' view of the network. The approach we took towards debugging and optimization was aided greatly by Jupyter Notebooks. Its cell-by-cell flow allowed us to develop each part of each network iteratively and make sure they functioned together. We focused heavily on the images being parsed through the network correctly and each part working in tandem; then we puzzled through why the outputs were not returning as anticipated.

Chapter 4

Results

After seeing the effects that the number of residual blocks had on time to train (as discussed in 3.3.2), we decided to alter the batch size of 200 training images. We did this because we thought with only 40 images, although the speed was desirable, there would be a largely visible bias as the image spread was only of one subgroup (flowers); 200 images for training had four subgroups. In all following image figures, they are organised into groups of 3 where:

- The leftmost image is the original image downsampled, the LR image
- The middle image is the image generated by the network from the LR image, the SR image
- The rightmost image is the ground-truth image, the HR image

4.1 Altering batch size

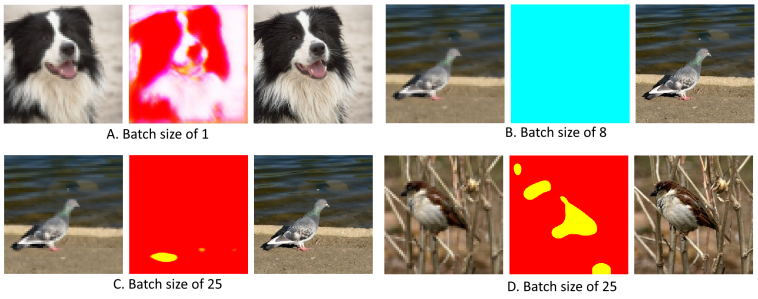


Figure 9: Altering batch sizes for 200 training images

We had originally decided on larger batch sizes as we knew there was a limit on time available for the project. We were also unable to use a GPU for training which slowed progress down massively. Figure 9 shows the best results for one epoch were returned from a batch size of 1, which was tested last and, in hindsight, should have been checked against first. The bottom left, and bottom right images show results from the first epoch versus the tenth epoch, with a training set of 200 images; this meant there was around a five-hour wait for not much visible improvement.

4.2 Dataset alterations

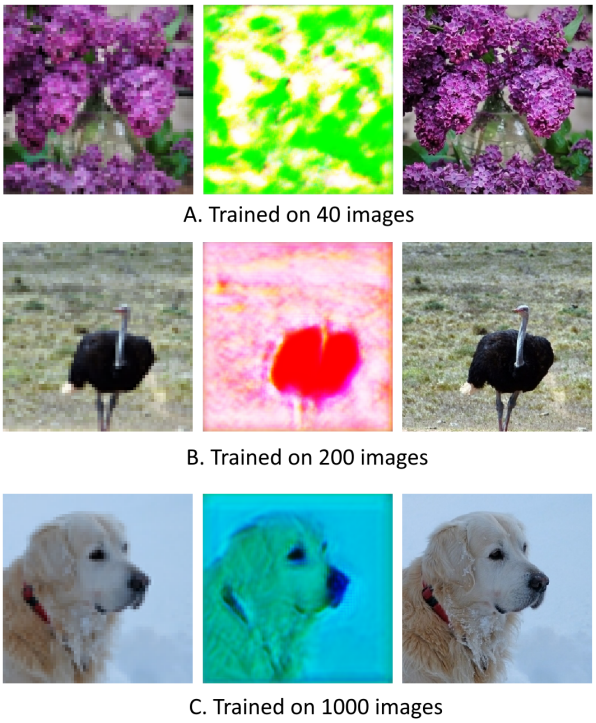


Figure 10: Showing the results after one epoch for various training set sizes

As shown in Figure 10, it is obvious that image C returned the most accurate result after one training epoch, followed by a 200 image training set and lastly the set with only 40 images. However, when we compare this with results from 3.3.2 we can see that the length of time for 16 residual blocks takes over 4 times longer for 1000 images. We decided it was pertinent to test both for more epochs to weigh up the loss of time against accuracy.

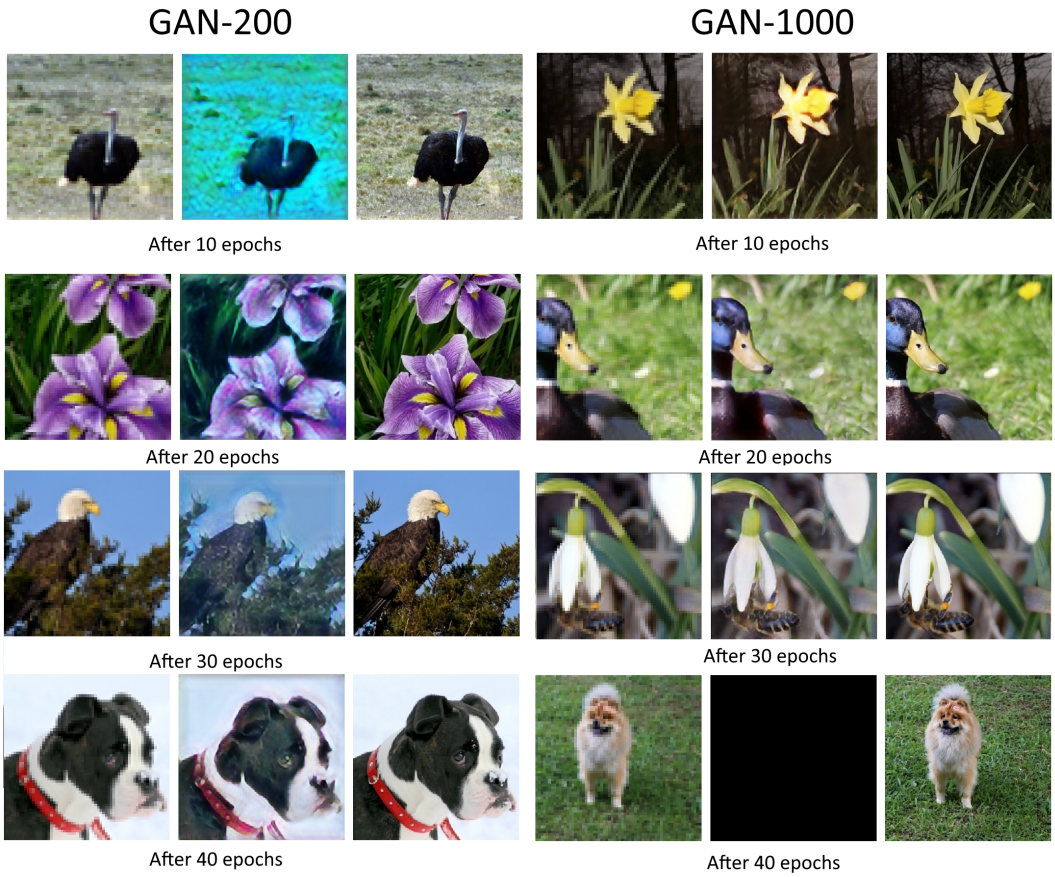
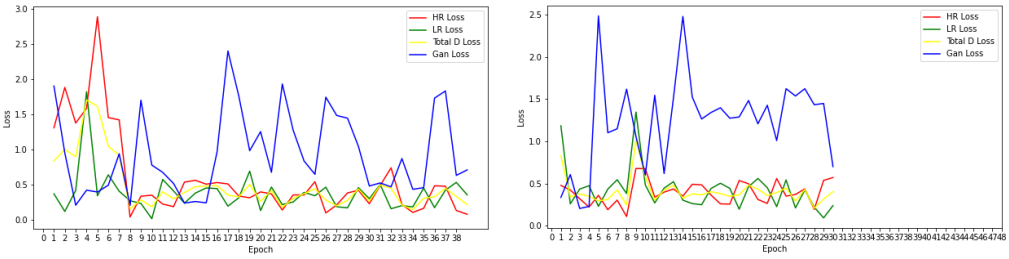


