UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Image-to-image domain translation using CycleGAN

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Furió Agustí, Juan Alejandro

Tutor: Casacuberta Nolla, Francisco

External cotutor: YOSHIDA, SHINICHI

ACADEMIC YEAR: 2021/2022

Resum

Els últims avanços en el camp de les xarxes neuronals han portat a l'auge de les xarxes generatives, les quals necessiten grans quantitats de dades per a entrenar-se. Aquestes dades moltes vegades existeixen en formats que no s'ajusten als requisits dels models d'aprenentatge supervisat. Això porta al camp de l'aprenentatge no supervisat, on els models poden treballar amb una varietat mes àmplia de conjunts de dades. En aquest treball s'aborda la capacitat d'un model d'aprenentatge no supervisat per a traslladar imatges entre dominis i la capacitat d'adaptació en conjunts de dades amb diferents classes dins de cada domini.

Paraules clau:
aprenentatge automàtic, gan, generació d'imatges, aprenentatge no supervisat

Resumen

Los últimos avances en el campo de las redes neuronales han llevado al auge de las redes generativas, las cuales necesitan grandes cantidades de datos para entrenarse. Estos datos muchas veces existen en formatos que no se ajustan a los requisitos de los modelos de aprendizaje supervisado. Esto lleva al campo del aprendizaje no supervisado, donde los modelos pueden trabajar con una variedad mas amplia de conjuntos de datos. En este trabajo se aborda la capacidad de un modelo de aprendizaje no supervisado para trasladar imágenes entre dominios y la capacidad de adaptación en conjuntos de datos con distintas clases dentro de cada dominio.

Palabras clave: aprendizaje automático, gan, generación de imágenes,
aprendizaje no supervisado

Abstract

In recent years, advances in the field of neural networks have led to the rise of generative networks, which require large amounts of data to train. These data often exist in formats that do not meet the requirements of supervised learning models. This leads to the field of unsupervised learning, where models can work with a wider variety of datasets. This work addresses the ability of an unsupervised learning model to translate images between domains and the ability to adapt on datasets with different classes within each domain.

Key words: machine learning, gan, image generation, unsupervised learning

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

During the last few years, the field of machine learning has experienced a boom due to its extraordinary progress in a wide variety of tasks. In particular what caught my attention were the generative models, especially dedicated to images and videos.

This project aims to expand the knowledge and expertise in the field of machine learning while developing a solution using a technique such as CycleGAN and implementing it with Tensorflow Keras, one of the most widely used frameworks for machine learning.

1.2 Objectives

This project aims to develop a machine learning model that translates images between different image domains. The model should be able to adapt to different domains if given the appropriate data.

To develop and evaluate the model, we have identified the following sub-objectives:

- Find a suitable dataset that fits the needs of this task and pre-process it so that it can be fed to the model
- Develop the deep neural network model itself
- Test the performance of the model in the given dataset and compare between the results.

1.3 Structure of the memory

In chapter 2 we explain the basics of machine learning and neural networks, diving into the optimization mechanism behind them and some useful approaches like CNN.

Next, in chapter 3, we discuss which technologies can be used to address the task of image-to-image translation and how they differ from our proposed model.

In chapter 4 is explained the idea and functioning of the CycleGAN, along with some auxiliary concepts.

The experimental setup is outlined in chapter 5, where we expose everything surrounding the development of the model: technologies used, dataset, functions, and network architecture.

Following with chapter 6, where we perform all the tests, train the final model and evaluate it.

In chapter 7 we finish with a brief comment on the project and possible future work.

CHAPTER 2
Theoretical Background

In this chapter, we introduce the basic concepts needed for a proper understanding of the entire report.

First, we will talk about machine learning, types of machine learning and introduce the concept of neural network. Then, we will dive into neural networks and its learning and optimization techniques. Finally, we will introduce some more advanced concepts that will be reviewed in later chapters.

2.1 Machine Learning

The so-called "machine learning" is nothing more than a field of study dedicated to researching and developing methods by which machines can "learn" and thus solve problems in their own way. This is really useful when faced with problems that are difficult to tackle with classic algorithmic methods. We will begin by explaining the starting point of the approach of the machine learning field.

2.1.1. Machine learning approach to problem-solving

In contrast to traditional programming, which is a manual process in which programmers have to specify the logic steps to follow and execute on their own, machine learning models learn from data.

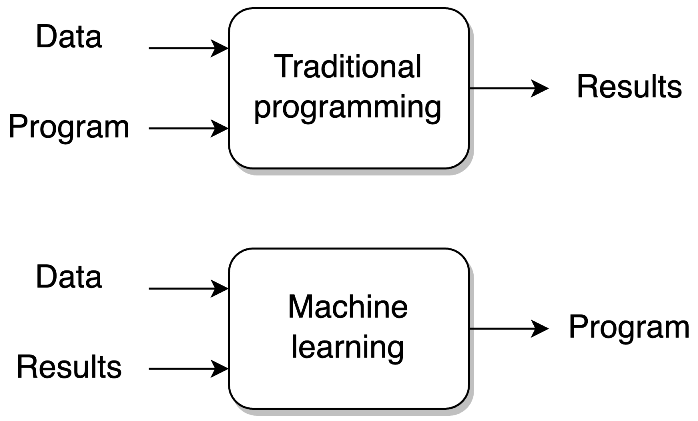


Figure 2.1: Machine learning vs tradicional programming

The goal of machine learning models is to learn the function f that maps the inputs x to the outputs y so that $f(x) = y$. Since learning f is really difficult in practice, we try to estimate f trying to learn the function h , so that $h(x) \approx y$

To accomplish this, these models are presented with many examples representing the task and they try to guess the underlying patterns and rules relating inputs and outputs.

