Digital avatar customization using Cycle Generative Adversarial Networks

Lucía González García

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

July 3, 2023

Supervised by: Rafael Fernández Beltrán

To my future self, to be used as motivation and inspiration for future projects.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Rafael Fernández Beltrán, for his dedication, help, orientation and advice that have made possible the elaboration of this project.

To my parents, for their support and education, and for the strength they have given me to get to where I am.

To Xavi for his support and advice, and for accompanying me throughout this process.

And to my family and friends for making me the person I am today.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

This document presents the Final Degree Work report of the Video Games Design and Development Degree by Lucía González García.

The primary aim of this project is to enhance the creation and personalization of digital avatars using Deep Learning techniques, particularly Generative Adversarial Networks (GAN) technology. The objective is to enable users to generate avatars based on their own photographs, such as selfies, and apply various desired image styles. To achieve this, an application will be developed, allowing users to upload their images and apply predefined styles such as manga, cartoons, and more. The outcome will be a harmonious combination of the original image and the selected style, resulting in a coherent visual output.

Keywords:

- Artificial Intelligence
- Deep Learning
- Generative Adversarial Networks (GAN)
- CycleGAN
- Image translation
- Digital avatars

Contents

Chapter 1

Introduction

Contents

This chapter reflects what the purpose of the work was in the beginning, why and how this project was going to be developed, which were the objectives initially fixed and how the idea started to be developed.

1.1 Work Motivation

The representation of people through digital avatars is a very important aspect of immersive user participation in many interactive applications, especially in entertainment-oriented applications such as video games [1][2][3]. Many users are attracted by the idea of being able to represent themselves in a virtual environment and make the character they control look like them in order to feel that they are the ones inside the game universe. This produces a sense of belonging and cohesion with the game, and gives the player a much more personal, immersive and satisfying experience. However, the design and customization of avatars are aspects that are often limited to a reduced set of images or pre-defined options. In this context, algorithms based on deep artificial intelligence have transformed the creation and manipulation of images by allowing the automated generation of realistic visual content, improving the quality of existing images, transferring artistic styles, recognizing and classifying objects in images, and facilitating the automatic editing and manipulation of images. For this reason I decided to investigate the application of these techniques, particularly the use of generative Neural Networks

and their derivatives, which have gained great relevance in recent times, with the aim of achieving this self-representation that the player is looking for his characters.

1.2 Objectives

Based on the motivation of the work, there are some goals to achieve:

- Learn how Generative Adversarial Networks technology and its derivatives work for digital image processing.
- Choose the most appropriate deep learning-based model for image-to-image translation
- Choose origin images and target images for translation
- Implement the selected model and evaluate its performance level for the proposed problem.
- Develop an application that allows the user to generate his own avatar with customized style.

1.3 Environment and Initial State

The original project idea was proposed by the supervisor along with a list of other project ideas from other teachers. This idea consisted of applying GAN technology (see Section 3.1.3) to perform an image transfer from a human face to a face with a predefined style, e.g. anime. Reviewing the different project ideas proposed by the teachers, without a doubt this was the one that caught my attention the most, since it combined two of the topics that most interested me, character/avatar customization and the different applications of deep learning.

As a player I consider that the customization of our characters is especially important for a correct immersion in the game, in fact, it is something to which players spend a lot of time trying to refine it as much as possible. Therefore, I find the idea of automating and simplifying this process very attractive. In addition, after almost finishing my degree, I have been able to discover that the Artificial Intelligence field is something I would like to dedicate myself professionally, for this reason, I am also interested in working with the different applications of deep learning and neural networks.

Currently, image generation with Deep Learning models has a variety of practical applications in industries such as graphic design, medicine, virtual reality, product design, image enhancement, data synthesis and artistic expression. These applications range from generating unique visual content to training diagnostic algorithms, creating virtual environments, and enhancing low-resolution images. In addition, Deep Learning models provide opportunities for creativity and artistic exploration. As research progresses, new applications and opportunities are expected to be discovered in this ever-developing

field. Specifically, image-to-image translation[4], which is the problem to be covered, is a growing and developing field with a variety of approaches and techniques. Some of these techniques are Generative Adversarial Networks (GANs), which consist of a generator and a discriminator that compete to produce realistic images; the cycle-consistency technique [5], which seeks to maintain consistency between the bidirectional translation of domains; Convolutional Neural Networks (CNNs), which stand out for their ability to capture visual features and complex patterns; or even pixel transformation as a common technique applied directly to image pixels.

CHAPTER 2

Planning and resources evaluation

Contents

This chapter deals with a technical part of the work. It also shows the planning that has been followed to complete the project and the resources used to accomplish that purpose.

2.1 Planning

In this section, the detailed time planning of the work, including all its tasks and subtasks are shown.

The main tasks that were planned at the beginning, in order to achieve the main objective of transforming a human face into a face of the chosen style, are the following in the Table 2.1.

Below is a Gantt Chart (see Figure 2.1) showing in detail the initial organization of the project along with its estimated duration, as well as the breakdown into subtasks. Throughout the development there have been some modifications with respect to the original planning. Some tasks have taken more hours than expected, such as the research and choice of the most appropriate type of GAN model for this project. And some others have taken less time, such as the development of the application. But in general, the tasks are quite close to the original planning.

To follow up the work Trello tool has been used to organize the wanted tasks (see Figure 2.2) to do and the wanted milestones to achieve, although as mentioned before,

TASK	ESTIMATED DURATION	FEBRUARY	MARCH	APRIL	MAY	JUNE	JULY
Study and learning of the GAN and styleGAN technology	**30h**						
Become familiar with the Google Colab environment and investigate TensorFlow, Keras and GAN.	10h						
Researching GAN examples in Colab	5h						
Become familiar with the GAN example in Colab	5h						
Read articles and study StyleGAN	10h						
Analysis of alternatives and choice of the deep learning based model	**30h**						
Researching and testing StyleGAN examples in Colab	5h						
Looking for alternatives, read articles and study CycleGAN	10h						
Researching and testing CycleGAN examples in Colab	10h						
Comparing CycleGAN and StyleGAN results	5h						
Implementation of the selected model	**80h**						
Investigate different implementations of the chosen model and choose the best base implementation.	10h						
Testing with the chosen model implementation	10h						
Modify input data	10h						
Modify and adjust parameters	20h						
Perform and analyze accuracy tests	30h						
Elaboration of a dataset for the generation of avatars	**10h**						
Search of datasets	5h						
Adjustment and elaboration of the dataset	5h						
Training and analysis of the model for the customization of digital avatars	**30h**						
Development of the user interface	**60h**						
Research on which tool to use for the development of the application and GUI.	5h						
Implement basic user interface	15h						
Implement Image Capture	10h						
Implement weight loading and image generation	20h						
Design and aesthetics	10h						
Documentation and presentation	**60h**						
Stadia and progress	10h						
Organize the analysis and design document, and define planning and other follow-up documents.	5h						
Analysis and design document development.	15h						
Drafting of the report	20h						
Prepare presentation and defense	10h						

Figure 2.1: Gantt chart

TASK	TIME
Study and learning of the GAN and styleGAN technology	30 h
Analysis of alternatives and choice of the deep learning based model	30 h
Implementation of the selected model	80 h
Elaboration of a dataset for the generation of avatars	10 h
Training and analysis of the model for the customization of digital avatars	30 h
Development of the user interface	60 h
Documentation and presentation	60 h

Table 2.1: Main tasks

along the process there have been modifications in the initial planning and therefore in the deadlines of the tasks and milestones. In addition, follow-up meetings have been held with the tutor both in person and remotely via Google Meet for the resolution of doubts and monitoring of the project.

2.2 Resource Evaluation

In this section, the resources evaluation (the human resources and the equipment necessary to develop and implement the work) are described, as well as the estimated cost of this resources.

- **Hardware:** Personal computer. Its initial cost was about 1000€, although, after 5/6 years, its current price will be much lower.
  - **CPU:** Intel Core i7-8750H
  - **GPU:** NVIDIA GeForce GTX 1050
    - Dedicated video memory: 4GB GDDR5
    - CUDA cores: 640
  - **RAM:** 16GB
  - **OS:** Microsoft Windows 10 Pro

  The GPU together with the CUDA cores will be the most important part to be able to perform the necessary computations of the deep learning model.

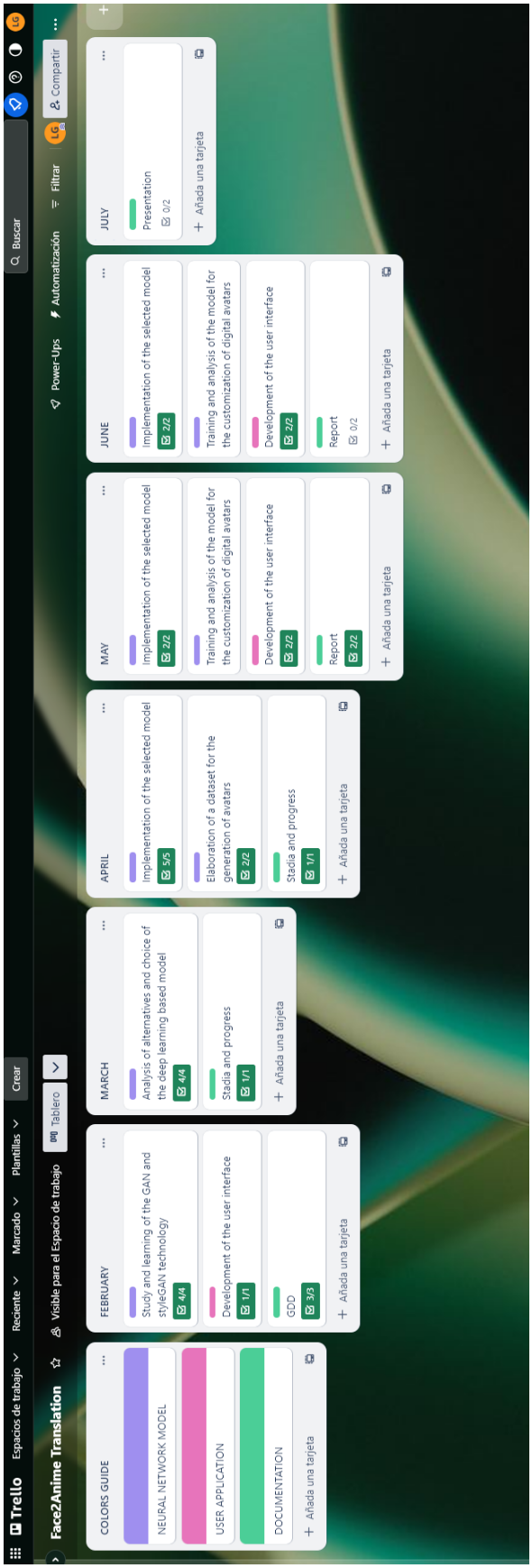

Figure 2.2: Trello

- **Software:**
  - **Google Colab (Free)[6]:** Execution virtual platform for performing network training, tests, trials and annotations. The free version has been used although it has some GPU usage limitations and therefore Some drawbacks have been found to be able to train the model. There are also several paid versions that include some improvements.
  - **Jupyter Notebook (Free)[7]:** Also an execution platform for performing network training, tests, trials and annotations. This tool has been used as a complement together with Google Colab to be able to perform the training of the model from the personal computer.
  - **Anaconda (Free) [8]:** For managing packages, libraries and programming environments.
  - **PyCharm (Free)[9]:** Programming environment to implement the application in Python.
  - **TensorFlow and Keras (Free)[10][11]:** Open Source Neural Network libraries written in Python for the implementation of the neural network model.
  - **Tkinter (Free)[12]:** Graphical library for the creation and development of desktop applications in python for the development of the user interface.
  - **OpenCV (Free)[13]:** Free library of artificial vision for image capture for the application.
  - **GitHub (Free)[14]:** Website for hosting projects using the Git version control system. It is primarily used for creating source code for computer programs.
  - **Trello (Free)[15]:** Project management and task organization software.
  - **TeXstudio (Free)[16]:** LaTeX editor for writing this report.
  - **Google Docs (Free):** Google tool for creating and editing documents that did not require too much quality.

- **Human resources:** Regarding human resources, the author of this project has been the only person in charge of the development of the project and therefore it has not implied any additional cost. However, the following is an estimate of the economic cost that would have been necessary to pay one person for the work done and the hours spent, if this had been a professional project.

  In the development of the project, two very important profiles stand out: data engineer for the development, research and analysis of the neural network model, whose average salary in Spain is 19 €/h according to Indeed website [17]; and a programmer, in this case a junior programmer, whose average salary in Spain is 10.59 €/h according to Indeed website [18]. The hours dedicated to this project have been 260 hours approx. as a data engineer and 50 hours approx. as a junior programmer. Therefore, the total cost would be 5470 € approx.

CHAPTER 3

System Analysis and Design

Contents

This chapter first presents a contextualization and definition of some terms to facilitate the understanding of the model that has been developed for this project and which will be explained in Section 4.1.1. It also presents the requirements analysis, the design and architecture of the proposed work, as well as the design of its interface.

3.1 Theoretical Framework

First, a brief explanation of some terms is shown in order to better understand the concept of neural networks and how they are specialized to arrive at the model that has been chosen.

3.1.1 Artificial Neural Network

A neural network is a method of artificial intelligence that teaches computers to process data in a way that is inspired by the way the human brain does. It is a type of machine learning process called deep learning, which uses interconnected nodes or neurons in a layered structure that resembles the human brain (see Figure 3.1). It creates an adaptive system that computers use to learn from their mistakes and continuously improve.

In this way, artificial neural networks attempt to solve complicated problems, such as summarizing documents or recognizing faces, with greater accuracy.

In this context, an artificial neuron consists of three main components: inputs, weights and an activation function. Inputs represent the signals or input values that the neuron receives from other neurons or the environment. Each input is associated with a weight, which determines the relative importance or impact of that input on the output of the neuron. The activation function is a mathematical function that takes the weighted sum of the inputs multiplied by their respective weights and transforms it into an output. This function introduces nonlinearities and defines how the neuron responds and generates an output depending on the inputs received. The output of an artificial neuron can be used as input for other neurons in the network, thus forming a network of connections that allows parallel information processing and the performance of complex tasks, such as pattern recognition, data classification or decision making.

