Generation of Content Through Antagonic Generative Networks

Generación de contenido mediante Redes Generativas Antagónicas

By
García Patiño Lenza, Guillermo
Alejandra Córdova, Daniela
Quiñones Pérez, Mario

Trabajo de fin de grado del Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Directed by
Antonio Alejandro Sánchez Ruiz-Granados

Madrid, 2021-2022

Abstract

This work is used to explore machine learning methods for generating images, in particular, we explore the Generative Adversarial Networks (or GANs) and how these are used to generate new synthetic samples for datasets. This project is mainly focused on the generation of images through different techniques. Most of our work consists of understanding these fairly new models and implementing them ourselves, to then use them to generate images and test how well each of them works.

Most machine learning methods need large sums of examples for them to understand how to fully operate in most cases, this is where GANs shine, as they can create new samples for these preestablished datasets that may not contain enough. Not only that but also the fact that this task requires a thorough understanding of the dataset makes the GAN an excellent judge for that dataset.

This research has led us to understand the strengths and shortcomings of each sort of model that we produced, as well as how the next one improved on it while giving up something in return, which in most cases was time because these types of models take a long time to provide useful findings that can then be analyzed.

Because these content generators aren't perfect, they cannot be employed in operations that need a high level of precision. However, this establishes a foundation from which other types of content generators can grow and evolve, leveraging their strengths to build stronger models. Another issue with these models is that the created content can only copy current ones and cannot develop new ones, resulting in bias.

In a nutshell, this study serves as a foundation for understanding current breakthroughs in generative machine learning algorithms and how they are used to generate new material from an existing dataset using GANs.

KeyWords

Style Transfer, GAN, Image Generation, Neural Networks, Machine Learning, Deep Learning, Artificial Intelligence

Resumen

Este trabajo se utiliza para explorar los métodos de aprendizaje automático de generación de imágenes, en particular exploramos las Redes Generativas Adversarias (o GANs) y cómo éstas se utilizan para generar nuevas muestras sintéticas para conjuntos de datos. Este proyecto se centra principalmente en la generación de imágenes mediante diferentes técnicas. La mayor parte de nuestro trabajo consiste en comprender estos modelos novedosos e implementarlos nosotros mismos, y así luego utilizarlos para generar imágenes y probar su funcionalidad.

La mayoría de los métodos de aprendizaje automático necesitan grandes sumas de ejemplos para entender cómo funcionan completamente en los distintos casos, aquí es donde las GANs brillan, ya que pueden crear nuevas muestras para estos conjuntos de datos preestablecidos que pueden no contener suficientes. No sólo eso, sino que el hecho de que esta tarea requiera un conocimiento profundo del conjunto de datos por parte del modelo, hace que el GAN sea un excelente juez para ese conjunto de datos.

Esta investigación nos ha llevado a comprender los puntos fuertes y débiles de cada tipo de modelo que hemos producido, así como la forma en que el siguiente lo mejoraba renunciando a algo a cambio, que en la mayoría de los casos era el tiempo, porque este tipo de modelos necesitan mucho tiempo para poder proporcionar resultados útiles que puedan ser analizados.

Como estos generadores de contenidos no son perfectos, no pueden emplearse en operaciones que necesiten un alto nivel de precisión. Sin embargo, esto establece una base a partir de la cual otros tipos de generadores de contenido pueden crecer y evolucionar, aprovechando sus puntos fuertes para construir modelos más sólidos. Otro problema de estos modelos es que los contenidos creados sólo pueden copiar los actuales y no pueden desarrollar otros nuevos, lo que provoca un sesgo.

En pocas palabras, este estudio sirve de base para comprender los avances actuales en los algoritmos de aprendizaje automático generativo y cómo se utilizan para generar nuevo material a partir de un conjunto de datos existente utilizando GANs.

Palabras Clave

Transferencia de estilos, GAN, Generación de Imagenes, Redes Neuronales, Aprendizaje automatico, Aprendizaje profundo, Inteligencia artificial

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There have been major advances in artificial intelligence, particularly machine learning, over the past decade. These breakthroughs have been made mostly because of the increase in processing capacity over time, as anticipated by Moore's law in 1965, which projected that the number of transistors in a given surface would double every year, as illustrated in Figure 1.1.

This increase in computational power has allowed the implementation of many techniques that had been considered previously, but had not been tested to their full potential due to the lack of computational power. For example, although multilayer perceptrons were proven to be able to approach any mathematical function, by the 1980s they were outclassed by support vector machines due to the inefficiency in the training process. Thanks to advancements in efficiency and computational power, multilayer perceptrons have grown into far more complicated structures like the ones presented in this work.

This increase in computation power has been used in recent years to solve mostly classification and regression problems using a variety of algorithms and models, in which the model must learn either to differentiate a sample between every class available or how the features of a given sample are related to another in order to predict its value. However, the aim of this work is to solve a much more complex problem. The models to be investigated in this work must understand not only the distinctions between the classes to which a sample can belong but also the exact attributes that essentially characterize each conceivable class. For example, while differentiating a cat from a dog is relatively simple, defining the specific characteristics of a dog and a cat is a considerably more difficult process. These models learn those specific properties to generate synthetic samples that incorporate the learning of the model.

There has been a lot of progress in recent years when it comes to creating videos, photos, and even music. Deepfakes are great examples of these advances: images or films of individuals that have been artificially made by transferring the facial expressions and gestures from one video to another, resulting in videos of people saying things they did not actually say. Other studies worth examining are NVIDIA's GauGAN [32], which can transform a segmentation map into a realistic image, and OpenAI's Dall E [37], which can generate faithful images based on a textual description provided by the user. Figures 1.3(a), 1.3(b) and 1.4 illustrate examples of what these two models can do.

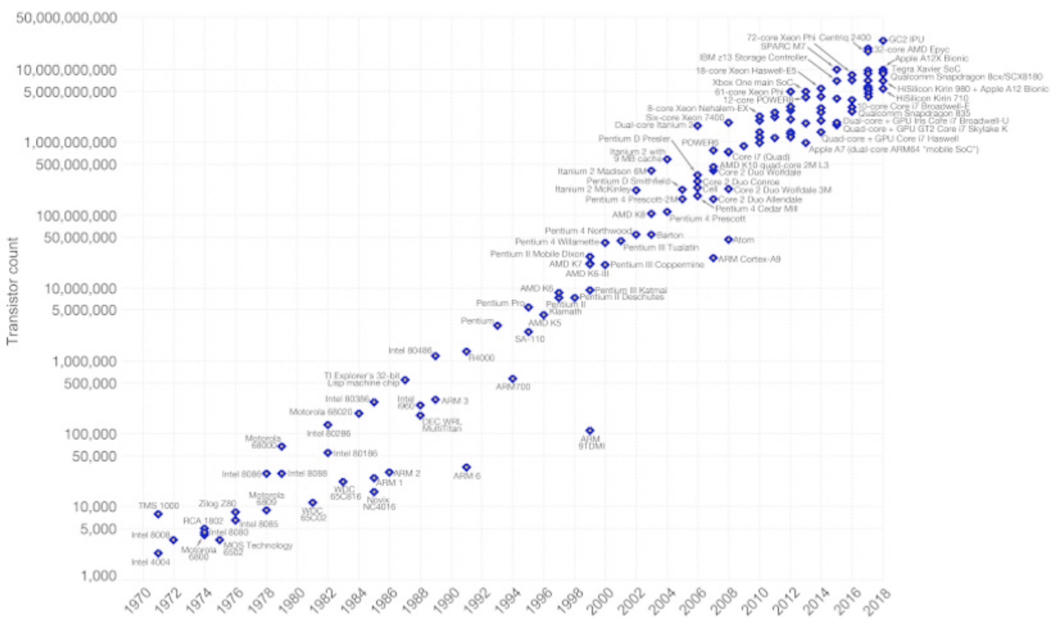


Figure 1.1: Growth of the transistor count over time

This is why image processing specialists are increasingly paying attention to the potential of generative adversarial networks (GANs). Image scaling, image shifting between domains (e.g. transitioning from daylight to night scenes), and many other applications benefit greatly from the usage of GANs in image production. To achieve these results, many modified GAN architectures have been developed with their own distinct properties for solving certain image processing challenge, although the baseline always stays the same.

In a GAN, two agents fight against each other: a Generator and a Discriminator, shown in Figure 1.2. The Generator produces an image that tries to mimic a real one, and then, that image is fed to the Discriminator, so it determines whether the image generated by the Generator is authentic or not. Initially, the Generator will produce low-quality images that the Discriminator will immediately identify as fake. Thanks to the Discriminator's decision, the Generator will learn to trick the Discriminator after collecting enough information, while the Discriminator will learn what a real image looks like by processing several real images. As a consequence, the generative model ends up producing highly realistic outcomes.

1.1 Objectives

- **O1:** Learn how to apply deep learning to generative processes.
- **O2:** Study the theory behind generative adversarial networks.
- **O3:** To develop a GAN able to produce synthetic samples within a very simple context.
- **O4:** To provide a GAN with means to control its output.

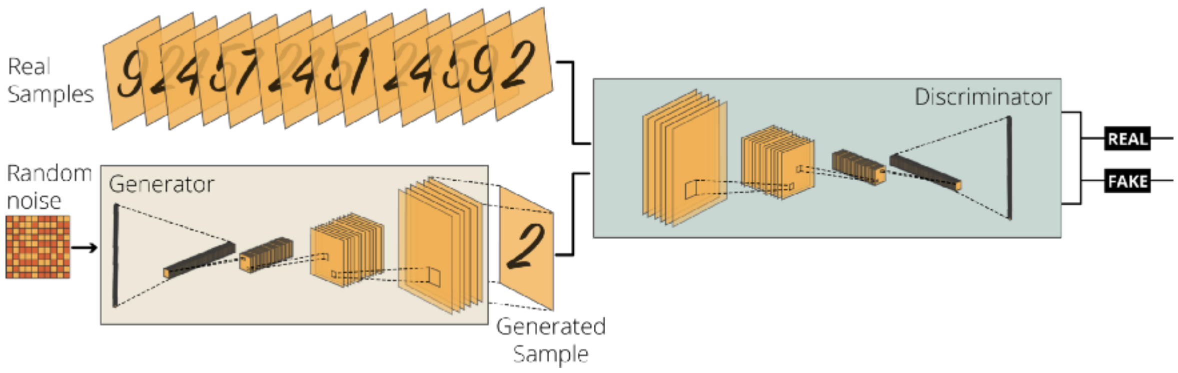
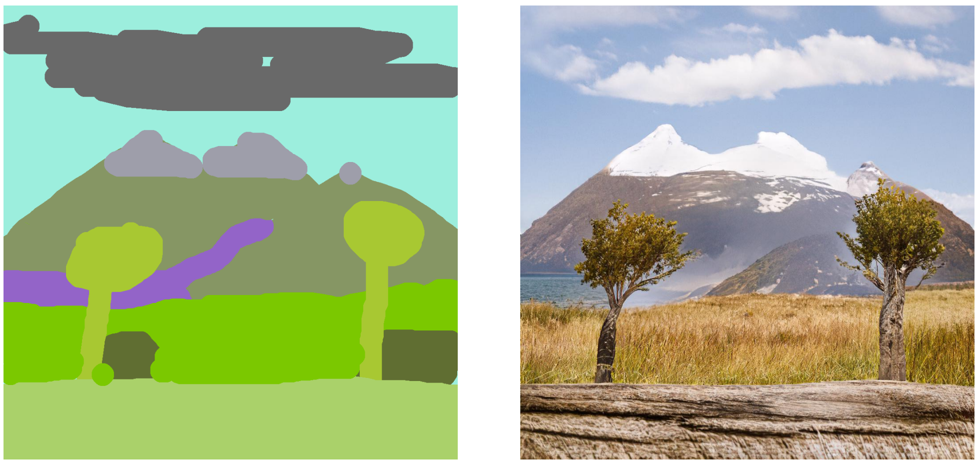


Figure 1.2: GANs (Generative Adversarial Networks) based on CNNs. Author: Yalçın, Orhan G [41].



(a) Input to the GAU GAN (b) GAU GAN's output

Figure 1.3: GAU GAN input and output example taken from [1]

- **O5:** Improve the quality of the generated samples.
- **O6:** Increase the control over the features of the generated samples.
- **O7:** Implement a GAN able to transfer features from one sample to another taking into account the context of both samples.

1.2 Road Map (Work Plan)

The work plan followed to achieve the objectives set was:

- During the first two months, the plan was to learn the basics about GANs and generative models. To that end, we watched OpenAI's specialization in GANs in coursera [39] and gathered some books and articles.
- Once we knew about the basics of GANs, we spent about a month trying to implement a single GAN using books like [18] as a reference.

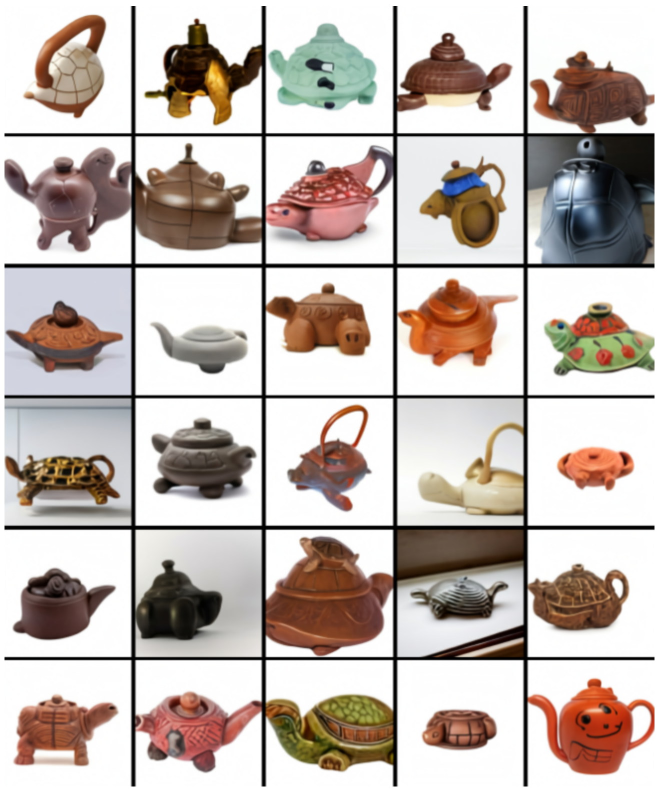


Figure 1.4: Dall E's output corresponding the input "A teapot imitating a turtle" taken from [4]

- Finally, during the next 5 months, we attempted to construct the four GAN architectures presented in this study and train them to produce the most faithful images possible. We also collected data on the training and made charts with it during the process.
- In addition, when implementing and developing the architectures, we created sketches, schemes, and tables, as well as writing this text. We concentrated on this effort, especially during the last two months, when there was little to do with the models.

Throughout this process, we met with the tutor of this work every two weeks to check on progress and set goals for the next two weeks. Furthermore, those discussions occasionally helped us solve crucial difficulties and gain new ideas from the instructor, allowing us to keep moving forward with the development of this project.

In addition, we used several tools to help us work as a team.

- During the first stages of the project, we used Google Drive to share the notes we took on the books and papers we read and on the coursera videos we watched. Additionally, we used Google Drive as a way to store links to different knowledge sources we found on the internet, so everyone in the team was able to access them easily.
- When we first started working on a GAN, we utilized Google Collab notebooks to conveniently share the code we developed. At the time, Google Collab was the

appropriate platform because it already provided a Python virtual environment with all the packages and modules we required, as well as GPUs and TPUs that let us run our code considerably quicker than any CPU.

- Later, once we had written a substantial amount of code, Google Collab became difficult to work with due to the increasing size of the notebook files, so we moved our code to a GitHub repository.

1.3 Memory structure

On chapter 2, we will introduce what knowledge is needed in order to understand the following chapters. Everything from what a neural network is to how they are used and for what purpose in our project will be described in this chapter.

The models we developed and how they work would be covered in the next four chapters. The chapter 3 discusses image generation and how it is accomplished. Additionally, this chapter presents a model that is able to generate simple images applying every concept so far.

The chapter chapter 4 relates a more advanced model that allows not only to control the features of the created samples, but to generate higher quality images by extending the input information.

Next, chapter 5 outlines techniques to increase the quality of the generated samples and to largely increase the degree of control over the generated sample's features to a point where it is possible to select the features that make it to the final sample.

The final model is shown in chapter 6, which relates a model capable of transferring features from one picture to another, placing the first image's style into a second image and vice versa.

On chapter 7, we sum up the conclusions we reached with this project, taking into account the objectives that are set on this very chapter. Furthermore, more lines of work will be proposed for future projects that may look to expand off this one.

1.4 Code

All the code used in this project can be found here:

GitHub Code(https://github.com/DanielaCordova/TFG—GANs).

1.4.1 Github's Structure

The repository is structured the following way:

Each Model described in this document, and that constitutes each chapter, is stored on its own folder, these folders contain 4 main files:

- ModelGenerator.py : this contains the class of the generator for said model.
- ModelDiscrimitor.py : the same as the previous, this file contains the discriminator for the models.
- trainingModel.py : this file contains the code capable of training these models, all the training details can be changed within it, such as the size of the image generated, the dataset used for its training, and if needed the previously trained model that it loads for it to resume its training from a previous run, among others.
- generateSamples.py : this file contains the code capable of generating samples from a previously trained model. Like the last file, within it, you can change the aspects of the generated images to an extent, as it has to match the generator that it is using.
- generateStyleMixing.py: this file is unique to the StyleGAN and given two sets of images it is able to mix their style using a pre-trained model.

The main folder additionally contains a number of files used throughout all the models:

- Constants.py: Which has the constant values that the other files can use.
- CustomLayers.py: Implements custom layers for both the Generators and Discriminators of each model.
- Blocks.py: Which has the declaration of most types of blocks that the models Generators and Discriminators are made of.
- ImageFunctions.py: Which implements all the logic that is used for handling images and datasets.
- Training.py: This implements all the functions needed for training each model.

Chapter 2

Generative Adversarial Networks

In this part, we will introduce the knowledge needed for understanding the rest of the document. This chapter would be divided into three sections:

- The first section, which provides an intuition on neural networks and tackles every component necessary to build one.
- Then, the second section delves deeper into a specific neural network architecture called generative adversarial networks, first giving an overview on them, and then explaining key concepts specific to this architectures.
- Finally, the last section introduces the frameworks and libraries used in this project.

2.1 Neural Networks

These are computing systems with interconnected nodes that work much like neurons in the human brain. Using algorithms, they can recognize hidden patterns and correlations in raw data, cluster and classify it, and – over time – continuously learn and improve.

Neural networks [7] are also ideally suited to help people solve complex problems in real-life situations. They can learn and model the relationships between inputs and outputs that are nonlinear and complex; make generalizations and inferences; reveal hidden relationships, patterns, and predictions; and model highly volatile data (such as financial time series data) and variances needed to predict rare events (such as fraud detection). As a result, neural networks can improve decision processes in areas such as text and voice recognition, medical and disease diagnosis, targeted marketing, financial predictions, or computer vision.

2.1.1 How Neural Networks Work

A simple neural network includes an input layer, an output (or target) layer, and, in between, a hidden layer. The layers are connected to each other by feeding the output of one to the input of another, and these connections form a “network” – the neural network – of interconnected nodes.

The simplest neural network node is called a perceptron. It consists of a single node that receives many inputs, and only one output. That output is calculated firstly by calculating a linear combination between the inputs and a vector of weights that are stored in the perceptron, and secondly by feeding the output of that linear combination to an activation function. The perceptron is patterned after a neuron in a human brain, so both of them behave similarly. Nodes are activated when there are sufficient stimuli or input. This activation spreads throughout the network, creating a response to the stimuli (output). The connections between these artificial neurons act as simple synapses, enabling signals to be transmitted from one to another. Signals traverse layers as they travel from the first input to the last output layer, and get processed along the way.

When posed with a request or problem to solve, the neurons run mathematical calculations to figure out if there’s enough information to pass on the information to the next neuron. Put more simply, they read all the data and figure out where the strongest relationships exist. In the simplest type of network, data inputs received are added up, and if the sum is more than a certain threshold value, the neuron “fires” and activates the neurons it’s connected to. This way, the computer learns to recognize patterns to solve the problem that it’s trying to be solved.

As the number of hidden layers within a neural network increases, deep neural networks are formed. Deep learning architectures take simple neural networks to the next level. Using these layers, data scientists can build their own deep learning networks that enable machine learning, which can train a computer to accurately emulate human tasks, such as recognizing speech, identifying images, or making predictions.

2.1.2 Types of Layers

Fully connected Layers

Fully connected layers are the most basic layers in a neural network. They represent each of the layers of a traditional multilayer perceptron model. Each of these layers group a set of perceptrons which get their inputs and produce their outputs according to the following formula:

$$z_i^{[l]} = \sum w_i^{[l]} a_i^{[l-1]} \tag{2.1}$$

Activation Layers

In this chapter we will introduce the activation layers that will be needed throughout this work.

- **Sigmoid Fuction**

The sigmoid function (Figure 2.1(a)) is a non linear activation function that outputs a value between 0 and 1 given for any given input. This function is often used in the output layers of models used to classify samples in two different categories.

$$Sigmoid(t) = \frac{1}{1+e^{-t}} \tag{2.2}$$

- **Hyperbolic Tangent Function**

  The hyperbolic tangent function (Figure 2.1(b)) is very similar to the sigmoid function. While the sigmoid function returns values between 0 and 1, the tangent function returns values between -1 and 1, but they both have an 'S' shape. This function is preferred over the sigmoid function when it comes to predictive models [14].

$$Tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}} \tag{2.3}$$

- **ReLU Function**

  The ReLU function (Figure 2.1(c)) outputs a value between 0 and ∞ for any given input. Precisely, this function outputs a 0 for any negative value, and outputs what it is given as input for any positive value. This activation function has much more sensitivity than the sigmoid and tanh functions.

$$ReLU(x) = max(0, x) \tag{2.4}$$

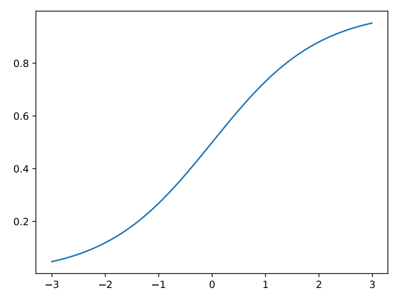
- **Leaky ReLU Function**

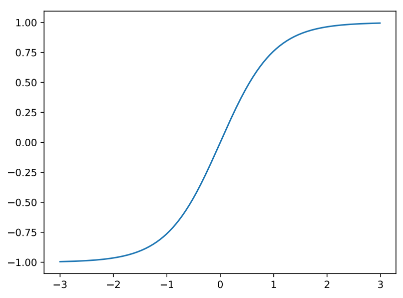
  The LeakyReLU (Figure 2.1(d)) function is similar to the ReLU function, but instead of outputting 0 for any negative value, this function outputs what it gets as an input scaled by a factor α which is usually set to 0.01. The reason to include this factor is to avoid the function having a 0 gradient for negative values so that it provides some feedback during backpropagation.

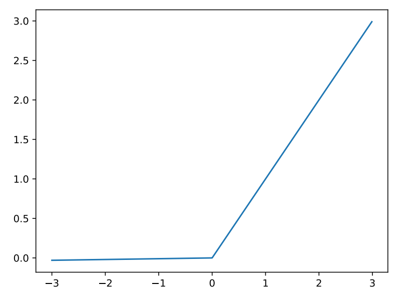
$$LeakyReLU(x) = max(\alpha x, x) \tag{2.5}$$

Convolutional Layers

A convolution is an operation performed over any tensor (a multidimensional array) to produce another tensor, although, in the context of this work, convolutions are always performed over an image (a 3-dimensional tensor). The way convolutions work is by sliding a filter (also called the kernel), which is just a square tensor with any amount of channels, across the input image and multiplying the filter and a slice of the image pixelwise, and then adding the results. For each time the filter is drawn across the input image, a scalar is generated to be part of the output image. More precisely, the position in the output image of a given scalar is determined by how many times the kernel has been slid across the image in each dimension.