Figure 11: Differences between 200 training images (GAN-200) and 1000 training images (GAN-1000) over various epochs

From this graphic we can see that after 30 epochs GAN-1000 delivers a more accurate output than GAN-200 after 40 epochs. Despite this, they are both very good quality images. 200 images took about 18 hours to train 40 times whereas 1000 images took about 5 days for 50 epochs.

As shown above, after 40 epochs the output was a black square for 1000 training images (see Figure A.7). This was due to the loss functions attempting to back-propagate a NaN (not a number); this is further demonstrated in the comparison of loss values (Figures A.9 and A.10).



GAN-200: The loss over 40 epochs

GAN-1000: The loss over 30 epochs

Figure 12: Smaller version of loss graphs shown in Figure A.8

This error is shown in Figure 12 as on GAN-1000 the graph stops recording values at epoch 30. The anomaly is verified by the fact that for GAN-200 training continued well past epoch 30 and losses were still being updated.

Seen above:

- **Gan Loss** - is the total loss of the network
- **LR Loss** - is the loss of the discriminator network when compared with the SR images from the generator
- **HR Loss** - is the loss of the discriminator network when comparing the HR image with the ground truth
- **Total D Loss** - is the total loss of the discriminator network obtained by addition of the two losses defined above and multipled by half

GAN Loss had a high variance despite increasing epoch number but you can clearly see a steady overall decline in the peak heights. Comparing this with LR and HR Loss one can see that they follow each other quite closely after 10 epochs despite starting at varying levels on epoch 1. This figure shows that overall the network is training in the correct manner and converging towards improvement. With more time it is apparent this would have stayed adversarial with the differences getting smaller over time.

4.3 Architectural adjustments

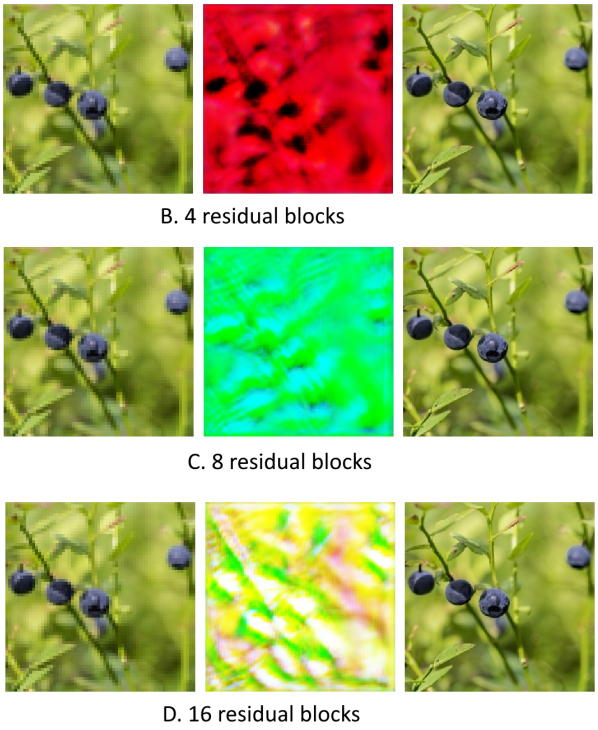


Figure 13: Altering number of residual blocks for 1 epoch

Figure 13 shows the development and improvements made to the generated image after one epoch with differing numbers of residual blocks. Two residual blocks held no comparable similarity and for this reason was not included in the graphic. As seen in the figure, there are vague similarities of general shapes of the image being retained. The core differences being that more residual blocks tend towards the true colour of the ground truth image.

Chapter 5

Conclusion and Future Work

5.1 Result findings

As seen from the table in 3.3.2, we can break down the training times, and when cross-compared with 4.3 we saw that 16 residual blocks yielded the best results early for the highest accuracy. Then when viewing 4.2, it was visible that 1000 training images yielded end results closer to the ground truth and got progressively more accurate more quickly, the main drawback is that it took over 4 times the time necessary for its 200 training image counterpart. We believed this happened for one of two reasons:

- The dataset with 200 images was simply too small, and the learning that occurred could not compare to 1000 images. This is backed by [8] and [9]
- The loss function could not adjust enough for this smaller dataset

5.2 Project Management

Comparing A.11 and A.12 it is clear that project's development evolved from what was originally planned in our initial document.

As seen in A.12 much more time had to be used for learning and debugging. We neglected to give enough margin for error and time for learning a completely new framework and therefore had to focus mainly on the initial aims; as redefined in 1.2.

Implementing a basic GAN went to plan but took far longer than expected due to using varying methods (including deprecations) from TensorFlow. As there was not much support

available online due to this concept still being in relative infancy, it took more time than anticipated to fix these bugs, especially while using a previously unused program language. The use of Jupyter helped as anticipated; code being executed cell-by-cell was more easily fixed, but with large networks being combined finding intricate issues within the networks proved difficult. Coupled with training the network on a CPU slowed the project down and meant extended goals could not be attempted. After the completion of this project we have a greater understanding of how we underestimated the scale and complexity of a seemingly small project and how difficult optimisation would be. In the future, we plan to give more time for the programming and planning of the network; hopefully with this cleared efficiently and with enough margin for error training will occur more smoothly. This was identified as a high probability risk in the risk analysis defined in our initial document and seen in Table B.2.