The way an artificial neural network learns to perform these tasks is through what is known as training. The training cycle of a neural network can be summarized in the following steps:

- **Data preparation:** Training data are collected, cleaned, and divided into training and test sets.
- **Neural network design and construction:** The network architecture is defined, including the number of hidden layers, the number of neurons per layer and the activation functions. Then, the network is built using a library or a deep learning framework, such as TensorFlow or PyTorch.
- **Initialization of the weights:** Before training, the weights of the network are initialized randomly or by some other method.
- **Forward propagation:** A training data set is taken and passed through the neural network from the input layer to the output layer. Calculations are performed on each neuron and the corresponding activation functions are applied.
- **Loss function calculation:** The output predicted by the network is compared to the real output of the training set using a loss function to measure the discrepancy.
- **Error backpropagation:** The gradient of the loss function with respect to the network weights is calculated and propagated backward through the network to adjust the weights of each neuron using optimization methods such as gradient descent. This allows the network to "learn" and adjust to improve its performance.
- **Repetition of the previous 3 steps:** The steps of forward propagation, loss function calculation, error backpropagation and weight update are repeated for several iterations or epochs, using different batches of training data at each iteration.
- **Model evaluation:** After training, the performance of the model is evaluated using test or validation data to determine its generalizability.

- **Model tuning and optimization:** If the model performance is not satisfactory, adjustments are made to the network architecture, hyperparameters (values that are set before training the model and control the behavior and performance of the network) or dataset to improve its accuracy.

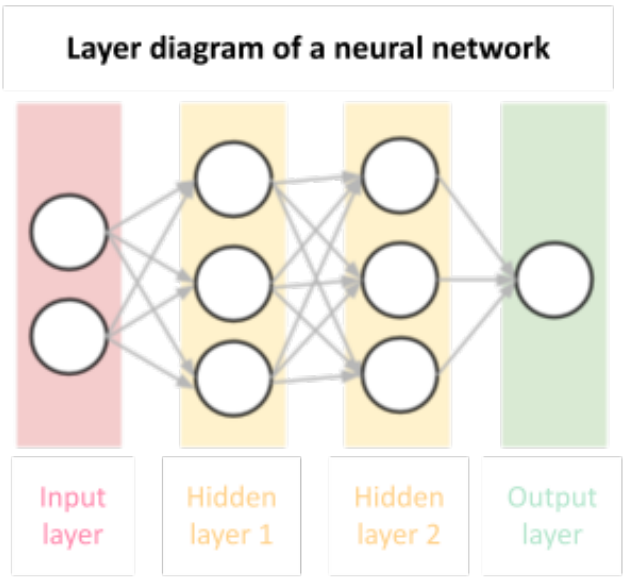

Figure 3.1: Layer diagram of a neural network

3.1.2 Convolutional Neural Network (CNN)

Convolutional Neural Networks or CNN is a type of multilayered Artificial Neural Network with supervised learning that processes its layers by mimicking the visual cortex of the human eye to identify different features in the inputs that ultimately enable it to identify objects and "see". To do this, the CNN contains several specialized hidden layers with a hierarchy. This means that the first layers can detect lines, curves and become specialized until they reach deeper layers that recognise complex shapes such as a face or the silhouette of an animal. This architecture is useful in several applications, mainly in image processing as well as video recognition and natural language processing tasks.

CNNs work as follows: To obtain the input layer, the network takes pixels from an image, each one will be an input neuron. It then performs a convolution that consists of applying a set of kernels to groups of nearby pixels, and generates a new layer of hidden neurons. Finally, it reduces the number of neurons using subsampling before performing

a new convolution (see Figures 3.2 and 3.3). It performs several convolutions until obtaining the output layer that will have the number of neurons corresponding to the classes we are classifying, for example, if we classify dogs and cats, there will be 2 neurons (see Figure 3.4).

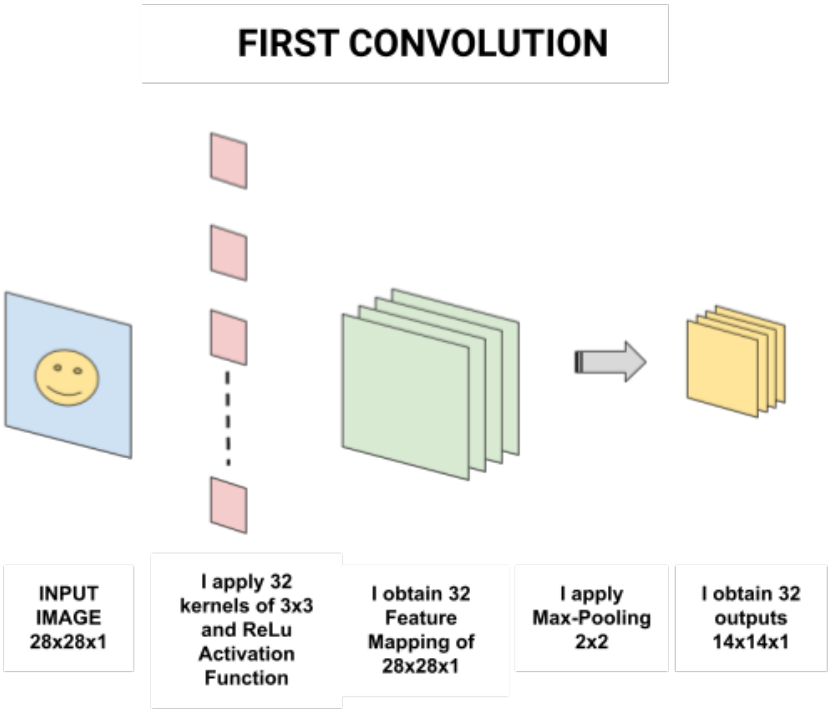

Figure 3.2: First convolution of a CNN

3.1.3 Generative Adversarial Network (GAN)

Generative Adversarial Networks or GANs, are networks that can learn to create samples, similar to the data we feed them with. The idea behind GAN is to have two competing neural network models. One, called Generator, initially takes "junk data" as input and generates samples. The other model, called the Discriminator, receives both samples from the Generator and the real training set and should be able to differentiate between the two sources. These two networks play a continuous game where the Generator learns to produce more realistic samples and the Discriminator learns to distinguish between real data and artificial samples (see Figure 3.5). These networks are trained simultaneously to finally achieve that the generated data cannot be distinguished from real data. Its main applications are the generation of realistic images, but also the improvement of existing images.

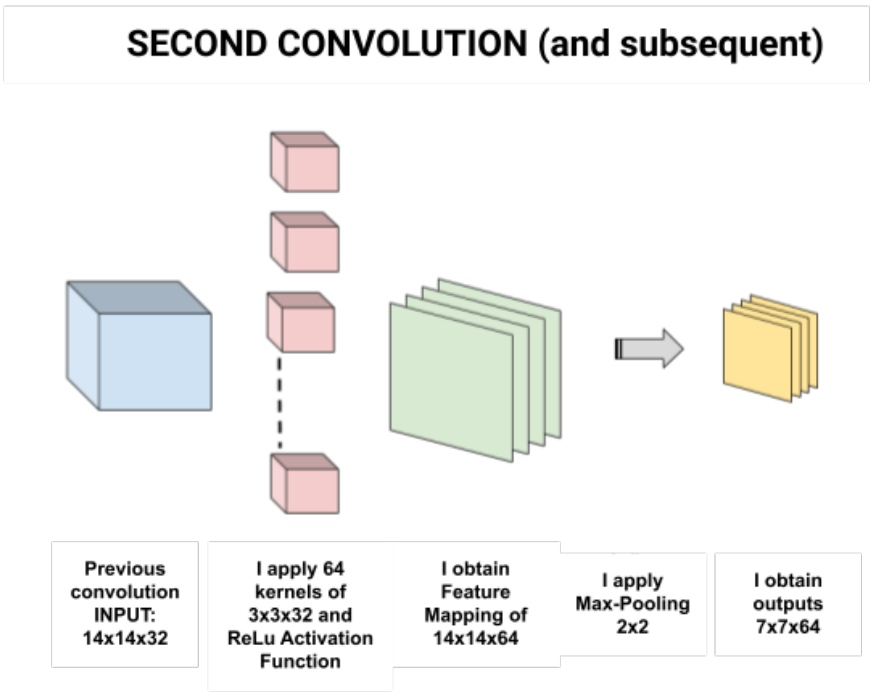

Figure 3.3: Other convolutions of a CNN

The Different Types of Generative Adversarial Networks (GANs) are:

- **Vanilla GAN:** The Vanilla GAN is the simplest type of GAN.The generator captures the data distribution meanwhile , the discriminator tries to find the probability of the input belonging to a certain class, finally the feedback is sent to both the generator and discriminator after calculating the loss function , and hence the effort to minimize the loss comes into picture.

- **Conditional Gan (CGAN):** In this GAN the generator and discriminator both are provided with additional information that could be a class label or any modal data. As the name suggests the additional information helps the discriminator in finding the conditional probability instead of the joint probability.

- **Deep Convolutional GAN (DCGAN):** This is the first GAN where the generator used deep convolutional networks , hence generating high resolution and quality images to be differentiated.

- **CycleGAN:** This GAN is made for Image-to-Image translations, meaning one image to be mapped with another image.

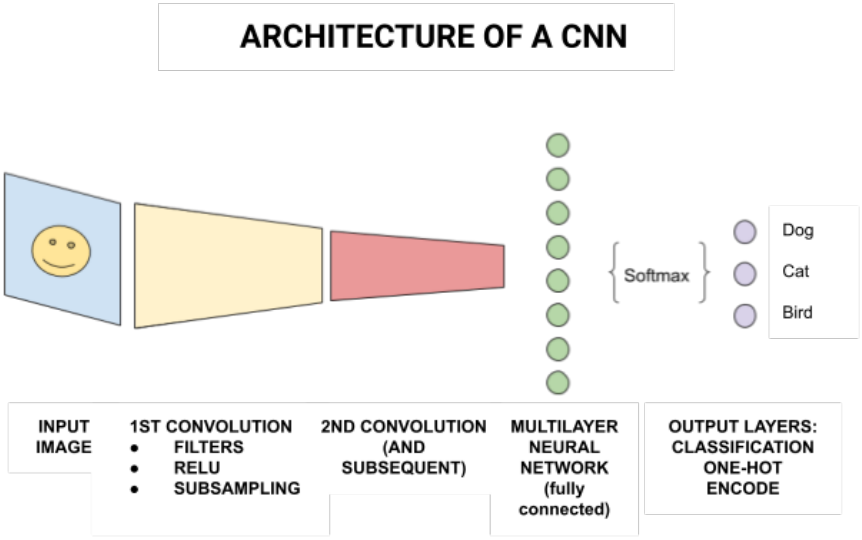

Figure 3.4: Architecture of a CNN

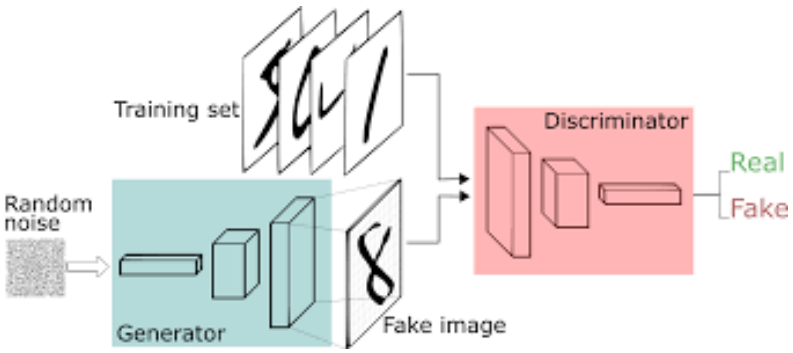

Figure 3.5: Architecture of a GAN

- **Generative Adversarial Text to Image Synthesis:** In this the GANs are capable of finding an image from the dataset that is closest to the text description and generate similar images.

- **Style GAN:** Other GANs focused on improving the discriminator in this case we improve the generator. This GAN is generated by taking a reference picture.

- **Super Resolution GAN (SRGAN):** The main purpose of this type of GAN is to make a low resolution picture into a more detailed picture. This is one of the most researched problems in Computer vision.

3.2 Requirement Analysis

In this section, the functional and non-functional requirements of the presented work will be detailed, but first, let's clarify how the application works, and then the requirements will be clearer.

As the importance of the project lies in the development of the Neural Network, which will be explained later (see Section 4.1.1), the application is not too complex and shows, in a basic way, an example of how the developed system works.

The application has been developed for Windows and when starting it, the user can choose whether to take a picture using the camera or upload it from the device (see Figure 3.10), then there is a loading period in which the new image is generated with the predefined style, and then it is displayed on the screen. Finally, the user can save the generated image on the device.

3.2.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. Once the previous explanation is clear, it is easy to identify which are the functional requirements in this project:

- **R1:** The user can take a photo.

- **R2:** The user can upload an image.

- **R3:** The Neural Network model can generate a new image.

- **R4:** The user can save de new image generated.

3.2.2 Non-functional Requirements

Non-functional requirements impose conditions on the design or implementation. In this project, the non-functional requirements are:

- **R5:** The User Interface is simple.
  The application is not too complex, but gives the user the necessary information to understand how the application works.

- **R6:** The application can be run on PC.
  Implementation and testing has been performed on Windows OS.

- **R7:** The image is quickly generated.
  The image-to-image translation is very fast since only the weights of the pre-trained model are loaded to generate the new image.

- **R8:** The generated image will be anime style.
  The style chosen for training the model is applied to the user's image.

3.3 System Design

This section presents the (logical or operational) design of the system to be carried out. In the following pages are defined the cases of use taken from the functional requirements (Tables 3.1, 3.2, 3.3 and 3.4), a case use diagram (see Figure 3.6), a class diagram (see Figure 3.7) and an activities diagram (see Figure 3.8). As the application is quite simple, they are not very complex. In addition, the application is composed of a user interface and a pre-trained model, however, the Neural Network model, for simplicity, is considered as a "Black Box", i.e., its operation is not known, only its input and output are known.

Requirement:	R1
Actor:	User
Description:	The user can take a picture of his face using de camera of de device.
Preconditions:	1. The user must be on the title screen. 2. The user must have clicked the "Take photo" button.
Steps normal sequence:	1. The user clicks the "Take photo" button. 2. Camera opens. 3. The user clicks the "Take photo" button. 4. The photo appears on the screen.
Alternative sequence:	None.

Table 3.1: Case of use «CU01. Take photo»

Requirement:	R2
Actor:	User
Description:	The user can upload an image of a human face from de device.
Preconditions:	1. The user must be on the title screen. 2. The user must have clicked the "Upload image" button.
Steps normal sequence:	1. The user clicks the "Upload image" button. 2. The file directories open. 3. The user selects the image he wants to upload. 4. The image appears on the screen.
Alternative sequence:	None.

Table 3.2: Case of use «CU02. Upload image»

Requirement:	R3
Actor:	Neural Network model
Description:	The Neural Network model generates a new image by combining the user's photo with the predefined style.
Preconditions:	1. The user must have clicked the "Generate image" button. 2. There is an image of a human face. 3. There is a pretrained neural network with predefined style.
Steps normal sequence:	1. The model preprocesses the image. 2. The weights of the network are applied to the added image and the style transfer is performed. 3. The new generated image is returned.
Alternative sequence:	None.