(a) Sigmoid function


(b) Tanh function


(c) ReLU function

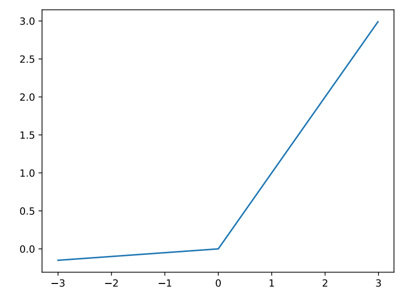

(d) LeakyReLU function

Figure 2.1: Loss Functions

For example, if the input data are color images, then each filter would have three channels (i.e. each having shape 3 × n × n) to match the three channels (red, green, blue) of the image

Additionally, to perform a convolution, it is necessary to understand what the stride and the padding parameter are.

- **Strides**

  The stride's parameter is the step size used by the layer to move the filters across the input. Increasing the stride, therefore, reduces the size of the output tensor. For example, when the stride is set up to two, the input tensor will be halved both in height and width.

- **Padding**

  The padding parameter stands for the size of the wrapping to be applied to the input data before performing the convolution itself. For example, a given image of size $3 \times 2 \times 2$ used as input for a convolutional layer set with padding one is first transformed into a $3 \times 4 \times 4$ image before calculating the convolution. The content of the wrapping can be set to anything from just ones, to a reflection of the input data or a slice of it repeated.

  Padding is usually used along with strides and filter size to set up the convolution to perform a certain action over the input image, such as upsampling or downsampling it.

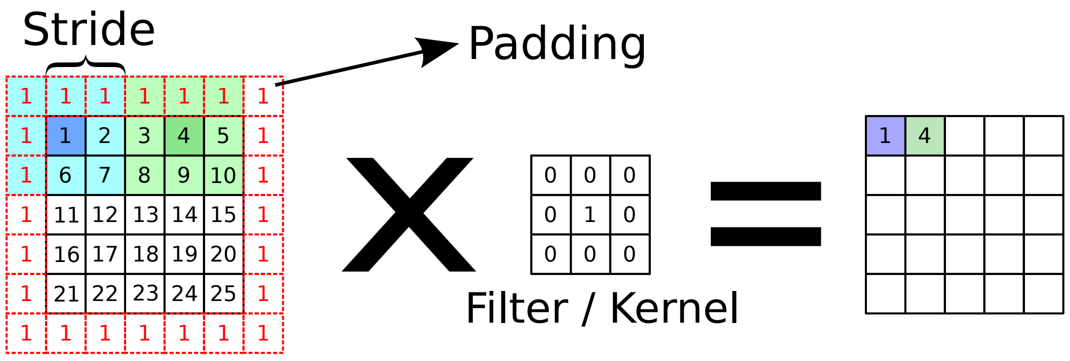


Figure 2.2: An image of a Convolutional layer with a 5 × 5 image , a kernel of 3 × 3 that its composed of zeros except the middle with a one, a padding of 1 and a stride of 3

So, to set up a convolution over a three dimensional input, it is necessary to specify several parameters: the number of channels of the input data, the number of channels desired in the output data, the kernel size, and the padding to be added to the image. Figure 2.2 shows the first two steps a convolutional layer takes when calculating a convolution over a 5 × 5 image using a filter of size 3 × 3, padding of 1, and a stride of 3.

Convolutional Transpose Layers

Convolutional transpose layers perform the opposite operation of convolutional layers. While convolutional layers were used to reduce the size of the input tensor, convolutional transpose layers serve the opposite purpose.

The way convolutional transpose layers increase the size of the input tensor is by adding padding data to the input tensor in two ways. First, padding data is injected between the input tensor data so that each component of the input tensor is separated from the other by a distance equal to the padding parameter in each dimension. Then, the input tensor is wrapped in more padding until the center of the filter can match the first component of the original input tensor. Finally, a traditional convolution is performed.

Pooling Layers

These layers are used to reduce the dimensions of a tensor using convolutions. Instead of multiplying the values in the kernel pixelwise with the image, the action performed per each stride of the kernel is either taking the average of the values on the kernel (Average Pooling) or their maximum value (Max Pooling). Figure 2.3 shows how a max pooling and an average pooling layer that use a kernel of size 2 × 2 and a stride of 2 operate over a 4 × 4 images.

Upsampling Layers

As an opposite to pooling layers, upsampling layers are used to increase the dimensions of a tensor using a known algorithm (linear, bilinear, nearest neighbor...) instead of using

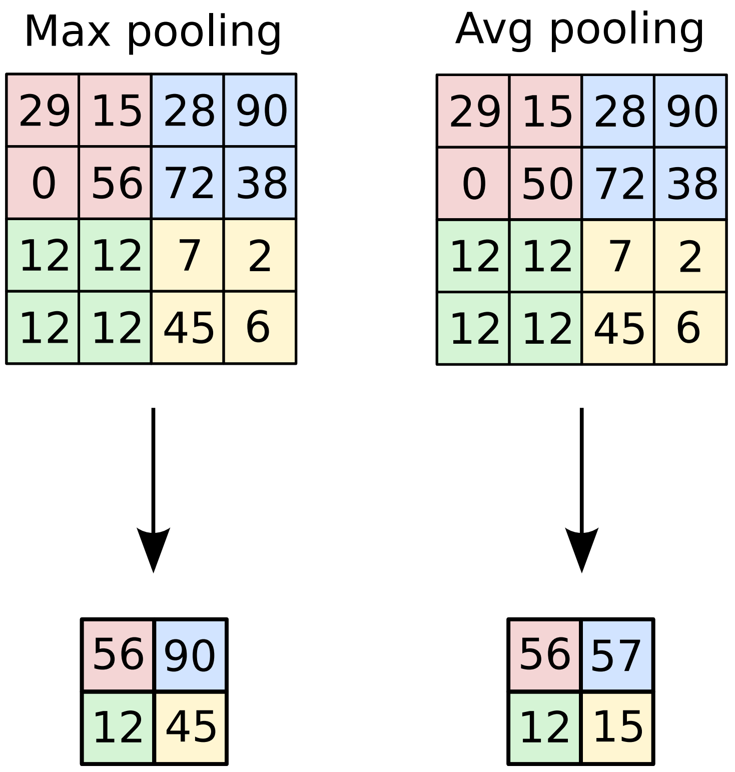


Figure 2.3: An example of a Max Pooling and an Average Pooling layers inputs and outputs. Both images use a kernel of 2 × 2 and a stride of 2. The Max pooling takes the maximum value of each stride of the kernel through the image of 4 × 4, while Avg pooling takes the average of the values from the stride

convolutions. Figure 2.4 shows the input and output of an upsampling layer over an image to double its size.

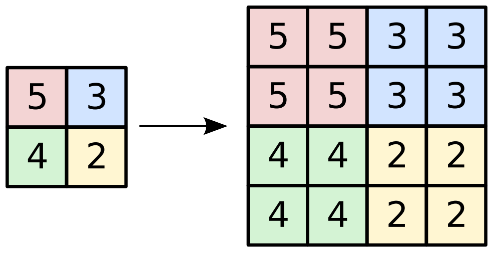


Figure 2.4: The effect of a nearest neighbor upsampling over a 2 × 2 image

Batch Normalization Layers

This layer uses the mean and the variance of the inputs in a certain node to normalize them. Using this kind of layer, a single sample can produce different results depending on the batch in which the sample is contained during testing. To avoid this issue, these layers gather the running mean and the variance of the different batches during training and then use them as a fixed value during testing.

$$\hat{y}_i^{[l]} = \frac{(z_i^{[l]} - \mu_{z_i^{[l]}})}{\sqrt{\sigma^2_{z_i^{[l]}} + \epsilon}} \tag{2.6}$$

$$y_i^{[l]} = \gamma * \hat{y}_i^{[l]} + \beta \tag{2.7}$$

Instance Normalization Layer

Instance normalization layers normalize their input the same way batch normalization layers do. The only difference between these two layers is that while batch normalization acts upon the entire batch taking into account the running mean and standard deviation, instance normalization acts over a part of the input tensor. For example, when taking images (three dimensional tensors) as an input, these layers normalize each channel of each input tensor independently without caring about the rest of the batch or the other channels of the image.

2.1.3 Cost Functions

Binary Cross Entropy

Binary cross entropy (BCE) cost function measures the effectiveness of a model at classifying samples between two different classes. It is designed to evaluate the cost of a binary sorter that produces a real value between 0 and 1 for each input. If that is taken into account and then being $y_i = 1$ for samples in the first class and $y_i = 0$ for samples in the second class, the BCE cost function forces the model to produce a value closer to 0 for $\hat{y}_i$ if the input is a sample from the second class, and a value closer to 1 if the input is a sample from the first class. On the other hand, if the difference between the tag (y_i) and the output of the model ($\hat{y}_i$) maximizes, the BCE goes to infinity.

$$BCE = -\frac{1}{m}\sum(y_i * log(\hat{y}_i) + (1 - y_i) * log(1 - \hat{y}_i)) \tag{2.8}$$

Although the use of this cost function may lead to some training problems, the BCE cost function is one of the most commonly used functions in GANs training.

2.1.4 Training using gradient descent

In this work, the goal of the model is to produce an output for a given sample that matches the class the input sample belongs to, and that class is the information the model has at its disposal to learn.

Feeding both the model's output and the correct tag to the cost function, the model gets feedback on how accurate its output was and finally, the gradient descent algorithm indicates to the model how to update the weights in each node to perform better next time (score lower on the cost function). This process is described in a schematic way in figure 2.5

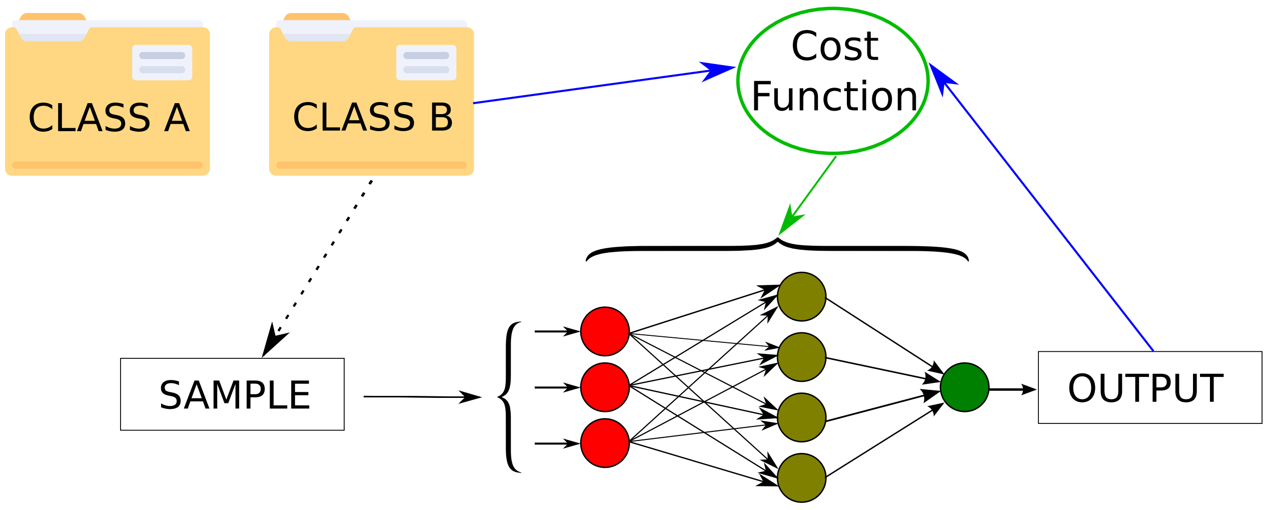


Figure 2.5: Supervised learning training step, in which the samples are passed through the network in order to get a result, then this results are compared to the samples that were used for the generation and the differences from the real samples and the output of the network are used as a cost function in order to update the weights of said network

Essentially, gradient descent calculates the partial derivatives of the cost function for each weight in the model and then evaluates those partial derivatives, thus obtaining a value (which is called the gradient) to update each weight.

This process has been refined and implemented in several algorithms provided by libraries such as pytorch [34] or tensorflow [3] to a point where each one offers a different trade-off between efficiency and speed.

The one used in the context of this work is the Adam optimizer [25] , provided by pytorch. It calculates the weights as shown in Equation 2.9 , using the gradients calculated using the weights used in the previous training step (g_t) and two set coefficients β_1 and β_2 usually set to a value close to 1.

The Adam optimizer is the one that optimizes the best cost function, but also the one that takes the most time to reach the minimum value of that function. Taking the equations above into account, the Adam optimizer updates the weights for the next iteration as shown in Equation 2.9.

$$
\begin{aligned}
&g_t = J(\theta_t) \\
&m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) \times g_t \\
&v_t = \beta_2 \times v_{t-1} + (1 - \beta_2) \times g_t^2 \\
&m_t = \frac{m_t}{1 - \beta_1^t} \\
&v_t = \frac{v_t}{1 - \beta_2^t} \\
&\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon} - \hat{m_t}
\end{aligned}
\tag{2.9}
$$

2.2 Generative Adversarial Networks

Generative adversarial networks (GANs) [15] are machine learning models whose task is to produce artificial samples similar to the ones used to train them. Having the model do a different task rather than classifying samples into two classes means that it is necessary to change certain aspects of both the neural network and the training process.

First, the model needs to become sort of an artist who is trying to produce a fake copy of, let's say, the Mona Lisa and then tries to sell it to an art critic. Imagine that the critic does not only determine if the painting the artist delivers is a fake one, but also tells the artist what to do to paint a more faithful copy. Then, the artist would end up painting perfect copies of the Mona Lisa, and the critic would have no other option but to randomly guess if what the artist delivers is either a fake copy or the real painting.

To be more specific, the GAN is made up of two submodels competing against each other, the generator, which plays the role of the artist, and the discriminator, which plays the role of the art critic. Although the artist-critic metaphor provides a clear idea of the interaction between the generator and the discriminator, there are many differences that are worth noting:

- Instead of creating a painting from scratch like the artist does, the generator needs a random noise vector as an input to produce a fake sample.
- Since the critic already knows what a real Mona Lisa looks like, the only one learning during the process is the artist. In contrast, at the beginning of the training of a GAN, neither the generator nor the discriminator has any kind of knowledge. During the training process, the generator has to learn using the feedback provided by the discriminator, and the discriminator has to do it by looking at real samples from a dataset.
- While it is quite clear that the discriminator ends up being a binary classifier, the task the generator performs is not so clear. To clarify the generator's task, it is necessary to think of the dataset used to train the GAN as a probability distribution that models the frequency in which certain features appear in real samples.

Taking this into account, what the generator is really doing is approximating the distribution in the dataset.

It is worth noticing the importance of the competition between the generator and the discriminator in the training process. On the one hand, if the discriminator never learned what a real sample looked like, the generator wouldn't learn how to produce faithful samples. On the other hand, if the generator didn't take advantage of the feedback provided by the discriminator, the duty of the discriminator would be too simple and there would be no point in training that model. Although the valuable part of a GAN ends up being the generator, the discriminator learning at a similar rate to the generator while being able to compete with the generator is the key to achieving a working GAN.

2.2.1 GAN architecture

The structure of the GAN is as shown in Figure 2.6, and the parts that compose it are as follows:

- **Training dataset**

  Dataset of real examples that we want to emulate with the Generator. It is the input to the Discriminator network.

- **Random noise vector**

  Vector of random numbers used by the Generator to synthesize fake examples. It is the input to the Generator network. The changes in this vector are used in order to increase diversity on the images generated by the Generator Network.

- **Generator network**

  This network takes in the random noise vector and outputs fake examples. Its goal is to generate fake examples as similar as possible to the real examples.

- **Discriminator network**

  This network takes in a real example (training set) or a fake example (Generator). For each example, the discriminator determinates the probability of whether the example is real.

2.2.2 Reaching a state of equilibrium

A GAN is fully trained when its reaches Nash equilibrium. The following requirements must be fulfilled:

- The Generator produces fake examples that are identical or almost identical to the real example in the training dataset.
- At best, the Discriminator can predict whether a particular case is real or fake at random (i.e., make a 50/50 guess).

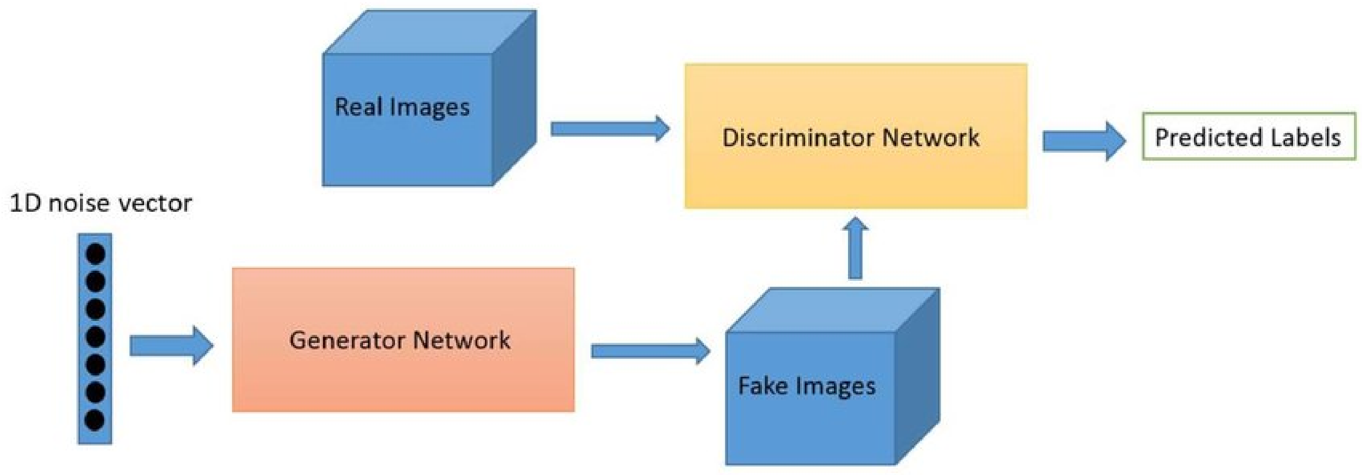


Figure 2.6: Simplified structure of a Generative Adversarial Network or GAN that uses both a generator, which takes a random noise and converts it into an image, and a discriminator which takes both the generated fake images and real images and discerns which ones are which

2.2.3 Evaluation

Given the task of generating fake samples of any kind, it is difficult to think of a way to measure how a model performs at it. On top of that, if the model needs not only to produce faithful copies of the original samples but also to generate a wide variety of them, the evaluation becomes even tougher. For example, it wouldn't be much of a success if a GAN trained to generate human faces only produced white male faces, and neither would be the desired result if the samples generated were diverse but fooled no human observer. Although the best case possible would be to have both high quality and diverse images, there is usually a trade-off between these two features, just as there is one between precision and recall when it comes to training a binary classifier.

In fact, the only module of the GAN that is being tested during the evaluation is the generator, while the discriminator is left behind. As previously mentioned, the generator's duty is to approximate the probability distribution that appears in the dataset used to train it. Taking that into account, it's easy to realize that evaluating a GAN equals measuring the difference between the distribution in the dataset and the one learned by the generator.

Inception score

The inception score (IS) [8] is a way of rating GANs based on the convolutional network named Inception. The rating is calculated feeding the same input to both models (the inception network and the model to rate), and comparing both outputs by either using cosine distance or by just taking the difference between the mean of the two outputs.

Fréchet inception distance

Fréchet inception distance (FID) [8] is a way to evaluate generative models that improve inception score. This measure originates from the Fréchet distance, which is named after the mathematician Maurice Fréchet. Essentially, the Frechét distance measures the distance between two given curves.

Intuitively, this distance can be thought of as the minimum leash a human walking forward along curve f needs to walk his dog, which walks forward along curve g if neither of them can walk backward.

As previously mentioned, the generator tries to model the distribution of the features of the real data, calculating the Frechét distance between the probability distribution produced by the model to rate and the produced by a reference model is a suitable measure to evaluate a generative model.

2.2.4 GAN Challenges

Oscillating Loss

This is the most common challenge in GAN training. When it comes to training a supervised learning model, it is important to ensure the training process converges to a point where the model performs correctly using the training data. In the context of generative adversarial networks, the training process consists of obtaining fake samples from the generator and using the feedback the discriminator provides on those fake samples and some real data to train both models. The nature of this process allows the generator and the discriminator to compete with each other to a point where let's say, the generator outperforms the discriminator while some training steps after, the discriminator outperforms the generator and neither of them performs better overall, thus the training of this neural networks hardly converges in a uniformed manner, as it can be seen in figure 2.7.

Uninformative Loss

As mentioned before, training a GAN actually consists of training the generator using as input a random noise vector and training the discriminator using both real and fake images produced by the generator. During this process, it is possible that the generator learns slower than the discriminator to a point where the discriminator differentiates perfectly the fake images from the real ones and stops learning. In this situation, the generator stops learning as well due to the discriminator performing perfectly and, thus, not giving any feedback to the generator.

Mode Collapse

When a GAN reaches mode collapse, some modes (for example, classes), are not well represented in the generated samples even though their frequency in the dataset proba-

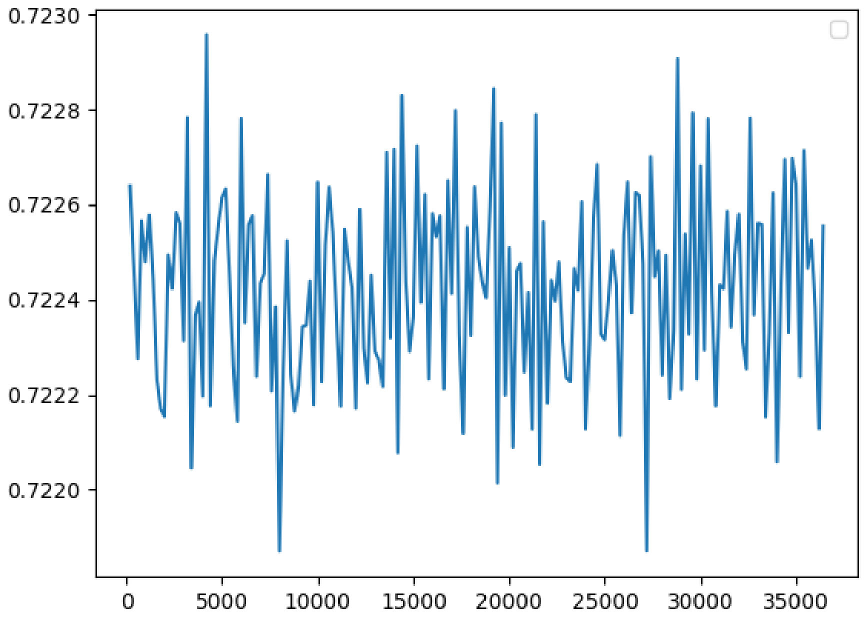


Figure 2.7: Image that shows how the loss function of the generator of one of our models doesn't converge, but instead oscillates, making the model not aware of how much it has learned

bility distribution is different from zero. For example, if a GAN that has been trained to generate human faces (using a dataset that is diverse enough) reaches mode collapse, the GAN may not generate Asian faces no matter how many samples it is asked to generate. Figure 2.8 shows the images created by a generator trained in a human faces dataset that has reached mode collapse.

Overgeneralization

Overgeneralization is kind of the opposite of Mode Collapse. Overgeneralization happens when a GAN learns things that should not exist based on real data. When a GAN overgeneralizes, modes (let's say modes are classes to match the previous example) that do not appear in the probability distribution of the dataset are present in the generated samples. For example, a cow with multiple bodies but only one head, or vice versa, may appear if a GAN trained to produce animal pictures overgeneralizes.

Slow convergence

This is a big problem with GANs and unsupervised settings, in generally the speed of convergence and available compute power are the main constraints—unlike with supervised learning, in which available labeled data is typically the first barrier.