5.3 Future Work

Due to the scale of this project relative to time and complexity, we have a keen interest to explore the concept further and with more depth:

- **Change the training to a GPU base as opposed to CPU for increased speed**
  We would be able to train for a greater amount of epochs with larger training sets. This would allow us to have output images generated far closer to the ground truth, from here we could plot graphs and understand how the loss functions work in tandem to update the network.

- **Exploration of the loss error propagation**
  As discussed in 4.2 an anomalous error occurred whilst training on 1000 images, due to time restrictions we were unable to test why this occurred. Returning to this project in the future this would be the first priority to understand and fix.

- **Much larger dataset with greater variance on images**
  This change would allow us to test with any image and hopefully see super-resolution results regardless of the content.

- **Training the network on much higher resolutions**
  As in [6] explore the option of a network being able to output full HD images or higher quality, for example, 2k or 4k.

- **Exploration of different loss functions**
  With this we could explore MSE versus Perceptual Loss ([8] [18]) and maybe explore more novel loss functions like [32].

- **Extended goal of a graphical user interface**
  We would like to have a trained model sitting behind a basic web application and allow users from across the globe to input custom images they would want resolving to a higher resolution.

5.4 Final Conclusion

In this project we successfully implemented a Generative Adversarial Network to provide a novel solution for the problem of Image Supersampling. We used a GAN instead of a CNN or interpolation algorithm to see if the output provided closer generations to the ground truth. After reviewing work related to our project and solutions derived from CNNs and interpolation we are pleased with the outcome. Despite long training times (this being due to the performance of ML training on a CPU) the accuracy retrieved from iterations as early as the 3rd epoch were converging accurately on the ground truth values. This was while using training sets that were a fraction of the size as recommended in literature of a similar nature. Previously having no experience with adversarial networks and deep learning problems, we have finished the project with a GAN that outputs images from 4,096 pixels to 65,536 pixels with a high degree of feature reconstruction comparable to the HR counterpart. We have developed stronger Python programming knowledge and have gained a deep and fundamental knowledge of machine learning as a topic with a clearer understanding of adversarial networks. Although the network had a few unclear issues that we could not solve by the end of the project, we observed very unique and interesting output and can see a clear future progression for the project should it be continued.

Bibliography

[1] Z. Wang, J. Chen, and S. C. H. Hoi, "Deep learning for image super-resolution: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2020. [Online]. Available: https://arxiv.org/pdf/1902.06068.pdf

[2] W. Yang, X. Zhang, Y. Tian, W. Wang, J. Xue, and Q. Liao, "Deep learning for single image super-resolution: A brief review," *IEEE Transactions on Multimedia*, vol. 21, no. 12, pp. 3106–3121, 2019. [Online]. Available: https://arxiv.org/pdf/1808.03344.pdf

[3] L. Lan, L. You, Z. Zhang, Z. Fan, W. Zhao, N. Zeng, Y. Chen, and X. Zhou, "Generative adversarial networks and its applications in biomedical informatics," *Frontiers in public health*, vol. 8, p. 164. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7235323/

[4] A. Hayat and D. Erb, "Building a simple generative adversarial network using tensorflow," Dec 2020. [Online]. Available: https://blog.paperspace.com/implementing-gans-in-tensorflow/

[5] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *CoRR*, vol. abs/1812.04948, 2018. [Online]. Available: http://arxiv.org/abs/1812.04948

[6] M. Chu, Y. Xie, L. Leal-Taixé, and N. Thuerey, "Temporally coherent gans for video super-resolution (tecogan)," *CoRR*, vol. abs/1811.09393, 2018. [Online]. Available: http://arxiv.org/abs/1811.09393

[7] A. Burnes, "Nvidia dlss 2.0: A big leap in ai rendering," Mar 2020. [Online]. Available: https://www.nvidia.com/en-gb/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/

[8] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network," *CoRR*, vol. abs/1609.04802, 2016. [Online]. Available: http://arxiv.org/abs/1609.04802

[9] T. Karras, M. Aittala, J. Hellsten, S. Laine, J. Lehtinen, and T. Aila, "Training generative adversarial networks with limited data," 2020.

[10] K. Gurney, *An Introduction to Neural Networks*. USA: Taylor Francis, Inc., 1997.

[11] T. Yiu, "Understanding neural networks," Aug 2019. [Online]. Available: https://towardsdatascience.com/understanding-neural-networks-19020b758230

[12] J. Brownlee, "A gentle introduction to cyclegan for image translation." August 2019. [Online]. Available: https://machinelearningmastery.com/what-is-cyclegan/

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

[14] A. Singh, "Fulfillment lies in the creating something," Nov 2018. [Online]. Available: https://towardsdatascience.com/fulfillment-lies-in-the-creating-something-d118db307405

[15] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," 2018. [Online]. Available: https://arxiv.org/pdf/1611.07004.pdf

[16] J. Brownlee, "A gentle introduction to generative adversarial networks (gans)," July 2019. [Online]. Available: https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

[17] ——, "A gentle introduction to stylegan the style generative adversarial network." May 2020. [Online]. Available: https://machinelearningmastery.com/introduction-to-style-generative-adversarial-network-stylegan/

[18] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," 2016.

[19] J. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," *CoRR*, vol. abs/1703.10593, 2017. [Online]. Available: http://arxiv.org/abs/1703.10593

[20] H. Bansal and A. Rathore, "Understanding and implementing cyclegan in tensorflow." [Online]. Available: https://hardikbansal.github.io/CycleGANBlog/

[21] P. Parsania and D. Virparia, "A comparative analysis of image interpolation algorithms," *IJARCCE*, vol. 5, pp. 29–34, 01 2016.

[22] A. Prajapati, S. Naik, and S. Mehta, "Evaluation of different image interpolation algorithms," *International Journal of Computer Applications*, vol. 58, 10 2012.

[23] P. Parsania and D. V.Virparia, "A review: Image interpolation techniques for image scaling," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 02, pp. 7409–7414, 01 2015.