Table 3.3: Case of use «CU03. Generate image»

Requirement:	R4
Actor:	User
Description:	The user can save in the device the image generated by the Neural Network model.
Preconditions:	1. the Neural Network model must have generated the image. 2. The user must have clicked the "Save image" button.
Steps normal sequence:	1. The user clicks the "Save image" button. 2. The file directories open. 3. The user selects the directory where the image is going to be saved. 4. The image saves on the device.
Alternative sequence:	None.

Table 3.4: Case of use «CU04. Save image»

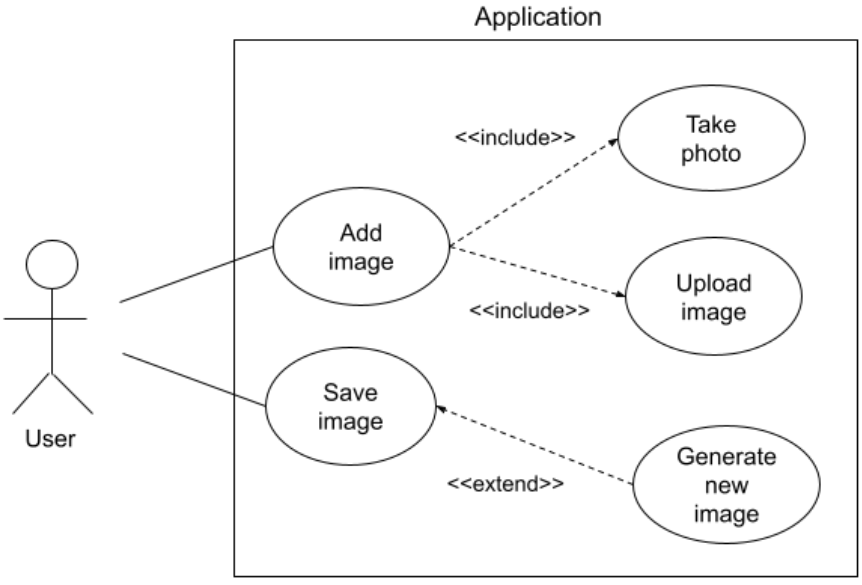

Figure 3.6: Case use diagram

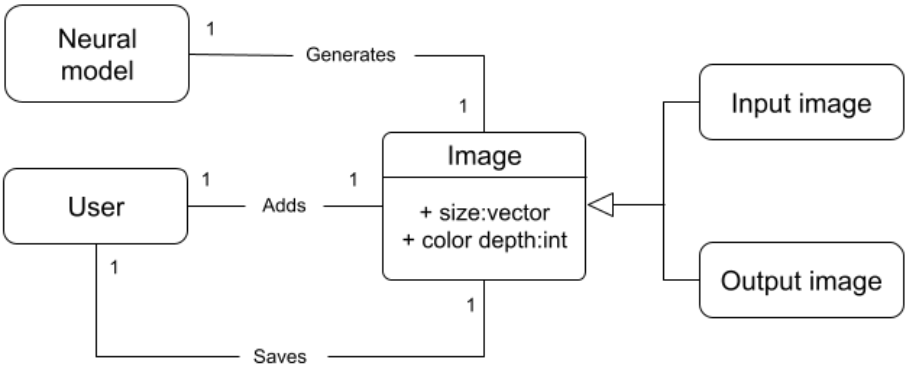

Figure 3.7: Class diagram

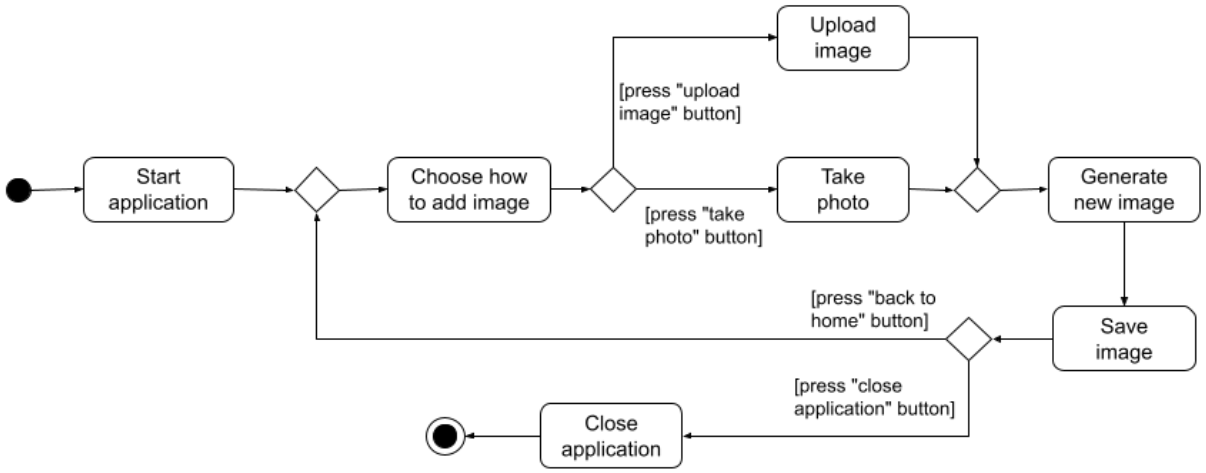

Figure 3.8: Activities diagram

3.4 System Architecture

For network training these are the minimum system requirements:

- Windows, Linux or macOS (in this case Google Colab and Jupyter Notebook execution platforms have been used)
- Python 3
- CPU or NVIDIA GPU + CUDA CuDNN

And in particular, to use the TensorFlow and Keras libraries, both for training the network and for generating the new image in the application (and therefore, for the application to work):

- Python 3.7–3.10
- Ubuntu 16.04 or later
- Windows 7 or later
- macOS 10.12.6 (Sierra) or later.

Although these are the minimum requirements to be able to train the model, for an efficient and fast training it is important to have a GPU with high capacity (CUDA cores are very relevant, as they can process tasks in parallel much faster and significantly accelerate the training time of neural networks [19]) or to have access to a virtual execution platform with more advanced GPUs such as Google Colab, or servers equipped with graphics cards with much more dedicated memory.

In addition, to use the application it is advisable to have a webcam to take the photo that we will convert, although it is not strictly necessary since it also offers the option of uploading an image from the computer.

3.5 Interface Design

As mentioned before, the most important part of this project is the implementation of the neural network and its training to achieve the most accurate results possible. The application is therefore a complement whose utility is to show the results obtained with Neural Network model. That is why the interface design is simple and precise, but gives the user the necessary information to understand how the application works. To implement the user interface, UI, the Python programming language has been used in the Pycharm programming environment together with the Tkinter library, which is in charge of the user interfaces. Section 4.1.2 explains in detail how the application has been implemented. In this section the aesthetics and design of the application will be explained.

All the elements of the application are combined to create a black and white aesthetic. The opening scene and the second scene, in which the choice between uploading an image and taking a photo is given, have a black and white image of a girl in anime style as a background (see Figures 3.9 and 3.10). This image is downloaded from the WallpaperBoat website [20] for free. To design the title (see Figure 3.9) it has been used the virtual tool CoolText [21] to manually customize the style, color and text. For the design of some buttons it has been used the virtual tool ButtonOptimizer [22] to also customize manually the style, color and typography of the buttons (see Figure 3.10). And the buttons with the camera and house icons (see Figure 3.11) are both downloaded from the Flaticon website [23][24] for free.

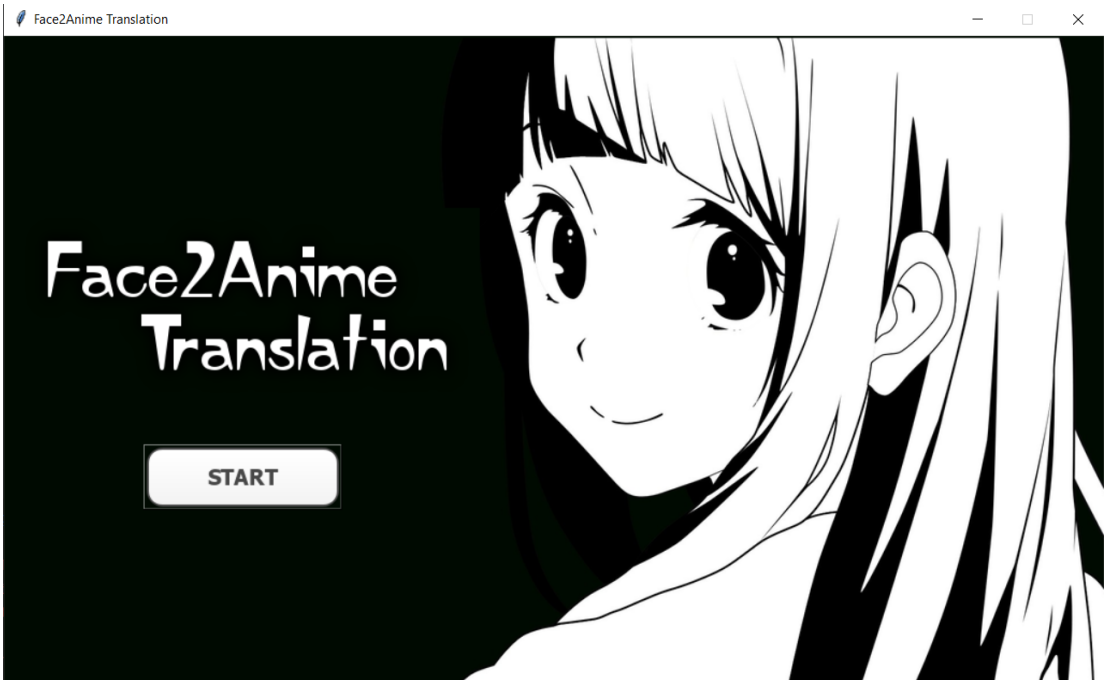

Figure 3.9: First screen of the application

Figure 3.10: Second screen of the application. Choice between uploading image or taking photo.

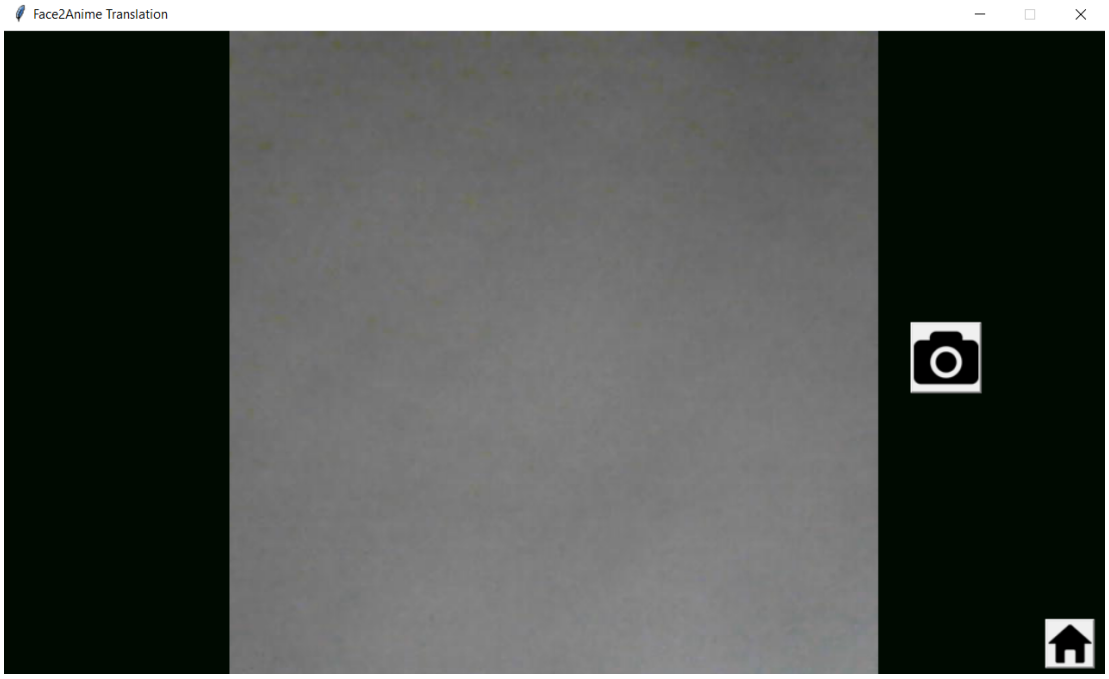

Figure 3.11: Take photo screen

Chapter 4

Work Development and Results

Contents

This chapter is an explanation of how the project has been developed since his start until the end of it. It also includes an assessment of the results and the deviations from the initial planning.

4.1 Work Development

This section shows in depth the work done and the decisions that have been taking along the process. It begins by explaining the work done with respect to the neural network, saying what type of model have been chosen. And then, the development of the application will be explained.

4.1.1 Neural Model Proposal

In this section it is going to be explained which neural network model has been implemented and how it has been done.

Despite the fact that to carry out this kind of projects we work with existing material and contents, this project was also directly oriented to research, since its importance also lies in finding out what kind of neural network is best suited to what is wanted to be achieved. This is one of the reasons why some modifications have arisen along the way.

Initially, together with the supervisor, it was decided to work with StyleGAN, which is a type of neural network that progressively increase the resolution of the generated

images and incorporate style features in the generative process. The main idea was to use a human face and apply anime style to it to generate a new image. While other GANs focus on improving the discriminator, StyleGAN tries to improve the generator by redesigning its architecture in a way that exposes novel ways to control the image synthesis process. This generator starts from a learned constant input and adjusts the "style" of the image at each convolution layer based on the latent code, therefore directly controlling the strength of image features at different scales. StyleGAN uses the baseline progressive GAN architecture and proposed some changes in the generator part of it. However, the discriminator architecture is quite similar to baseline progressive GAN. Traditionally the latent code is provided to the generator through an input layer, i.e., the first layer of a feedforward network (see Figure 4.1(a)). The StyleGAN model departs from this design by omitting the input layer altogether and starting from a learned constant instead (see Figure 4.1(b)).