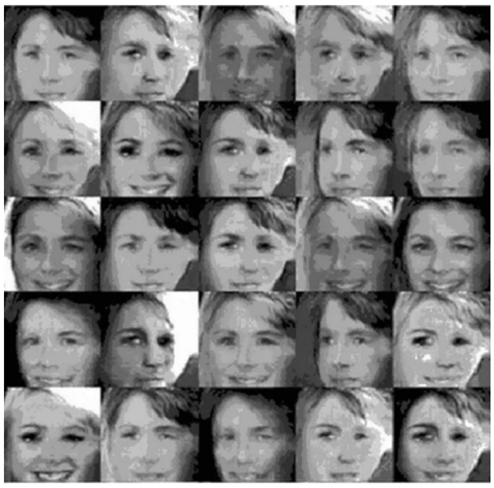


Figure 2.8: Image that represents mode collapse, on this image we can clearly see that even though this was trained in a dataset with images from different types of faces, all the generated ones look similar as if they were originated from the same face base

Hyperparameters

Hyperparameters are the training variables of the model, which are selected by the developers in order to increase the effectiveness of a given model. The number of layers and nodes per layer and the activation function are some of the parameters that are needed. GANs are really sensitive to these values, and small changes in these can lead to really different results.

2.2.5 Stop training

Traditionally, the training of any kind of learning model stops when the value of the cost function decreases less than a set value or when the training loop has run for long enough. In the context of this work, the training process stops when Nash equilibrium is reached. However, in practice, checking if Nash equilibrium has been reached is sometimes hard, so other criterion are used such as reaching a set value for the FID or the IS, but these can take too long to meet. Taking into account every argument stated above, in this work, the training process is stopped once the generator provides are realistic enough.

2.3 Frameworks and libraries

The code developed in the context of this work has been written in Python [2] using the PyTorch [34] framework.

PyTorch allows the creation and training of neural networks and the specification of how data travels through the defined architectures. The choice of PyTorch for this work was made mainly because it eases the calculation of gradients given any function, which is of a lot of use to train GANs.

Additionally, it provides a lot of pre-built layers which are very useful to build the architectures that will be described in this project, as well as a very efficient implementation of tensors that GPUs are able to compute. Pytorch also includes utilities to load, save, and iterate through datasets, which will be of use in this work due to datasets being structured in folders containing images. Finally, Pytorch supports the saving and loading of a given model during the training process, so the training can be paused and resumed, which is convenient due to the training of some architectures presented being so long.

Alongside PyTorch, the Python wrapper for Matplotlib [19] has been used to generate and save the plots of the loss function through the training process.

Chapter 3

Simple GAN: Image Generation

In this chapter, the basic knowledge about GANs explained in the previous chapter will be put into practice in order to achieve a simple GAN able to produce the most possible realistic handwritten digits.

3.1 Definition of a Simple GAN

The architecture of the GAN used in this chapter is the same as mentioned in 2.

The main component is the Generator, which essentially plays the role of the artist who tries to copy the Mona Lisa, taking a random noise vector and producing a fake sample.

Additionally, the GAN includes a Discriminator playing the role of the art critic as it takes the image the Generator has produced and determines whether that image is real or produced by the Generator. Moreover, during the training process, the Discriminator learns what a real image looks like by processing real images taken from a dataset, as shown in Figure 3.1.

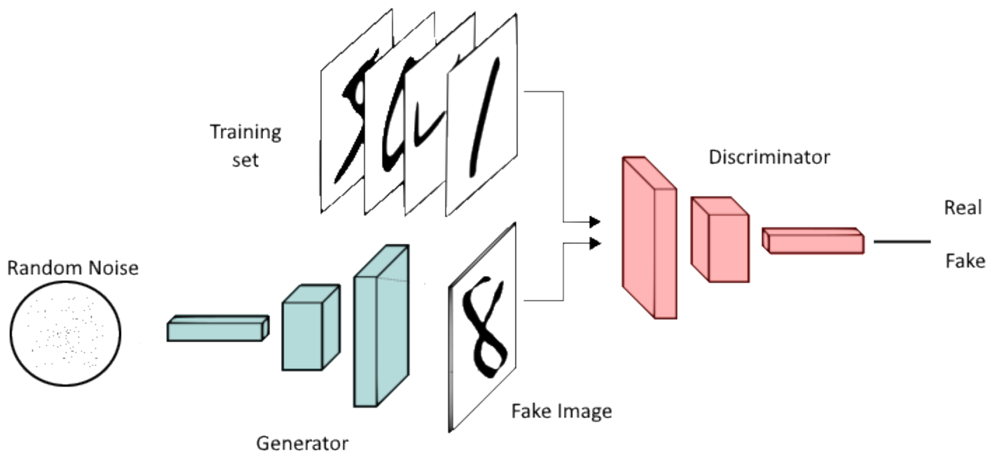


Figure 3.1: Basic components of a GAN

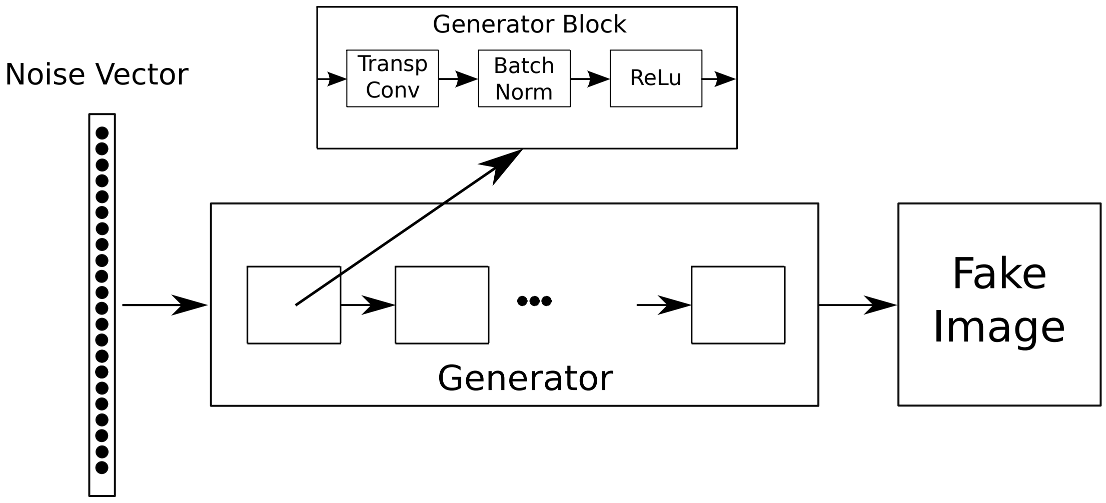


Figure 3.2: A graphic view of the generator's architecture

3.2 Architecture

3.2.1 Generator

The generator is made up of a concatenation of blocks that consist of a convolutional layer, a batch normalization layer, and a ReLU as an activation function as shown in Table 3.1 . The blocks are all the same, but the last one is made up of a convolutional layer, and a Tanh as an activation function. The input vector traverses each block until the final output of the generator ends up being a 1 channel image of size 56 $\times$ 56. A scheme of this module is shown in 3.2, and a detailed report on the layers used in this model and its output sizes in 3.2.

Layer(type)	**Output Shape**	**Num Params**
ConvTranspose2d	[512, 6, 6]	2.097.408
BatchNorm2d	[512, 6, 6]	512
ReLU	[512, 6, 6]	0

Table 3.1: Simple Generator block that takes an input of size (512 $\times$ 3 $\times$ 3)

Layer (type)	**Output Shape**	**Num Params**
Simple Generator Block 1	[512, 3, 3]	296.448
Simple Generator Block 2	[512, 13, 13]	295.296
Simple Generator Block 3	[512, 27, 27]	73.920
ConvTranspose2d	[3, 56, 56]	3.075
Tanh	[3, 56, 56]	0

Table 3.2: Simple Generator

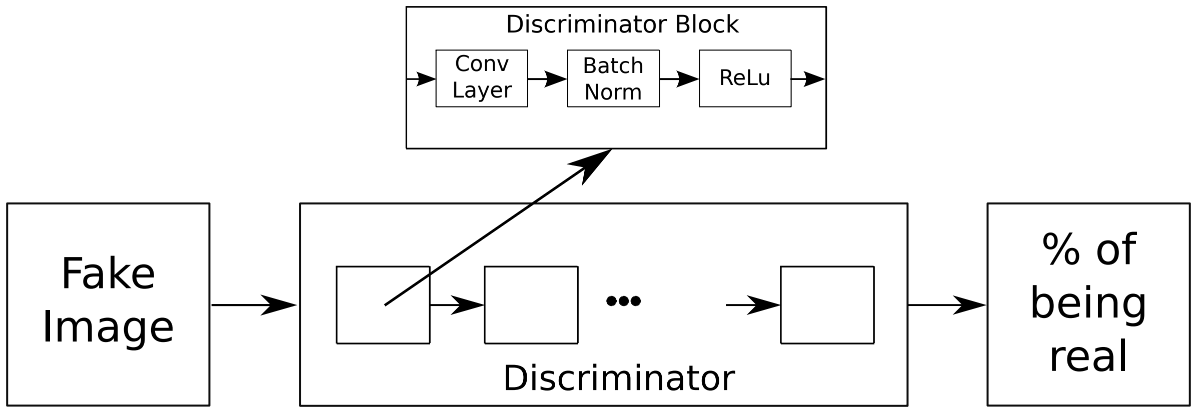


Figure 3.3: Scheme of the discriminator's architecture

3.2.2 Discriminator

The Discriminator used for this GAN is just a copy of the generator except for the last block, which is formed by a convolutional layer and a sigmoid as an activation function, whose output sizes are detailed in Table 3.3.

The only difference between the two modules is that, while the generator upsamples the initial noise vector that it receives as input, the Discriminator needs to gradually decrease the input tensor to produce a number between 1 and 0 that indicates how close the image is to be real or fake. Figure 3.3 shows a graphic view of the architecture of this module. Additionally, table 3.4 provides more details on the layers used in this model.

Moreover, it is worth noticing that the Discriminator uses convolutional layers instead of transposed convolutional layers since, even though both layers can be set to produce any output size, transposed convolutional layers ease the upsampling of their input when it comes to coding them, and convolutional layers do it at downsampling their input.

Layer(type)	**Output Shape**	**Num Params**
Conv2d	[512, 12, 12]	131.200
BatchNorm2d	[512, 12, 12]	256
ReLU	[512, 12, 12]	0

Table 3.3: Simple Discriminator block that takes an input of size (512 × 26 × 26)

Layer (type)	**Output Shape**	**Param #**
Simple Discriminator Block 1	[64, 26, 26]	3.264
Simple Discriminator Block 2	[128, 12, 12]	131.456
Simple Discriminator Block 3	[256, 5, 5]	525.056
Conv2d	[3, 1, 1]	12.291

Table 3.4: Simple Discriminator

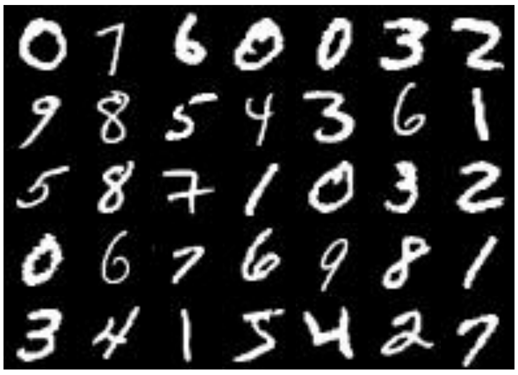


Figure 3.4: Handwritten digits images contained in the MNIST dataset

3.3 Training

The discriminator has to discern between real and fake images, as such the loss function used in this model was BCE Loss Function 2.8 combined with a Sigmoid Layer 2.2. We use the Adam optimizer for both the Generator and Discriminator, with a learning ratio of 0.003 and a batch size of 16 during training.

The hyperparameters used to train this model are :

- **Batch size** : 16
- **Learning rate** : 0.003
- **Loss function** : BCE as explained in section 1.4 of chapter 2
- **Image resolution** : 55×55
- **Adam Betas** : 0.9 and 0.999

3.4 Experiments and Results

3.4.1 MNIST Dataset

This dataset consists of 60000 images of numbers written by hand separated into classes from 0 to 9. Each class contains around 6000 images as shown in, and all the images are 28 by 28 pixels each. All images are black and white, making the training of the models faster, as there was no need to learn the specific coloring of the image. Figure 3.4 shows 35 random images sampled from this dataset.

3.4.2 First Experiment: One Class Dataset

Once we had built a basic GAN architecture, we then tried to prove it and were able to generate faithful images in a very reduced scope. To that end, we trained the GAN on a

Figure 3.5: Handwritten images of number 8 used in this experiment

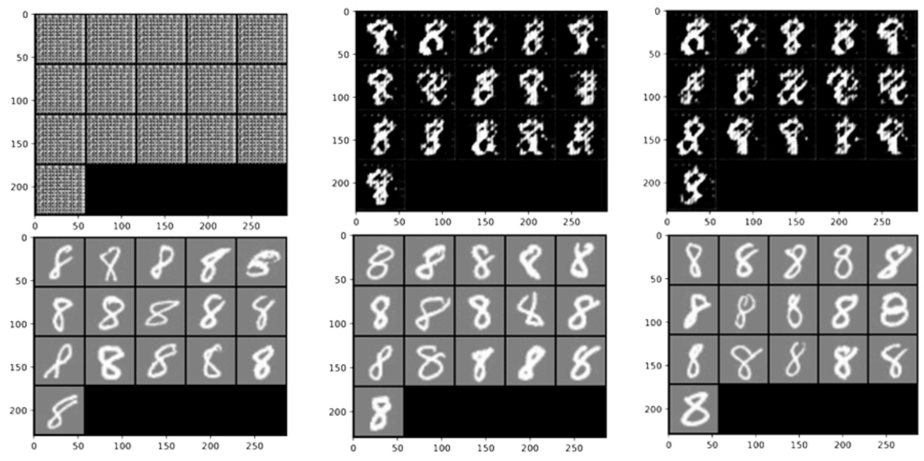


Figure 3.6: Training results after 1 (left), 4 (middle) and (right) epochs. The upper row shows what the GAN outputed, while the lower row displays real samples.

subset of the original MNIST dataset made of only handwritten images of the number 8 like the ones shown in Figure 3.5.

The following images are the result of the generator from the first epochs (1 and 8) to the later epochs (16 to 64) in which the generator has learned how to create real looking numbers. These images contain the results of the generators on the top row, while the bottom one consists of real images from the dataset.

Obtaining these images took around 1 hour and a half to generate, running the experiment on an NVIDIA 1660 paired with a Ryzen 5 3600 and 16 GB of RAM available.

As shown in 3.6 on the very first epoch the generator doesn't really do a good job at creating an image of an eight. In fact, it tries all the pixels of the image until it reaches a state in which it learns the shape of the number vaguely on the fourth and continues learning until the eighth. Even though it isn't exactly an eight, the progress is noticeable.

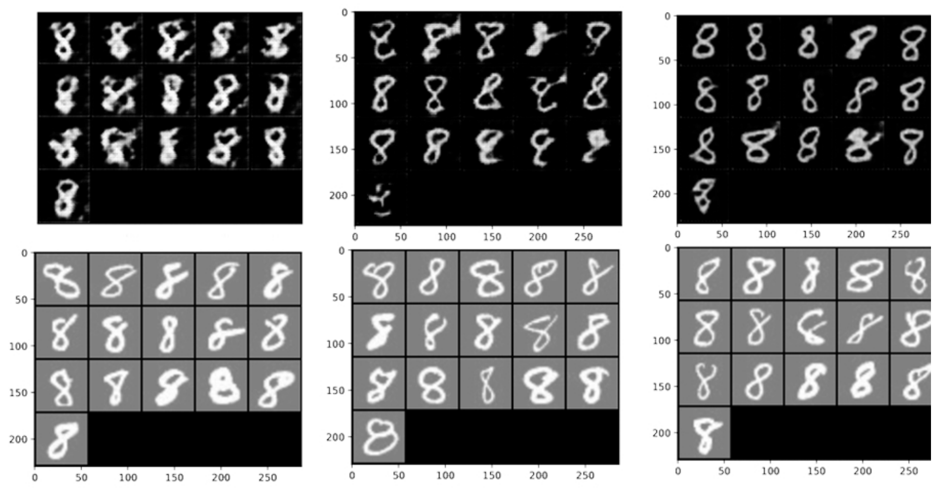


Figure 3.7: Training results after 16 epochs (left), 32 epochs (middle) and 64 epochs (left). The upper row shows what the GAN outputed, while the lower row displays real samples.

In the later stages of training, the GAN is as shown in 3.7, the number eight is more visible and the generator tries different shapes in order to match the real images from the dataset. At the begging in epoch 16, the shapes are clearly distinct, but by the 64th epoch, the shape of the numbers are much alike even though the background is still the same black color different from the gray shade of the real images.

Finally, in the last images, 3.8, we can see the loss function of both the generator and the discriminator. The discriminator is separated between the real images, the fake images losses, and the loss of both combined. It is evident that the discriminator learns far too quickly and consistently beats the generator; even on later epochs, where the shapes of the images are very identical, the discriminator can detect the slightest change and make a very accurate guess. The generator's loss, on the other hand, keeps oscillating while it learns how to create a more realistic looking eight, even ending with higher loss values than it began with, due to the discriminator learning at a much faster pace. Notice that even though the Discriminator loss details the loss for fake samples and real samples, that information is not clearly visible in Figure 3.8 as the scale needed to plot them along with the Generator loss is too large.

3.4.3 Second Experiment: Two classes Dataset

Following the promising results of the GAN trained with a single class dataset, a more complicated dataset was employed in the training procedure to demonstrate the GAN's full potential. More precisely, this dataset was made of two classes that contained both images from ones and sevens. Similar to previous figures, in the ones following the top row represents the generated fake images, while the bottom represents the real ones that the top ones are trying to imitate.

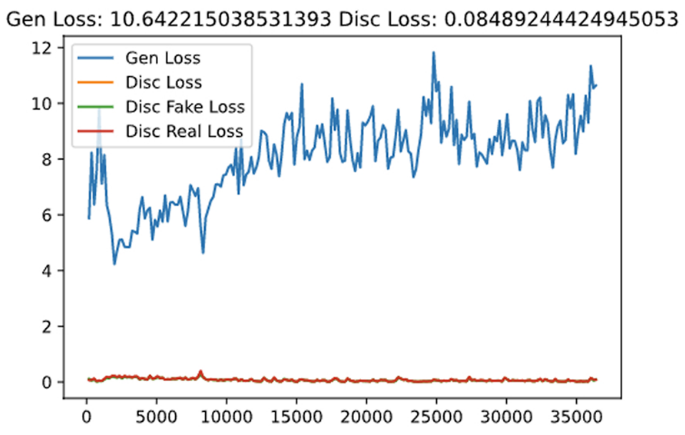


Figure 3.8: Loss function over time measured in batches

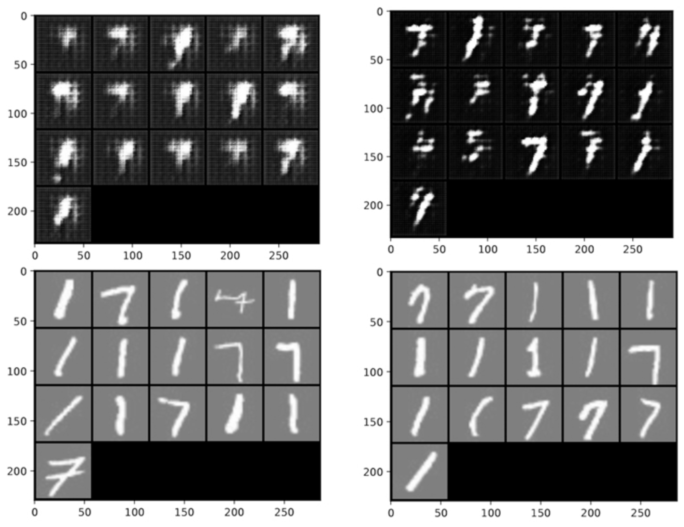


Figure 3.9: Training results after 1 (left) and 4 (right) epochs. The upper row shows what the GAN outputed, while the lower row displays real samples.

As seen on the first image in Figure 3.9, the generator starts the same as the previous section 3.4.2 in which it tries to roughly create the shape that most resembles a one and a seven combined. By the fourth epoch, it reaches a point where the shape of the number can be seen in some way.

But by the eighth epoch, as shown in Figure 3.10, it starts showing promising results. Due to the simplistic shapes of both the classes used during the training and the number of images on this one relative to the previous one (more than double) the training process takes way fewer epochs in order to harbor convincing results. In the latest epochs, the numbers have almost no difference from their original ones. Still, the generator doesn't differentiate between one and seven. Due to the similarities between both images, the generator creates a random image that doesn't discriminate between ones and sevens and creates one of them indiscriminately.

As seen on Figure 3.11, the same as Figure 3.4.2 the discriminator loss doesn't get too high for the same reasons as before, only in this case it has one more reason not to increase,

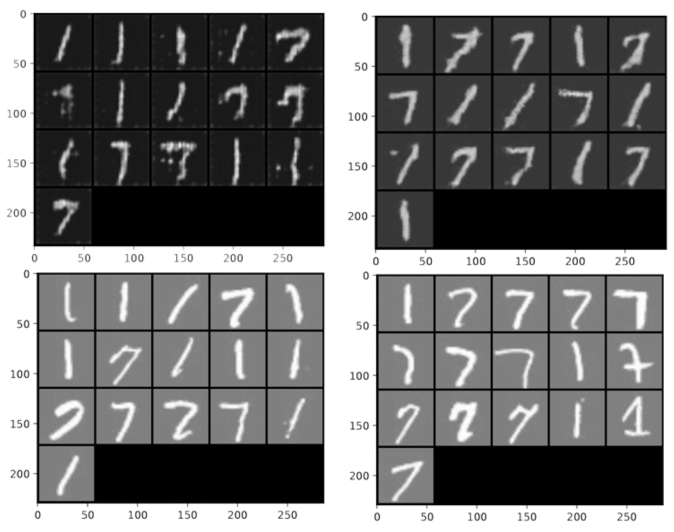


Figure 3.10: Training results after 8 epochs (left), 15 epochs (left). Top row showing training results and bottom row showing real samples.

and that is the generator incapability of creating a specific number given a class. The generator oscillates in its loss while still learning how to generate a good enough looking number, while still not knowing which class it belongs to.

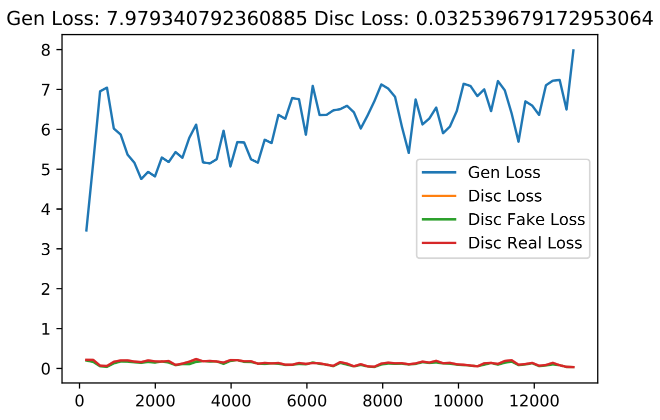


Figure 3.11: Loss function over time measured in batches for a two class dataset

3.4.4 Third Experiment: More than two classes Datasets

As the previous experiment shows promising results, we move on to more complex datasets with more than two classes or classes with not many similarities. It resulted in the generator not learning while trying to create a mixture of all classes of images at the same time, producing blurry and random images that don't resemble the originals in any way, shape, or form as it is shown in Figure 3.12.

Due to this problem in creating images for the more complex dataset, we need a GAN that takes into account which class the generated images has to represent, for that purpose

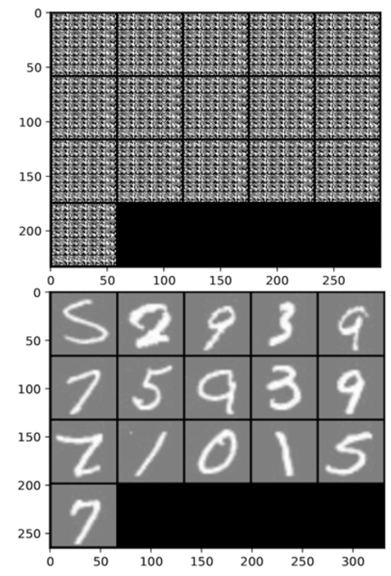


Figure 3.12: Training results after 64 epochs of the GAN with a dataset of all the numbers

we move onto a more advanced GAN on chapter 4.