[24] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *CoRR*, vol. abs/1501.00092, 2015. [Online]. Available: http://arxiv.org/abs/1501.00092

[25] Y. Zhang, Y. Tian, Y. Kong, B. Zhong, and Y. Fu, "Residual dense network for image super-resolution," *CoRR*, vol. abs/1802.08797, 2018. [Online]. Available: http://arxiv.org/abs/1802.08797

[26] J. Wimpee, "Finding nash equilibria in two-player, zero sum games," 2008.

[27] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *CoRR*, vol. abs/1606.03498, 2016. [Online]. Available: http://arxiv.org/abs/1606.03498

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

[29] J. Yang, J. Wright, T. S. Huang, and Y. Ma, "Image super-resolution via sparse representation," *IEEE Transactions on Image Processing*, vol. 19, no. 11, pp. 2861–2873, 2010.

[30] S. Gu, W. Zuo, Q. Xie, D. Meng, X. Feng, and L. Zhang, "Convolutional sparse coding for image super-resolution," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1823–1831.

[31] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 184–199.

[32] X. Zhu, L. Zhang, L. Zhang, X. Liu, Y. Shen, and S. Zhao, "Generative adversarial network-based image super-resolution with a novel quality loss," in *2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, 2019, pp. 1–2.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[34] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90. [Online]. Available: https://jupyter.org/documentation

[35] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[37] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[38] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[39] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015, cite arxiv:1511.06434Comment:

Under review as a conference paper at ICLR 2016. [Online]. Available: http://arxiv.org/abs/1511.06434

[40] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," 2016.

[41] W. Shi, J. Caballero, L. Theis, F. Huszar, A. Aitken, C. Ledig, and Z. Wang, "Is the deconvolution layer the same as a convolutional layer?" 2016.

[42] Z. Ding, X. Liu, M. Yin, W. Liu, and L. Kong, "TGAN: deep tensor generative adversarial nets for large image generation," *CoRR*, vol. abs/1901.09953, 2019. [Online]. Available: http://arxiv.org/abs/1901.09953

[43] J. Kim, J. K. Lee, and K. M. Lee, "Accurate image super-resolution using very deep convolutional networks," *CoRR*, vol. abs/1511.04587, 2015. [Online]. Available: http://arxiv.org/abs/1511.04587

[44] H. Liu, Z. Ruan, P. Zhao, C. Dong, F. Shang, Y. Liu, and L. Yang, "Video super resolution based on deep learning: A comprehensive survey," 2020.

[45] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

Appendix A

Supplementary Data

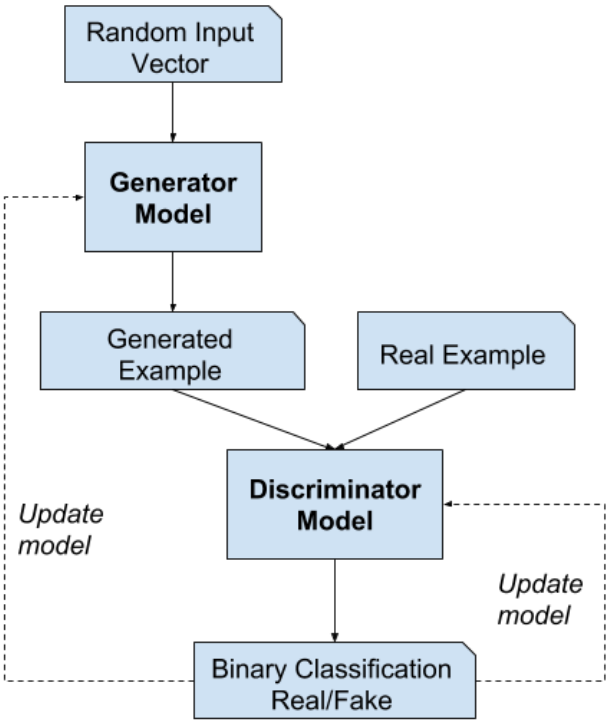


Figure A.1: Basic architecture of a GAN [16]

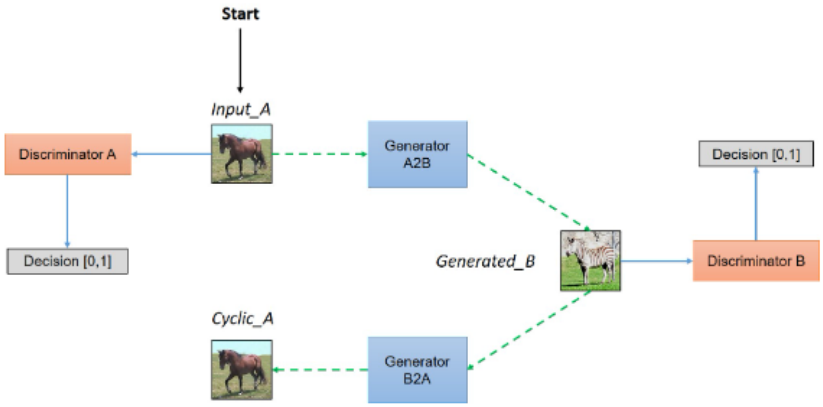


Figure A.2: Basic architecture of a CycleGAN with input into a discriminator, with the input being the original image for that domain [20]

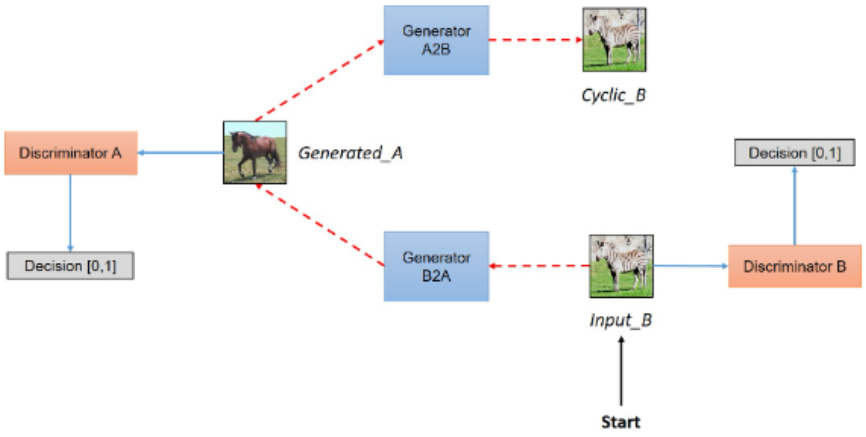


Figure A.3: Basic architecture of a CycleGAN with input into the other discriminator, this time the input is the generated image [20]