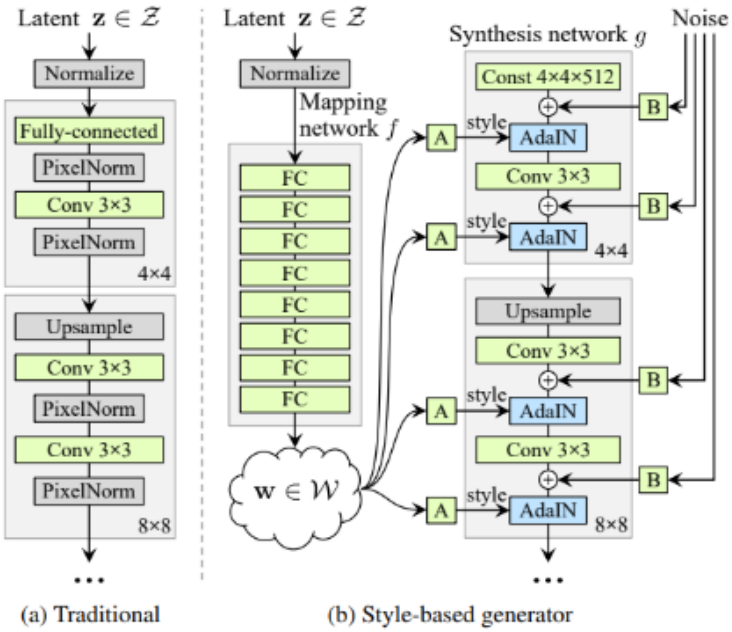

Figure 4.1: Architecture of a StyleGAN.

It started by working with the implementation provided by Nvidia [25], however, this implementation generated many errors due to version incompatibility between Google Colab and the Tensorflow library. After trying to solve these errors in a failed way, it was decided to try other sources and among them, work began with Soon Yau Cheong's implementation [26], which worked perfectly. However, after doing some tests with the code and after talking with the supervisor, it was concluded that StyleGAN could not achieve the desired results, since what wanted to be done was an Image-to-image translation, and although StyleGAN is a widely used generative neural network (GAN)

architecture for generating realistic images, it is not specifically designed for image-to-image translation due to the lack of matched training data, the ability to capture and align accurate features, and several other limitations. So, finally, it was decided that it was more appropriate to use CycleGAN, which is explained below. There are also other architectures and approaches specifically designed for this type of problem, such as JoJoGAN, Pix2Pix and UNIT, which may offer better results and deal with some of the limitations mentioned, and which are still under development.

CycleGAN

This project has implemented Cycle Generative Adversarial Network model, or CyceGAN for short, which was first proposed in the paper "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" by Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros in 2017 [5].

Image-to-image translation is a class of vision and graphics problems where the goal is to learn the mapping between an input image and an output image using a training set of aligned image pairs. However, for many tasks, paired training data will not be available. CyceGAN presents an approach for learning to translate an image from a source domain X to a target domain Y in the absence of paired examples. This method, basically, can learn to capture special characteristics of one image collection and figuring out how these characteristics could be translated into the other image collection.

As illustrated in Figure 4.2(a), this model includes two mappings $G : X- > Y$ and $F : Y- > X$. In addition, it introduces two adversarial discriminators DX and DY , where DX aims to distinguish between images x and translated images $F(y)$; In the same way, DY aims to discriminate between y and $G(x)$. Also, it works whit two types of losses, adversarial losses for matching the distribution of generated images to the data distribution in the target domain; and cycle consistency losses to prevent the learned mappings G and F from contradicting each other. Adversarial losses alone cannot guarantee that the learned function can map an individual input x_i to a desired output y_i. To further reduce the space of possible mapping functions, we argue that the learned mapping functions should be cycle-consistent: as shown in Figure 4.2(b), for each image x from domain X, the image translation cycle should be able to bring x back to the original image, i.e., $x- > G(x)- > F(G(x)) \approx x$. We call this forward cycle consistency. Similarly, as illustrated in Figure 4.2(c), for each image y from domain Y, G and F should also satisfy backward cycle consistency: $y- > F(y)- > G(F(y)) \approx y$.

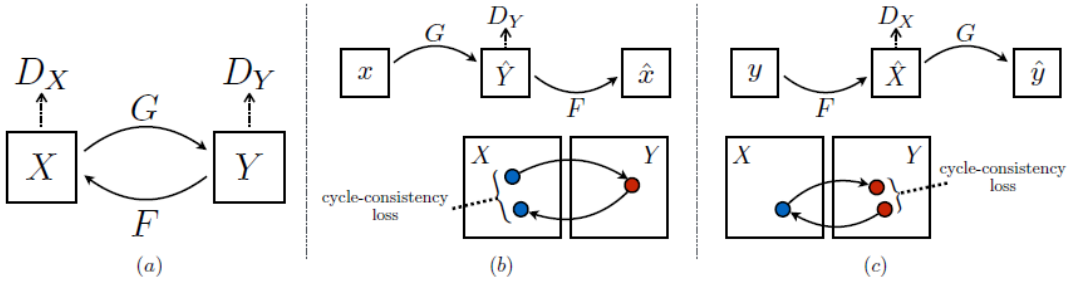

Figure 4.2: (a) This model contains two mapping functions $G : X -> Y$ and $F : Y -> X$, and associated adversarial discriminators DY and DX. DY encourages G to translate X into outputs indistinguishable from domain Y , and vice versa for DX and F. To further regularize the mappings, we introduce two cycle consistency losses that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x -> G(x) -> F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y -> F(y) -> G(F(y)) \approx y$.

Datasets

Initially, a search of the existing datasets of the possible domains to translate was made, the target being a "cartoon" model (anime, disney style, 3D model, etc.) after reviewing the content of the main sources to obtain this information, it was finally found that the anime style was the one with the most content and wealth of datasets on the Internet and therefore open to more possible adjustments, so it was decided to apply this style.

To feed the system and train the neural network, a dataset containing 4 subsets has been created. One subset with 3000 images of anime faces for training (see Figure 4.3), another with 3000 images of human faces also for training (see Figure 4.4), and two others with 100 images each for testing, one with anime faces and the other with human faces.

To create this dataset it have been used as a base the images from 3 internet datasets, Anime Faces vs Human Faces from Kaggle [27] which contains both anime and human faces, Human Faces from Kaggle too [28] which contains human faces, and the last one from a GitHub project by Lmtri1998 [29], from which only the anime faces have been used. From these datasets it have been selected the most appropriate images and discarded some images with excessively low resolution or that could contain noise (see Figure 4.5) and negatively affect the training, for example sunglasses, masks or other elements that could hide part of the facial features, although the images with vision glasses have been kept to see how the model works with those elements. On the other hand, these images with noise have not been discarded from the test set for future analysis of model accuracy. In addition, the images of human faces contained in the dataset created and used for training the model are very varied, both men and women, elderly, young and children. However, the images of anime faces are somewhat more

limited, and although it has been tried to have as much variety as possible, female faces predominate over male faces, and this could slightly affect the learning of the model. The choice of dataset is very important, as it affects the way the model learns during training and the quality of the final result.

Figure 4.3: Sample of human face images from the used dataset

Software and Development

For the development of the Neural Network it has been used Python on the Google Colab execution platform and on the Jupyter Notebook execution platform, where it have been performed network trainings, tests, trials and annotations, together with the TensorFlow and Keras (for neural networks and deep learning), Numpy (for high-level mathematical operations, vectors and large multidimensional matrices) and Matplotlib (for two-dimensional graphics generation) libraries.

As a base it has been used A-K-Nain's implementation of the CycleGan neural network [30] where he solves the image-to-image translation problem by converting images of horses to images of zebras. On this base, modifications have been made to adjust the CycleGAN model and its parameters to our requirements and our dataset.

Throughout the process it has been followed a cyclic work strategy, this means that the tasks have not been developed one after the other, but we are constantly going back on what is already worked to adjust and improve the model and make it more and more accurate. The detailed code explained in detail can be found in Appendix A.1.

Figure 4.4: Sample of anime face images from the used dataset

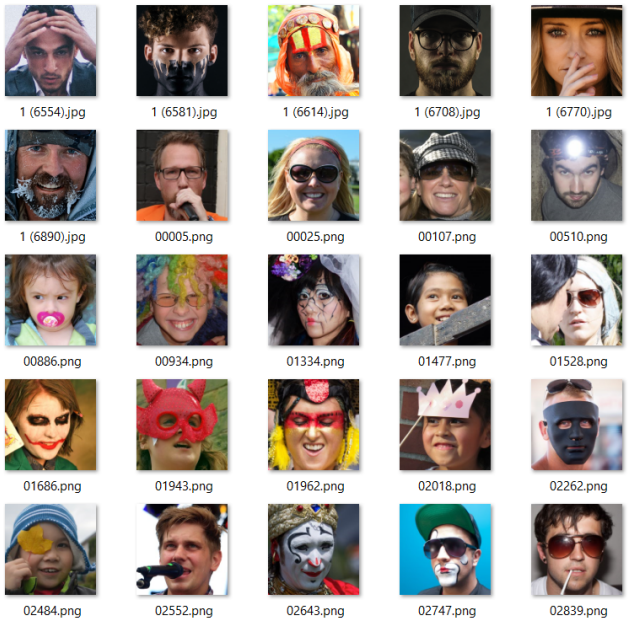

Figure 4.5: Sample of images discarded because they contain noise

The first step was to create the dataset with the appropriately selected images and then converting it into the Tensorflow Dataset type so that the images can be processed by the TensorFlow library and subsequently be used in training. This last is implemented in the model and does it automatically by passing it a path to any set of images with well specified subsets. Since two different programming environments (Google Colab and JupyterNotebook) were used to carry out this project, the paths to the dataset were different for each too. For Google Colab, the dataset was stored in Google Drive, while for Jupyter Notebook, the dataset was stored in the computer's local directories. This dataset has also been modified throughout the development of the project to better fit the requirements of the model and its different trainings.

After this, some parameters were adjusted in the code and the training sessions were started. For each training it was possible to choose the number of epochs to be performed. An epoch refers to a complete pass of the entire training dataset during the training process of a model. During an epoch, the model processes and adjusts its parameters using all examples in the training dataset. After an epoch, the model's performance is evaluated using a validation or test dataset, and metrics such as accuracy, loss, or other relevant metrics are recorded. These metrics provide an indication of how the model is improving as the training process progresses. The number of epochs used in training depends on the problem, the size of the data set and other factors; and it is important to adjust the number of epochs to obtain a balance between model fit and generalizability to new data, since a very high number of epochs can lead to overtraining the model and distorting the results (see Figure 4.6). For this reason, in this project, different trainings have been performed with different number of epochs (from 10 to 240) to see which one fits better to this model. In addition, in each training some parameters are modified , such as the batch size that specifies the number of training samples to be used in each training step to update the model parameters. It is common to divide the training dataset into smaller batches instead of using all samples at once. Having a larger batch size can speed up training since fewer weight updates are performed, but it can also require more GPU memory. On the other hand, a smaller batch size can lead to more frequent weight updates and can allow for more stable training. And this is where a problem has arisen in the development of this project, since for training in JupyterNotebook the GPU of the personal computer was somewhat limited and only supported batch of size 1 and therefore the model did much slower training, while the Google Colab GPU supported at most a batch of 6. This is one of the reasons why it was decided to use two different programming environments, as well as to perform several trainings simultaneously. Other parameters that have been modified in the different trainings are the sizes of the input and output images, and the buffer size for data shuffling or randomization operations.

Finally, after each training, the results were analyzed, both the quality and accuracy of the generated images, as well as the loss, which is a measure that quantifies the discrepancy between the model predictions and the real values of the training data. Loss is a function that is used during training to guide the model toward better predictive ability. The objective of the model is to minimize the loss, i.e. to reduce the discrepancy

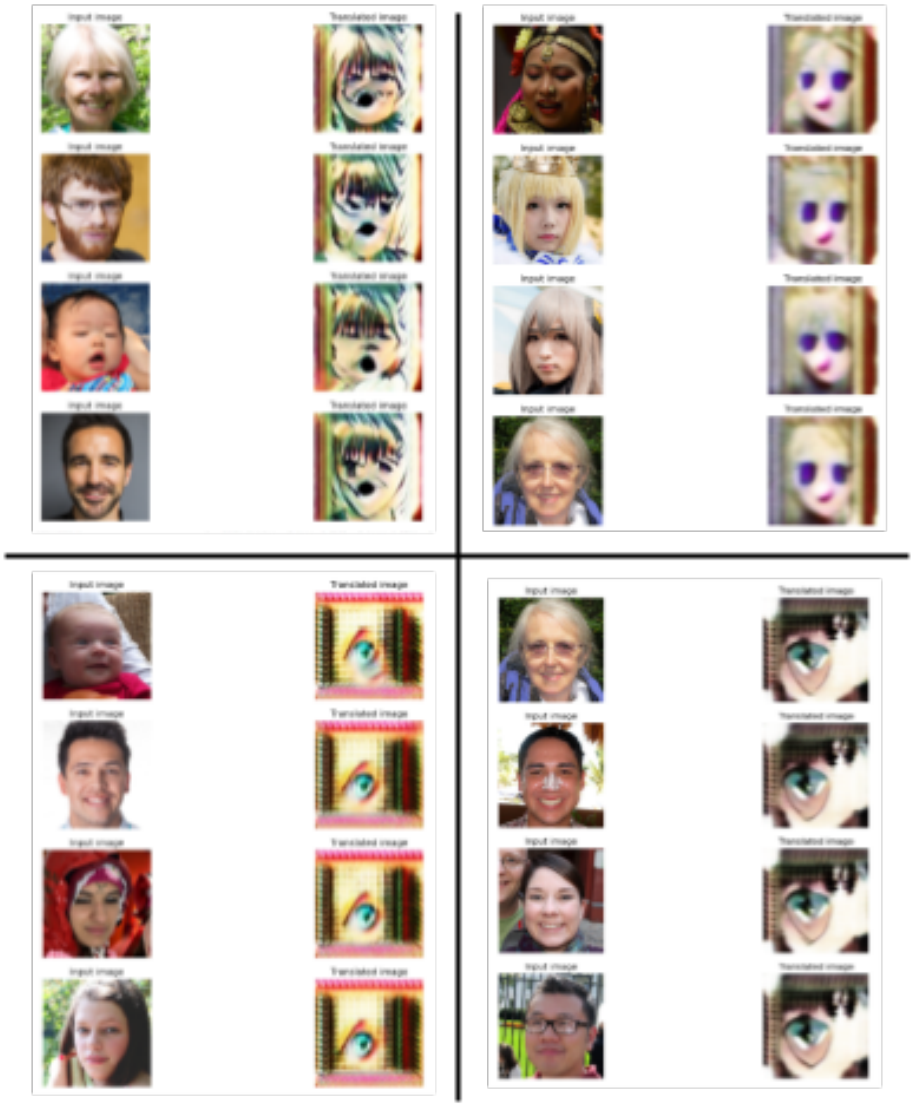

Figure 4.6: Example of results of some training sessions that ended in overtraining

between the predictions and the actual values. Depending on the results obtained, some parameters of the model were modified and another training was performed and so on until the result obtained was as close as possible to what we wanted to achieve with this project.

As this model is a slow convergence model, the training processes are slow and very long. The time spent only in the training for this project is approximately 520 hours approx., which may seem like a lot, but a project of this magnitude requires many more. In Section 4.2 the results of some of these trainings are explained in more detail, as well as the final result that has been possible to achieve with the limitations of hardware, software and time.