3.4.5 Conclusions

In this chapter, we applied the basic notions about GANs to create a simple architecture able to produce images fairly similar to the ones used during its training. First, we tested the architecture on a one class dataset, and the Simple GAN proved to be capable of achieving realistic images. Then, we expanded the scope of training this GAN with a dataset containing images from 2 similar classes and tested the Simple GAN. The results produced by this test were worse than the ones obtained through the first experiment, but still were somehow similar to the real samples. Finally, we ran one last experiment using the whole MNIST dataset. The results of this experiment proved that the Simple GAN was not powerful enough to deal with so many classes.

Thanks to these experiments, we learned about the basic structure of a GAN, and well as its limits. They clearly showed that, although maybe this architecture would produce higher quality images by stacking more blocks in both the Generator and the Discriminator, this architecture's potential is limited.

Additionally, these experiments served us to get down to work with Pytorch and the tools it provides to build and train models, and also its utilities to manage datasets.

Chapter 4

Conditional GAN: Conditional Image Generation

GANs are capable of producing examples, but by using simple GANs we can not specify any of the features of the data samples or classes the GAN would generate. The downside to the standard GANs is that if any feature is desired to be present in the generated samples, the only choice to generate a fake sample with said feature is by trial and error. This makes the model very slow to provide an example that meets the desired conditions, and also produces a lot of undesired images to get rid of.

Fortunately, conditional GANs [30] tackle and solve this challenge pretty well with almost no change to the standard GAN architecture.

4.1 Definition of a Conditional GAN

Conditional GANs are generative adversarial networks that contain additional information on the input to condition the Generator and Discriminator during training. This auxiliary data might theoretically be anything, such as a class classification, a set of tags, or even a written explanation. In the context of this work, this additional information will be a label indicating one of the multiple classes.

The fact that the Conditional Generator's input is extended with the label of the sample to generate, implies that both the Conditional Generator and the Conditional Discriminator need to learn a way to map the features of the samples in each class to their label. Then, the Conditional Generator learns this in order to provide control to the user over the features of the generated sample. Furthermore, the Conditional Discriminator now needs to detect images paired with the wrong label regardless of whether the image is real or fake, and fake images paired with the correct label.

As a result, producing realistic-looking data via the Conditional GAN Generator is insufficient to fool the Discriminator. It's also important that the examples it generates match their labels. After the Generator has been fully trained, the Conditional GAN can synthesize any sample from any class by feeding it the desired label. To clarify the idea that this architecture introduces, 4.1 shows a sketch of how everything is put together.

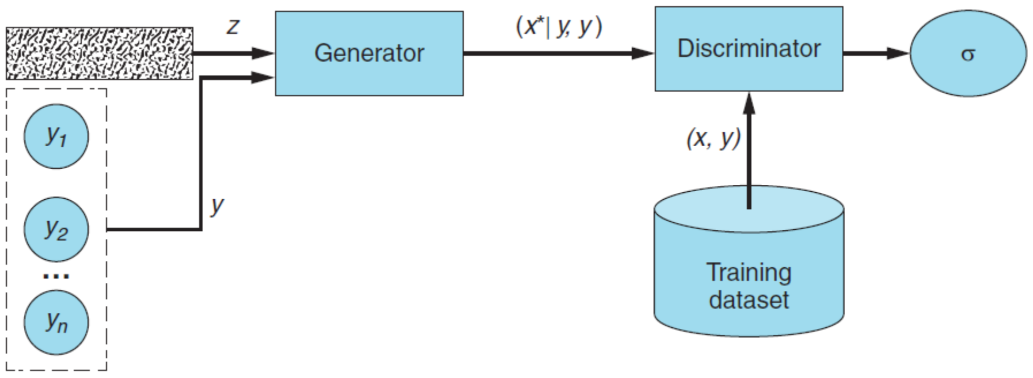


Figure 4.1: An image of a Conditional GAN

4.2 Architecture

4.2.1 Generator

As the dataset used is originally split into categories differentiating each fruit, it is easy to encode the label of an image using a one-hot vector. So, to sum up, being z the random noise vector the generator takes as input and being y the label of the sample desired, the generator produces $G(z, y) = x^*|y$ (read as "x* given that, or conditioned on, y")

The purpose of this false example is to resemble a real case for the supplied label as closely as possible (in the eyes of the Discriminator).

At a lower level, the Conditional GAN Generator consists of a series of blocks, each one performing a convolution, a batch normalization, and an activation function on their input. More precisely, the convolution performs an upsample on the input tensor while halving its number of channels. The layers along with the output layers are shown in Tables 4.2 and 4.1 and in Figure 4.2.

The reason for this is that, doubling the size of the input tensor while halving its number of channels is a way of maintaining the amount of information of the random noise vector fed to the Conditional GAN Generator as its shape is being changed along its way through the Generator using the kernels in the convolutional layers.

Layer (type)	**Output Shape**	**Num Params**
ConvTranspose2d	[256, 6, 6]	2.097.408
BatchNorm2d	[256, 6, 6]	512
ReLU	[256, 6, 6]	0

Table 4.1: Conditional Generator Block that takes an input of size (512 × 3 × 3)

Finally, we use ReLu activation in all layers except for the output layer where we use the Tanh function in consequence of previous experimentation [36], where it was discovered

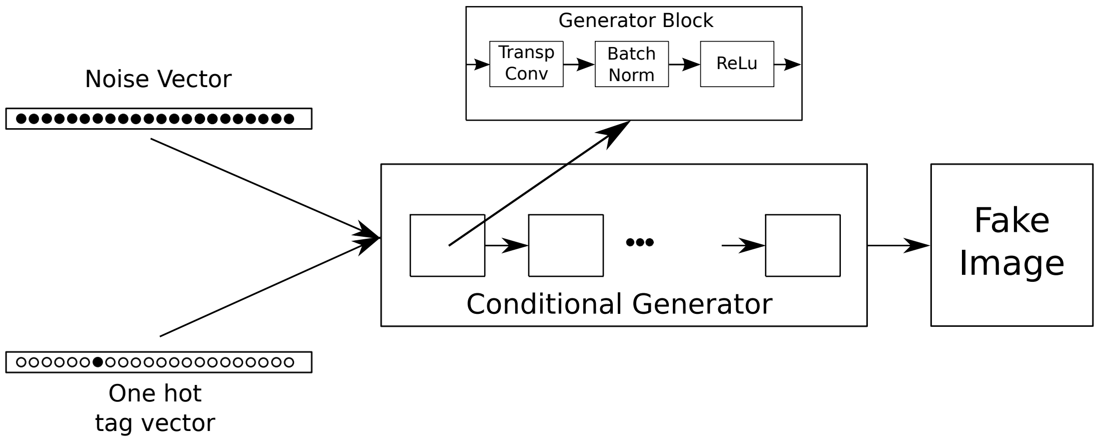


Figure 4.2: Sketch of the Conditional Generator

Layer (type)	**Output Shape**	**Num Params**
ConvTranspose2d	[512, 3, 3]	365.568
ConvTranspose2d	[256, 6, 6]	2.097.408
BatchNorm2d	[256, 6, 6]	512
ReLU	[256, 6, 6]	0
ConvTranspose2d	[128, 13, 13]	295.040
BatchNorm2d	[128, 13, 13]	256
ReLU	[128, 13, 13]	0
ConvTranspose2d	[64, 27, 27]	73.792
BatchNorm2d	[64, 27, 27]	128
ReLU	[64, 27, 27]	0
ConvTranspose2d	[3, 56, 56]	3.075
Tanh	[3, 56, 56]	0

Table 4.2: Conditional Generator

that, by adopting a limited activation, the model was able to learn to saturate and cover the training distribution's color space more quickly.

4.2.2 Discriminator

The Discriminator receives real examples with labels (x, y), and fake examples with the label used to generate them, $(x{*}|y, y)$. On the real example-label pairs, the Discriminator learns how to recognize real data and how to recognize matching pairs. On the Generator-produced examples, it learns to recognize fake image-label pairs, thereby learning to tell them apart from the real ones. The Discriminator outputs a single probability indicating its conviction that the input is a real, matching pair. Its goal is to learn to reject all fake examples and all examples that fail to match their label, while accepting all real example-label pairs,

Also, the Discriminator is never explicitly trained to reject mismatched pairs by being trained on real examples with mismatching labels; its ability to identify mismatched pairs is a by-product of being trained to accept only real matching pairs.

As well as the discriminator in the previous chapter, the Conditional Discriminator is not much more than a mirror of the generator. It performs the opposite operation to the generator, meaning that it downsamples the input tensor while doubling its number of channels, and finally generates a single number as an output. The detailed set of layers used in this module are shown in Tables 4.4 4.3 and in Figure 4.3

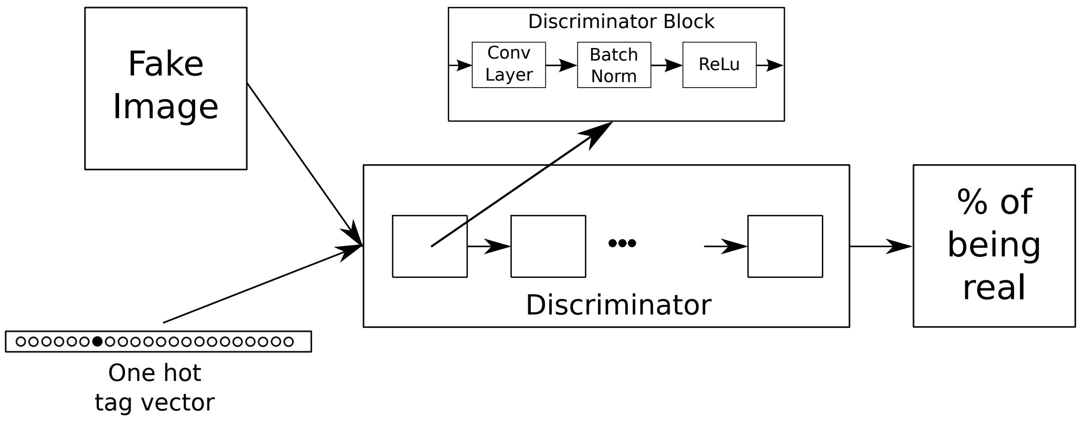


Figure 4.3: Sketch of the Conditional Discriminator

Layer (type)	Output Shape	Num Params
Conv2d	[128,14, 14]	131.200
BatchNorm2d	[128,14, 14]	256
LeakyReLU	[128,14, 14]	0

Table 4.3: Conditional Discriminator block that takes a tensor of size (64 × 31 × 31) as input

Layer (type)	Output Shape	Num Params
Conv2d	[64, 31, 31]	18.496
BatchNorm2d	[64, 31, 31]	128
LeakyReLU	[64, 31, 31]	0
Conv2d	[128,14, 14]	131.200
BatchNorm2d	[128,14, 14]	256
LeakyReLU	[128,14, 14]	0
Conv2d	[256, 6, 6]	524.544
BatchNorm2d	[256, 6, 6]	512
LeakyReLU	[256, 6, 6]	0
Conv2d	[18, 2, 2]	73.756

Table 4.4: Conditional Discriminator

We use batch Normalization all along the conditional GAN because it is crucial for the generator to start learning, since it prevents the collapse of all samples into a single point.

4.3 Training Details

The conditional GAN was finally trained a conditional on fruits images from the dataset fruits 360 dataset conditioned on their class labels (131 classes), encoded as one-hot vectors.

Since the discriminator has to differentiate between two classes (real and fake), the adversarial loss function used in this GAN is the BCE Loss function (2.8) combined with a Sigmoid layer. Furthermore, we use Adam optimizer for the Generator and Discriminator with a learning rate of 0.0002. Additionally, a batch size of 15 was used during training.

The generator network starts with a noise vector of size 79 (which is the sum of a noise vector of size 64 and the number of classes in the dataset). During the training process, the generated image is fed to the discriminator to obtain a tag for that image. Then, the tag obtained is compared to the one used to generate that image, and the gradients are calculated to modify the weights. Additionally, real images along with real tags are also fed to the discriminator to train it.

To sum up, the hyperparameters used to train this model are :

- **Batch size** : 15
- **Learning rate** : 0.002
- **Loss function** : BCE
- **Image resolution** : 64 $\times$ 64
- **Adam Betas** : 0.9 and 0.999

4.4 Experiments and results

4.4.1 Fruits-360 Dataset

The training dataset has 67692 images in total divided into 131 classes (types of fruits). These classes contain around 500 images per class as shown in, although some of them have almost double the images like the class 'Grapes Blue' which has almost a thousand images. All images are 100x100 pixels.

In contrast with the MNIST dataset, this dataset contains colorful images that are considerably bigger. Because of this, using this dataset to train a GAN significantly increases the complexity of the process.

It is worth noticing that, out of the 131 classes of fruits, 114 are round shaped and 17 are not. This may lead to more difficulties making accurate images of the non round fruits.

4.4.2 Conditional image generation

Finally, to train the Conditional GAN, a subset of the Fruits-360 dataset was used due to the memory needed to allocate the one-hot vectors and attach them to real and fake images is excessive.

This subset of the fruits 360 dataset consists of every sample in each of the 15 classes selected among the 131 available. More precisely, these classes are: clementine, avocado, banana, cocos, eggplant, kaki, kiwi, lemon, orange, pear, limes, raspberry, watermelon, mango, and maracuja. There are 482 images of each class in this dataset, adding up to 7230 images.

In the following figures, the generator output after 1, 4, 45, and 64 epochs are shown in the upper row, while in the lower row, there are real samples of the class fed into the generator to produce the corresponding output in the upper row. This experiment was run on an NVIDIA 2060 Super GPU paired with an Intel I5-9400 CPU and 16 GB of RAM and took approximately 3 hours to complete.

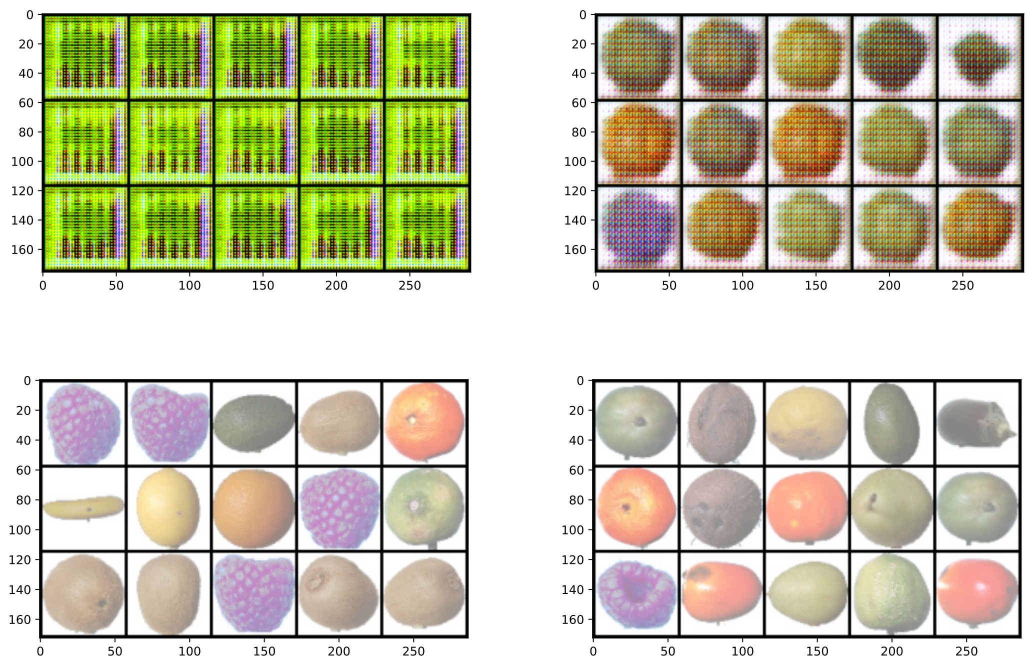


Figure 4.4: Training results after 1 (left) and 4 (right) epochs. Real samples (lower row) in contrast to fake samples (upper row)

As Figure 4.4 shows, after one epoch the conditional GAN is able to generalize that every image in the dataset shares a common background, although it is not able to learn much more from one epoch. After four epochs, the model has effectively learned the background color as well as the fact that the majority of the fruits in the dataset are round. Furthermore, the conditional Gan has been able to match different colors to distinct classes.

In later stages of the training process shown in Figure 4.5, after 45 epochs, the Conditional GAN is able to generate fake samples using colors fairly similar to those of the real samples and has perfected the shape of most of the classes. Only the ones that are the least comparable to others, such as the raspberry or eggplant, are nevertheless unrealistic, and the images have some granularity in the colors when they could have been plainer. Finally, after 64 epochs, most of the shapes in the fake samples match their actual counterparts,

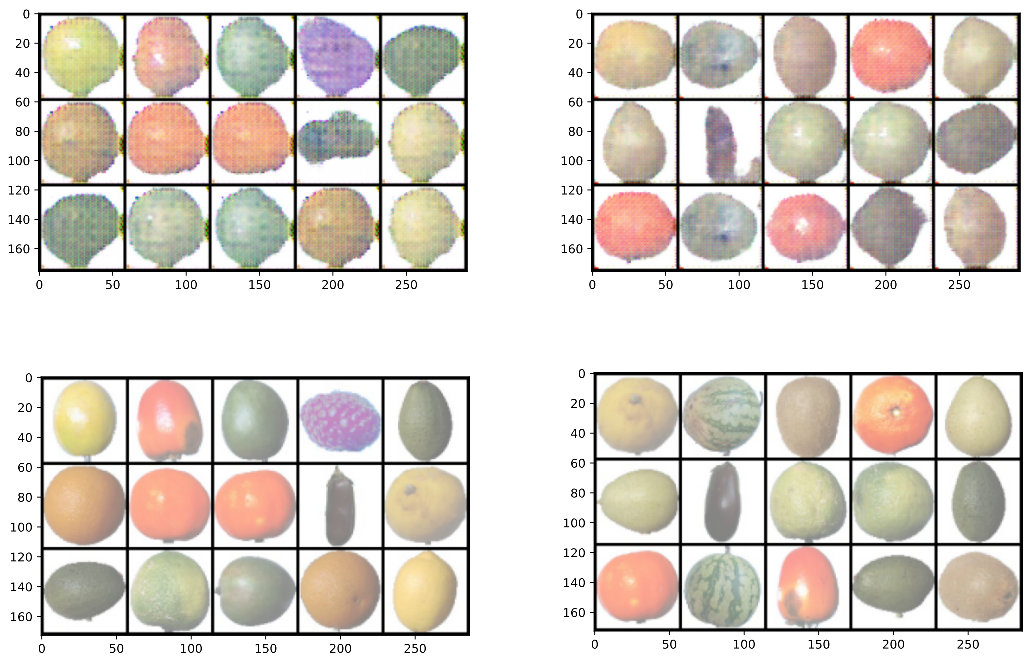


Figure 4.5: Training results after 45 epochs (left) and 64 epochs (right). Real samples (lower row) in contrast to fake samples (upper row)

and the colors are nearly identical. The sole disadvantage of these final generated photos is that the color is still not perfect because, upon closer inspection, a human viewer may discern numerous pixels in each sample.

Finally, the cost function graph over time shown in Figure4.6 demonstrates that, even though those pixels are invisible to a human observer, the discriminator is still capable of distinguishing generated samples from actual ones very effectively. It is important to notice how, although the legend of the plot provides detail on the loss function values calculated with real and fake samples, the scale needed to plot that information along the generator loss function makes the lines representing those values almost invisible.

4.4.3 Conclusions

During this chapter, we have developed some improvements over the Simple GAN to enable it to learn from both bigger and colorful images. Essentially, this was achieved by attaching a one-hot vector to the random noise input to indicate the class to which the generated sample should belong. Additionally, that one-hot vector is also added to the Discriminator's input, so it learns to tell apart fake image-tag pairs from the real ones.

The downside to this improvement comes from the fact that extending the input size limits the size of the input vector, the batch size used during training, and the amount of classes included in the dataset (because the size of the one-hot vector encoding the labels

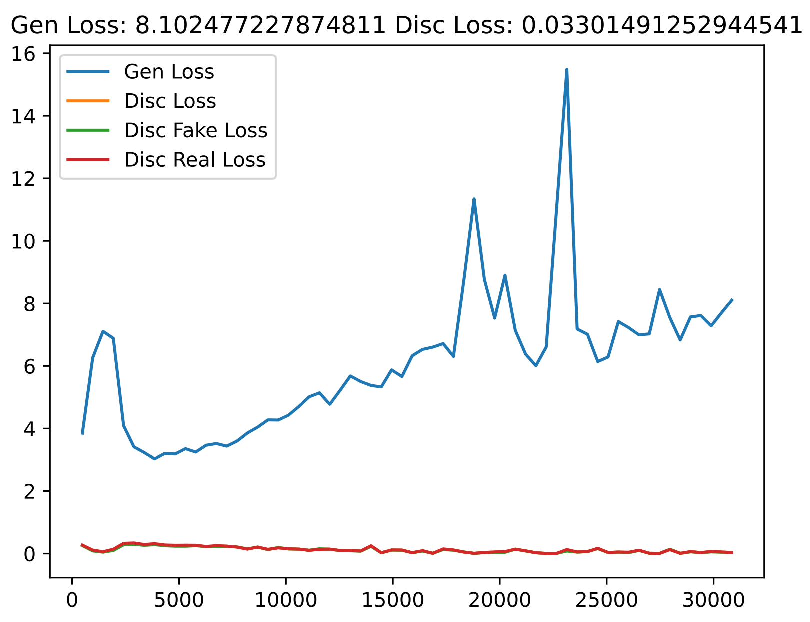


Figure 4.6: Loss function over time (measured in batches)

grows linearly with the number of classes).

Finally, the most important fact that this experiment along with the ones carried out in the previous chapter reveal is that, only building GANs from convolutional layers and batch normalization layers is not enough to produce truly realistic images.

Chapter 5

StyleGAN: Style Image Generation

As shown in the previous chapter, Conditional GANs are capable of generating images of a given class through a one-hot vector attached to the noise used as input. However, picking the image class to generate is not always sufficient to generate realistic enough images. The conditional GAN's results are somewhat realistic and can sometimes trick a human, but it has been proven that a Conditional GAN may not be adequate for more complex datasets with larger or less clear images.

In this context, StyleGANs [24] can be used to achieve both higher quality images and more precise control over the generated image. Also, because StyleGANs produce a more diversified output, the possibility of a mode collapse during training is lowered simply by modifying the network's architecture.

5.1 Definition of a StyleGAN

When looking at an image, one unconsciously takes apart the content of the image from its style. For example, both Da Vinci's Mona Lisa and Monet's Woman with a Parasol share a woman in their content, but clearly, those women are not painted in the same style. Another example of this concept is that, although two people may wear the same t-shirt, they would probably do it in different styles.

Using these examples as a reference, a style can be defined in the context of this work as a variation of any feature in an image. Relating the definition with examples, in the paintings, the women's hair can vary its style being long, short, curly, or straight, while the t-shirts present variations in their style by changing their color or the length of their sleeves.

The StyleGAN is named after this concept because not only it can generate highly realistic images, but because it is capable of changing and mixing styles thanks to two different techniques: random noise injection and style mixing. A deeper insight into this architecture's capability to alter and work with styles will be presented further in this work.