2.2 Types of machine learning

Machine learning approaches can be divided into three categories depending on the nature of the given data or the expected output.

- **Supervised learning**: The most widely used paradigm, it focuses on learning the mapping between the input data and known targets (sometimes called labels). It is widely used for tasks such as OCR and image classification, but also for more exotic tasks, such as sequence generation and image segmentation.
- **Unsupervised learning**: Since this approach lacks the target output we are looking for, most of the unsupervised learning techniques are focused on data analysis, grouping, or interesting transformations without the aid of labels. But, as we shall see below, this is not always the case if the right techniques are applied.
- **Reinforcement learning**: The last one considers intelligent agents in a certain environment and learns to make decisions that maximize a certain utility function.

2.3 Neural Networks

In this section, we will explain the theoretical basis of neural networks, starting with *perceptron*, the fundamental unit of all NNs, and we will go deeper to finish into the field of CNNs and GANs.

2.3.1. Basic concepts

Weights

The parameters of neural networks are called weights. These are the mechanism by which networks learn to perform assigned tasks and are usually represented as matrices of floating-point numbers.

Activation function

A very important key to solve complex problems is the activation function. These functions can do some useful things like adding non-linearity, transform the output of a network to resemble a probability, generate normalized data, etc.

For now, it is enough to understand the most basic and widely used activation function, the Rectified Linear Unit (ReLU), which works as follows.

$$ReLU(x) = x^{+} = \max(0, x) \tag{2.1}$$

Where x is the input

We will further discuss these activations in chapter 4.

2.3.2. Perceptron

The idea of the perceptron [1] is inspired by the biological functioning of neurons in our brain. Biological neurons receive electric signals from other neurons and consequently produce a result, activating or not certain parts of their structure.

Basic neuron

A basic neuron looks like this.

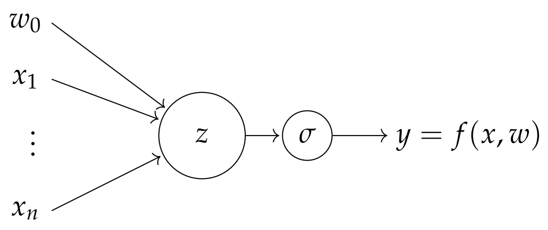


Figure 2.2: Basic neuron. Adapted from [1]

Where $x = \{x_1, \cdots, x_n\}$ is the input vector, $w = \{w_0, \cdots, w_n\}$ are the parameters (called weights) of the given neuron and σ is the activation function.

The neuron function is

$$\sigma\left(\sum_{i=1}^{n}(w_i x_i) + w_0\right) \tag{2.2}$$

Usually written in uniform notation as a dot product, assuming x has a first component $(x_0 = 1)$ so the bias $(b \equiv w_0)$ can be included in the result. The simplified formula becomes as follows:

$$\sigma(w \cdot x) \tag{2.3}$$

If we replicate the same operation for several output neurons, we get a perceptron with multiple outputs 2.3. This type of layer is usually called a **fully connected** layer. For each output neuron, we have different weights which will be adjusted when the network is trained. This comes in handy when our task requires more than one number as an output.

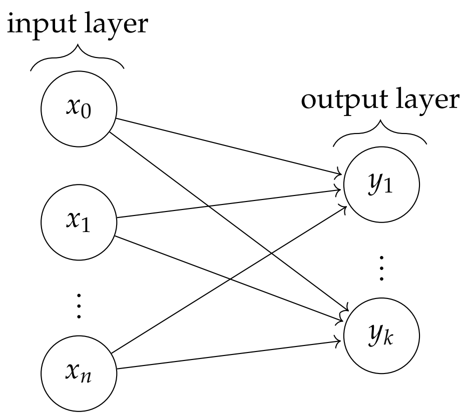
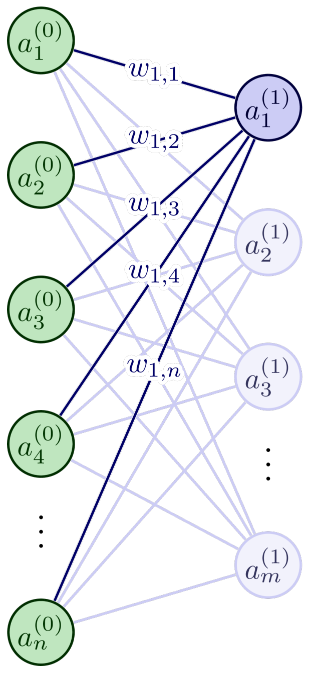


Figure 2.3: One-layer perceptron. Source [2]