4.1.2 User Application Development

This section explains how the application has been developed and in Appendix A.2 the most relevant code fragments are shown.

Since the neural network training has been performed in Python, the PyCharm programming environment has been used to implement the application in Python. In addition, the Tkinter tool has been used for the development of the graphical user interface and the OpenCV library for camera detection and other image interactions. In this project, the application is a tool to show the user the work done by the model trainings. The application is responsible for applying the weights generated by the model to an image of a human face that can be either taken with the computer's webcam or uploaded from the device, and transform it to anime style.

As mentioned above, the Tkinter library and some of its elements have been used to design the user interface. Some of the elements that have been used are the buttons with the **`Button()`** function, the labels with the **`Label()`** function to add text or images, and for reading these images the **`PhotoImage()`** function. In order to upload an image from the computer, the function **`filedialog.askopenfilename()`** has been used, which opens a new window to select files from the local directories (see Figure 4.7). In this case it has been specified to detect only PNG files since the **`PhotoImage()`** function does not allow other image formats such as JPG or JPEG. To take a picture with the computer's webcam (if available, if not, an error message is displayed on the screen. See Figure 4.8) it has been used the OpenCV library which is in charge of starting the webcam with the function **`cv2.VideoCapture(0)`** (in this case, 0 refers to the first camera detected by default) and generate an image for each frame to later display it on the screen with the Tkinter library in a loop and thus generate the video effect in real time. To take the picture, simply save the last generated frame and interrupt the loop. Finally, to generate the new image with the anime style the TensorFlow and Keras libraries are used again to read the weights of the pre-trained model and do the image-to-image translation, for this the CycleGan model class and some of its functions must be defined again. This process is performed four times to provide the user with four different transformations by applying four different weights.

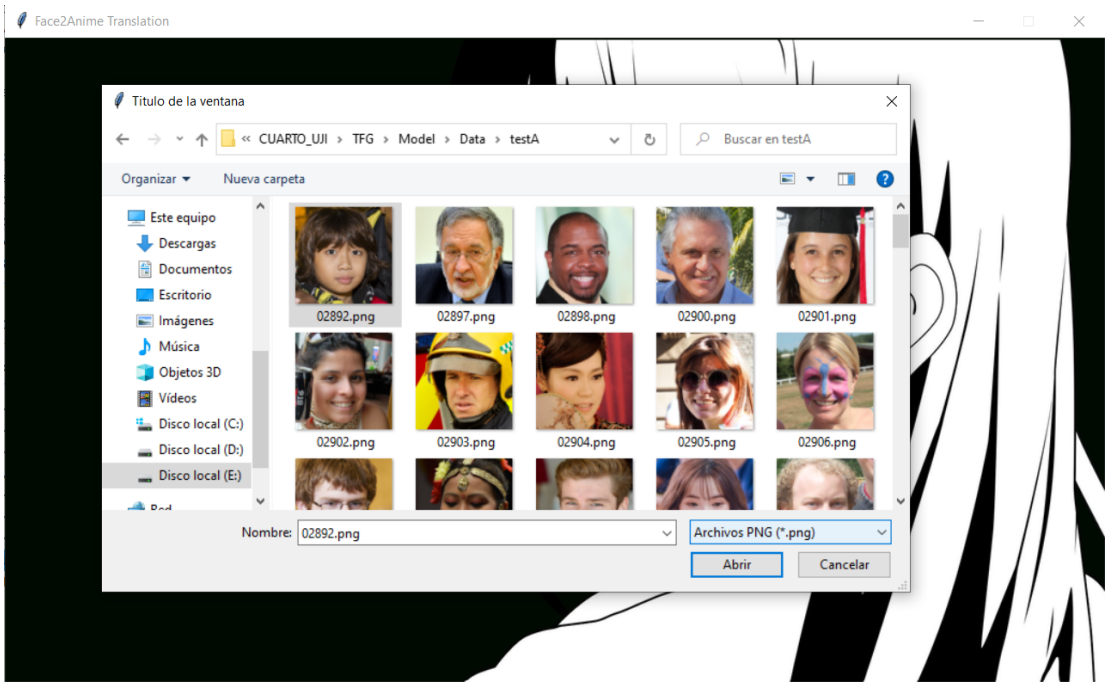

Figure 4.7: File selection window generated by **`filedialog.askopenfilename()`** Tkinter function

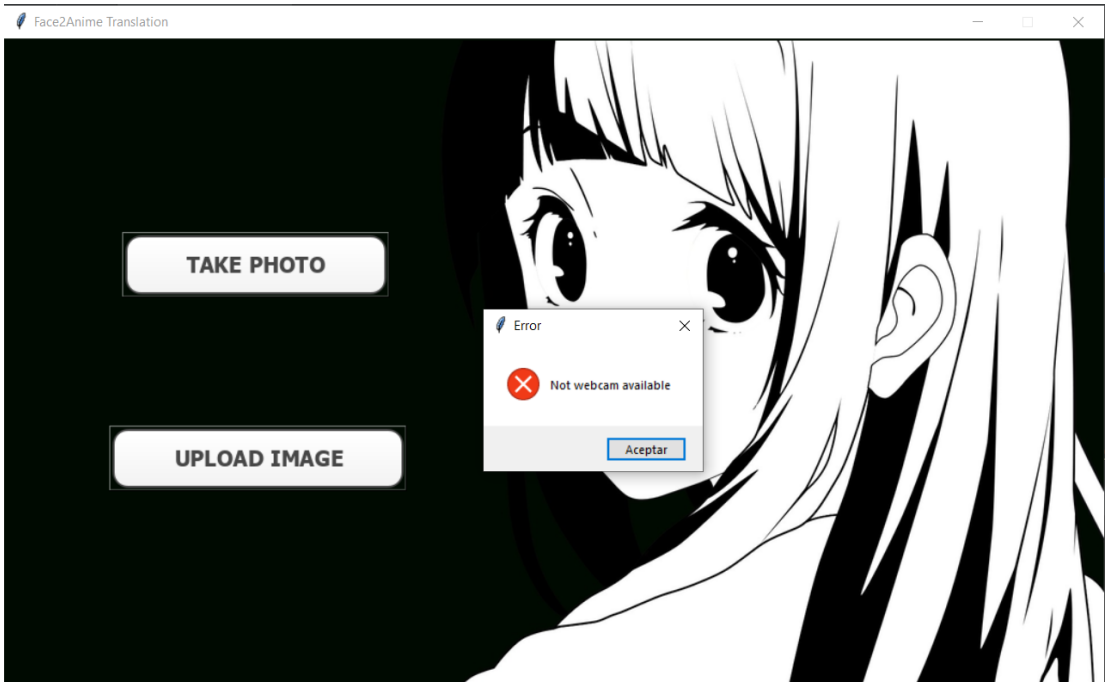

Figure 4.8: Error message when webcam is not available

4.2 Results

A total of over 40 training sessions have been conducted and examined, exploring various combinations of parameter values such as the number of epochs, batch size, image size, and more. The outcomes of selected training sessions are presented below, along with the altered parameters and their impact on the model. Finally, the conclusions drawn from these experiments are summarized.

Figure 4.9 shows two of the first trainings that were performed in Google Colab, the first one (A) is the result of 10 epochs with 1000 sample images, while the second one (B) is the result of 8 epochs with 2000 samples. In the example it can be seen that a higher number of samples leads to a higher quality result, however increasing the number of samples also increases the time it takes for the model to learn (the first one took 2 hours and the second one 4 hours).

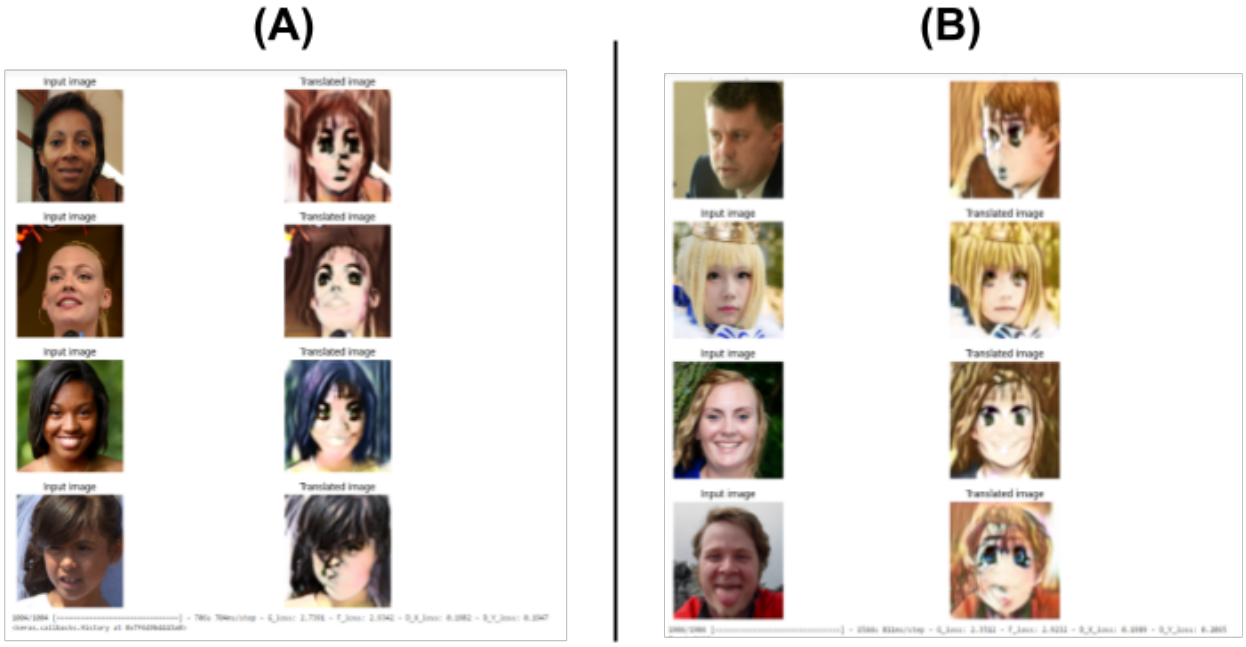

Figure 4.9: (a): 10 epochs - 1000 samples - 2h approx. (b): 8 epochs - 2000 samples - 4h approx

Figure 4.10 shows two of the trainings that were performed in Jupyter Notebook, both are the result of 56 epochs with 2000 sample images, but this time we introduce a new modification in the learning_rate variable. The learning_rate of the first one (A) is 2e-4 , while the learning_rate of the second one (B) is 2e-3. The learning rate determines the step size that the optimization algorithm takes to update the model weights during training. If the learning rate is too high, the algorithm may skip local minima and diverge. If it is too low, the model may take a long time to converge to an acceptable

solution. Therefore, finding an appropriate learning rate is important to ensure that the model converges quickly and accurately. After different trainings with different learning rate values, it has been concluded that the default value in the base implementation of the model (2e-4) was quite adequate.

Figure 4.10: (a): 56 epochs - 2000 samples - learning_rate=2e-4 - 65h approx. (b): 56 epochs - 2000 samples - learning_rate=2e-3 - 61h approx

During the development of the project, the dataset was modified to make it more efficient and 1000 more samples were added, since with 2000 samples it reached over-training very quickly (around epoch 60). Figure 4.11 shows the results of training 60 epochs with 2000 samples (A) and the results of training 80 epochs with 3000 samples (B). In addition, in training B the value of image size and buffer size have also been modified to half the value of training A, which significantly reduces the training time.

As mentioned above, two different programming environments (Google Colab and Jupyter Notebook) were used to train the model. This is due to the limitations that each of them has and that has been a drawback in the development of the project. Google Colab has GPU usage limitations and therefore it was only possible to train in 4-hour intervals and leaving 24 hours between each training. On the other hand, Jupyter Notebook did not have this limitation, but the GPU capacity of the personal computer was more reduced and therefore the trainings took longer to complete. In addition, for the same reason, only batch size 1 trainings could be performed, while in Google Colab

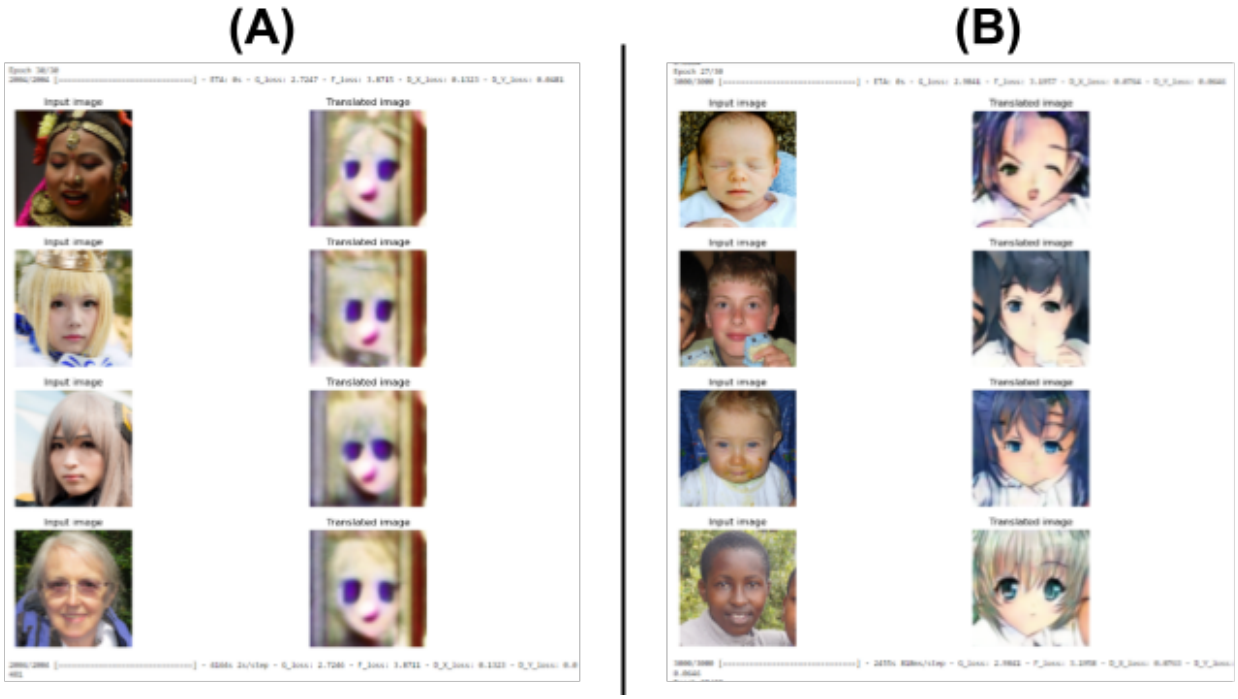

Figure 4.11: (a): 60 epochs - 2000 samples - learning_rate=2e-4 - buffer_size=256 - input_img_size=(256, 256, 3) - 70h approx. (b): 80 epochs - 3000 samples - learning_rate=2e-4 - buffer_size=128 - input_img_size=(128, 128, 3) - 53h approx

the batch size could be up to 6. Figure 4.12 shows a training in Google Colab (A) of 30 epochs with batch_size=2, and another training in JupyterNotebook (B) of 30 epochs with batch_size=1.