To handle styles, StyleGAN's Generator takes a random noise vector and a constant vector as an input. The idea behind having two different inputs is that the constant vector will travel through several convolutional layers and end up as the generated image, whereas the random noise vector is processed by some fully connected layers to generate other that represents the styles that will be applied to the random noise vector at various points during the generation process. The StyleGAN Discriminator, on the other hand, is quite similar to the Conditional GAN Discriminator, but it also applied some changes for it to keep up with the Generator. A whole intuition of the complete architecture is given in Figure 5.1

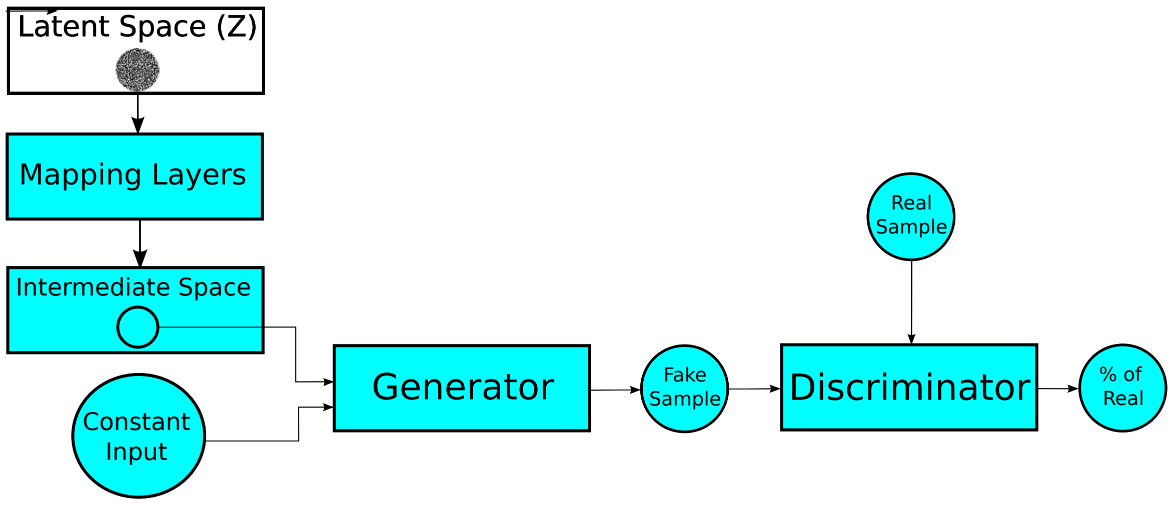


Figure 5.1: The StyleGAN's architecture takes a random vector and a constant vector as an input. The random vector first is ran through the mapping layers to produce another vector w that then will be inputed to the Generator. Then, the data flows through the model as it does in previous GAN architectures.

5.1.1 StyleGAN Components

These style components are significant advancements made primarily in the style generator architecture to allow style usage throughout the architecture.

- **Random Noise Injection**

  The goal of this component is to perturb the present image to obtain different variations of it. This adds more noise to the model, and depending on which blocks of the network this is added, the changes will be coarser or finer. This noise is sampled from a normal distribution and concatenated to the vector resulting from a generator block's convolutional layer preceding the other layers in the block. The extent to which this noise influences the image is determined by a learning factor λ.

  The changes with Stochastic Noise can be as subtle as the orientation of a wisp of hair, as seen in Figure5.2. This adds variance to the image, which can then be used for all subsequent images generated.

- **Mapping Network**

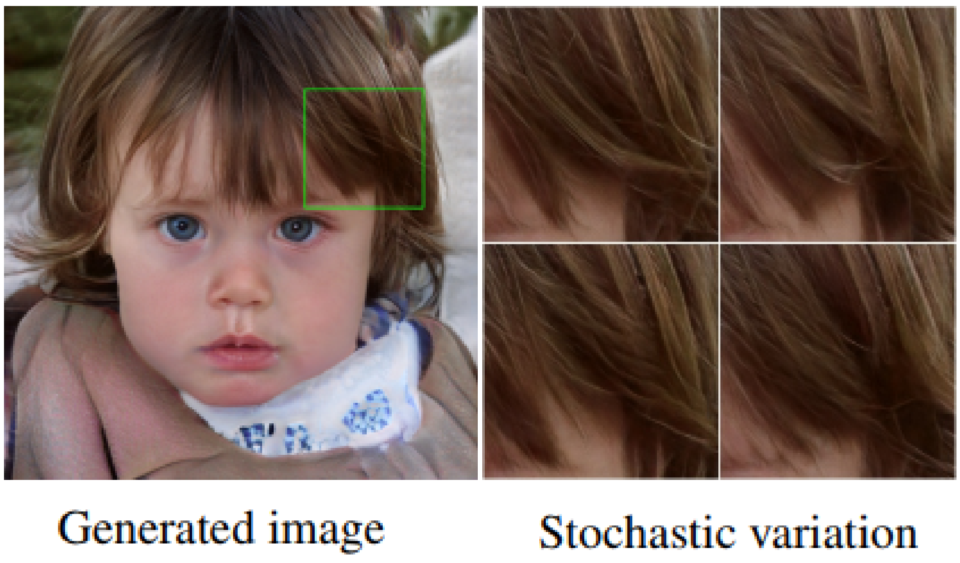


Figure 5.2: Stochastic Noise

As well as the Conditional GAN generator, the StyleGAN generator also takes a random noise vector z as input. Each vector component's value is drawn at random from a normal distribution. The noise is then sent into a neural network called a mapping network, which is constructed by stacking fully connected layers and activation functions to form an intermediate vector w.

The goal of these mapping layers is to disentangle the values of the vector, that is, to transform the initial vector z into another vector w of the same size, whose components map to the features of the generated image more precisely. This way, changing values of the vector w provides more control over the style of the generated image than changing those in the vector z. The structure of this mapping network is shown in Figure 5.3

- **Progressive Growing**

  As shown by Karras et al.[23], this component is utilized to facilitate the production of a high definition image by gradually training the generator and discriminator from lower resolution images to higher resolution ones.

  It starts by, for example, using the Generator to make 4×4 images and the Discriminator to distinguish between them and the real ones. To make it more difficult for the Discriminator, the original photos would be downsampled to 4×4.

  When both models are sufficiently trained, the next step is to double everything, so that instead of 4×4 images, the generator output an 8×8 image and the discriminator receives 4×4 images to compare. This process will be repeated until the resulting image is the required size, as shown in Figure 5.4.

  However, during this process, neither the Generator nor the Discriminator employs either the produced or upsampled outputs, but rather both images. Because the new convolutional layer has not yet been trained, the upsampled version is largely used at first. As the training progresses, the proportion of both images used in the combination gradually increases, favoring the convolutional one until it is the sole one used. The combination is as follows: 5.1

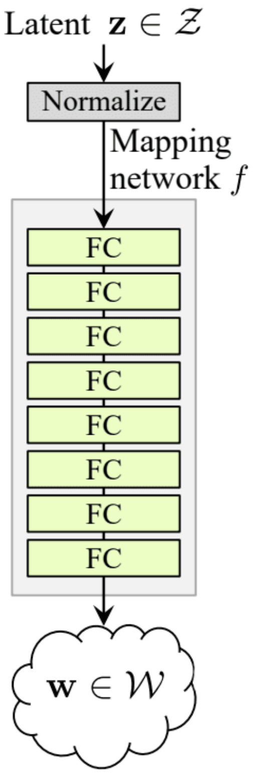

Latent $\mathbf{z} \in \mathcal{Z}$
Normalize
Mapping network f
FC
FC
FC
FC
FC
FC
FC
FC
$\mathbf{w} \in \mathcal{W}$

Figure 5.3: Mapping network structure

$$image = (1 - \alpha) * U + C * \alpha \tag{5.1}$$

Where alpha gradually increases until it becomes 1, being U the output of the algorithm to upsample the image and C the output of the convolutional block. The procedure is then repeated with the next layers. Figure 5.5 shows how a StyleGAN would grow from using 16 × 16 images to 32 × 32

- **Adaptive Instance Normalization**

  Adaptive instance normalization (AdaIn) is the mechanism used by the StyleGAN Generator to integrate the noise vector w produced by the mapping network with the image at several points as the image travels through the StyleGAN Generator. Essentially, it extracts two scalars α and β from the intermediate noise vector w and uses them to modify the image.

  Let us dissect this procedure. Similarly to batch normalization, as seen in equation 2.6, AdaIn normalizes each value in an instance by taking the mean and standard deviation of the values in the instance into account. An instance in this context is a set that contains every value in a certain channel of an image.

  The adaptive procedure begins once normalization is complete. The image is scaled and shifted by two factors, as illustrated in equation 5.2, where α is the scaling factor and β is the shifting factor. These scalars are obtained by propagating the intermediate noise vector w through two separate sets of fully connected layers.

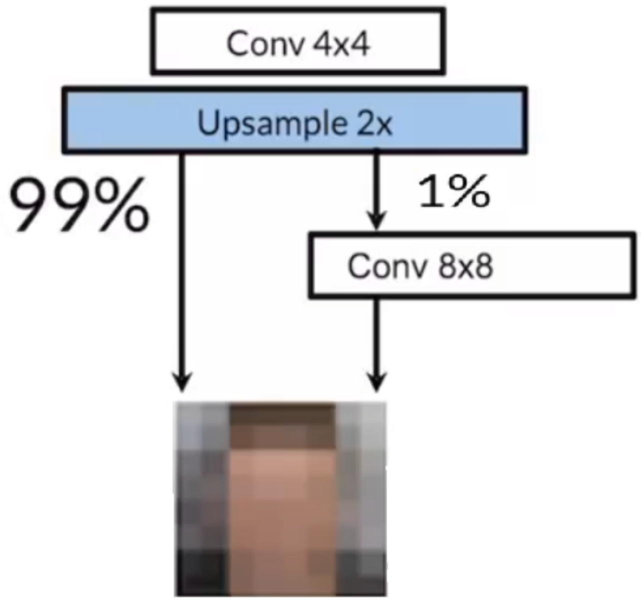


Figure 5.4: Initial state of the generator in the training process

$$x_i' = \alpha * \hat{x}_i + \beta \tag{5.2}$$

Along with progressive growing, adaptive instance normalization is an important component of the style-based approach. Due to progressive growing, the same style vector creates different variations on the resulting image depending on where it is used during the generation process. The earlier the style vector is inputted in the generating process, the larger the difference in the final image. This way, by using different style vectors along the training process, the StyleGAN generator allows more precise control over the generated image.

- **Style Mixing**

  When training the Generator, each layer of it is given $w1$, the vector that the mapping network creates. However, this isn't limited to only one vector for each layer. Another $w2$ can be added to different layers created from different noise vectors passed through the same mapping layer. This new vector, $w2$, can be used in different parts of the network.

  For example, if $w1$ is used in the first half of the network this would control the coarser features of the resulting image, such as shape and color, while $w2$ would modify finer features if used in the later blocks of the architecture.

  This method increases diversity during training due to this mixing in styles getting more diverse outputs.

  As shown in Figure 5.6, depending on which layers of the network the source B (which would be our $w2$) is added in, the Style of the original A ($w1$) changes in

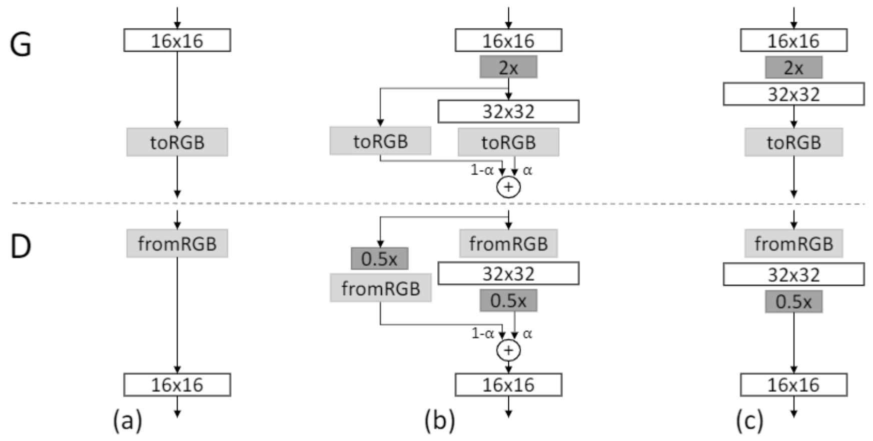


Figure 5.5: This example demonstrates the transformation of 16X16 images (a) to 32X32 images (b) (c). During the transition (b), we treat the higher-resolution layers as if they were residual blocks, with a weight that increases linearly from 0 to 1. Both the toRGB and fromRGB layers use 1X1 convolutions to convert feature vectors to RGB colors.

different ways. For coarser details, the whole shape and age of the person change, but for finer details only the color of the hair and the background change.

In a nutshell, Style Mixing is a way to intertwine two images, to create a new onthatch has the styles of both, controlling the level of detail and the degree to which the changes affect the image by selecting how deep the style vector is added. Additionally, this can be extended to any number of vectors used as input at different points in the architecture to change different styles in the output image.

- **Noise Truncation**

  Noise truncation is a way of trading off between diversity and fidelity by truncating the normal distribution used to generate the noise vector. This is done after training the StyleGAN using a hyperparameter and is another way of controlling the features of the output image. The difference between a truncated and a not truncated normal distributions can be seen in Figure 5.7

  When training generative models, the dataset used to train the discriminator strongly influences the quality of the images generated. Due to this, features poorly represented in the dataset are not well learned by the generator, thus it may end up producing poor quality images when trying to represent those features.

  Noise truncation addresses this issue by truncating the normal distribution noise vectors are taken. This way, the values of the noise vector are closer to the mean of the distribution and represent features the generator is able to generate properly.

- **Pixel Normalization of the Feature Vector**

  StyleGAN are prone to generator and discriminator magnitudes going out of control as a result of competition. To avoid this, the feature vector is normalized at each pixel to the unit length in the generator after each convolutional layer.

- **Loss Functions**

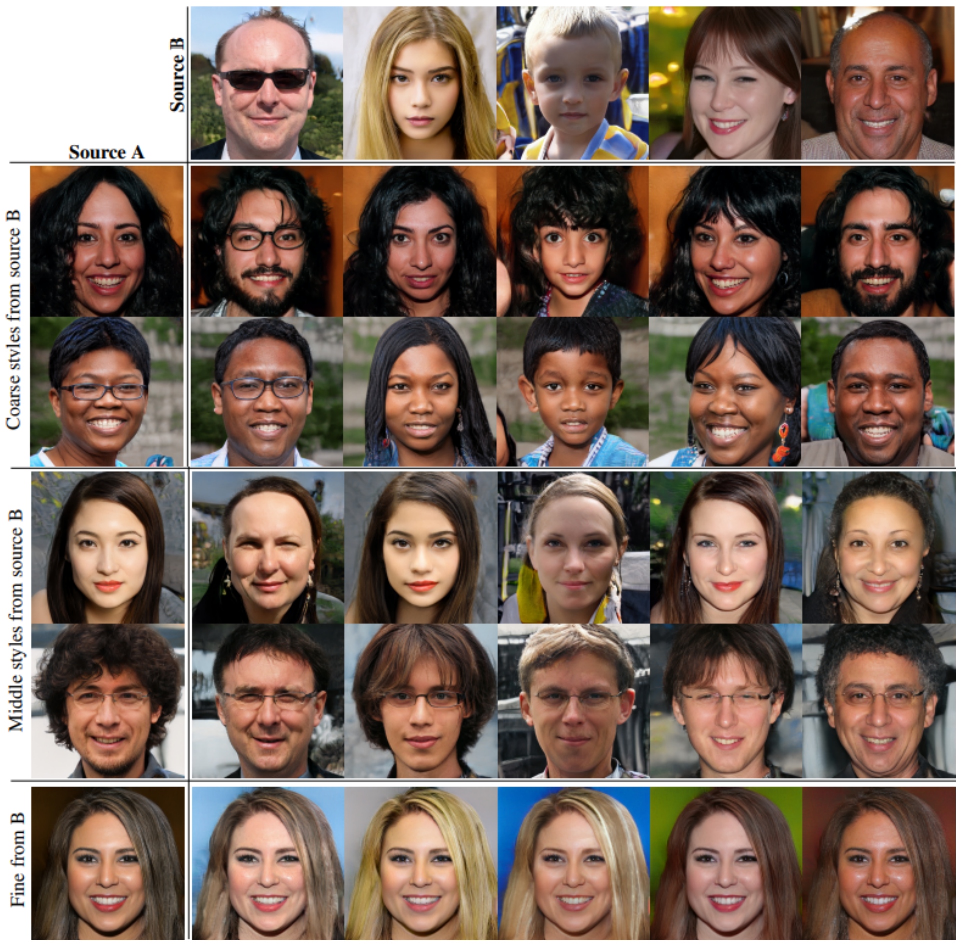


Figure 5.6: Style Mixing

Finally, to train this architecture, a new loss function, the logistic loss, was used. This new loss function is calculated as shown in the next equation, where fp stands for the prediction of the Discriminator for fake samples, and rp is the prediction of the Discriminator for real samples.

$$LogisticLoss(x) = E(softplus(fp) + softplus(-rp)) \tag{5.3}$$

The function softplus applied to the prediction emitted by the Style Discriminator acts like a ReLU function but softens the slope to make a smoother transition to where the function behaves as the identity. Figure 5.8 shows a plot of this function, which depends on two parameters:

- **Beta**: which is used in the smoothening of the function
- **Threshold**: which marks the point where the function starts behaving like the identity.

Bearing these two parameters in mind, the function is calculated as follows:

$$softplus(x) = \begin{cases} \frac{1}{\beta} \times log(1 + e^{\beta x}) & if \quad x \times \beta \leq Threshold \\ x & if \quad x \times \beta > Threshold \end{cases} \tag{5.4}$$

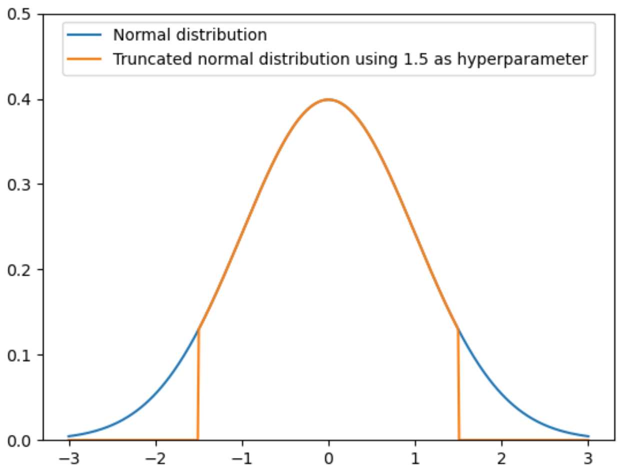


Figure 5.7: Difference between a truncated and not truncated normal distributions

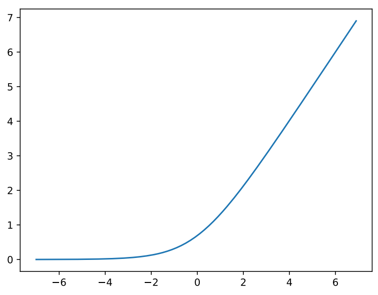


Figure 5.8: Softplus function

A regularization function called R1 regularization [29] is introduced in addition to these calculations. This technique uses a penalty and gradient regularization to train generative adversarial networks. Through gradient penalty, it penalizes the discriminator for deviating from Nash Equilibrium only on real data. In a nutshell, R1 is the gradient norm, which specifies how quickly the weights will be changed.

5.2 Architecture

5.2.1 Generator

The style generator uses every component previously mentioned in order to create a high definition image to which you can impose a style. Before explaining the whole Generator architecture, Figure 5.9 shows a sketch of how every component is put together.

The Generator consists of 3 main elements. The block in charge of mapping the styles (Mapping Network), the block in charge of progressive growing, injecting the vector of styles that has been produced by the mapping layer and adding variability to the images,

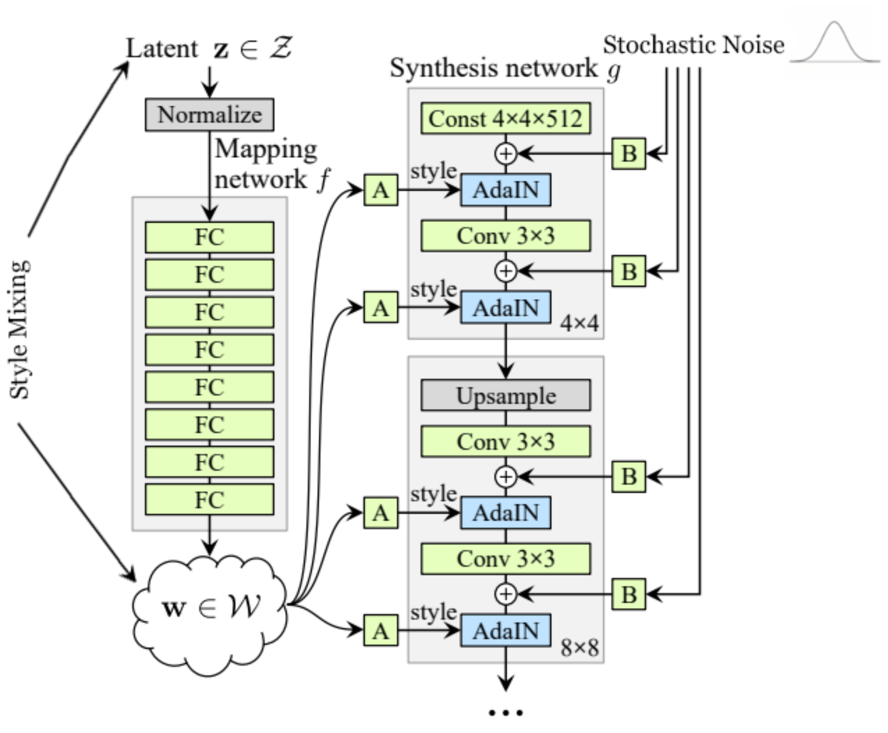


Figure 5.9: Structure of a StyleGAN Generator that has two blocks, a 4x4, which is the initial block that starts the GAN's training and takes the vector w as it's input, and a 8x8 block which takes the output of the previous block and does the same process. This structure also shows the stochastic noise and style mixing and where they are added onto the initial structure which helps with diversity and variation to outputs. Image taken from [24].

and finally a functionality to truncate the images within stipulated limits. Table 5.5 shows a full report on the layers and blocks employed in this architecture.

It begins with a learned constant. A non-linear mapping network $f(Z) = W$ first produces w given a random vector z sampled from latent space Z. An 8-layer multilayer perceptron (MLP) is used to implement the mapping f. The mapping network applying affine transformations to the vector z can be thought of as a way to draw samples for each style from a learned distribution. The effects of each style are localized in the network, which means that changing a specific subset of the styles will affect only certain aspects of the image. The detailed report on the layers used in this mapping network is shown in Tables 5.1 and 5.2.

Then, each block of the synthesis network (Table 5.4) learns affine transformations to specialize w to styles $y = (ys, yb)$ that control adaptive instance normalization (AdaIN) (Eq.5.2) operations contained in the blocks detailed in Table 5.3. To understand why this localization is necessary, consider how the AdaIN operation first normalizes each channel to zero mean and unit variance before applying scales and biases based on the style. The new per-channel statistics, as specified by the style, change the relative importance of features for the subsequent convolution operation, but they are independent of the

Layer (type)	**Output Shape**	**Num Params**
EqualizedLinearPropia	[512]	262,656
LeakyReLU	[512]	0

Table 5.1: Mapping Network Block

Layer (type)	**Output Shape**	**Num Params**
Mapping Network Block-1	[512]	262,656
Mapping Network Block-2	[512]	262,656
Mapping Network Block-3	[512]	262,656
Mapping Network Block-4	[512]	262,656
Mapping Network Block-5	[512]	262,656
Mapping Network Block-6	[512]	262,656
Mapping Network Block-7	[512]	262,656
Mapping Network Block-8	[512]	262,656

Table 5.2: Mapping Network

original statistics due to normalization. As a result, each style can only control one convolution before being overwritten by the next AdaIN operation.

Layer (type)	**Output Shape**	**Num Params**
NoiseLayer	[512, 4, 4]	512
ReLU	[512, 4, 4]	0
InstanceNorm2d (Adain)	[512, 4, 4]	0
EqualizedLinear	[1024]	525,312

Table 5.3: Adain Block (4x4 Block Example)

To improve the localization of the styles in the generated images, mixing regularization [24] is implemented in the StyleGAN Generator. Mixing regularization consists of applying style mixing during the training process by first running two different inputs, $z1$ and $z2$ through the mapping network to generate two intermediate latent vectors $w1$ and $w2$. Then, at a set point during the training process, the latent vector used in the AdaIN blocks is switched from $w1$ to $w2$. This regularization technique prevents the network from making the assumption that adjacent styles are correlated.

Finally, by introducing explicit noise inputs, the generator is given a direct way to generate stochastic detail. It is fed a single-channel dedicated noise image to each layer of the synthesis network. By using learned per-feature scaling factors, the noise image is broadcasted to all feature maps and then added to the output of the corresponding convolution. The reason why the stochastic variations are not introduced the traditional way in this architecture is because it is difficult and not always successful to hide the periodicity of general signals. This architecture avoids these issues entirely by including per-pixel noise after each convolution. The noise only has to affect stochastic aspects of the system, leaving the overall composition and high-level aspects such as identity unaffected.