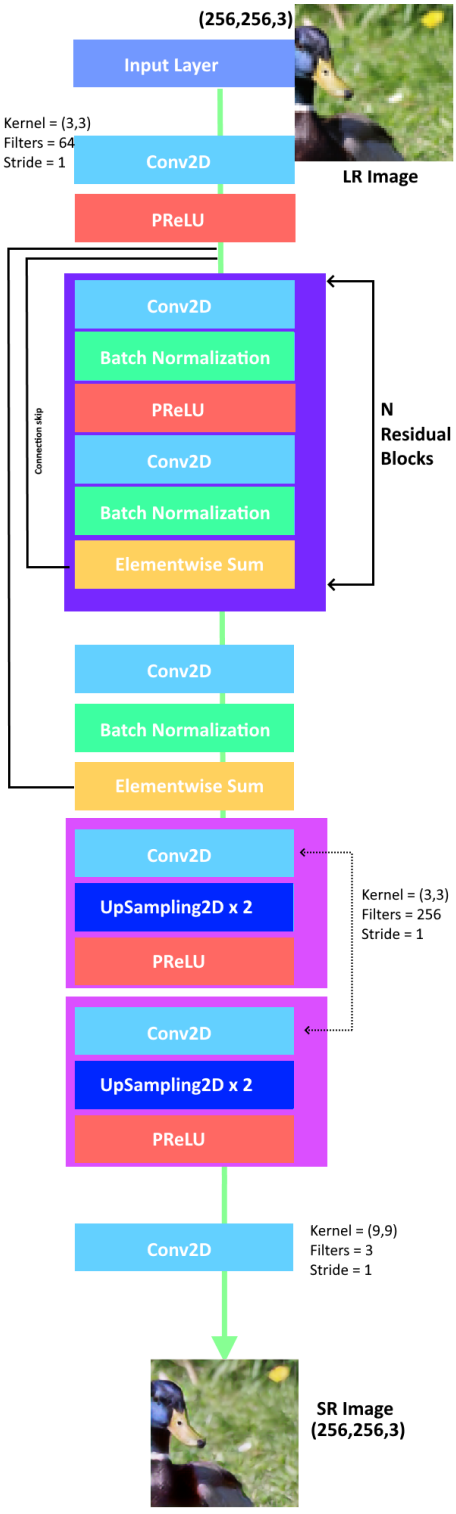


Figure A.4: Architecture for the Generator

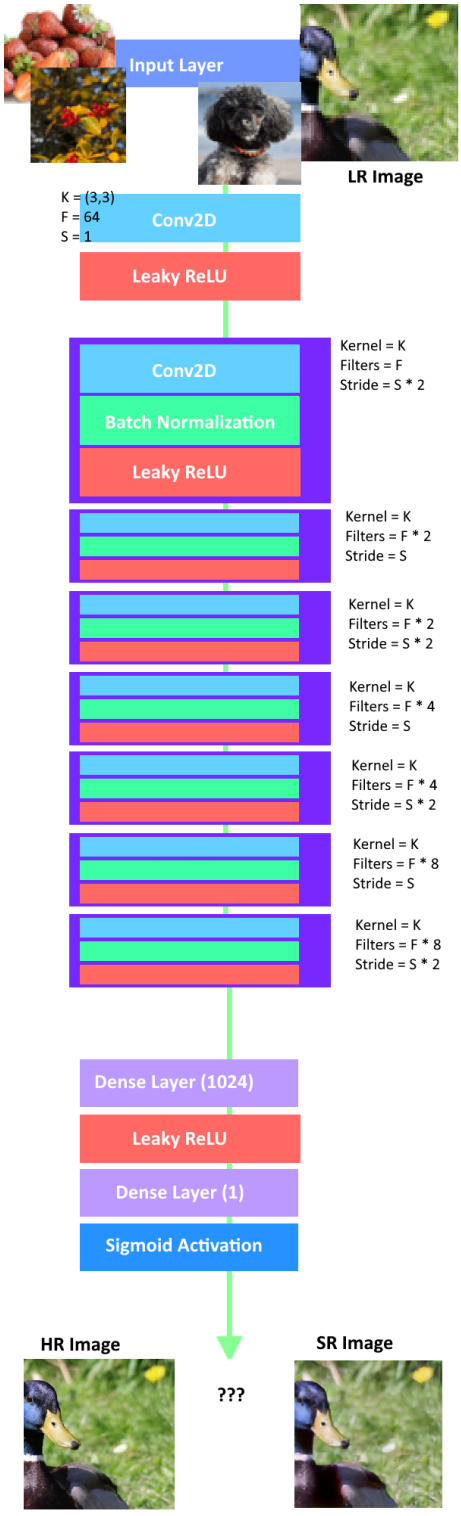


Figure A.5: Architecture for the Discriminator

```
==========================================================================================
Total params: 1,001,731
Trainable params: 1,001,091
Non-trainable params: 640
------------------------------------------------------------------------------------------
```

2 blocks

```
==========================================================================================
Total params: 1,150,595
Trainable params: 1,149,443
Non-trainable params: 1,152
```

4 blocks

```
==========================================================================================
Total params: 1,448,323
Trainable params: 1,446,147
Non-trainable params: 2,176
------------------------------------------------------------------------------------------
```

8 blocks

```
==========================================================================================
Total params: 2,043,779
Trainable params: 2,039,555
Non-trainable params: 4,224
```

16 blocks

Figure A.6: Changes to number of residual blocks affecting the trainable parameters