Finally, it has been concluded that the best results are obtained in batch 1 training, between 130 and 180 epochs, with 3000 samples and with the following parameters: learning_rate=2e-4, buffer_size=128 and input_img_size=(128, 128, 3). Figure 4.13 shows one of the best results obtained.

All the tasks and subtasks that were initially planned have been completed and the proposed milestones have been achieved. However, due to time constraints, the training results are not of the best possible quality, although if more time were available and longer training sessions could be carried out, they would be significantly improved.

This project could be applied as an API or plugin to be used as a tool for the automatic generation of avatars in video games and other interactive applications.The complete project (model implementation and user application) is available at GitHub.

Figure 4.12: (a): 30 epochs - 3000 samples - learning_rate=2e-4 - buffer_size=128 - input_img_size=(128, 128, 3) - batch 2 - 8h approx. (b): 30 epochs - 3000 samples - learning_rate=2e-4 - buffer_size=128 - input_img_size=(128, 128, 3) - batch 1 - 20h approx

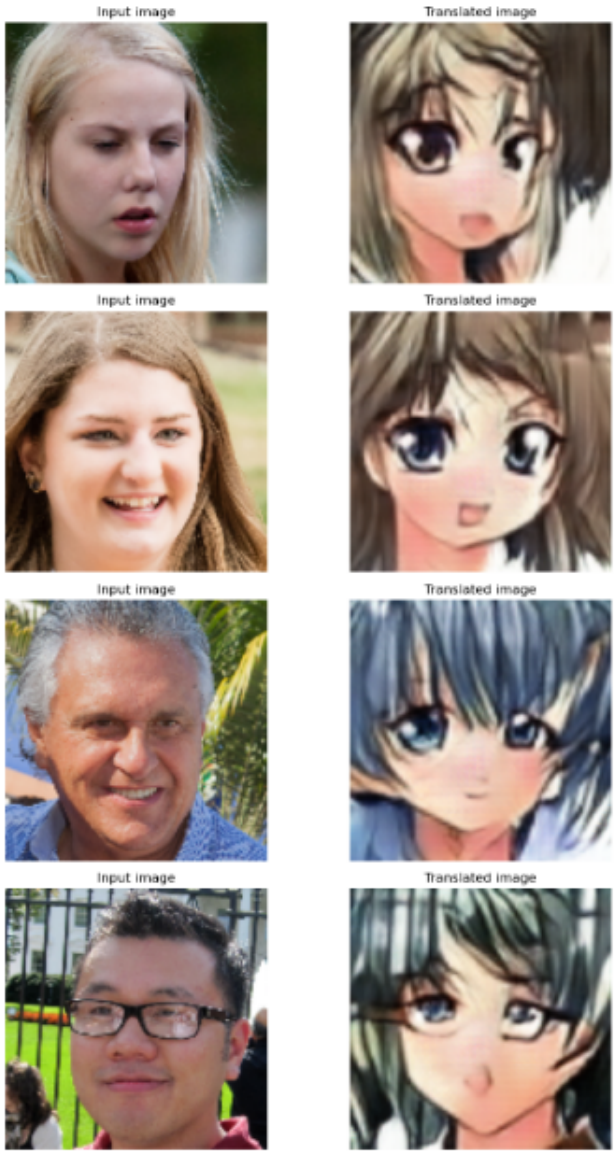

Figure 4.13: 180 epochs - 3000 samples - learning_rate=2e-4 - buffer_size=128 - input_img_size=(128, 128, 3) - batch 1

Chapter 5

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

Although this project had the added difficulty that it was a totally new field for me and that nothing has been studied in the career about it, it has been very interesting for me to investigate and learn how this type of neural network models work. And although the results are not what I expected to get at the beginning of the project, this has not been a demotivation for me, on the opposite, I consider that I underestimated the difficulty of embracing a field like this and that it really is a terrain where it is difficult to move without any previous knowledge. For that reason, I am satisfied with the results since I have invested a great effort to achieve them.

Regarding report, it has been somewhat complicated for me to explain the technical concepts of the implementation of this model, since they should be simple enough to be understood by a person without knowledge in this area, but at the same time they should be technical enough for a project of this magnitude. In summary, it was difficult to find the middle ground between the unintelligible and the vulgar.

Currently, there exist numerous ethical dilemmas surrounding artificial intelligence as a result of its rapid advancement and the potential consequences it may bring to society. Personally, I hold the belief that artificial intelligence is making a positive impact on technological advancements, offering us a multitude of applications to enhance our products and overall quality of life. Undoubtedly, its utilization can greatly benefit society, enabling us to surpass limitations that would otherwise be insurmountable for human beings. Therefore, it remains crucial to persist in research and advancements within this field, striving for continual improvement.

5.2 Future work

In terms of improving this project, I would like to spend more hours training and tuning the model, as well as improving the dataset to achieve more accurate results. To improve the dataset I would add more samples and make a more meticulous selection of the images. In addition, I would include classifications of faces by gender, age, and other determining features, to help the model better associate both domains (human-anime). I would also like to add new domains (such as cartoon images like Disney, etc) to do the image-to-image translation of human faces to other styles.

Another improvement I could add would be to use a deep learning model in the user application to discriminate images uploaded or photos taken that are not human faces, in order to avoid erroneous translations. And as I mentioned before, I would like to use this project as an API so that it can be used by third party applications.

In addition to working on improving this own project, as I mentioned at the beginning, the customization and the self-representation of avatars in video games is something that especially attracts my attention, as well as the different applications of deep learning and neural networks. That is why in the future I would like to continue researching this field, and in particular, to investigate how to apply this system to 3D models in order to modify automatically the polygonal mesh of the characters in 3D video games.

Bibliography

[1] Szolin, K., Kuss, D. J., Nuyens, F., & Griffiths, M. D. Exploring the user-avatar relationship in videogames: A systematic review of the proteus effect. https://doi.org/10.1080/07370024.2022.2103419. Accessed: 2023-06-23.

[2] Turkay, S., & Kinzer, C. K. The effects of avatar-based customization on player identification. https://doi.org/10.4018/ijgcms.2014010101. Accessed: 2023-06-23.

[3] Rahill, K. M., & Sebrechts, M. M. Effects of avatar player-similarity and player-construction on gaming performance. https://doi.org/10.1016/j.chbr.2021.100131. Accessed: 2023-06-23.

[4] Pang, Y. Image-to-Image Translation: Methods and Applications. https://arxiv.org/abs/2101.08629. Accessed: 2023-06-23.

[5] J. Zhu. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. https://arxiv.org/abs/1703.10593. Accessed: 2023-06-23.

[6] Google colaboratory. https://colab.research.google.com/.

[7] Project jupyter. https://jupyter.org/.

[8] Anaconda | The world's most popular data science platform. https://www.anaconda.com/.

[9] Pycharm: el ide de python para desarrolladores profesionales, por jetbrains. https://www.jetbrains.com/es-es/pycharm/.

[10] Tensorflow. https://www.tensorflow.org/?hl=es-419.

[11] Keras: Deep learning for humans. https://keras.io/.

[12] Tkinter — Interface de python para tcl/tk. https://docs.python.org/es/3/library/tkinter.html.

[13] Opencv - Open computer vision library. https://opencv.org/.

[14] Github: Let's build from here. https://github.com/.

[15] Manage your team's projects from anywhere | Trello. https://trello.com/.

[16] Texstudio - A latex editor. https://www.texstudio.org/.

[17] Sueldo de data engineer/a en españa. https://es.indeed.com/career/data-engineer/salarie. Accessed: 2023-06-23.

[18] Sueldo de programador/a junior en españa. https://es.indeed.com/career/programador-junior/salaries. Accessed: 2023-06-23.

[19] Tecnopc. ¿Qué son los núcleos CUDA? • Tarjetas Gráficas PC. https://tarjetasgraficaspc.com/que-son-nucleos-cuda/. Accessed: 2023-06-23.

[20] Admin. 30 Anime Black and White Wallpapers - Wallpaperboat. https://wallpaperboat.com/anime-black-and-white-wallpapers. Accessed: 2023-06-23.

[21] Generador de Logos y Gráficos. https://es.cooltext.com/.

[22] Call-to-Action Button Generator - Design buttons & download as CSS PNG. https://buttonoptimizer.com/.

[23] Flaticon. Photo camera interface symbol for button Icon - 45010. https://www.flaticon.com/free-icon/photo-camera-interface-symbol-for-button_45010. Accessed: 2023-06-23.

[24] Flaticon. Home button Icon - 61972. https://www.flaticon.com/free-icon/home-button_61972. Accessed: 2023-06-23.

[25] T. Karras. A Style-Based Generator Architecture for Generative Adversarial Networks. https://arxiv.org/abs/1812.04948. Accessed: 2023-06-23.

[26] K. Team. Keras documentation: Face image generation with StyleGAN. https://keras.io/examples/generative/stylegan/. Accessed: 2023-06-23.

[27] Anime Faces vs Human Faces. https://www.kaggle.com/datasets/sanyam1992000/anime-faces-vs-human-faces. Accessed: 2023-06-23.

[28] Human Faces. https://www.kaggle.com/datasets/ashwingupta3012/human-faces. Accessed: 2023-06-23.

[29] Lmtri. Face2Anime-using-CycleGAN/datasets at main · lmtri1998/Face2Anime-using-CycleGAN. https://github.com/lmtri1998/Face2Anime-using-CycleGAN/tree/main/datasets. Accessed: 2023-06-23.

[30] K. Team. Keras documentation: CycleGAN. https://keras.io/examples/generative/cyclegan/. Accessed: 2023-06-23.

Appendix A

Source code

A.1 Neural Network Model

This section explains in detail the structure of the implemented model and how it works.

Setup

```
import os
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

import tensorflow_addons as tfa
autotune = tf.data.AUTOTUNE
```

This code snippet imports the following libraries: os, numpy (renamed as np), matplotlib.pyplot (renamed as plt), tensorflow (renamed as tf), and tensorflow_addons (renamed as tfa). It also configures the autotune variable to use TensorFlow's "AUTOTUNE" option.

These libraries are used for different purposes:

- **os:** Provides functions for interacting with the operating system, particularly for file and directory-related operations.
- **numpy:** A popular library for numerical computations in Python, commonly used for manipulating arrays and performing efficient mathematical operations.

- **matplotlib.pyplot:** A visualization library in Python that provides a MATLAB-like interface for creating plots and visualizations.
- **tensorflow:** An open-source machine learning library developed by Google. It is used for building and training machine learning models, especially neural networks.
- **tensorflow.keras.layers:** A sub-module of TensorFlow's Keras API that offers pre-defined layers used for constructing neural network models.
- **tensorflow_addons:** An additional library for TensorFlow that provides extra implementations of advanced algorithms and layers to enhance and extend TensorFlow's capabilities.
- **autotune:** A variable set to utilize TensorFlow's "AUTOTUNE" option, which automatically selects the best performance configuration based on the execution context.

Prepare the dataset

```
# Define the path of our local dataset
dataset_path = "./Data"

# Function to obtain image paths
def get_image_paths(dataset_dir):
        image_paths = []
        for root, _, files in os.walk(dataset_dir):
                for file in files:
                        if file.endswith(".jpg") or file.endswith(".png"):
                                image_path = os.path.join(root, file)
                                image_paths.append(image_path)

        return image_paths

# Obtain the routes of the training and test images.
train_human_paths = get_image_paths(os.path.join(dataset_path, "trainA"))
train_anime_paths = get_image_paths(os.path.join(dataset_path, "trainB"))
test_human_paths = get_image_paths(os.path.join(dataset_path, "testA"))
test_anime_paths = get_image_paths(os.path.join(dataset_path, "testB"))

# Create the datasets for training and testing
train_human_ds = tf.data.Dataset.from_generator(lambda: train_human_paths, output_types=tf.string)
train_anime_ds = tf.data.Dataset.from_generator(lambda: train_anime_paths, output_types=tf.string)
test_human_ds = tf.data.Dataset.from_generator(lambda: test_human_paths, output_types=tf.string)
test_anime_ds = tf.data.Dataset.from_generator(lambda: test_anime_paths, output_types=tf.string)



# Define the standard image size.
orig_img_size = (158, 158)
# Size of the random crops to be used during training.
```

```
input_img_size = (128, 128, 3)
# Weights initializer for the layers.
kernel_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
# Gamma initializer for instance normalization.
gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

buffer_size = 128
batch_size = 1


def normalize_img(img):
        img = tf.cast(img, dtype=tf.float32)
        # Map values in the range [-1, 1]
        return (img / 127.5) - 1.0


def preprocess_train_image(path):
        img = tf.io.read_file(path)
        img = tf.image.decode_jpeg(img, channels=3)

        # Random flip
        img = tf.image.random_flip_left_right(img)
        # Resize to the original size first
        img = tf.image.resize(img, [*orig_img_size])
        # Random crop to 256X256
        img = tf.image.random_crop(img, size=[*input_img_size])
        # Normalize the pixel values in the range [-1, 1]
        img = normalize_img(img)
        return img


def preprocess_test_image(path):
        img = tf.io.read_file(path)
        img = tf.image.decode_jpeg(img, channels=3)

        # Only resizing and normalization for the test images.
        img = tf.image.resize(img, [input_img_size[0], input_img_size[1]])
        img = normalize_img(img)
        return img
```

This code fragment is responsible for loading and preparing images for use in the training and testing process of the machine learning model. First, image paths are obtained within a specific directory and datasets are created using the obtained image paths. These datasets are used for training and testing the model. Then, image preprocessing functions are defined for training and testing. These functions read the images from the paths, apply preprocessing operations such as random flipping, resizing and normalization, and return the preprocessed images.

Create Dataset objects

```
# Apply the preprocessing operations to the training data
```

```
train_human = (
        train_human_ds.map(preprocess_train_image, num_parallel_calls=autotune)
        .cache()
        .shuffle(buffer_size)
        .batch(batch_size)
)
train_anime = (
        train_anime_ds.map(preprocess_train_image, num_parallel_calls=autotune)
        .cache()
        .shuffle(buffer_size)
        .batch(batch_size)
)

# Apply the preprocessing operations to the test data
test_human = (
        test_human_ds.map(preprocess_test_image, num_parallel_calls=autotune)
        .cache()
        .shuffle(buffer_size)
        .batch(batch_size)
)
test_anime = (
        test_anime_ds.map(preprocess_test_image, num_parallel_calls=autotune)
        .cache()
        .shuffle(buffer_size)
        .batch(batch_size)
)
```

Here the preprocessing operations are applied together with the **cache()** method to cache the data in memory to improve performance and the **shuffle(buffer_size)** method to randomize the data with a specified buffer size. Finally, the data is batched (batch_size) using the **batch()** method.