To sum up, each block ends up performing the following:

- A transposed convolution and a Upsampling layer to upsample the input image using learned weights.

- Then, performs the random noise injection to produce stochastic variation on the image.
- Runs the result through an LeakyReLU
- Takes the latent vector w , calculates the style shift and the style scale coefficients using the AdaIn multilayer perceptrons, normalizes the input and finally applies the calculated coefficients to the results.

To enhance the performance of the generators, the previous sequence can be altered to perform the upsampling, the noise injection and the adaptive instance normalization twice in each block. This way, each block learns even more with each training step.

Layer (type)	**Output Shape**	**Num Params**
Upscale2d	[512, 32, 32]	0
Conv2d	[512, 32, 32]	0
Adain Block	[512, 32, 32]	0
Conv2dPropia	[512, 32, 32]	2,359,808
Adain Block	[512, 32, 32]	0

Table 5.4: Synthesis Network Block (32x32 output Block Example). Input Shape: [512, 16, 16]

Component	**Output Shape**	**Num Params**
PixelNormLayer-1	[512]	0
Mapping Network-2	[10, 512]	0
Truncation-3	[10, 512]	0
ConstantLayer-4 (4x4)	[10, 512]	0
Adain Block-5 (4x4)	[512, 4, 4]	525,824
Conv2dPropia-6 (4x4)	[512, 4, 4]	2,359,808
Adain Block-7 (4x4)	[512, 4, 4]	525,824
Synthesis Network Block (8x8)	[512, 8, 8]	5,771,264
Synthesis Network Block (16x16)	[512, 16, 16]	5,771,264
Synthesis Network Block (32x32)	[512,16, 16]	5,771,264
Conv2dPropia (64x64)	[3, 64, 64]	13,827
Synthesis Network Block (64x64)	[256, 64, 64]	2,295,808
Conv2dPropia (64x64)	[3, 64, 64]	6,915

Table 5.5: Style Generator

5.2.2 Discriminator

As previously stated, the StyleGAN's Discriminator is extremely similar to the Conditional GAN's one. This neural network is built with convolution and average pooling layers to decrease the size of the input image in half, a batch normalization layer, and finally an activation function (ReLU). To avoid outsmarting the generator, this discriminator must use progressive growing because the work at hand is simpler than creating genuine samples from start, and the StyleGAN Discriminator's design is significantly less complex than the StyleGAN Generator's. The detailed order and output size of the layers can be seen in Table 5.7.

Layer (type)	**Output Shape**	**Num Params**
Conv2d	[512, 64, 64]	2.359.808
LeakyReLU	[512, 64, 64]	0
Conv2d	[512, 32, 32]	1.179.904
AvgPool2d	[512, 32, 32]	0
LeakyReLU	[512, 32, 32]	0

Table 5.6: Discriminator Block (64x64 Block Example)

Layer (type)	**Output Shape**	**Param #**
Conv2d	[512, 64, 64]	2.359.808
Discriminator Block (64X64)	[512, 32, 32]	2.359.808
Discriminator Block (32X32)	[512, 16, 16]	2.359.808
Discriminator Block (16X16)	[512, 8, 8]	2.359.808
Discriminator Block (8X8)	[512, 4, 4]	2.359.808
Discriminator Block (4X4)	[512, 2, 2]	2.359.808
LeakyReLU	[512]	0
EqualizedLinearPropia	[512]	4.194.816
LeakyReLU	[512]	0
EqualizedLinearPropia	[1]	513

Table 5.7: Style Discriminator

5.3 Training Details

During the training process, the fact that progressive growing forces the StyleGAN to work with different image sizes, allows us to specify the values of certain hyperparameters for each intermediate size. More precisely, these hyperparameters are the number of epochs spent learning to generate images of a certain size and the batch size used while learning to produce images of that size. Accordingly, these hyperparameters will be shown as a list, whose elements relate to the corresponding element in the size list.

In both architectures, image sizes grow from 4 to 64, doubling in each step. Additionally, we use leaky ReLU with a negative slope equal to 0.2 and equalized learning rate in all layers, as Karras et al. [23]. The mapping network in the Generator is composed of 8 fully connected layers, and the dimensionality of all input and output activations to be used as noise in both architectures is 512.

The following hyperparameters were used to train this model:

- **Image resolution to be generated** : [4,8,16,32,64]

- **Number of epochs corresponding to the resolution**: [4, 4, 4, 4, 8, 16]
- **Batch size corresponding to the resolution**: [32, 32, 32, 16, 8, 4]
- **Learning rate** : 0.003
- **Loss function** : Logistic Loss
- **Mapping Layers** : 8
- **Latent Size** : 512
- **Truncation psi** : 0.7
- **Truncation cutoff** : 8
- **Adam Betas** : (0, 0.99)

5.4 Experiments and Results

For all the experiments conducted in this chapter, we used the Fruits-360 dataset explained in previous chapters.

5.4.1 Initial Training

Following the pattern of previous chapters, once we had implemented the StyleGAN, we first tried to generate the highest possible quality.

To this end, we began training this GAN with components from the Conditional GAN, such as the BCE loss function 2.8. The Generator started correctly during the initial depths after training for multiple batches and depths because the images it generated were of low resolution, but as the resolution increased, it began to worsen, resulting in poorly drawn fruits. It ceased appropriately separating the colors and shapes of the fruits and began drawing everything similarly. Figure 5.10 shows the output images from this function, while image 5.12 illustrates the deterioration of the Generator versus the Discriminator based on the loss function.

When we examined the results, we saw that the GAN was not learning to separate the parts that placed the style on each fruit at resolutions greater than 16. After analyzing and experimenting with various loss functions, we discovered that the logistic 5.3 works pretty well and that the training produces good results.

We can hypothesize that it is related to the logistic loss function's regularization function. Each fruit is given a unique identifier, and each identifier is given its own feature. If the regularization function is not supplied, the model will be entirely overfitted, resulting in misgenerated fruits. This is owing to the model's inability to minimize the loss to zero in all samples, causing the weights of each indicator feature to wildly fluctuate. This is more likely with high-dimensional data (for example, resolutions more than 16x16) with

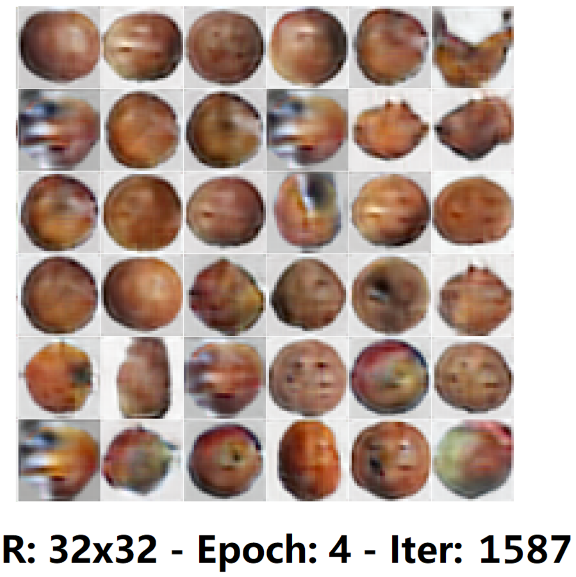


Figure 5.10: StyleGAN outcome with BCE Loss at the end of 64x64 image generation.

feature crossings, where numerous unexpected crossovers occur in only one occurrence each [16].

Figure 5.11 shows the output of the generator at the beginning and end of training, given a resolution using the logistic loss function.

The generator has learned some of the basic colors that represent the simplest fruits in 4x4 resolution. For the 8x8 resolution, it was discovered that most fruits are circular and thus represent them in this manner. In addition, the generator can now tell which color is the main color of the fruit it is studying. For the 16x16 resolution, it can do the same thing as in the previous resolution, but now it begins to define specific characteristics of each fruit, specifically the shape. The generated images can already be recognized as fruits at 32x32 resolution, with only those with a rather unique shape remaining to be correctly learned. Finally, for the 64x64 resolution, it learns very specific fruit characteristics such as valleys, discolorations, leaves, skin style, and so on.

To obtain the images previously shown, we had to train the StyleGAN for 10 hours on a computer packing an NVIDIA 2060 Super GPU, an Intel I5-9400 CPU, and 16 GB of RAM.

Finally, we plot two graphs of the cost function over time, one at 4x4 and one at 64x64 pixels. The first one, Figure 5.13, shows that the Generator started with huge losses, but was able to match and compete with the Discriminator using the loss function 5.3. Then, in the image showing the losses at depth 64x64, Figure 5.14, it is clear that the Generator was on par with the Discriminator throughout the training.

The time it took to train is depicted in the graph as depicted in Figure 5.15.

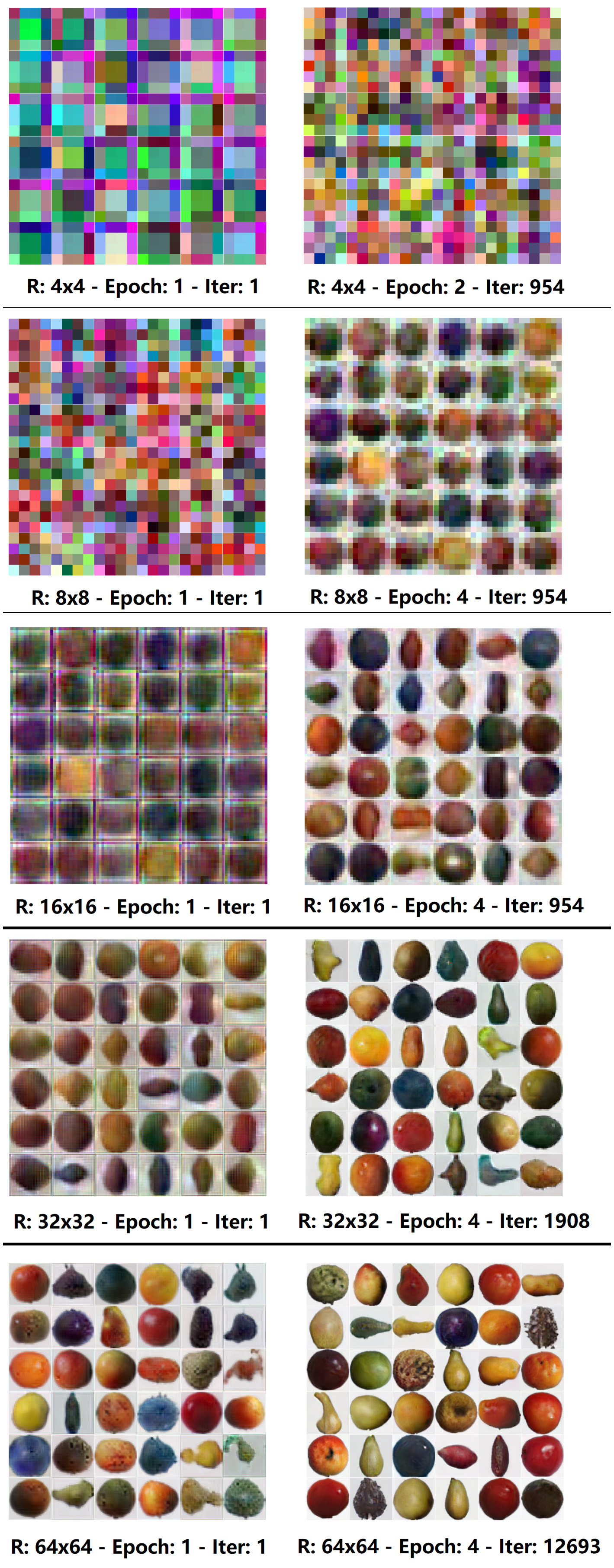

R: 4x4 - Epoch: 1 - Iter: 1
R: 4x4 - Epoch: 2 - Iter: 954
R: 8x8 - Epoch: 1 - Iter: 1
R: 8x8 - Epoch: 4 - Iter: 954
R: 16x16 - Epoch: 1 - Iter: 1
R: 16x16 - Epoch: 4 - Iter: 954
R: 32x32 - Epoch: 1 - Iter: 1
R: 32x32 - Epoch: 4 - Iter: 1908
R: 64x64 - Epoch: 1 - Iter: 1
R: 64x64 - Epoch: 4 - Iter: 12693

Figure 5.11: StyleGAN Results with Logistic Loss

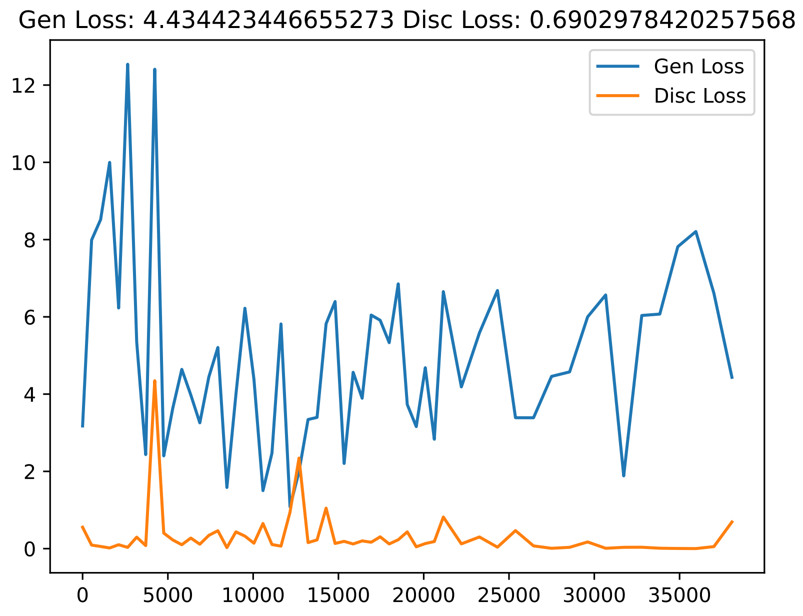


Figure 5.12: BCE Loss function over time (measured in batches) in depth 32x32

5.4.2 Style Mixing

After the StyleGAN had proven its capability to generate realistic images, we trained the StyleGAN for the nights for 2 full weeks in order to generate even higher resolution images. Additionally, we needed such images because we wanted to test how exactly the difference in the places where the style vector is added changes the final sample.

After training had been completed, the StyleGAN could be used to generate image modifications by combining styles. We took a tomato and a white grape image and ran it through the Generator along with many style vectors extracted from other fruits inserted in different blocks in order to produce finer or coarser adjustments to the image, as can be seen in Figures 5.16 and 5.17.

5.4.3 Conclusions

In this chapter, we have addressed a new architecture for the Generator while adapting the Discriminator to provide the model with the ability to alter the style of an image. This is achieved primarily thanks to both progressive growing and adaptive instance normalization. Additionally, there are some other components introduced into the architecture like the Logistic Loss function, or the Random Noise Injection which help in producing more realistic images and increasing the diversity of the samples generated respectively.

However, with an improvement in the quality of the results comes a significant rise in training time. This, combined with the increased number of possible hyperparameter combinations, makes training much more difficult, as the time required to acquire usable feedback from the model increases significantly.

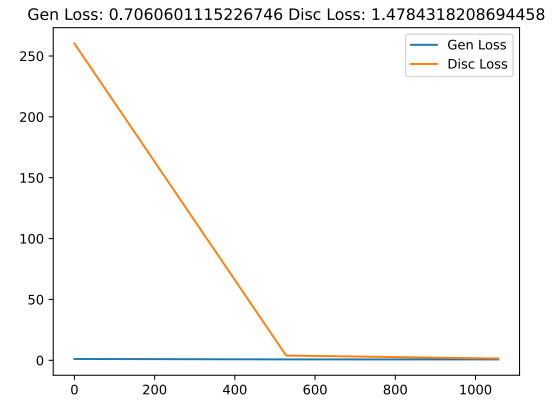


Figure 5.13: Loss function over time (measured in batches) in depth 4x4

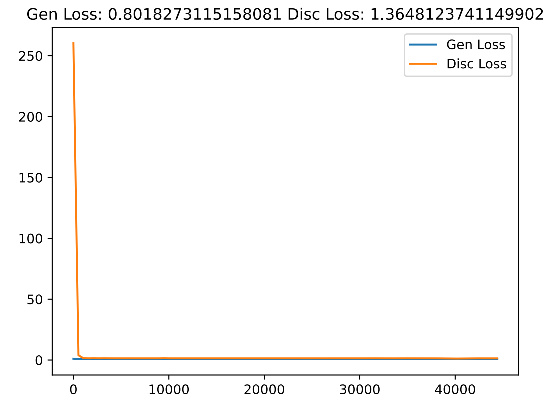


Figure 5.14: Logistic Loss function over time (measured in batches) in depth 64x64

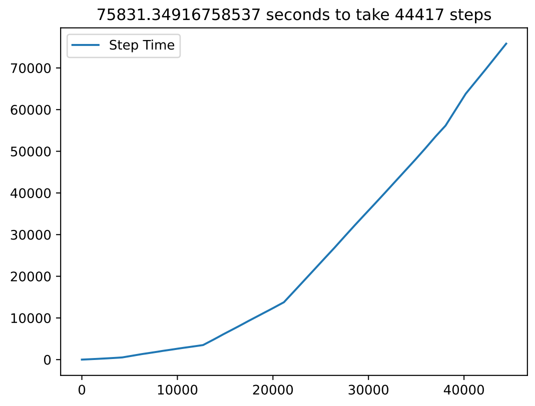


Figure 5.15: StyleGAN batch training time Graph

Through conducting these experiments, we learned how styles are dealt with in the field of image processing and also, some useful techniques to increase the realism of the generated images other than adding power to the model by stacking blocks on top of each other.

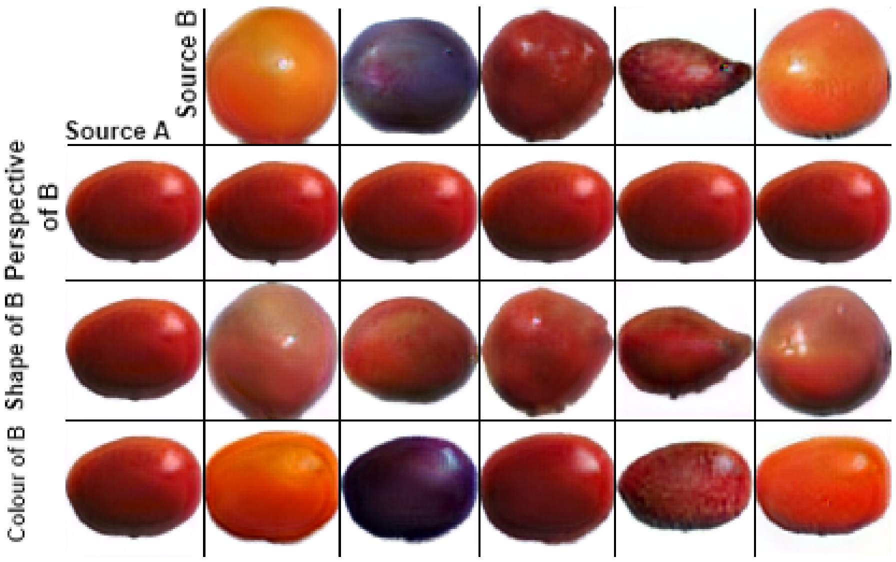


Figure 5.16: The StyleGAN developed three rows of images that combined various attributes of various fruits to create a variety of styles. The first row merges the tomato's style and attempts to shift the viewpoint or position of the tomato so that it is similar to the style of the other fruits; however, because all the fruits have similar perspectives, the tomato does not change. The second row changes the shape of the tomato by merging it with the shapes of the other fruits, resulting in tomatoes of varied shapes. Finally, the last row aims to change the color style of the tomato by creating tomatoes with the skin of other fruits.

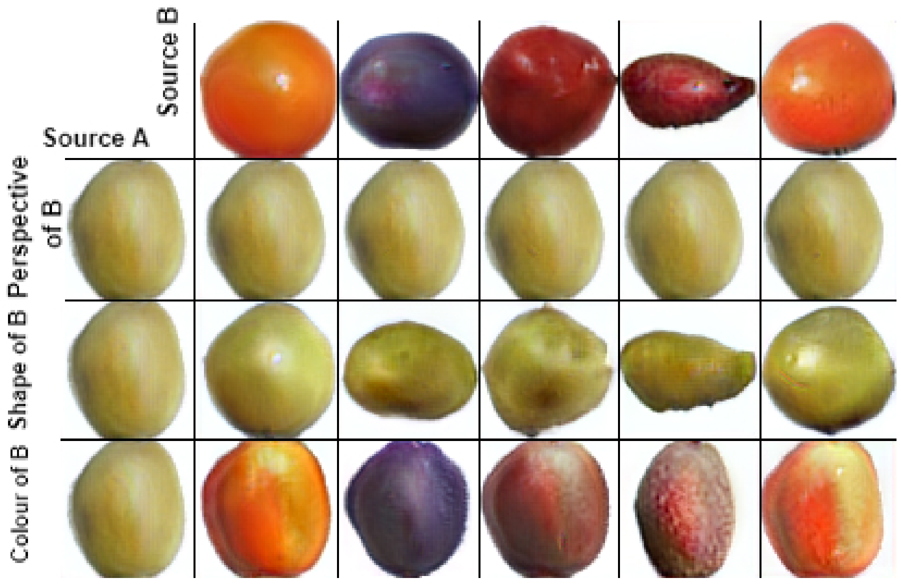


Figure 5.17: The StyleGAN in this case utilizes a white grape as an example of a picture to be mixed. As in the preceding scenario, the most noticeable cases occur when a combination of styles is used to alter the form and skin of the grape.

Chapter 6

CycleGAN : Style Semantic Transfer

Following the explanation of architectures that learnt how to define styles and classes, the next step would be to develop a model that learned how to transfer features or styles between images. The goal is to create a texture from a source image in its entirety while constraining the texture synthesis to preserve the semantic content (such as an object) of a target image. It is a texture transfer algorithm that constrains a texture synthesis method by using feature representation from a convolutional neural network.

Translating images from one to another is about discovering the mapping between an input image and an output image based on a set of training images. For example, if the model were to generate an apple with pear skin or vice-versa, the model must be able to define the common characteristics between two photos (apple and pear) and transfer the necessary attribute to the other image.

To make this possible, a neuronal network architecture should be capable of retaining most of the attributes of an image that allows it to be recognized as part of a specific class while also being capable of changing some characteristics. That is, an architecture that recognizes an image as an apple but can transform it into a pear's skin.

Notice how what we aim to do in this chapter is different from what was achieved in the last one. The StyleGAN built an image from scratch bearing a vector that somehow described the style to be applied to the final image. The model provided in this chapter, on the other hand, attempts to learn what content is shared between two images and to swap the style of the common content.

6.1 Definition of a CycleGAN

A CycleGAN [42] is essentially two modules facing each other, where each one learns a mapping function from one class to another. If the CycleGAN is set to transform images from class A to class B and vice-versa, it receives a sample from class A, and first runs that input through the first module which produces a fake sample of class B. Then, that fake sample goes through the second module, producing a new fake sample of class A. Each module in the CycleGAN is in the end a GAN, so it contains a Generator and a Discriminator, making up to 2 Generators and 2 Discriminators contained in one CycleGAN.

The CycleGAN learns by comparing the final sample of class A to the original one because, in the end, if the final sample resembles the original one, that means that both the transformation from class A to class B and the one from class B to class A have worked correctly. Furthermore, assuming the CycleGAN has been appropriately trained, the two components can be separated to produce two independent models capable of transferring features from one class to a sample of another class.

By using unsupervised learning, CycleGAN can learn a mapping function from one image domain to another. This implies that the dataset doesn't need to contain samples of a class transformed into the other. Instead of learning the features of some transformed samples, CycleGAN learns to perform this transformation by telling Generator Networks to learn a mapping from domain X to what seems to be a picture from domain Y and vice-versa.