Figure A.7: Anomaly caused by NaN backpropagation

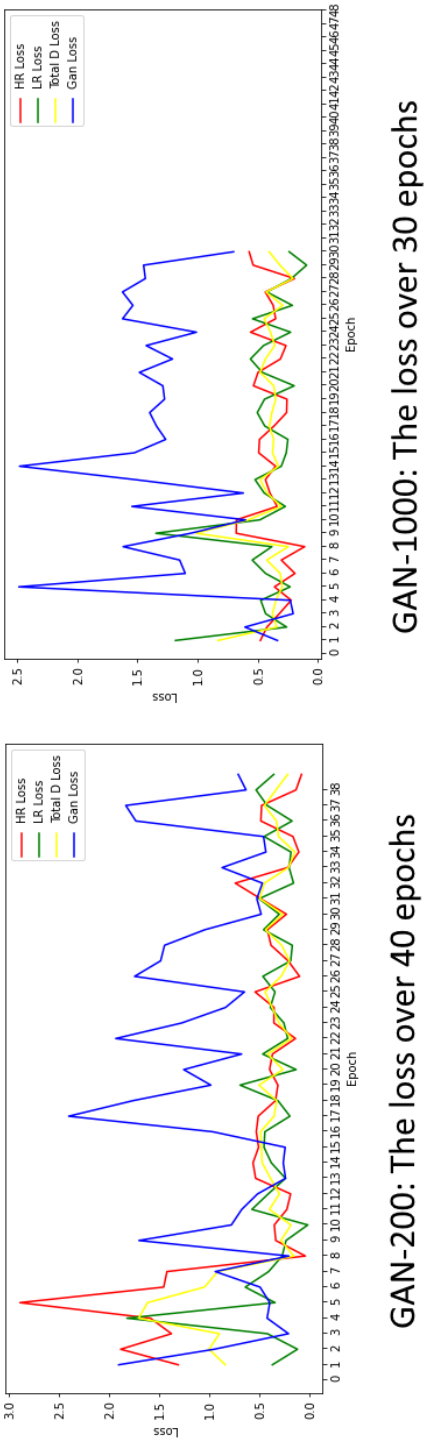


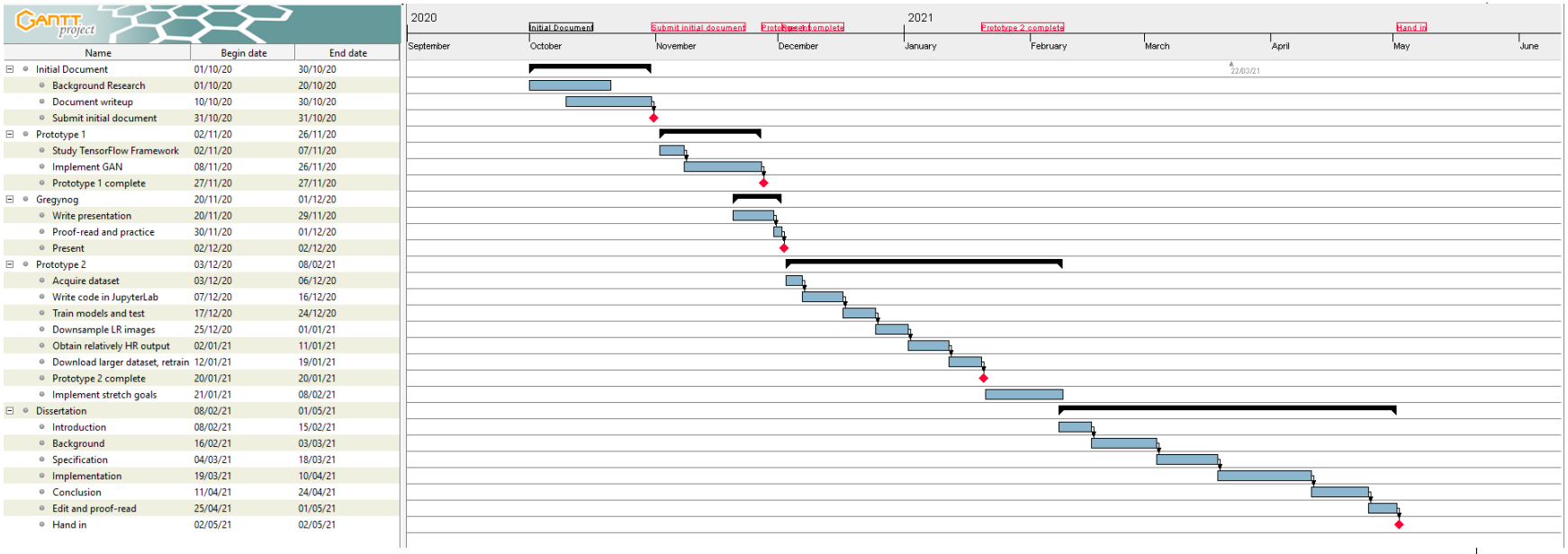
Figure A.8: Relative loss graphs

	A	B	C	D	E	F	G	H	I
1	Epoch	Loss for High Resolution	Loss for Low Resolution	Total Loss	Loss for the GAN				
2	(1	1.311210632	0.372448713	0.841829672	'['	0.063673988	0.063483506	1.904828072	']')
3	(2	1.88561058	0.122034758	1.003822669	'['	0.05467039	0.054575112	0.95277077	']')
4	(3	1.378902435	0.421673089	0.900287762	'['	0.084687077	0.084665954	0.211238787	']')
5	(4	1.588974833	1.823729873	1.706352353	'['	0.05268411	0.05264163	0.424805701	']')
6	(5	2.890505791	0.345064431	1.617785111	'['	0.051502515	0.051462695	0.398210257	']')
7	(6	1.456363082	0.643013597	1.049688339	'['	0.05830805	0.058258489	0.495613635	']')
8	(7	1.423799396	0.409618586	0.916708991	'['	0.040068302	0.039974056	0.942444801	']')
9	(8	0.043125026	0.27448386	0.158804443	'['	0.037283711	0.037262902	0.20810613	']')
10	(9	0.338801891	0.236235231	0.287518561	'['	0.073076114	0.072905712	1.704056978	']')
11	(10	0.353255183	0.019141788	0.186198485	'['	0.027445968	0.027367814	0.781539321	']')
12	(11	0.227681637	0.576719701	0.402200669	'['	0.060215715	0.060147896	0.678191006	']')
13	(12	0.190592408	0.413465738	0.302029073	'['	0.032153241	0.03210124	0.520007372	']')
14	(13	0.539223433	0.241552547	0.39038799	'['	0.030368814	0.030344695	0.241189852	']')
15	(14	0.565688252	0.382839262	0.474263757	'['	0.027864655	0.027838271	0.263849646	']')
16	(15	0.509824455	0.456063777	0.482944116	'['	0.03596868	0.035944186	0.244952396	']')
17	(16	0.53298676	0.447380364	0.490183562	'['	0.040344864	0.040248059	0.968047261	']')
18	(17	0.513425708	0.197492808	0.355459258	'['	0.035162292	0.034921899	2.403936863	']')
19	(18	0.341328055	0.316687316	0.329007685	'['	0.039382581	0.039206322	1.762568235	']')
20	(19	0.317149043	0.694803834	0.505976439	'['	0.026105586	0.026006989	0.985981643	']')
21	(20	0.400134027	0.13548249	0.267808259	'['	0.02847941	0.028353736	1.256740093	']')
22	(21	0.371468186	0.46912238	0.420295283	'['	0.03424843	0.034180503	0.679257631	']')
23	(22	0.141760677	0.219655693	0.180708185	'['	0.011922117	0.01172863	1.934869885	']')
24	(23	0.357335597	0.257276595	0.307306096	'['	0.033547789	0.033420138	1.276517153	']')
25	(24	0.355926484	0.393324137	0.37462531	'['	0.020440778	0.020356812	0.839659035	']')
26	(25	0.545266807	0.345956415	0.445611611	'['	0.024070077	0.024005022	0.650550365	']')
27	(26	0.101128571	0.467558205	0.284343388	'['	0.023030529	0.022855844	1.74684155	']')
28	(27	0.215189993	0.189385593	0.202287793	'['	0.031238362	0.031089665	1.486967087	']')
29	(28	0.385818303	0.173464835	0.279641569	'['	0.010155186	0.01001039	1.447960138	']')
30	(29	0.422339469	0.460516632	0.44142805	'['	0.023206403	0.023101723	1.046801805	']')
31	(30	0.232084215	0.303901672	0.267992944	'['	0.020269843	0.020221442	0.484011114	']')
32	(31	0.490364999	0.504368722	0.497366861	'['	0.019441916	0.019389167	0.527490973	']')
33	(32	0.742650688	0.161220089	0.451935388	'['	0.027086085	0.027038915	0.47169587	']')
34	(33	0.22048074	0.205346182	0.212913461	'['	0.01694547	0.016858075	0.873956144	']')
35	(34	0.108432971	0.186881095	0.147657033	'['	0.016351933	0.016308211	0.437213928	']')
36	(35	0.166951805	0.454287648	0.310619727	'['	0.01919529	0.019149225	0.460659891	']')
37	(36	0.487092316	0.175744921	0.331418619	'['	0.011628892	0.011455517	1.733747005	']')
38	(37	0.480335563	0.424570322	0.452452943	'['	0.01012598	0.009942492	1.83486855	']')
39	(38	0.136816025	0.535433531	0.336124778	'['	0.010878519	0.010815022	0.634971917	']')
40	(39	0.081582911	0.358235836	0.219909374	'['	0.011733562	0.01166225	0.713112593	']')