Visualize some samples

```
_, ax = plt.subplots(4, 2, figsize=(10, 15))
for i, samples in enumerate(zip(train_human.take(4), train_anime.take(4))):
        human = (((samples[0][0] * 127.5) + 127.5).numpy()).astype(np.uint8)
        anime = (((samples[1][0] * 127.5) + 127.5).numpy()).astype(np.uint8)
        ax[i, 0].imshow(human)
        ax[i, 1].imshow(anime)
plt.show()
```

This fragment shows some examples of the images in the dataset.

Building blocks used in the CycleGAN generators and discriminators

```
class ReflectionPadding2D(layers.Layer):
        """Implements Reflection Padding as a layer.

        Args:
        padding(tuple): Amount of padding for the
```

```
        spatial dimensions.

        Returns:
        A padded tensor with the same type as the input tensor.
        """

        def __init__(self, padding=(1, 1), **kwargs):
                self.padding = tuple(padding)
                super().__init__(**kwargs)

        def call(self, input_tensor, mask=None):
                padding_width, padding_height = self.padding
                padding_tensor = [
                        [0, 0],
                        [padding_height, padding_height],
                        [padding_width, padding_width],
                        [0, 0],
                ]
                return tf.pad(input_tensor, padding_tensor, mode="REFLECT")


def residual_block(
        x,
        activation,
        kernel_initializer=kernel_init,
        kernel_size=(3, 3),
        strides=(1, 1),
        padding="valid",
        gamma_initializer=gamma_init,
        use_bias=False,
):
        dim = x.shape[-1]
        input_tensor = x

        x = ReflectionPadding2D()(input_tensor)
        x = layers.Conv2D(
                dim,
                kernel_size,
                strides=strides,
                kernel_initializer=kernel_initializer,
                padding=padding,
                use_bias=use_bias,
        )(x)
        x = tfa.layers.InstanceNormalization(gamma_initializer=gamma_initializer)(x)
        x = activation(x)

        x = ReflectionPadding2D()(x)
        x = layers.Conv2D(
                dim,
                kernel_size,
                strides=strides,
                kernel_initializer=kernel_initializer,
                padding=padding,
                use_bias=use_bias,
```

```
        )(x)
        x = tfa.layers.InstanceNormalization(gamma_initializer=gamma_initializer)(x)
        x = layers.add([input_tensor, x])
        return x


def downsample(
        x,
        filters,
        activation,
        kernel_initializer=kernel_init,
        kernel_size=(3, 3),
        strides=(2, 2),
        padding="same",
        gamma_initializer=gamma_init,
        use_bias=False,
):
        x = layers.Conv2D(
                filters,
                kernel_size,
                strides=strides,
                kernel_initializer=kernel_initializer,
                padding=padding,
                use_bias=use_bias,
        )(x)
        x = tfa.layers.InstanceNormalization(gamma_initializer=gamma_initializer)(x)
        if activation:
                x = activation(x)
        return x


def upsample(
        x,
        filters,
        activation,
        kernel_size=(3, 3),
        strides=(2, 2),
        padding="same",
        kernel_initializer=kernel_init,
        gamma_initializer=gamma_init,
        use_bias=False,
):
        x = layers.Conv2DTranspose(
                filters,
                kernel_size,
                strides=strides,
                padding=padding,
                kernel_initializer=kernel_initializer,
                use_bias=use_bias,
        )(x)
        x = tfa.layers.InstanceNormalization(gamma_initializer=gamma_initializer)(x)
        if activation:
                x = activation(x)
        return x
```

This code fragment defines several functions and a class to implement the CycleGAN style neural network architecture.

The **`ReflectionPadding2D()`** class implements the reflection padding method in two dimensions. This layer is used to add the reflection padding to the convolution input, which allows to better preserve the image features.

The **`residual_block()`** function implements a residual block of the neural network. Residual blocks are important to help the network learn complex nonlinear transformations. In this case, two 2D convolutions with a ReLU activation function and an instance normalization layer are used. The reflection padding layer is also applied to the residual block input.

The **`downsample()`** and **`upsample()`** functions implement down-sample and up-sample layers, respectively. Downsampling is used to reduce the image size and increase the number of image features, while upsampling is used to increase the image size and reduce the number of features.

Build the generators

The generator consists of downsampling blocks: nine residual blocks and upsampling blocks. The structure of the generator is the following:

```
"""
c7s1-64 ==> Conv block with ‘relu‘ activation, filter size of 7
d128 ====|
|-> 2 downsampling blocks
d256 ====|
R256 ====|
R256     |
R256     |
R256     |
R256     |-> 9 residual blocks
R256     |
R256     |
R256     |
R256 ====|
u128 ====|
|-> 2 upsampling blocks
u64  ====|
c7s1-3 => Last conv block with ‘tanh‘ activation, filter size of 7.
"""
```

```
def get_resnet_generator(
        filters=64,
        num_downsampling_blocks=2,
        num_residual_blocks=9,
        num_upsample_blocks=2,
        gamma_initializer=gamma_init,
        name=None,
):
```

```
        img_input = layers.Input(shape=input_img_size, name=name + "_img_input")
        x = ReflectionPadding2D(padding=(3, 3))(img_input)
        x = layers.Conv2D(filters, (7, 7), kernel_initializer=kernel_init, use_bias=False)(
                x
        )
        x = tfa.layers.InstanceNormalization(gamma_initializer=gamma_initializer)(x)
        x = layers.Activation("relu")(x)

        # Downsampling
        for _ in range(num_downsampling_blocks):
                filters *= 2
                x = downsample(x, filters=filters, activation=layers.Activation("relu"))

        # Residual blocks
        for _ in range(num_residual_blocks):
                x = residual_block(x, activation=layers.Activation("relu"))

        # Upsampling
        for _ in range(num_upsample_blocks):
                filters //= 2
                x = upsample(x, filters, activation=layers.Activation("relu"))

        # Final block
        x = ReflectionPadding2D(padding=(3, 3))(x)
        x = layers.Conv2D(3, (7, 7), padding="valid")(x)
        x = layers.Activation("tanh")(x)

        model = keras.models.Model(img_input, x, name=name)
        return model
```

The function starts by creating an image input using the Keras **`Input()`** class. Then, a reflection padding is performed on the image using the **`ReflectionPadding2D()`** class. A 7x7 convolutional layer with a number of filters is applied, followed by instance normalization and ReLU activation.

Then, a number of downsampling blocks are applied, each reducing the spatial resolution of the image by half. After the downsampling blocks, a number of residual blocks are applied to allow for connection hops in the network.

After the residual blocks, a number of upsampling blocks are applied, each doubling the spatial resolution of the image. Finally, a final padding reflection block is applied, followed by a 7x7 convolutional layer with 3 filters and a hyperbolic tangent activation (tanh).

The model is compiled using the image input and the output generated by the last convolutional layer, and returned as an instance of the Keras **`Model()`** class.

Build the discriminators

The discriminators implement the following architecture: C64->C128->C256->C512

```
def get_discriminator(
        filters=64, kernel_initializer=kernel_init, num_downsampling=3, name=None
```

```
):
        img_input = layers.Input(shape=input_img_size, name=name + "_img_input")
        x = layers.Conv2D(
                filters,
                (4, 4),
                strides=(2, 2),
                padding="same",
                kernel_initializer=kernel_initializer,
        )(img_input)
        x = layers.LeakyReLU(0.2)(x)

        num_filters = filters
        for num_downsample_block in range(3):
                num_filters *= 2
                if num_downsample_block < 2:
                        x = downsample(
                                x,
                                filters=num_filters,
                                activation=layers.LeakyReLU(0.2),
                                kernel_size=(4, 4),
                                strides=(2, 2),
                        )
                else:
                        x = downsample(
                                x,
                                filters=num_filters,
                                activation=layers.LeakyReLU(0.2),
                                kernel_size=(4, 4),
                                strides=(1, 1),
                )

        x = layers.Conv2D(
                1, (4, 4), strides=(1, 1), padding="same", kernel_initializer=kernel_initializer
        )(x)

        model = keras.models.Model(inputs=img_input, outputs=x, name=name)
        return model


# Get the generators
gen_G = get_resnet_generator(name="generator_G")
gen_F = get_resnet_generator(name="generator_F")

# Get the discriminators
disc_X = get_discriminator(name="discriminator_X")
disc_Y = get_discriminator(name="discriminator_Y")
```

This code fragment defines a function to create a discriminator that takes an input image and produces a single-valued output indicating whether the image is real or false. The discriminator is constructed using a series of convolutional and down-sampling layers that reduce the spatial resolution of the image and increase the number of channels. Finally, a 1x1 convolution layer is used to produce a single-valued output.

After constructing the get_discriminator function, the code also defines two ResNet generators, gen_G and gen_F, using the **get_resnet_generator()** function. Next, the code creates two discriminators, disc_X and disc_Y, using the **get_discriminator()** function.

Build the CycleGAN model

```
class CycleGan(keras.Model):
        def __init__(
                self,
                generator_G,
                generator_F,
                discriminator_X,
                discriminator_Y,
                lambda_cycle=10.0,
                lambda_identity=0.5,
        ):
                super().__init__()
                self.gen_G = generator_G
                self.gen_F = generator_F
                self.disc_X = discriminator_X
                self.disc_Y = discriminator_Y
                self.lambda_cycle = lambda_cycle
                self.lambda_identity = lambda_identity

        def call(self, inputs):
                return (
                        self.disc_X(inputs),
                        self.disc_Y(inputs),
                        self.gen_G(inputs),
                        self.gen_F(inputs),
                )

        def compile(
                self,
                gen_G_optimizer,
                gen_F_optimizer,
                disc_X_optimizer,
                disc_Y_optimizer,
                gen_loss_fn,
                disc_loss_fn,
        ):
                super().compile()
                self.gen_G_optimizer = gen_G_optimizer
                self.gen_F_optimizer = gen_F_optimizer
                self.disc_X_optimizer = disc_X_optimizer
                self.disc_Y_optimizer = disc_Y_optimizer
                self.generator_loss_fn = gen_loss_fn
                self.discriminator_loss_fn = disc_loss_fn
                self.cycle_loss_fn = keras.losses.MeanAbsoluteError()
                self.identity_loss_fn = keras.losses.MeanAbsoluteError()

```

```
46          def train_step(self, batch_data):
47                  # x is Human and y is anime
48                  real_x, real_y = batch_data
49
50                  # For CycleGAN, we need to calculate different
51                  # kinds of losses for the generators and discriminators.
52                  # We will perform the following steps here:
53                  #
54                  # 1. Pass real images through the generators and get the generated images
55                  # 2. Pass the generated images back to the generators to check if we
56                  #    we can predict the original image from the generated image.
57                  # 3. Do an identity mapping of the real images using the generators.
58                  # 4. Pass the generated images in 1) to the corresponding discriminators.
59                  # 5. Calculate the generators total loss (adverserial + cycle + identity)
60                  # 6. Calculate the discriminators loss
61                  # 7. Update the weights of the generators
62                  # 8. Update the weights of the discriminators
63                  # 9. Return the losses in a dictionary
64
65                  with tf.GradientTape(persistent=True) as tape:
66                          # Human to fake anime
67                          fake_y = self.gen_G(real_x, training=True)
68                          # Anime to fake human -> y2x
69                          fake_x = self.gen_F(real_y, training=True)
70
71                          # Cycle (Human to fake anime to fake human): x -> y -> x
72                          cycled_x = self.gen_F(fake_y, training=True)
73                          # Cycle (Anime to fake human to fake anime) y -> x -> y
74                          cycled_y = self.gen_G(fake_x, training=True)
75
76                          # Identity mapping
77                          same_x = self.gen_F(real_x, training=True)
78                          same_y = self.gen_G(real_y, training=True)
79
80                          # Discriminator output
81                          disc_real_x = self.disc_X(real_x, training=True)
82                          disc_fake_x = self.disc_X(fake_x, training=True)
83
84                          disc_real_y = self.disc_Y(real_y, training=True)
85                          disc_fake_y = self.disc_Y(fake_y, training=True)
86
87                          # Generator adverserial loss
88                          gen_G_loss = self.generator_loss_fn(disc_fake_y)
89                          gen_F_loss = self.generator_loss_fn(disc_fake_x)
90
91                          # Generator cycle loss
92                          cycle_loss_G = self.cycle_loss_fn(real_y, cycled_y) * self.lambda_cycle
93                          cycle_loss_F = self.cycle_loss_fn(real_x, cycled_x) * self.lambda_cycle
94
95                          # Generator identity loss
96                          id_loss_G = (
97                                  self.identity_loss_fn(real_y, same_y)
98                                  * self.lambda_cycle
99                                  * self.lambda_identity
```

```
                        )
                        id_loss_F = (
                                self.identity_loss_fn(real_x, same_x)
                                * self.lambda_cycle
                                * self.lambda_identity
                        )

                        # Total generator loss
                        total_loss_G = gen_G_loss + cycle_loss_G + id_loss_G
                        total_loss_F = gen_F_loss + cycle_loss_F + id_loss_F

                        # Discriminator loss
                        disc_X_loss = self.discriminator_loss_fn(disc_real_x, disc_fake_x)
                        disc_Y_loss = self.discriminator_loss_fn(disc_real_y, disc_fake_y)

                        # Get the gradients for the generators
                        grads_G = tape.gradient(total_loss_G, self.gen_G.trainable_variables)
                        grads_F = tape.gradient(total_loss_F, self.gen_F.trainable_variables)

                        # Get the gradients for the discriminators
                        disc_X_grads = tape.gradient(disc_X_loss, self.disc_X.trainable_variables)
                        disc_Y_grads = tape.gradient(disc_Y_loss, self.disc_Y.trainable_variables)

                        # Update the weights of the generators
                        self.gen_G_optimizer.apply_gradients(
                                zip(grads_G, self.gen_G.trainable_variables)
                        )
                        self.gen_F_optimizer.apply_gradients(
                                zip(grads_F, self.gen_F.trainable_variables)
                        )

                        # Update the weights of the discriminators
                        self.disc_X_optimizer.apply_gradients(
                                zip(disc_X_grads, self.disc_X.trainable_variables)
                        )
                        self.disc_Y_optimizer.apply_gradients(
                                zip(disc_Y_grads, self.disc_Y.trainable_variables)
                        )

                        return {
                                "G_loss": total_loss_G,
                                "F_loss": total_loss_F,
                                "D_X_loss": disc_X_loss,
                                "D_Y_loss": disc_Y_loss,
                        }
```

This code snippet defines a CycleGAN model in TensorFlow as a subclass of keras.Model. The class constructor initializes the generators (gen_G and gen_F), discriminators (disc_X and disc_Y), and other parameters such as lambda values for cycle loss and identity loss. The **`call()`** method defines the forward pass of the model, where it passes the input through the discriminators and generators, returning the outputs. The **`compile()`** method sets up the optimizers and loss functions for training the model. The

train_step() method defines a single training step, where it performs operations for training the CycleGAN model. This includes generating fake images, calculating losses for generators and discriminators, calculating gradients, and updating the weights of the networks. And finally, the method returns a dictionary containing the calculated losses during the training step.