So, the mapping functions between two domains X and Y learned by the model would be.

$$\begin{aligned} G &: X \to Y \\ F &: Y \to X \end{aligned} \tag{6.1}$$

Given training samples x ϵ X and y ϵ Y and data distributions denoted by $x\ data(x)$ and $y\ data(y)$, the architecture consists of two adversarial discriminators D_X and D_Y, where D_X aims to distinguish between images x and translated images F(y); in the same way, D_Y aims to discriminate between y and G(x).

In addition, two new loss functions are introduced. The first one, adversarial loss, is used to match the distribution of generated images to the distribution of data in the target domain, while the second one, cycle consistency losses, keeps the learned mappings G and F from contradicting one another. Both mapping functions are subjected to adversarial losses.

In Figure 6.1 it can be seen the complete structure of a CycleGAN that transforms horses into zebras and vice versa. The first generator transforms a horse into a zebra, and then this generated image is passed on to the other generator to make a horse.

6.2 Architecture

6.2.1 Generator

The generators in each GAN of the CycleGAN are a modified U-Net [38], which is an architecture that specializes in processing images to generate segmentation maps built by joining two separate modules. Within the U-Net architecture, one is an encoder in which each block is built by stacking a convolution, a batch normalization layer, and a leaky ReLU function. Then this encoder is followed by a series of nine blocks which

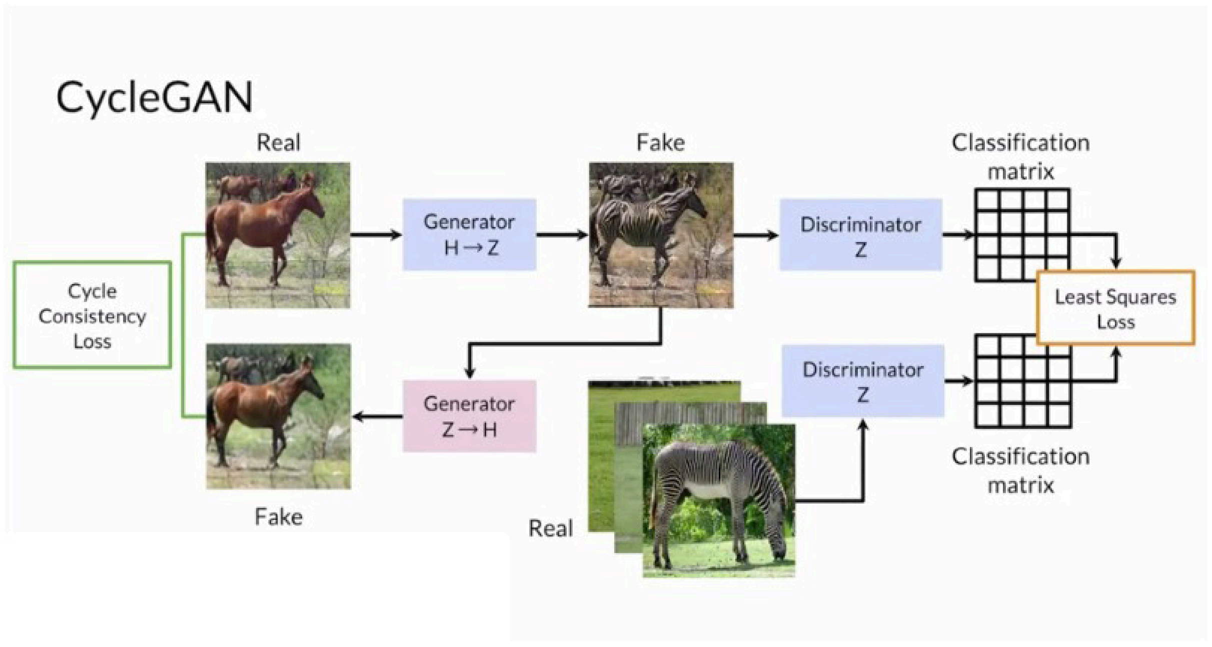


Figure 6.1: Complete Structure of a CycleGAN which has been trained with images of horses and zebras

perform two convolutions and an Instance Normalization layer while having the output size unchanged as shown in 6.1. Then the Decoder downsamples the image by creating the same organization as the Encoder, only backward to ensure the output image is the same size as the real images from the dataset. Table 6.2 provides more detail about the layers used and their output sizes. The entire structure can be seen on Figure 6.2

6.2.2 Discriminator

The discriminator used is called PatchGAN [20]. This discriminator outputs a matrix of values, each of which corresponds to a ”patch” or region, that are sections of the image, as seen in Figure 6.3. This value ranges between 0 and 1, with 0 representing a fake part of the image and 1 representing a real one. All of these terms are combined into a value matrix, with each number representing a patch. On this discriminator, the label for fake isn’t 0, but a matrix filled with zeroes.

Both Discriminators have a much simpler structure. Each has a series of four blocks that contain a convolution to upsample the input image and an Instance Normalization layer, as shown in Table 6.3. Following that, a convolutional is used to transform the vector image into a 1×1 vector, which is then utilized as the discriminator’s prediction on how much of this image is real or fake.

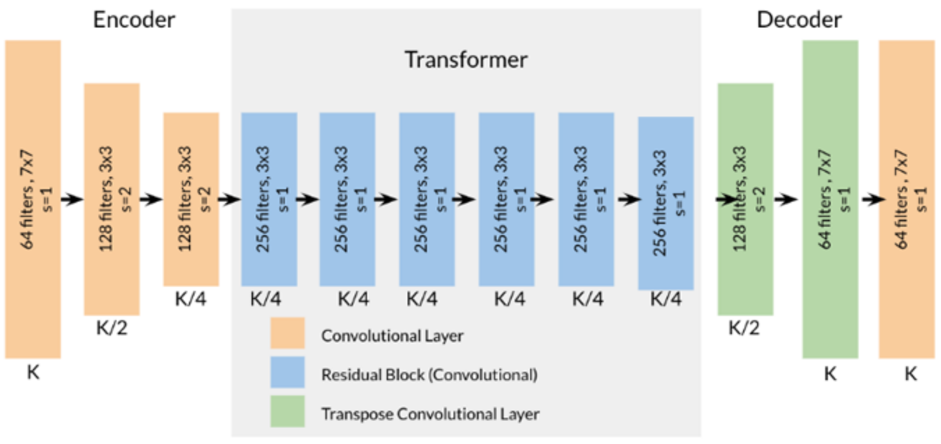


Figure 6.2: CycleGAN Generator structure

6.2.3 CycleGAN Loss Functions

Cycle Consistency Loss

For each of the parts that conform to the CycleGAN, it is added a loss term that calculates the pixel differences between the real image that is used as the input, to the generated image that has passed through a cycle of the generators. On the first Generator $G : X \to Y$ the loss function goes:

$$\sum(x - F(G(x))) \tag{6.2}$$

Where the difference is between the real image (x) and the image that the generator G creates $(G(x))$ which is then passed through F $(F(G(x)))$. On the other hand, the second generator $F : Y \to X$ loss function goes:

$$\sum(y - G(F(y))) \tag{6.3}$$

Which calculate the same function but for the other types of images y.

Both terms are added to the loss function which is shared between both generators multiplied by a factor λ as follows:

Layer (type)	**Output Shape**	**Param #**
Conv2d-4	[-1, 128, 50, 50]	18,496
InstanceNorm2d-5	[-1, 128, 50, 50]	0
ReLU-6	[-1, 128, 50, 50]	0
Conv2d-7	[-1, 128, 50, 50]	73,856
InstanceNorm2d-8	[-1, 128, 50, 50]	0
ResidualBlock-15	[-1, 128, 50, 50]	0

Table 6.1: Residual Block

Layer (type)	**Output Shape**	**Param #**
Conv2d-1	[-1, 32, 200, 200]	4,736
FeatureMapBlock-2-2	[-1, 32, 200, 200]	0
Conv2d-3	[-1, 64, 100, 100]	4,736
InstanceNorm2d-4	[-1, 64, 100, 100]	0
ReLU-3	[-1, 64, 100, 100]	0
ContractingBlock-3	[-1, 64, 100, 100]	0
Conv2d-3	[-1, 128, 50, 50]	4,736
InstanceNorm2d-4	[-1, 128, 50, 50]	0
ReLU-3	[-1, 128, 50, 50]	0
ContractingBlock-3	[-1, 128, 50, 50]	0
Residual Block 1 to 9 (6.1)	[-1, 128, 50, 50]	0
ConvTranspose2d-64	[-1, 64, 186, 186]	73,792
InstanceNorm2d-65	[-1, 64, 186, 186]	0
LeakyReLU-66	[-1, 64, 186, 186]	0
ConvTranspose2d-67	[-1, 32, 180, 180]	18,464
InstanceNorm2d-68	[-1, 32, 180, 180]	0
LeakyReLU-69	[-1, 32, 180, 180]	0
Conv2d-70	[-1, 3, 200, 200]	4,707
InstanceNorm2d-71	[-1, 3, 200, 200]	0
LeakyReLU-72	[-1, 3, 200, 200]	0

Table 6.2: CycleGAN Generator

$$\iota_{cyc} = \lambda \times (\sum (x - F(G(x))) + \sum (y - G(F(y)))) \tag{6.4}$$

Adversarial Loss

The adversarial loss function used is the least squares loss function, which is commonly used in CycleGAN. This loss function is used to help with training stability, and more specifically with the vanishing gradient problem. This function uses a method that minimizes the sum of square residuals between the prediction D(x) and the label (1 for real images and 0 for fakes).

- **Discriminator**

  On the discriminator, the functions go as follows:

Layer (type)	**Output Shape**	**Param #**
Conv2d-1	[64, 197, 197]	3,136
LeakyReLU-2	[64, 197, 197]	0
Conv2d-3	[128, 194, 194]	131,200
InstanceNorm2d-4	[128, 194, 194]	0
LeakyReLU-5	[128, 194, 194]	0
Conv2d-6	[256, 191, 191]	524,544
InstanceNorm2d-7	[256, 191, 191]	0
LeakyReLU-8	[256, 191, 191]	0
Conv2d-9	[512, 188, 188]	2,097,664
InstanceNorm2d-10	[512, 188, 188]	0
LeakyReLU-11	[512, 188, 188]	0
Conv2d-12	[1, 188, 188]	513

Table 6.3: CycleGAN Discriminator

$$E_x[\![(D(x)-1)^2]\!] + E_z[\![(D(G(z))-0)^2]\!] \tag{6.5}$$

Where E_x are the real images, whose predictions from the discriminator are compared to the real ground truth 1, and E_z are the generated images that are passed to the discriminator to achieve the difference between the prediction of the discriminator and its fake ground truth 0. This equation can be simplified into:

$$E_x[\![(D(x)-1)^2]\!] + E_z[\![(D(G(z)))^2]\!] \tag{6.6}$$

- **Generator**

  On the generator's side, the functions go as follows:

$$E_z[\![(D(G(z))-1)^2]\!] \tag{6.7}$$

  As the generator's role is to create an image as realistic as possible, the loss function reflects the distance between those images and the real ground truth 1.

Both terms are unified into an adversarial loss term by adding them up, so the term ends up like this:

$$\iota_{adv} = E_z[\![(D(G(z))-1)^2]\!] + E_x[\![(D(x)-1)^2]\!] + E_z[\![(D(G(z)))^2]\!] \tag{6.8}$$

Identity Loss

This is an optional loss term that is introduced to the loss function to aid with color preservation in outputs. When putting an image y that goes through the mapping $G : X \rightarrow Y$ the result should be the same input y. This loss term calculates the pixel difference between y and $G(y)$ which should be the same images.

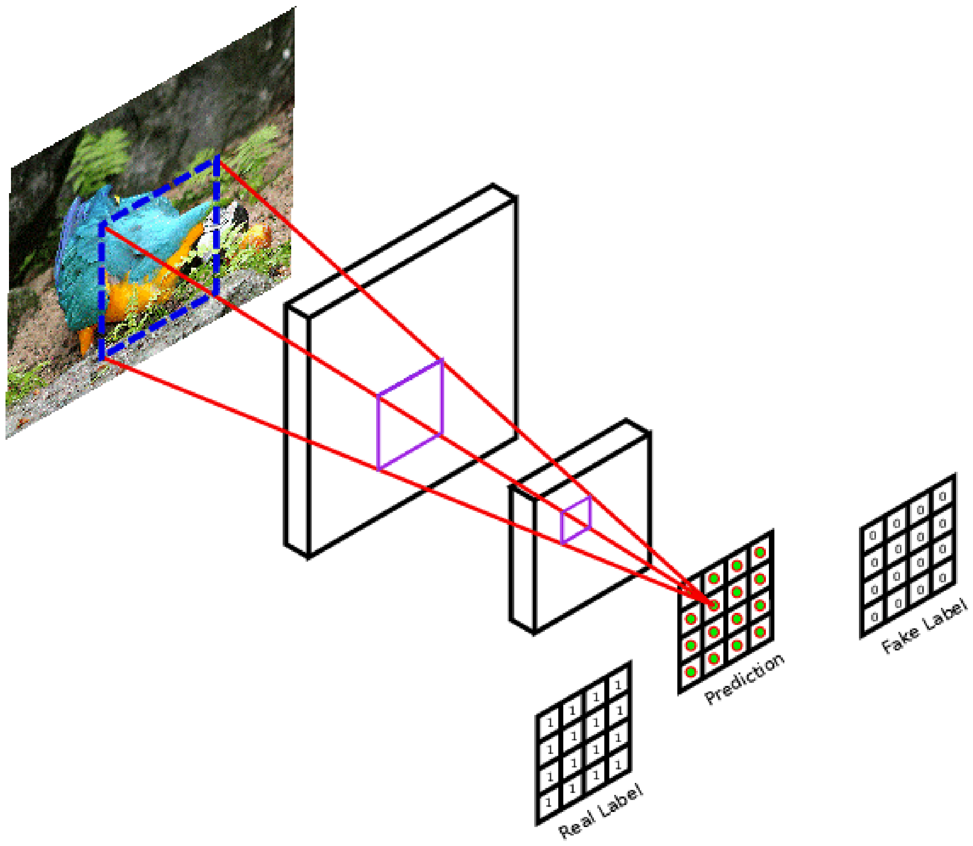


Figure 6.3: PatchGAN patch division

$$\iota_{id} = \sum [\![x - F(x)]\!] + \sum [\![y - G(y)]\!] \tag{6.9}$$

Full Loss Function

Taking every loss term previously explained, the CycleGAN loss function ends up as follows:

$$Loss = \iota_{adv} + \lambda_1 \times \iota_{cyc} + \lambda_2 \times \iota_{id} \tag{6.10}$$

6.3 Training Details

Because this sort of GAN only requires two classes to learn, only two of the classes from the dataset specified in 4.4.1 were used for training the CycleGAN.

The two fruits that were originally chosen were the apple and the pear. More precisely, the classes Apple Red 1 and Pear from the original dataset. Which results in a dataset consisting of 984 images.

The hyperparameters used to train this model are :

- **Batch size** : 1
- **Learning rate** : 0.002
- **Loss function** : The mix of Cycle Consistency Loss, Least Squares Loss and Identity Loss presented in Equation 6.10
- **Adam Betas** : 0.9 and 0.999
- **Image resolution** : 200 × 200

6.4 Experiments and Results

Once the CycleGAN was fully implemented, the model was trained using only the pears and the apples, leaving the rest of the fruits unused. The objective was to test which features the CycleGAN was able to transfer between the two fruits and the quality of the images it produced.

The following figures show the generator's output at the start and end of the first epoch, and the remainder are the outputs after each epoch, which are epochs 3, 15, and 20. The lower row displays the generated image's output, while the upper row displays the real sample of the class used by the generator to generate each matching result.

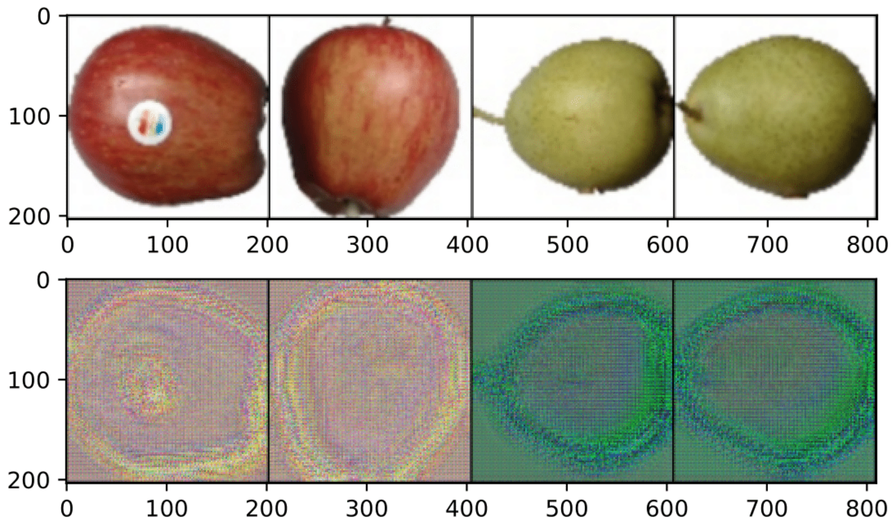


Figure 6.4: Training results at the beginning of the first epoch for apples and pears.

At the start of training, as shown in Figures 6.4 and 6.5, it is clear that the GAN only tried to change its colour while maintaining its shape to create the image, but by the fourth epoch the colour was completely mimicked, and the finer details were the only ones left (As demonstrated in Figure 6.6).

By epoch 15 , as shown on Figure 6.7, it is clear that the color and skin have been replicated, resulting in an excellent result. Finally, at epoch 20 , in Figure 6.8, the fruit has changed color to the other type but has begun to try to change the shape, indicating

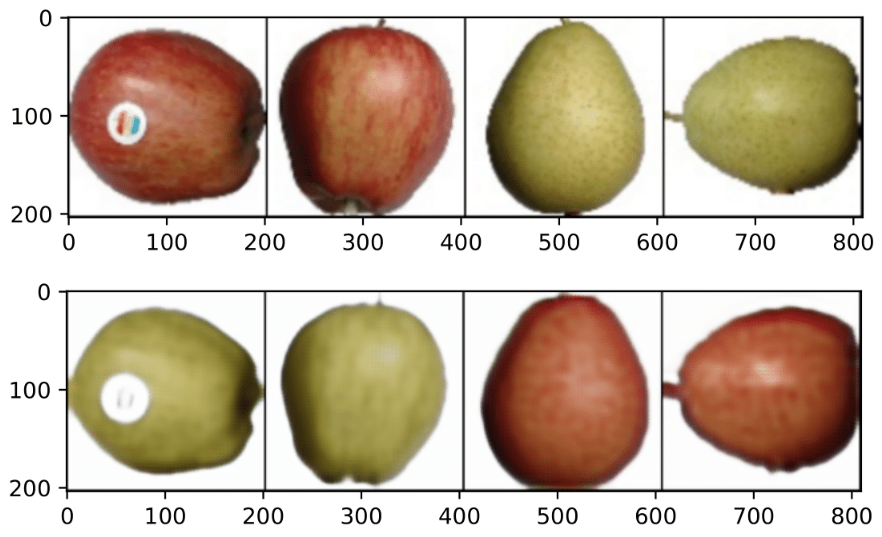


Figure 6.5: Training results at the end of the first epoch for apples and pears.

that the GAN is attempting to learn to transfer the shape, which is not what we want. As a result, we must halt the training.

Due to the small size of the Dataset for training and the similarities in shape and form of the two types of fruit, the training took less than 20 epochs to reach a point where the learning rate was insufficient to justify the continuation, as seen in Figure 6.9, and thus the training was assumed to be completed. The results show that even the finer details like the shine and the rind texture were changed in order to mimic the other fruit.

6.5 Conclusions

During this chapter, we've shown how to combine Patch GANs with U-Nets to create the CycleGAN, a model capable of transferring features from one domain to another by learning two mapping functions. This architecture attempts to solve a slightly different generative challenge in which the image must remain unmodified after being converted into another and then restored to the original image to complete the cycle. As a result, the loss function had to be modified to include a term that measured the difference between the input and output images, in addition to the one that considered the image's realism.

This experiment served us to peek into a different kind of generative problem and learn how to approach it using GANs. It provided us with a broad enough perspective to continue learning about more difficult challenges connected to this one, such as creating and applying segmentation maps in computer vision tasks.

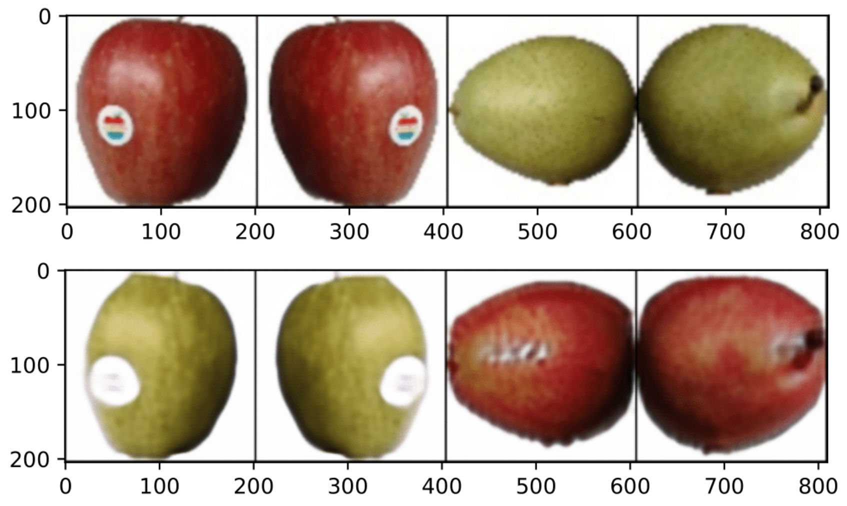


Figure 6.6: Training results at the end of the third epoch for apples and pears.

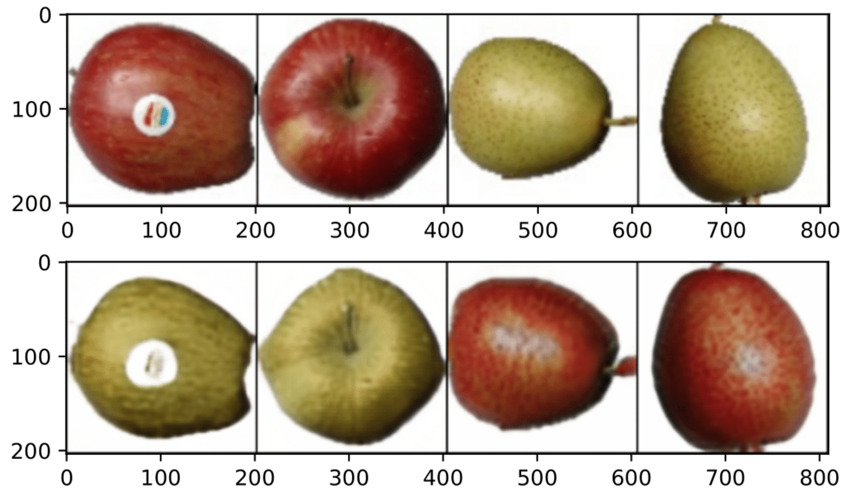


Figure 6.7: Training results after 15 epochs for apples and pears.

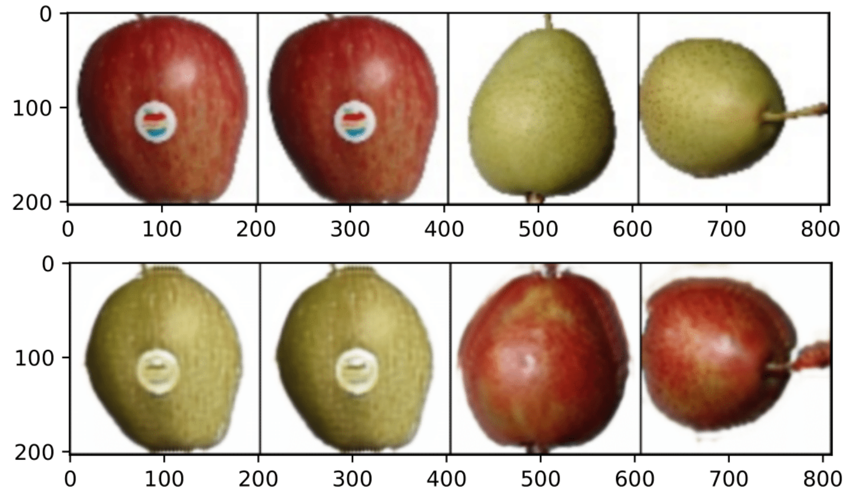


Figure 6.8: Training results after 20 epochs for apples and pears.