Figure A.9: Numerical loss values for training on 200 images

	A	B	C	D	E	F	G	H	I
1	Epoch	Loss for High Resolution	Loss for Low Resolution	Total Loss	Loss for the GAN				
2	(1	0.476638794	1.182846427	0.82974261	'['	0.056929063	0.056896	0.334201	']')
3	(2	0.422192335	0.25775978	0.339976057	'['	0.039589822	0.039529	0.605089	']')
4	(3	0.318999708	0.433069587	0.376034647	'['	0.05409408	0.054074	0.203505	']')
5	(4	0.220150232	0.473222911	0.346686572	'['	0.0541844	0.054162	0.22776	']')
6	(5	0.359356761	0.228901967	0.294129364	'['	0.044145621	0.043897	2.486346	']')
7	(6	0.188817218	0.433178157	0.310997687	'['	0.026096176	0.025986	1.101299	']')
8	(7	0.302155852	0.544280171	0.423218012	'['	0.016260291	0.016145	1.148014	']')
9	(8	0.107752047	0.380721182	0.244236615	'['	0.025154237	0.024992	1.618021	']')
10	(9	0.677587926	1.346182346	1.011885136	'['	0.026294477	0.026189	1.055892	']')
11	(10	0.675223231	0.480594039	0.577908635	'['	0.016254442	0.016194	0.599752	']')
12	(11	0.337415725	0.268808007	0.303111866	'['	0.016387586	0.016233	1.54521	']')
13	(12	0.396925926	0.444540501	0.420733213	'['	0.009348005	0.009286	0.618302	']')
14	(13	0.434540093	0.520890892	0.477715492	'['	0.022480702	0.022329	1.517854	']')
15	(14	0.349840194	0.302253962	0.326047078	'['	0.020325376	0.020077	2.480106	']')
16	(15	0.490207285	0.261038572	0.375622928	'['	0.018150697	0.017998	1.524899	']')
17	(16	0.48369801	0.247855052	0.365776531	'['	0.020254213	0.020128	1.264016	']')
18	(17	0.362660199	0.440110415	0.401385307	'['	0.004054883	0.003921	1.341102	']')
19	(18	0.258387268	0.502426684	0.380406976	'['	0.011326198	0.011186	1.397027	']')
20	(19	0.255221218	0.441922098	0.348571658	'['	0.007767741	0.00764	1.27347	']')
21	(20	0.534244657	0.195107669	0.364676163	'['	0.018409122	0.01828	1.287789	']')
22	(21	0.496102005	0.461054325	0.478578165	'['	0.005649056	0.005501	1.48305	']')
23	(22	0.309864789	0.559283018	0.434573904	'['	0.007448888	0.007328	1.207427	']')
24	(23	0.263174236	0.45007214	0.356623188	'['	0.00845738	0.008315	1.426957	']')
25	(24	0.558242142	0.22509259	0.391667366	'['	0.009383942	0.009283	1.008312	']')
26	(25	0.349479705	0.541698933	0.445589319	'['	0.01831218	0.01815	1.623189	']')
27	(26	0.36954838	0.21044144	0.28999491	'['	0.01500115	0.014847	1.536923	']')
28	(27	0.435956925	0.422307312	0.429132119	'['	0.007129985	0.006968	1.623283	']')
29	(28	0.192384541	0.218663424	0.205523983	'['	0.00400053	0.003857	1.433381	']')
30	(29	0.536460459	0.091736287	0.314098373	'['	0.01005899	0.009914	1.44818	']')
31	(30	0.569819629	0.236990541	0.403405085	'['	0.00798837	0.007918	0.699832	']')
32	(31	nan	nan	nan	'['	nan	nan	nan	']')
33	(32	nan	nan	nan	'['	nan	nan	nan	']')
34	(33	nan	nan	nan	'['	nan	nan	nan	']')
35	(34	nan	nan	nan	'['	nan	nan	nan	']')
36	(35	nan	nan	nan	'['	nan	nan	nan	']')
37	(36	nan	nan	nan	'['	nan	nan	nan	']')
38	(37	nan	nan	nan	'['	nan	nan	nan	']')
39	(38	nan	nan	nan	'['	nan	nan	nan	']')
40	(39	nan	nan	nan	'['	nan	nan	nan	']')
41	(40	nan	nan	nan	'['	nan	nan	nan	']')
42	(41	nan	nan	nan	'['	nan	nan	nan	']')
43	(42	nan	nan	nan	'['	nan	nan	nan	']')
44	(43	nan	nan	nan	'['	nan	nan	nan	']')
45	(44	nan	nan	nan	'['	nan	nan	nan	']')
46	(45	nan	nan	nan	'['	nan	nan	nan	']')
47	(46	nan	nan	nan	'['	nan	nan	nan	']')
48	(47	nan	nan	nan	'['	nan	nan	nan	']')
49	(48	nan	nan	nan	'['	nan	nan	nan	']')
50	(49	nan	nan	nan	'['	nan	nan	nan	']')