[1] https://davidstutz.de/illustrating-convolutional-neural-networks-in-latex-with-tikz/
[2] https://davidstutz.de/illustrating-convolutional-neural-networks-in-latex-with-tikz/

Therefore, the network output can be expressed in a matrix form (Figure 2.4), which makes it possible to perform network training on GPUs due to its fast implementation of matrix operations. In the end, all operations used in neural networks are applied in a matrix form on GPU modules.



$$= \sigma\left(w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ w_{2,0} & w_{2,1} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \ldots & w_{m,n} \end{pmatrix}\begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$a^{(1)} = \sigma\left(\mathbf{W}^{(0)}a^{(0)} + \mathbf{b}^{(0)}\right)$$

Figure 2.4: Forward propagation in one neuron. Source [3]

2.3.3. Multilayer perceptron

Once we understand the basics of the perceptron, we can make the behavior more complex by adding more layers between input and output. This is what is called a Multi-Layer Perceptron (MLP) [2].

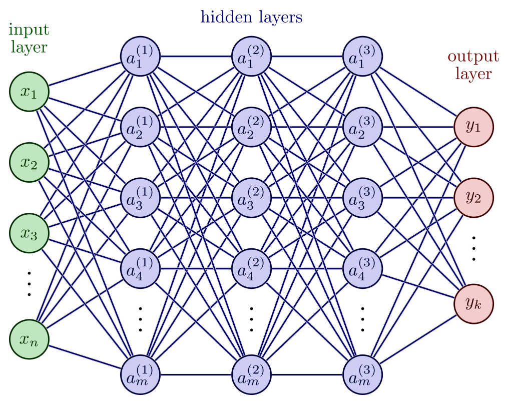


Figure 2.5: Multilayer perceptron with three hidden layers. Source [4]

Hence, the output of a neuron in a given layer is the dot product of the outputs of the previous layers and the weights corresponding to that neuron, plus the bias. Repeating this along all the layers is what is called forward propagation.

[3]https://tikz.net/neural_networks/
[4]https://tikz.net/neural_networks/

This type of network has proven to be very effective in solving a wide variety of problems, especially in classification and regression.

2.3.4. Loss

To know how well our model performs, we can compare the output of the model with the expected output or **target**. For that purpose, we use a loss function, a function that returns the "distance" between the expected and the real output, that is, how close our output is to what it should be.

In current models, several loss functions are used, although the most common are cross-entropy [3] for classification problems.

$$-\sum_{i=1}^{N} y_i \log(p_i) \tag{2.4}$$

Where y are the labels or targets representing the classes of the samples and p are the predicted probabilities of these classes.

In addition, Mean Squared Error (MSE) [4] is used for regression problems.

$$\sum_{i=1}^{N} (x_i - y_i)^2 \tag{2.5}$$

Where x are the predicted values and y are the target values.

2.3.5. Gradients and learning rate

On the basis of this loss function, we can also obtain the gradients of the loss function in relation to the weights of the network at a specific time. We can move in the direction of gradients trying to reduce the loss value (Figure 2.6).

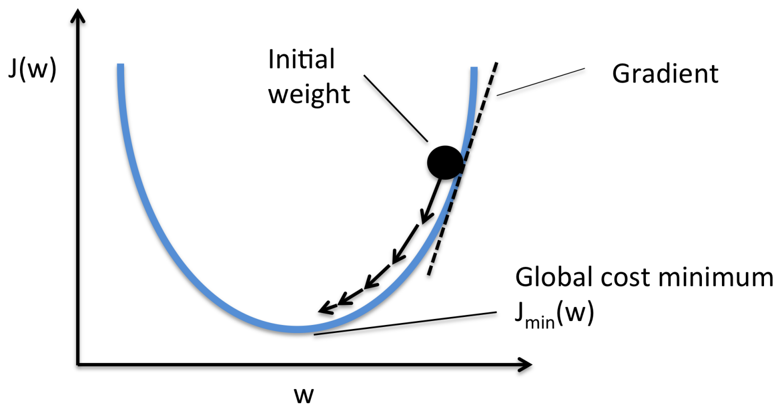


Figure 2.6: Gradient descend. Each arrow represents one learning step. Source [5]

The parameter that determines how much we have to move in the direction of the gradients is called *learning rate*.

If the learning rate is too low, the training will take a lot of time, since at each step the weights will not even change. On the contrary, if it is too high, the network could

[5]https://sebastianraschka.com/faq/docs/gradient-optimization.html

have convergence problems, since the network could jump over a minimum of the loss function. That is why it is especially important to properly adjust the learning rate to find the balance between speed and quality.

Given the weights of the network at a time $\theta_{(t)}$, the learning rate η, and the value of the loss function $\mathcal{L}(\theta)$, we can update the weights after each training step with the following formula.

$$\theta_{(t+1)} = \theta_{(t)} - \eta \nabla \mathcal{L}(\theta_{(t)}) \tag{2.6}$$

In practice, better results are obtained by using a learning rate scheduler or, even better, an optimizer, which will be discussed in more detail in chapter 5.

2.3.6. Backpropagation

Backpropagation [5] uses SGD to compute the gradient of the loss function with respect to the weights of the network and update those weights to reduce the loss error and optimize the predictions of the network.