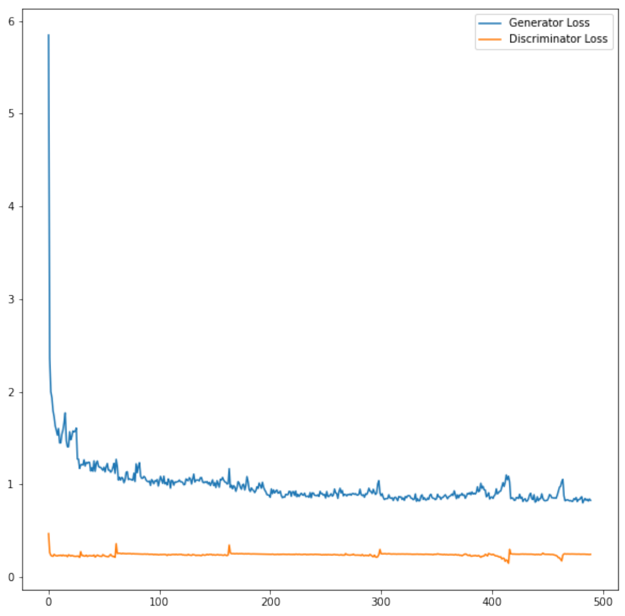


Figure 6.9: Loss function over time (measured in batches)

Chapter 7

Conclusions

7.1 Summary

During this project, four different generative models were discussed, implemented, and tested, each more complicated and powerful than the one before it:

- The basic GAN, which was capable of learning and reproducing the characteristics of a single class to provide acceptable instances.
- Then, a Conditional GAN, which produced better results than the simple one due to its capability to learn features from multiple classes thanks to the extended class information in its input, and also provided control over the features of the generated samples.
- The Style GAN came after the Conditional GAN. This greatly enhanced the quality of the generated samples, significantly expanded the user's control over the characteristics in the generated images, and vastly expanded the diversity of the created samples. All of these accomplishments were made possible by modifications made to the generator to allow it to adapt image styles, including the AdaIn blocks, progressive growth, and the mapping network.
- Finally, the Cycle GAN. This last architecture took the style manipulation to the next level by being able to transfer styles from one sample to another and vice versa thanks to the introduction of U-Nets and Patch GANs as its Generator and Discriminator.

Each one of these GANs has managed to produce results that resembled the data set that was used to train them, but it is clear that not all of them would manage to fool a human observer.

The Simple GAN, even though it managed to produce images that would fool a human discriminator when trained only with one class, was not even able to completely resemble the real samples when trained with two classes, and neither was it capable of generating anything similar to the dataset used to train it when it contained more than two different classes.

The Conditional GAN, on the other hand, significantly improved the results of the previous model because, even when forced to learn features from more than ten classes and to work with three-channel images containing colors, it managed to achieve results that were closer to the dataset than the Simple GAN. The disadvantage of this model is that it did not fully learn some features, such as the color patterns in fruits like watermelon or the shape of classes like eggplant.

The Style GAN was the first to provide results with enough quality to effectively fool a human observer, but not for every class in the dataset. With fruits that share some features, it performed notably better than the others, as the oranges or the apples it produces are far more realistic than the pineapple or the pumpkin. Additionally, it was able to merge the features from different classes to produce images with decent quality.

Finally, the Cycle GAN produced excellent results that may deceive a human observer, although it was trained with fairly similar classes.

In the end, the objectives stated in the introduction of this work were fulfilled through these achievements:

- **O1 and O2:** We learned how to apply deep learning to generative processes and the theory behind GANs, as we have managed to implement several GAN architectures that have been proved to work correctly.
- **O3:** This was achieved through the successful results obtained from the Simple GAN.
- **O4:** The prove of the achievement of this objective is present both in the Conditional GAN chapter and in then Style GAN chapter, as both architecture allow the user to control the output through an extended input.
- **O5:** A proof of this improvement is present along all this work, as the results of each architecture have increasingly improved the later they are shown in this work.
- **O6:** The fact that the Conditional GAN only supports the choice of the class to which the output belongs, while the Style GAN supports more precise control over the output thanks to the style vector, along with the results obtained by both GANs, are sufficient proof that this objective has been achieved.
- **O7:** This objective has been fulfilled due to the success at implementing the Cycle GAN and at obtaining realistic results from it.

7.2 Conclusions

We now understand and comprehend how to deal with images in deep learning processes and how GANs can execute this generative task in a variety of ways as a result of all we have learned from carrying out this study.

In general, GANs are extremely complex models that cover a wide range of techniques. Their complexity rises from simple convolutions that are very simple to more difficult procedures such as AdaIn or progressive growing. GANs gain additional capabilities and are able to control the samples they create in increasingly accurate ways with each step that raises their complexity.

However, training these architectures has been a challenging process. Almost all the provided models required at least an hour to generate photos similar to those in the dataset. This, combined with the fact that the more complicated the model, the more hyperparameters and training settings are available, resulting in a long learning procedure. For the simplest architectures, such as the Simple GAN or the Conditional GAN, it only took a few hours to generate data that could be used as feedback to assess and deduce the influence of a modification in the architecture or hyperparameters during the training process. The Style GAN, in particular, took the longest due to progressive growing. Because producing 64×64 images required the model to first learn how to produce lower resolution images. The training procedure took approximately 10 hours using an NVIDIA 2060 Super GPU combined with an Intel i5-9400 CPU.

We discovered via numerous experiments that their time complexity increases extremely rapidly with the resolution and realism of the desired sample. Additionally, we have shown that another aspect that quickly climbs their time complexity is the degree of control over the generated images. This is especially noticeable in the Style GAN, where progressive growing is both vital to the quality of the generated images and control over the features in those images, as well as the bottleneck that causes the architecture to take so long to deliver acceptable results. In comparison, the Simple GAN or the Conditional GAN takes far less time to achieve their best results, albeit producing far worse images.

Finally, GANs are very promising models for obtaining high-quality results and, once trained, are quite rapid at delivering output. However, the common user lacks the processing capacity required to train these technologies to produce results that are relatively substantial in size and capable of tricking a human observer in a fair amount of time. In the future, if GPU's computation power continues to increase while remaining affordable, GANs may become a tool available to the average user, with many more applications in addition to those already described.

7.3 Future Work

Finally, we believe that this study can be utilized as a baseline in the future to explore the following work lines:

- Implement and test other models such as variational autoencoders [26] to broaden knowledge about deep learning in content generation.
- Implement and test more GAN architectures like the ProGAN [12], the PatchGAN [21], or the GauGAN [33], and try to find a way to combine the strengths of each model to produce a better one, just like the Style GAN does.

- Exploring bias and fairness on GANs to ensure that these models, just like traditional classification or regression models, don't present these problems.
- Generate other types of content such as text or music. In this work only images have been dealt with, so as every domain needs to be treated in a particular way, it would be interesting to learn how the fact that both music and video consist of elements in a precise order is dealt with in deep learning.
- Perfect the results provided by the presented models. Although the models presented in this work have produced decent results, the images generated by the Conditional GAN and the Simple GAN leave much to be desired, while the Cycle GAN and the Style GAN produce higher quality images. Bearing this in mind, it would be interesting to find a way to produce better images with the same amount of training.

Chapter 8

Personal Contributions to the Project

8.1 Guillermo García Patiño Lenza

As part of the team, I watched the online lessons of the Coursera GANs specialization and took notes, so we were able to consult them and acquire the necessary knowledge to implement the models correctly. Additionally, during the first months, I also read and took notes on some papers published related to generative models and read books in order to get some ideas on what could be achieved using GANs.

Afterward, when we began implementing the models, I contributed to the first implementation of the Simple GAN using the Google Collab python notebooks by calculating and adjusting the input and output sizes and by making them more accessible through a function that returned them.

Moving forwards, while implementing the Conditional GAN, I wrote the code that allowed to attach the tags to their corresponding samples. Additionally, during that time, I coded the original Conditional Generator and fixed an issue where the Conditional Discriminator wasn't outputting a single value per sample by fixing the kernel sizes and padding values of the convolutions in the Conditional Discriminator.

When we were to implement the Style GAN, as working in Collab Notebooks was becoming a bit of a mess, I updated our GitHub repository with the Style GAN code we had developed to that moment written and structured in regular .py files. This code contained classes that implemented both the Style Generator and the Style Discriminator, as well as utilities to plot the loss functions during the training process and to save the generated images in a selected folder.

Then, I began implementing a Style Discriminator from scratch. The first functional version of this Discriminator wasn't able to distinguish correctly a fake sample from a real one, so, advised by the director of this work, I began to tear apart each component from the Discriminator and testing simpler versions of it to detect the components that were causing the malfunction of this module.

First, I implemented a simple convolutional discriminator that was capable of telling apart samples from two different classes. Then, I tried to use progressive growing in that discriminator. This way, I managed to obtain a Discriminator who learned to differentiate between two classes much faster than the original one but still worked poorly with our Generators. After some more testing and re-reading of the bibliography, I discovered the main reason was that the progressive growing hadn't been implemented correctly. Apparently, in a progressive growing discriminator, the input must be previously downsampled using an algorithm and then ran through only the necessary blocks instead of processed all the way through the architecture with every α set to 0 except for the last one. Another fix that was implemented along this testing process was the use of fromRGB and toRGB blocks, whose function was to turn the image into a tensor with more channels to increase the information the Discriminator works with and to transform a tensor produced by the Discriminator into an image that could be merged with the downsampled one using equation 5.1.

Once the implementation of the Style Discriminator was completed, the GitHub repository contained a lot of duplicated code, since I had to implement a single function to train only the discriminator that contained the same code as the function to train the whole GAN.

To solve this, I turned the training functions into class objects. This way, each GAN architecture would have its own trainer class that offered methods to either train each module separately or the whole GAN. In addition, I implemented utilities to save the plots of the loss function and the generated samples of the models. Additionally, as training was taking more and more time for each try we made, I also implemented utility methods to save the state of the GAN periodically and load it to resume the training at any point.

Also, to speed up the training process, I wrote a Python script that preprocessed a given dataset turning each image into tensors, applying some initial transformations, and saving them all together into a .pt file. Finally, I also created some classes similar to the ones provided by pytorch that allowed us to create subsets of a given dataset.

Next, Daniela and I tried to implement and test the Style Generator. However, this time the testing process was far easier since Daniela had previously studied the Style GAN paper and achieved a useful pytorch implementation for the Style Generator. This way, I just wrapped her implementation into a class that featured the methods required to work with the trainer class I previously developed.

To finish the Style GAN implementation, I coded and tested how different cost functions worked with our architecture to find out that, in the end, the logistic loss explained in 5.3 was the one that made the model work the best, while the BCE 2.8 made the model produce almost random images and promoted mode collapse during training.

When it was time to write this document, I collaborated by writing, correcting, and generating images for most of the chapters.

In chapter 1 I wrote the introduction and the subsections for the objectives and the road map.

Then, in chapter 2 I elaborated the subsections explaining the types of layers in a neural network, the BCE, the gradient descent and the Adam algorithm, and the different ways of evaluating how well a GAN performs, the GAN challenges and the frameworks and libraries used in the project. Additionally, I also created most of the images in that chapter either by using matplotlib or inkscape (a free open source vector graphics editor).

Afterward, in chapter 3, I redacted the motivation section, the explanation of the Simple GAN architecture, and generated the sketches for the architectures of both the Conditional Generator and the Conditional Discriminator.

Moreover, in the next chapter, I collaborated in the writing of subsections for the motivation, the explanation of the architecture, the training details, and the results.

Next, in chapter 5 I collaborated on the redaction of the motivation section, the explanation of the Style GAN Generator, the Style Discriminator, its components, and the new loss function.

Additionally, in the chapter that explains the Cycle GAN, I helped correct and reviewing the motivation subsection, the explanation for the Cycle GAN architecture, and its loss functions.

Also, I wrote the final chapter that contains both the conclusions and the lines of work we suggest continuing our work in the future.

Finally, we first delivered a draft of this document to the director of this work, and he provided us with lots of changes to the structure of the chapters and to some explanations, so a reader would understand better what we tried to present. I took charge of changing the structure of every chapter following those directions, and also of reading again the whole document to check if the new structure made sense.

8.2 Daniela Alejandra Córdova Porta

I began my research for the TFG by taking the GANs course on Coursera, as did the rest of my colleagues, and in combination with this, I completed minor practical activities relating to the supplied topic and read papers connected to the architectures seen in the course. In addition, I thoroughly researched the theory utilizing books devoted to the subject.

After finishing the course and discovering that we could start building GANs, I started working on the first Conditional GAN in Google Collab. I started by creating the necessary classes, layering them according to the theory, and then implementing the code to import the Kaggle dataset into the Google Collab notebook. I began customizing it and adding features and inputs to ensure that it produced respectable results, i.e. more or less recognized fruits.

We spent hours analyzing the results and searching for papers to see what they suggested for refining the architecture because we learned it for black and white photos and we were using a fruit dataset. In other words, we wanted the GAN to be able to generate color visuals that looked like a specific fruit. I began customizing it and adding features and inputs to ensure that it produced acceptable results, that is, identifiable fruits. This meant adding new layers and changing how the GAN handled stride and padding on different layers.

We proceed with the implementation of a more complex GAN after receiving encouraging replies from the professor. In my case, I was responsible for the construction of the CycleGAN 6. I started researching articles and getting the training I needed to create this GAN by reading books and evaluating how some new layers are built in Pytorch or how they differ from the ones used in the ConditionalGAN 4. I tweaked it for several weeks to reach the intended outcomes because it required continual study and reading of related papers because we couldn't get the styles across appropriately. We were able to achieve good results after discovering that the problem was not with the architecture, but with the images that we were utilizing, which were too small for the architecture, resulting in poor learning. This first GAN was implemented in Google Collab for the first time, and then we moved it to a Jupyter notebook to train the GAN for a longer period of time. We ended up converting the code to Python files for better understanding and to be able to discover what issues were occurring and how the images were being changed. I trained this GAN and subsequent GANs using the Pycharm development environment.

Then I assisted my colleagues in the construction of the StyleGAN 5, and since several weeks had gone by with no satisfactory results, I made it my responsibility to read the StyleGAN [24] and ProGAN [23] articles step by step and observe how they were implemented in TensorFlow. We discovered that many layers and changes were not being used in our Pytorch implementation.

To avoid dragging difficulties, I designed an architecture from scratch and began comparing it to what my colleagues had built. I constructed the architectures: mapping network, synthesis network, and layers that were missing. The new layers I had to construct were specifically relevant to the notion of Equalized Learning Rate and Pixel Normalization of the Feature Vector. This was explained in the original ProGAN article but not in the

Style article, therefore we overlooked it when constructing the GAN.

After we found that they produced some but not particularly good results, I began to experiment with different loss functions to see what was going on. I determined that the Logistic was the optimal loss function for our dataset because the Generator was unable to understand the various elements that comprise a fruit's style and was overlearning when it reached dimensions bigger than 16x16. Training this GAN was one of the most difficult aspects of the project, as the model needed to be trained for several days in order to produce respectable results. In many cases, after hours of training, some layer or outcome produced a mistake, forcing us to restart.

There were also issues with early results in some of the models, which were whitish, but with regular study of how a dataset is loaded and how the dataset modification influences the images, I was able to acquire better results. As a consequence, the fruits produced by CycleGAN and StyleGAN are non-whitish and vibrant.

In terms of the TFG report, I contributed to all chapters, both theoretically and in the presentation of the GAN architecture or outcomes. I contributed to the State of the Art Chapter 2 by writing portions, completing others started by colleagues, and modifying their wording for greater understanding.

Then I wrote a portion of the architecture and theory of the ConditionalGAN 4, StyleGAN 5, and CycleGAN 6, as well as their outcomes. I created the tables with my colleagues and spent some time arranging them for better understanding and connecting them with the graphics in the respective chapters. I designed some of the visual features shown in prior chapters and provided certain conceptual parts that were required in others. In brief, I helped build all of the chapters by adding information, amending portions, and fixing typos. For the project's completion, my colleagues and I fixed the code and cleaned up the files on Github.

8.3 Mario Quiñones Pérez

As a member of the team, like my teammates, I watched the online lessons from Coursera GAN's specialization [39] and completed the exercises that were given. During the first month, all of us read books and papers and investigated the topic and everything that was needed in order to understand GANs and what results could we expect from our work. I also read about other models such as Variation Encoders [26] in order to further understand what was special about GANs and why they are more famous in the image generation field.

The first models were created using Google Collab, where we programmed in python [2] the first model that is explained during this work, the Simple GAN 3 . I was responsible for passing this model to its .py format much further into the project when we decided to change it into a GitHub repository. This process was easy as another colleague, Guillermo, made a class for training models which could be partially used. To this end I only needed to add a script that could process a dataset into a .pt file where all the images would be one channel vectors, which would mean a black and white image. This was used in order for the Simple GAN to train faster with much less information to process, as they were one channel vectors instead of three channel ones.

The first Style generator 5 was created by all of us, but I was in charge of trying to fix its problems. As it wasn't performing nearly as well as we imagined and still being at is the initial stage where it wasn't yielding any results. I tried changing its structure multiple times , as well as only training the generator and not the discriminator in order to ascertain if said generator was truly learning, but to no avail. During this process, as it wasn't yielding any promising results, it was decided that it needed to be rearranged from scratch, and Daniela mostly created this new model.

To help with training, and in order to pass it to the GitHub repository, the Simple and Cycle GANs were passed to .py and its training classes were created in order to make use of them. During the training of both models, certain aspects of them were changed in order to achieve better results, like the image size of both of them, as larger images would tend to throw an exception of out of memory, as well as the batch size and kernel size to improve efficiency while still obtaining good results.

Given how the Simple GAN model could process results that achieved what we expected on a one-class dataset, I set on testing the limits of said model, reaching the conclusion that even a two-class dataset was too complex for the said model, as shown in chapter 3 on the results section.

I helped fix bugs in the conditional GAN as its generator wasn't taking the labels of the dataset correctly, concatenating the image vector with the label in a way that the shapes of the resulting vector didn't match what the model was expecting. I also added the possibility to log the time for each epoch and the step it took to train the models in order to further understand the time needed for the models to produce good enough results and to graph the StyleGAN curve of learning and time spent on training on each step of progressive growing.

My colleagues and I all contributed to the code's repository by restructuring it and making sure it all worked after the changes. In particular, I moved both the Simple and Cycle GANs to their respective folders and added all the README files to the project.

During the realization of this document, all of us participated by writing and correcting the document. My work in particular specialized in completing chapters by adding training details and results and other minor changes that were requested by the professor, and fixing internal references and cites. Also, I contributed by rephrasing paragraphs whose initial wording wasn't on par with the rest of the document.

In the first chapter 1, I contributed by writing the first iteration of the memory structure before some changes were needed in this document, needing to also change its structure. I helped with the introduction, and made the code structure subsections.

In the following chapter, chapter 2, I contributed by adding images and fixing problems that were pointed out by our tutor. Also contributing by adding to both, GAN Architecture as well as challenges, and the Convolutional layers subsections.

In chapter 3, I added the training details and results of said training, by giving examples of all the subsections of the dataset that was used and how well this GAN performed on them. Also, I added the tables which will be later divided in two by my colleagues.

All the dataset used during this project were described by me, and all the parts that were deemed important.

on Chapter 5, I contributed to describing why a Style GAN is called so, stating where the Style came from, by adding to its components both style mixing and Random Noise injection. Also, I contributed to other parts, like the explanation of progressive growing or the mapping network. I also helped with the definition of the Generator and discriminator of the Style and how they are connected, as well as adding the tables of the generator structure.

On Chapter 6, I added all the loss functions that are introduced with the Cycle GAN and how they are all used in order to improve the learning of said model. I explained both the Generator and Discriminator and added the details and results of its training.

Bibliography

[1] Gaugan test website. `http://gaugan.org/gaugan2/`. Accessed: 2022-21-05.

[2] Python official documentation. `https://docs.python.org/3.10/`. Accessed: 2022-23-05.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Open AI. Dalle test website. `https://openai.com/blog/dall-e/`. Accessed: 2022-21-05.

[5] David Bau, Jun-Yan Zhu, Hendrik Strobelt, Bolei Zhou, Joshua B. Tenenbaum, William T. Freeman, and Antonio Torralba. GAN dissection: Visualizing and understanding generative adversarial networks. *CoRR*, abs/1811.10597, 2018.

[6] Yoann Boget. Adversarial regression. generative adversarial networks for non-linear regression: Theory and assessment, 2019.

[7] Peter J. Braspenning, Frank Thuijsman, and A. J. M. M. Weijters, editors. *Artificial Neural Networks: An Introduction to ANN Theory and Practice*, volume 931 of *Lecture Notes in Computer Science*. Springer, 1995.

[8] Min Jin Chong and David A. Forsyth. Effectively unbiased FID and inception score and where to find them. *CoRR*, abs/1911.07023, 2019.

[9] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial feature learning. *CoRR*, abs/1605.09782, 2016.

[10] Vincent Dumoulin, Ethan Perez, Nathan Schucher, Florian Strub, Harm de Vries, Aaron Courville, and Yoshua Bengio. Feature-wise transformations. *Distill*, 2018.

[11] David Foster. *Generative deep learning*. O'Reilly Media, Sebastopol, CA, July 2019.

[12] Hongchang Gao, Jian Pei, and Heng Huang. Progan: Network embedding via proximity generative adversarial network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '19, page 1308–1316, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[15] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. *Commun. ACM*, 63(11):139–144, 2020.

[16] Google. Logistic regression: Loss and regularization |machine learning crash course |google developers. `https://developers.google.com/machine-learning/crash-course/logistic-regression/model-training?hl=es_419`. Accessed 2022-16-05.

[17] J. Hany and G. Walters. *Hands-On Generative Adversarial Networks with Pytorch 1. X: Implement Next-Generation Neural Networks to Build Powerful GAN Models Using Python*. Packt Publishing, 2019.

[18] John Hany and Greg Walters. *Hands-On Generative Adversarial Networks with PyTorch 1.x*. Packt Publishing, Birmingham, England, December 2019.

[19] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[20] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[21] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2016.

[22] Yoshifumi Ito, Wataru Shimoda, and Keiji Yanai. Food image generation using a large amount of food images with conditional GAN: ramengan and recipegan. In Ichiro Ide and Yoko Yamakata, editors, *Proceedings of the Joint Workshop on Multimedia for Cooking and Eating Activities and Multimedia Assisted Dietary Management, MADiMa@IJCAI 2018, Mässvägen, Stockholm, Sweden, July 15, 2018*, pages 71–74. ACM, 2018.

[23] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017.

[24] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[26] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[28] Jakub Langr and Vladimir Bok. *GANs in Action*. Manning Publications, New York, NY, October 2019.

[29] Lars M. Mescheder. On the convergence properties of GAN training. *CoRR*, abs/1801.04406, 2018.

[30] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[31] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. *CoRR*, abs/1610.09585, 2016.

[32] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Gaugan: semantic image synthesis with spatially adaptive normalization. pages 1–1, 07 2019.

[33] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Gaugan: Semantic image synthesis with spatially adaptive normalization. In *ACM SIGGRAPH 2019 Real-Time Live!*, SIGGRAPH '19, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[35] Thomas Pinetz, Daniel Soukup, and Thomas Pock. Impact of the latent space on the ability of {gan}s to fit the distribution, 2020.

[36] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. November 2015.

[37] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021.

[38] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[39] Eda Zhou Sharon Zhou, Eric Zelikman. Specialized program: Generative adversarial networks (gans). `https://www.coursera.org/specializations/generative-adversarial-networks-gans#courses.html`, 2021. Accessed 14/05/22.

[40] Michael Soloveitchik, Tzvi Diskin, Efrat Morin, and Ami Wiesel. Conditional frechet inception distance. *CoRR*, abs/2103.11521, 2021.

[41] Orhan G Yalçın. Image generation in 10 minutes with generative adversarial networks | towards data science, Sep 2020.

[42] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *CoRR*, abs/1703.10593, 2017.