Create a callback that periodically saves generated images

```
class GANMonitor(keras.callbacks.Callback):
        """A callback to generate and save images after each epoch"""

        def __init__(self, num_img=4):
                self.num_img = num_img

        def on_epoch_end(self, epoch, logs=None):
                _, ax = plt.subplots(4, 2, figsize=(12, 12))
                for i, img in enumerate(test_human.take(self.num_img)):
                        prediction = self.model.gen_G(img)[0].numpy()
                        prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
                        img = (img[0] * 127.5 + 127.5).numpy().astype(np.uint8)

                        ax[i, 0].imshow(img)
                        ax[i, 1].imshow(prediction)
                        ax[i, 0].set_title("Input_image")
                        ax[i, 1].set_title("Translated_image")
                        ax[i, 0].axis("off")
                        ax[i, 1].axis("off")

                        prediction = keras.preprocessing.image.array_to_img(prediction)
                        prediction.save(
                                "generated_img_{i}_{epoch}.png".format(i=i, epoch=epoch + 1)
                        )
                plt.show()
                plt.close()
```

This code is a method that is executed at the end of each epoch during the training of the deep learning model. The method generates a translated image (prediction) from each input image (img), visualizes the images generated by the model during the training process and saves each generated image in a PNG file using the Keras **save()** function. This allows the user to observe how the model is improving as the training progresses.

Train the end-to-end model

```
# Loss function for evaluating adversarial loss
adv_loss_fn = keras.losses.MeanSquaredError()

# Define the loss function for the generators
def generator_loss_fn(fake):
        fake_loss = adv_loss_fn(tf.ones_like(fake), fake)
        return fake_loss
```

```
8
9
10 # Define the loss function for the discriminators
11 def discriminator_loss_fn(real, fake):
12         real_loss = adv_loss_fn(tf.ones_like(real), real)
13         fake_loss = adv_loss_fn(tf.zeros_like(fake), fake)
14         return (real_loss + fake_loss) * 0.5
15
16
17 # Create cycle gan model
18 cycle_gan_model = CycleGan(
19         generator_G=gen_G, generator_F=gen_F, discriminator_X=disc_X, discriminator_Y=disc_Y
20 )
21
22 # Compile the model
23 cycle_gan_model.compile(
24         gen_G_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
25         gen_F_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
26         disc_X_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
27         disc_Y_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
28         gen_loss_fn=generator_loss_fn,
29         disc_loss_fn=discriminator_loss_fn,
30 )
31
32 # Callbacks
33 plotter = GANMonitor()
34 checkpoint_filepath = "./model_checkpoints/cyclegan_checkpoints.{epoch:03d}"
35 checkpoint_dir = os.path.dirname(checkpoint_filepath)
36 model_checkpoint_callback = keras.callbacks.ModelCheckpoint(
37         filepath=checkpoint_filepath,
38         save_weights_only=True
39 )
40
41 # Here we will train the model for 30 epochs.
42 cycle_gan_model.fit(
43         tf.data.Dataset.zip((train_human, train_anime)),
44         epochs=30,
45         callbacks=[plotter, model_checkpoint_callback],
46 )
```

This code fragment defines the adversarial loss function and the loss functions of the generators and discriminators. A cycle_gan_model object representing the CycleGAN model is created and compiled, using the previously defined generators and discriminators, and callbacks are defined to display the images generated during training and to store the model weights during training. Finally, the cycle_gan_model model is trained using the training data (train_human and train_anime) for a number of epochs, in this example 30.

Test the performance of the model.

```
1 # Load the checkpoints
```

```
latest = tf.train.latest_checkpoint(checkpoint_dir)
cycle_gan_model.load_weights(latest).expect_partial()
print("Weights loaded successfully")

_, ax = plt.subplots(4, 2, figsize=(10, 15))
for i, img in enumerate(test_human.take(4)):
        prediction = cycle_gan_model.gen_G(img, training=False)[0].numpy()
        prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
        img = (img[0] * 127.5 + 127.5).numpy().astype(np.uint8)

        ax[i, 0].imshow(img)
        ax[i, 1].imshow(prediction)
        ax[i, 0].set_title("Input image")
        ax[i, 1].set_title("Translated image")
        ax[i, 0].axis("off")
        ax[i, 1].axis("off")

        prediction = keras.preprocessing.image.array_to_img(prediction)
        prediction.save("predicted_img_{i}.png".format(i=i))
plt.tight_layout()
plt.show()
```

Finally, this code fragment loads the previously saved trained model weights using **`tf.train.latest_checkpoint()`** to obtain the path to the most recent weights file, generates and displays the translated images using the CycleGAN model, and saves the translated images as PNG files.

A.2 User Application

This section explains in detail some of the most important code fragments for the development of the application.

Generate user interface

To generate a user interface, a root and its corresponding **`Frame()`** are created. Each frame works as a box that contains elements of the Tkinter library, inside it the buttons, labels, images, etc. are added.

```
root = Tk()
root.title("Face2Anime Translation")
root.geometry("1024x600")
root.resizable(False, False)
root.config(bg="#000a01")

initFrame = Frame()
initFrame.config(bg="#000a01", width="1024", height="600")
initFrame.pack()

root.mainloop()
```

Buttons and labels generation

Buttons and labels are generated as follows. First, images are assigned to each element and then their parameters, such as background color, size and position, are modified. In the case of adding an image, the **`Label()`** object is used, and to generate a button, the **`Button()`** object is used together with the function that is called when it is pressed.

```
# Background image
imageBg = PhotoImage(file="imgs/ejemploFondo3.png")
bgLabel = Label(initFrame, image=imageBg, text="Background")
bgLabel.config(bg="#000a01")
bgLabel.place(x=60, y=0)

# Start button
imageBtStart = PhotoImage(file="imgs/buttonStart.png")
btnStart = Button(initFrame, text="Start", image=imageBtStart, command=startApp)
btnStart.place(x=130, y=380)
btnStart.config(bg="#000a01")
```

Start Camera

To start the camera, first a cameraObject is created using **`cv2.VideoCapture(0)`**, which represents the webcam and checks if the webcam could be opened correctly, if the camera could not be opened an error window appears on the screen indicating that the camera is not available. If the camera has been opened correctly, a new frame is loaded and the **`displayCamera())`** function is called. In this function, some transformations are performed on the image, such as converting it from BGR to RGB, flipping it horizontally and resizing it. Finally a recursive call to this function is scheduled after 10 milliseconds. This allows to continuously update the captured image in real time.

```
def initCamera():
        global captureFrame
        global btnFrame
        global captureLabel

        global cameraObject

        # Load webcam
        messagebox.showinfo("Loading␣webcam",
         "Wait␣while␣the␣webcam␣starts,␣it␣may␣take␣a␣few␣seconds.")
        cameraObject = cv2.VideoCapture(0)
        cameraObject.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
        cameraObject.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

        if cameraObject is not None:
                retval, photo = cameraObject.read()
                if retval == True:
                        # Hide Init Frame
                        initFrame.pack␣forget()

                        # Create Capture Frame
```

```
                        captureFrame = Frame()
                        captureFrame.config(width="600", height="600")
                        captureFrame.pack()

                        captureLabel = Label(captureFrame)
                        captureLabel.config(width="600", height="600")
                        captureLabel.pack()

                        # Create Buttons Frame
                        btnFrame = Frame()
                        btnFrame.config(bg="#000a01", width="212", height="600")
                        btnFrame.place(x=812, y=0)

                        # Camera Button
                        btnCamera = Button(btnFrame, text="Camera",
                         image=imageBtCamera, command=takePhoto)
                        btnCamera.place(x=30, y=270)

                        # HomeButton
                        btnHome = Button(btnFrame, text="Home", image=imageBtHome, command=goHome)
                        btnHome.place(x=155, y=545)

                        displayCamera()

                else:
                        messagebox.showerror("Error", "Not␣webcam␣available")
```

```
def displayCamera():
        global photo
        if cameraObject is not None:
                retval, photo = cameraObject.read()
                if retval == True:
                        photo = cv2.cvtColor(photo, cv2.COLOR_BGR2RGB)  # Change color to rgb
                        photo = cv2.flip(photo, 1)      # Flip horizontally
                        img = Image.fromarray(photo)
                        img = img.resize((800, 600))  # 640x480 -> 800x600
                        imgTk = ImageTk.PhotoImage(image=img)
                        captureLabel.configure(image=imgTk)
                        captureLabel.image = imgTk
                        captureLabel.after(10, displayCamera)
                else:
                        captureLabel.image = ""
                        cameraObject.release()
```

Upload image

To upload an image, **filedialog.askopenfilename()** is used to display a file selection dialog box, and where it is specified that only PNG files are allowed. This function returns the path to the file.

```
def uploadImage():
        file = filedialog.askopenfilename(title="Upload␣image",
```

```
                filetypes=[("Archivos PNG", "*.png")])
```

Generate images

To generate the four images, a series of preprocessing is first performed on the input image so that it can be used by the model to generate the output images. Then the pre-trained model weights are loaded (four different weights), and the new images are generated and assigned to a label for display on the user interface.

```
def generateImage(path):
    global btnGenerate

    input_img_size = (256, 256, 3)
    messagebox.showinfo("Generating images...",
     "Wait while images are being generated, it may take a few seconds.")

    # Image preprocessing.
    imgTF = tf.io.read_file(path)
    imgTF = tf.image.decode_jpeg(imgTF, channels=3)
    imgTF = tf.image.resize(imgTF, [input_img_size[0], input_img_size[1]])
    imgTF = tf.cast(imgTF, dtype=tf.float32)
    imgTF = (imgTF / 127.5) - 1.0
    imgTF = tf.expand_dims(imgTF, 0)

    # Get the generators
    gen_G = get_resnet_generator(name="generator_G")
    gen_F = get_resnet_generator(name="generator_F")

    # Get the discriminators
    disc_X = get_discriminator(name="discriminator_X")
    disc_Y = get_discriminator(name="discriminator_Y")

    # Create cycle gan model
    cycle_gan_model = CycleGan(
    generator_G=gen_G, generator_F=gen_F, discriminator_X=disc_X, discriminator_Y=disc_Y
    )

    # Compile the model
    cycle_gan_model.compile(
    gen_G_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
    gen_F_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
    disc_X_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
    disc_Y_optimizer=keras.optimizers.legacy.Adam(learning_rate=2e-4, beta_1=0.5),
    gen_loss_fn=generator_loss_fn,
    disc_loss_fn=discriminator_loss_fn,
    )

    # Load the checkpoints
    weight_file = "checkpoints/cyclegan_checkpoints.001"
    cycle_gan_model.load_weights(weight_file).expect_partial()
    print("Weights loaded successfully")
    prediction1 = cycle_gan_model.gen_G(imgTF, training=False)[0].numpy()
```

```
44      prediction1 = (prediction1 * 127.5 + 127.5).astype(np.uint8)
45      prediction1 = keras.preprocessing.image.array_to_img(prediction1)
46      img1 = prediction1.resize((300, 300))
47      img1 = ImageTk.PhotoImage(img1)
48
49      weight_file = "checkpoints/cyclegan_checkpoints.020"
50      cycle_gan_model.load_weights(weight_file).expect_partial()
51      print("Weights loaded successfully")
52      prediction2 = cycle_gan_model.gen_G(imgTF, training=False)[0].numpy()
53      prediction2 = (prediction2 * 127.5 + 127.5).astype(np.uint8)
54      prediction2 = keras.preprocessing.image.array_to_img(prediction2)
55      img2 = prediction2.resize((300, 300))
56      img2 = ImageTk.PhotoImage(img2)
57
58      weight_file = "checkpoints/cyclegan_checkpoints.008"
59      cycle_gan_model.load_weights(weight_file).expect_partial()
60      print("Weights loaded successfully")
61      prediction3 = cycle_gan_model.gen_G(imgTF, training=False)[0].numpy()
62      prediction3 = (prediction3 * 127.5 + 127.5).astype(np.uint8)
63      prediction3 = keras.preprocessing.image.array_to_img(prediction3)
64      img3 = prediction3.resize((300, 300))
65      img3 = ImageTk.PhotoImage(img3)
66
67      weight_file = "checkpoints/cyclegan_checkpoints.012"
68      cycle_gan_model.load_weights(weight_file).expect_partial()
69      print("Weights loaded successfully")
70      prediction4 = cycle_gan_model.gen_G(imgTF, training=False)[0].numpy()
71      prediction4 = (prediction4 * 127.5 + 127.5).astype(np.uint8)
72      prediction4 = keras.preprocessing.image.array_to_img(prediction4)
73      img4 = prediction4.resize((300, 300))
74      img4 = ImageTk.PhotoImage(img4)
75
76
77      captureLabel = Label(captureFrame, image=img1)
78      captureLabel.image = img1
79      captureLabel.place(x=0, y=0)
80
81      captureLabel = Label(captureFrame, image=img2)
82      captureLabel.image = img2
83      captureLabel.place(x=300, y=0)
84
85      captureLabel = Label(captureFrame, image=img3)
86      captureLabel.image = img3
87      captureLabel.place(x=0, y=300)
88
89      captureLabel = Label(captureFrame, image=img4)
90      captureLabel.image = img4
91      captureLabel.place(x=300, y=300)
92
93
94      btnGenerate.destroy()
95      btnSave = Button(btnFrame, text="SaveImage", image=imageBtSave,
96       command=lambda: saveImage(prediction1, prediction2 ,prediction3 ,prediction4))
97      btnSave.place(x=70, y=270)
```

```
98      btnSave.config(bg="#000a01")
```

Save images

Finally, the four images are saved if the user clicks on the button "Save Images".

```
1 def saveImage(img1, img2, img3, img4):
2         messagebox.showinfo("Images_saved", "Images_saved_successfully.")
3         img1.save("predicted_img_1.png")
4         img2.save("predicted_img_2.png")
5         img3.save("predicted_img_3.png")
6         img4.save("predicted_img_4.png")
```