Going back to a simple example, the weights of the neurons are updated starting from the output, as shown in Figure 2.7

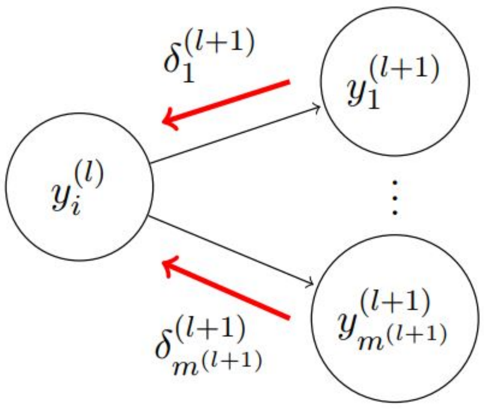


Figure 2.7: Backpropagation step. Source [6]

Once evaluated for all output units, the errors δ_n^{l+1} of the neuron y_n^{l+1} in layer $(l+1)$ can be propagated backward to neurons in the previous layer, up to the weights of the first layers.

2.3.7. CNN

A Convolutional Neural Network (CNN) [6] is a type of NNs designed to work with data that are assumed to be images. In fully connected networks, each neuron is connected to all neurons in the previous layers. Let us assume that we have a square image of 256x256x256 (3 color channels); we would need a total of 196608 weights for only one layer. When the number of layers increases, this becomes unfeasible due to the huge size of the network weights. Moreover, it does not take into account the spatial dependencies between near pixels in the same image.

To address this problem, we use layers that apply the convolution operation. A kernel or convolution matrix is a small matrix that is used for identification and modification

[6]https://davidstutz.de/illustrating-convolutional-neural-networks-in-latex-with-tikz/

tasks. The kernel moves through an image, performing an element-wise multiplication with the corresponding pixels of the image, as shown in Figure 2.8.

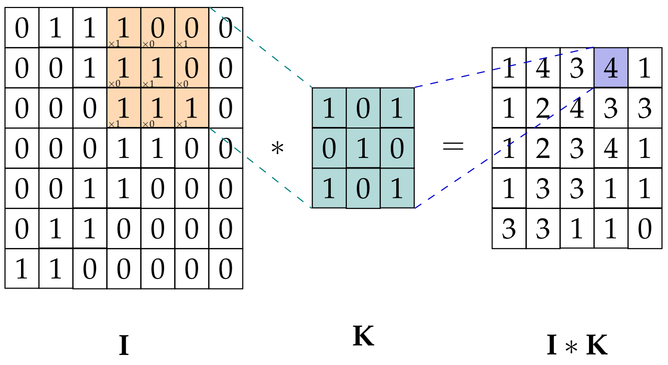


Figure 2.8: Convolution operation. The result of the convolution step (violet) is the sum of numbers in the element-wise matrix multiplication between the image area on which the convolution is applied (orange) and the convolution matrix (aquamarine). Source [7]

This process is capable of recognizing dependencies between pixels and saving them in feature maps, which shows the activation of a pattern in the image in Figure 2.9

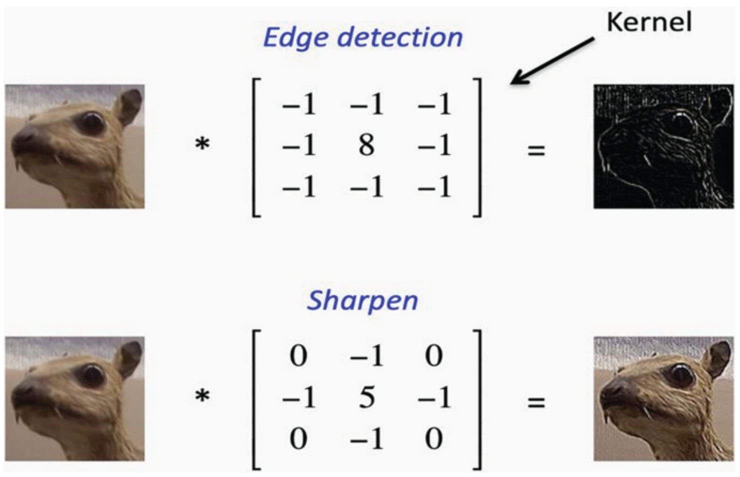


Figure 2.9: Kernel feature map activation.
On the left the original image, on the right the feature map after applying the convolution with each kernel. Source[7]

The convolution layers have different parameters that can be adjusted depending on the desired output images. The stride $S = (S_H, S_W)$ is the number of pixels that the kernel moves with each step. Consequently, a stride of 2 would produce an image that is around half the size of the original in both dimensions (width and height). The kernel size (K_H, K_W) is just the size of the convolution matrix used for the operation. The padding (P_W, P_H) is the amount of zeros (or other values) added in the borders of the image; this is done to maintain the original image size.

Given an image with dimensions (I_H, I_W), the size of the feature map (F_H, F_W) resulting from the convolution can be determined by following these formulas 2.3.7.

$$F_H = \frac{I_H + 2 * P_H - K_H}{S_H} + 1 \qquad F_W = \frac{I_W + 2 * P_W - K_W}{S_W} + 1$$

[7]https://tikz.net/neural_networks/

The convolution operation was first used in combined networks to extract dependencies between pixels that represent some features in order to pass them to a set of fully connected layers. This represented a breakthrough in the field of image classification models.