Figure A.10: Numerical loss values for training on 1000 images



Name	Begin date	End date
Initial Document	01/10/20	30/10/20
Background Research	01/10/20	20/10/20
Document writeup	10/10/20	30/10/20
Submit initial document	31/10/20	31/10/20
Prototype 1	02/11/20	26/11/20
Study TensorFlow Framework	02/11/20	07/11/20
Implement GAN	08/11/20	26/11/20
Prototype 1 complete	27/11/20	27/11/20
Gregynog	20/11/20	01/12/20
Write presentation	20/11/20	29/11/20
Proof-read and practice	30/11/20	01/12/20
Present	02/12/20	02/12/20
Prototype 2	03/12/20	08/02/21
Acquire dataset	03/12/20	06/12/20
Write code in JupyterLab	07/12/20	16/12/20
Train models and test	17/12/20	24/12/20
Downsample LR images	25/12/20	01/01/21
Obtain relatively HR output	02/01/21	11/01/21
Download larger dataset, retrain	12/01/21	19/01/21
Prototype 2 complete	20/01/21	20/01/21
Implement stretch goals	21/01/21	08/02/21
Dissertation	08/02/21	01/05/21
Introduction	08/02/21	15/02/21
Background	16/02/21	03/03/21
Specification	04/03/21	18/03/21
Implementation	19/03/21	10/04/21
Conclusion	11/04/21	24/04/21
Edit and proof-read	25/04/21	01/05/21
Hand in	02/05/21	02/05/21

Figure A.11: Original Gantt Chart

Name	Begin date	End date
Initial Document	01/10/20	30/10/20
Background Research	01/10/20	20/10/20
Document writeup	10/10/20	30/10/20
Submit initial document	31/10/20	31/10/20
Prototype 1	02/11/20	04/12/20
Study TensorFlow Framework	02/11/20	15/11/20
Understand GAN	16/11/20	04/12/20
Review similar projects	05/12/20	05/12/20
Gregynog	20/11/20	01/12/20
Write presentation	20/11/20	29/11/20
Proof-read and practice	30/11/20	01/12/20
Present	02/12/20	02/12/20
Prototype 2	03/12/20	14/04/21
Acquire dataset	03/12/20	12/12/20
Write code in JupyterLab	13/12/20	06/01/21
Debug code	07/01/21	15/02/21
Downsample LR images	16/02/21	23/02/21
Obtain relatively HR output	24/02/21	05/03/21
Begin Training	06/03/21	29/03/21
Prototype 2 complete	30/03/21	30/03/21
Train various architectures	06/03/21	14/04/21
Dissertation	04/04/21	27/04/21
Introduction	04/04/21	07/04/21
Background	08/04/21	13/04/21
Implementation	14/04/21	18/04/21
Conclusion	19/04/21	21/04/21
Edit and proof-read	22/04/21	27/04/21
Hand in	28/04/21	28/04/21
Project Presentation	30/04/21	30/04/21

Figure A.12: Updated Gantt Chart

Appendix B

Supplementary Tables

	Minutes to train for 1 epoch for X residual blocks			
Training set size	**2 blocks**	**4 blocks**	**8 blocks**	**16 blocks**
40 images	5	6	8	12
200 images	12	18	25	30
1000 images	40	60	90	130
1200 images	45	80	130	170

Table B.1: Changes to dataset sizes causing training time increases

Risk Analysis			
Risk	Impact	Probability	Description
Training network takes longer than anticipated due to throttling	High	Medium	Run a smaller dataset as a prototype to see if there is any throttling or not, if there is throttling on a smaller dataset then move training to Google Colab. The impact of this is high as if not addressed early will be detrimental to progress on implementation. Whereas the probability of this is medium, although my laptop has a dedicated NVIDIA GPU with the latest machine learning tensorcores in it, it is not the newest graphics card and subsequently may struggle.

Dataset far larger than anticipated and slows everything down	High	Low	When developing the code, use a dataset in the magnitude of 10^3 samples and make sure everything works as intended. The impact of this is high as it will massively slow down development if starting with a dataset too large. The probability is low, as for the project just start on a smaller dataset. Start with a dataset of HR images with resolution around 256×256 pixels and downsample towards 64×64 pixels. Once thoroughly tested and debugged, slowly rotate towards a larger dataset with higher resolution images.
Loss of project due to data corruption	High	Low	Make sure that data is backed up in multiple places, cloud and physical. Although this will get more difficult with a large dataset have consistent backups throughout making use of Git, Microsoft OneDrive and an external HDD. The impact is high as data loss and subsequent project loss would mean failure of this module, probability is low as I often back up my computer.

Lose dataset and have to download again	High	Low	Similar reasoning as above, a loss of the dataset would not be as major as losing the whole project; although depending on bandwidth could be very time consuming redownloading the set of samples again. As above, make sure at least one backup is readily available and has the dataset that is being used.
Training network takes longer than anticipated due to bugs in code	Medium	Medium	Medium impact because it will slow development down significantly, scaling as the dataset and project size scale. Medium probability as bugs in code are frequent. This leads to a conscious decision for using Jupyter for development and debugging due to segmented code, then if an automated testing suite is required migrate the debugged code to PyCharm .
Catching COVID-19	Medium	Low	Medium impact as I don't know how badly this will affect me, it could range from hospitalisation to just a few days out of commission. Low probability as my household is on the whole self-isolating and very good at wearing masks.

Dropping my laptop and breaking it	Medium	Low	Medium impact as my laptop is necessary for the project but I do have a backup, whilst my current laptop is under premium warranty. Low probability as the computer rarely leaves my desk but this would cause a stall in my working, to prevent this I will be extra careful whilst carrying my laptop around.

Table B.2: Risk Analysis from initial document