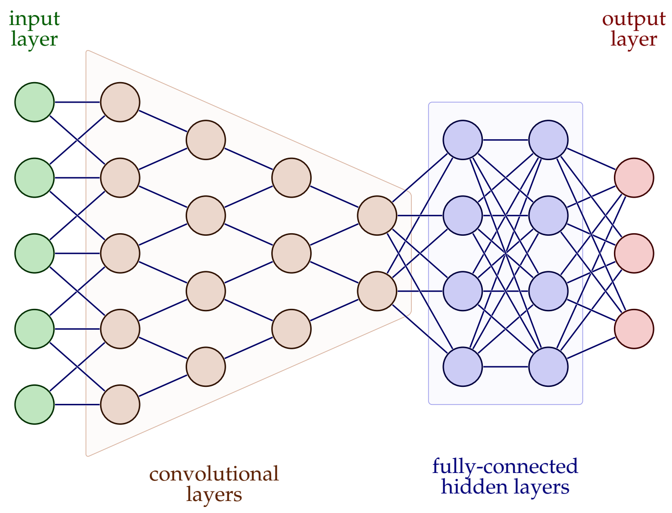


Figure 2.10: A scheme of a CNN followed by fully connected layers. Source[8]

Even so, convolution layers can also be used to modify an image if we do not add any fully connected layer before the output of the network.

2.3.8. Normalization

Dropout

Another technique that can be applied is dropout [8], commonly used during the training of DNN to prevent overfitting (when the networks learns to fit too much to the training data and as a result it becomes unable to generalize correctly) that consists on shutting down some random neuron at training time. This helps networks not to rely on a single or small set of features, and therefore decreases their likelihood of being overly dependent on available data.

The following image (2.11) shows the "inactive" neurons in a NN using dropout.

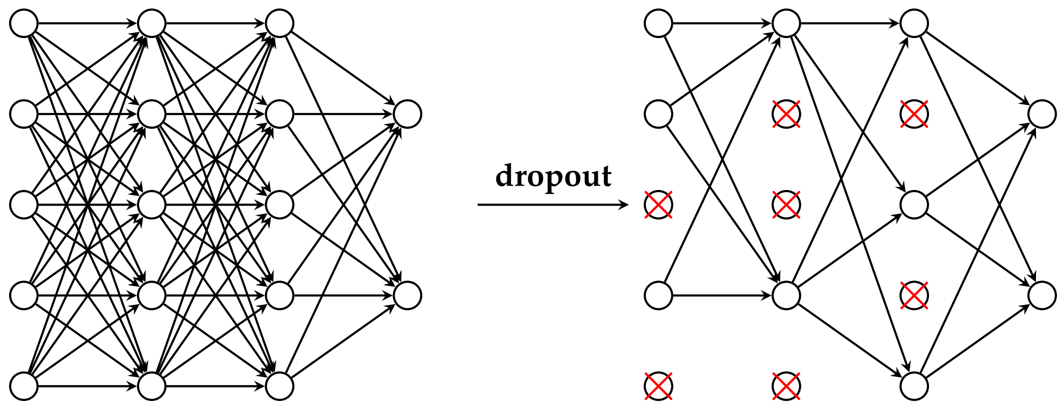


Figure 2.11: Dropout in a Fully Connected NN. Source [9]

[8]https://tikz.net/neural_networks/
[9]https://tikz.net/neural_networks/

Batch Normalization

Batch Normalization [9] (also called batch norm) is a technique for training DNN that standardizes the input of a layer for each training step. This has the effect of stabilizing the learning process and dramatically reduces the number of training epochs required to train deep networks.

Even though batch norm is the most well-known of them, there are more variations of the same idea. There is another layer, Instance Normalization, that standardizes each element of the batch independently. We will discuss more about Instance Normalization in Chapter 5

2.3.9. GAN concept

The mechanism of the Generative Adversarial Network (GAN)[10] is really intuitive, two networks (a generator and a discriminator) compete against each other in a way that one's gain means the other's loss, that is, a zero-sum game. That way they force each other to improve. The generator learns how to produce more realistic samples and the discriminator learns to discern between real and generated samples.

Usually, the generator takes some sort of input (often just random noise) and generates fake samples based on that input.

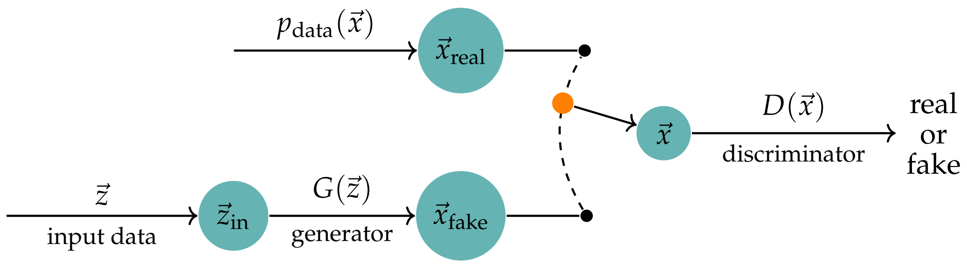


Figure 2.12: GAN concept. Adapted [10]

Different problems arise during the training of GANs.

- Non-convergence. The model loss oscillates, destabilizing and not finding a good equilibrium between generator and discriminator
- Mode collapse. The model fails to generalize; this normally occurs due to the generator learning faster than the discriminator, it gets stuck at a local minimum, and it stops improving.
- Vanishing gradient. Conversely, if the discriminator learns to always beat the generator, the generator will always get a high loss, leading to low gradients that cannot backprop throughout the entire network.

The GAN game objective function is

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[1 - \log D(G(z))] \tag{2.7}$$

[10]https://tikz.net/neural_networks/

CHAPTER 3
State of the Art

In this chapter, we will explain other techniques and networks that can be used to face the image-to-image translation task and their main differences with respect to our proposal.

Image-to-image translation is a task that consists in transferring images from a source domain to a target domain without losing the image content. Various techniques have been proposed, achieving both realistic and non-realistic results. Some of them have been widely used and others have been only a curious line of research or an ingenious application of already existing networks.

Let us take a look to the most interesting and successful ones and how they are related.

3.1 DeepDream

DeepDream [11] is a network created by a Google engineer that combines detail injection and upsampling to create hallucination-like visuals. The detail injection is based on maximizing particular activations of the Inception [12] network, thus injecting the pattern learned by the network into the image. It uses the backpropagation algorithm, but instead of changing the weights of the network, it modifies the original image.

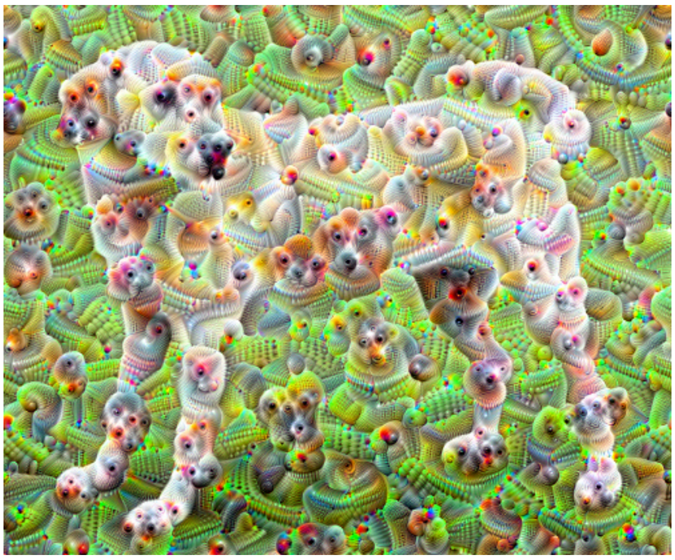


Figure 3.1: An image of a dog modified using DeepDream [11]

However, this network not only makes it very difficult to control the result of the generated image, since we do not know the specific purpose of the activation of the layers, but also does not generate realistic images like we aim in this project.

3.2 Variational Autoencoders

The VAE [13] architecture works with two components, an encoder and a decoder. The former tries to encode the input data into its representation in a latent space, whereas the latter tries to reconstruct the input from the latent representation. The reconstructed image should be as close as possible to the original.

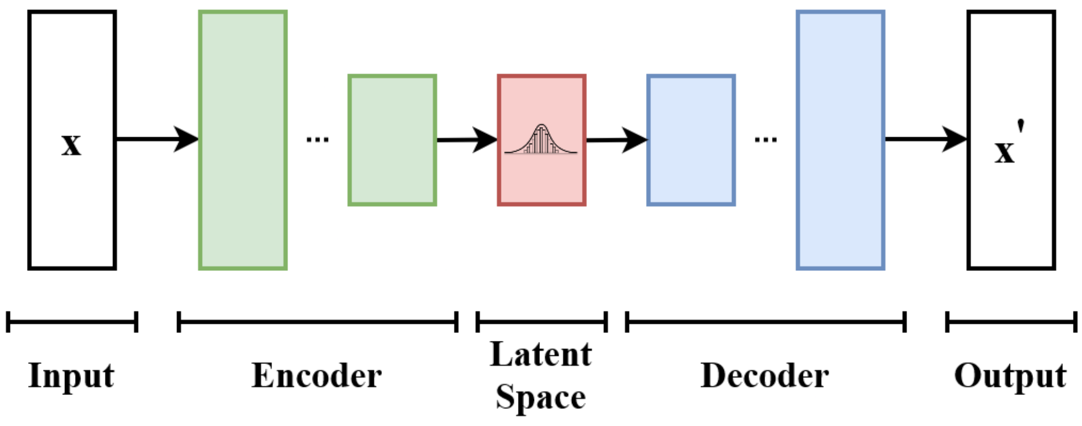


Figure 3.2: VAE scheme. Source [1]

In the latent space there may be directions that encode high-level characteristics of the original data; we call this a concept vector. For example, given an original space of faces, there could be a concept vector in the latent space that represents a smile. Adding this vector to the latent representation of a specific face would make the reconstructed face look more smiling. The opposite is also true; subtracting the concept vector from the latent representation would make the reconstructed face look sad.

Although this is a powerful tool, concept vectors are sometimes unpredictable as well as difficult to find.

3.3 Conditional Adversarial Networks (pix2pix)

If you take the idea of GANs and replace the input of a random sample of a latent space with a real image at the input, plus an expected output image, you get the pix2pix network [14].

This network not only learns the mapping between input and output using the generator, but also learns a loss function by the discriminator.

As opposed to our proposed model (which will be discussed in the next chapter), pix2pix network uses supervised learning (seen on page 4) – i.e., it learns from paired data. Since these models have a very reliable way of evaluating their results (by com-

[1]https://commons.wikimedia.org/wiki/File:VAE_Basic.png

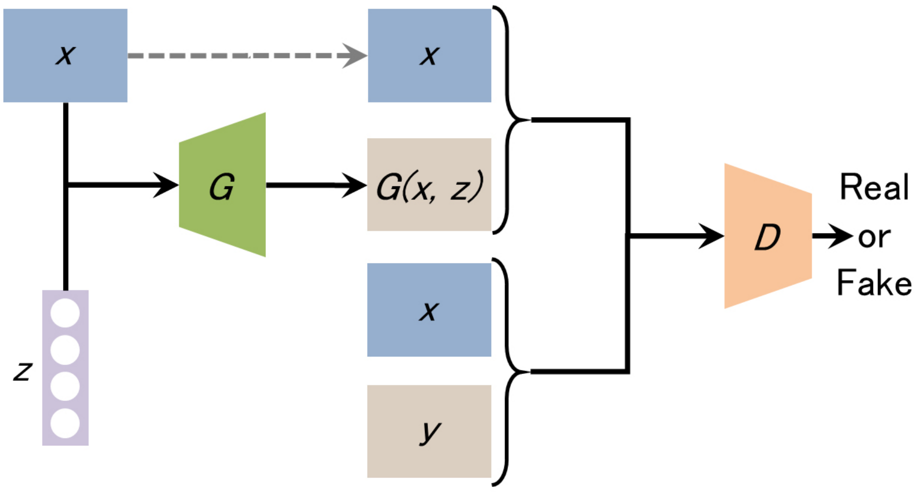


Figure 3.3: Scheme of a pix2pix model

paring them with the target images), they are usually more powerful and perform better than the unsupervised ones.

Therefore, they can achieve better results at the cost of having a tighter set of constraints on the data set.

Pix2pix has been used to approach tasks such as image domain translation or image segmentation, as can be seen in Figure 3.4 and has become a standard of multipurpose image-to-image models.

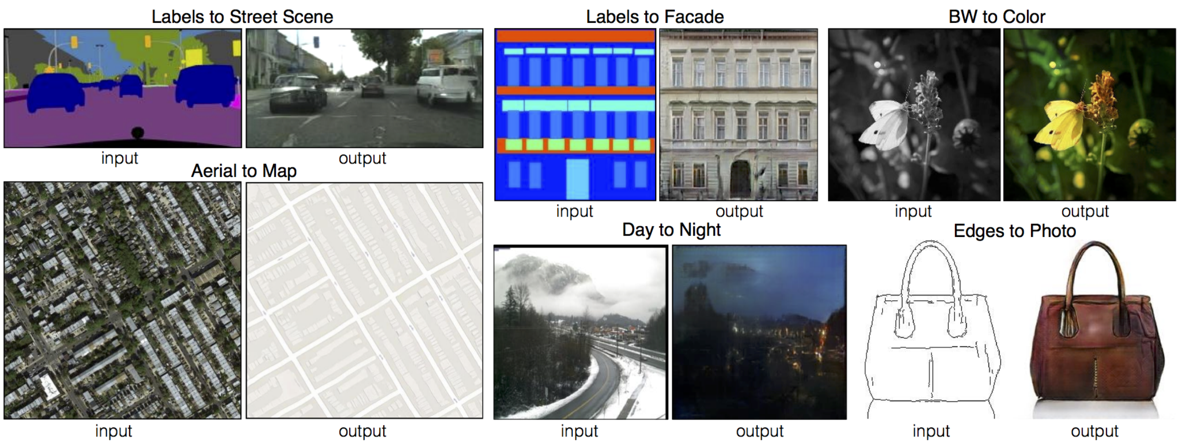


Figure 3.4: Examples of applications of pix2pix network [14]

CHAPTER 4

Proposal

In AI, the same problem can be addressed with a wide variety of approaches and tools. In this chapter we will see which one has been chosen for this project and the main reasons.

4.1 CycleGAN

Cycle consistent Generative Adversarial networks (CycleGAN) [15] is a variation of the basic GAN model. It uses two GANs in conjuction with a cycle-consistency loss to address the task of unsupervised image-to-image translation.

Nowadays, we can find huge amounts of data (especially visual data, videos and images) on the internet. CycleGAN can use data that does not meet the strict requirements of supervised learning models. The lack of need for paired 4.1 data makes this model much more viable than its supervised analogous, the pix2pix network.

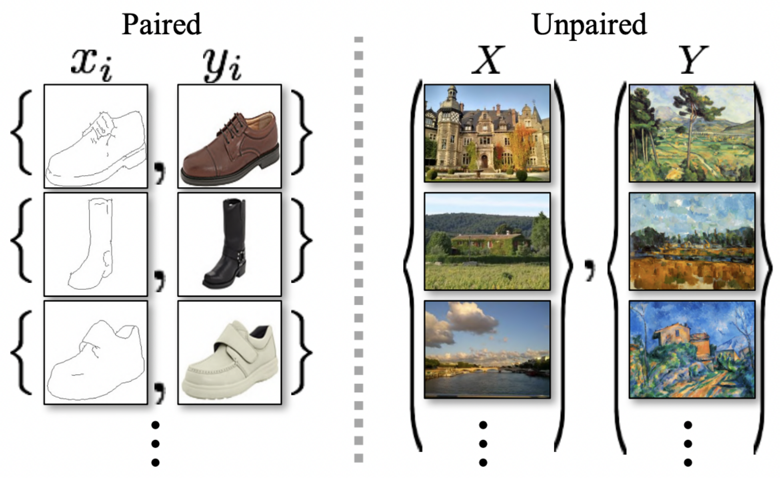


Figure 4.1: Examples of paired and unpaired datasets

CycleGAN model works by translating images from one domain to another with a generator, and translating back to the original domain with another generator.

To check if the translation has been done properly, we use a discriminator that learns to classify if an image pertains to a domain. In addition, we add a cycle consistency loss that forces the original and the reconstructed images to be as similar as possible. Lastly, we use a last loss called identity loss to improve the results, as the authors suggest in the original paper [15].

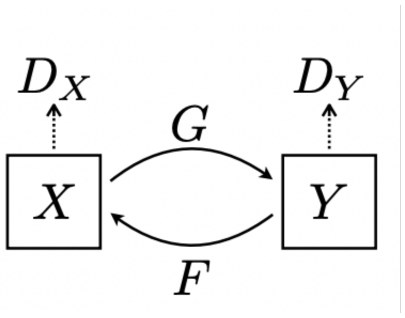


Figure 4.2: CycleGAN scheme

To express the problem in a formal way:

There are two domains:

$$X \qquad\qquad Y$$

Two discriminators:

$$D_X : X \to [0,1] \qquad\qquad D_Y : Y \to [0,1]$$

And two generators:

$$G : X \to Y \qquad\qquad F : Y \to X$$

Adversarial loss

Following the basic GAN proposal, applied to two generators and two discriminators, G tries to generate images that pertain to domain Y and D_Y tries to classify real images from domain Y mixed with translated samples generated by G, and the equivalent for generator F and discriminator D_X.

$$\mathcal{L}_{GAN}(G, D_Y) = E_{y \sim p_{data}(y)}[\log D_Y(y)] + E_{x \sim p_{data}(x)}[\log(1 - D_Y(G(x))]$$

$$\mathcal{L}_{GAN}(F, D_X) = E_{x \sim p_{data}(x)}[\log D_X(x)] + E_{y \sim p_{data}(y)}[\log(1 - D_X(F(y))]$$

Cycle consistency loss

After generating the fake images, we can feed the image generated by one generator to the other. We compare the reconstructed image to the original one, trying to match them. The cycle consistency loss is usually multiplied by a weight, λ

$$\mathcal{L}_{cycle}(G, F) = E_{x \sim p_{data}(x)} \|F(G(x)) - x\| + E_{y \sim p_{data}(y)} \|G(F(y)) - y\|$$

Identity loss

Given a generator that translates images from a source domain to a target domain, we expect it to perform that task. If the image receives as an input an image from the target domain, the generator should keep that input as it is, without changing anything.

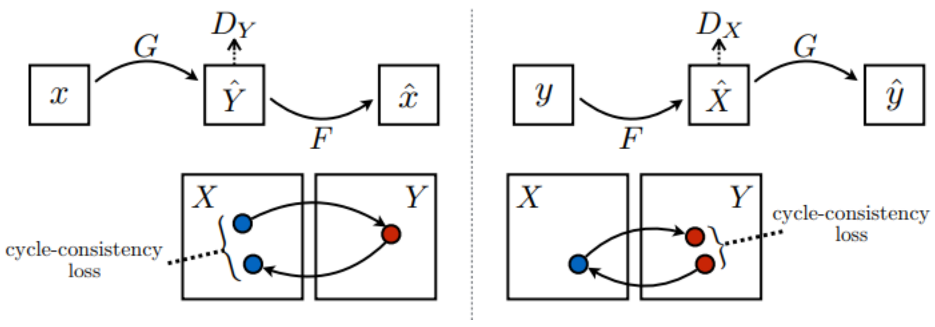


Figure 4.3: Cycle-consistency Loss. Source [15]

The identity loss attempts to impose this condition assuring that the generated image remains as close as possible to the original one. It does this by computing the distance between the two images. It has a weight, usually γ, by which it is multiplied, along with λ, the cycle consistency weight

$$\mathcal{L}_{identity}(G,F) = E_{x\sim p_{data}(x)}(x)\|F(x)-x\| + E_{y\sim p_{data}(y)}\|G(y)-y\|$$

Total loss

To compute the total loss, we perform the weighted sum of all the losses:

$$\mathcal{L}_T(G,F,D_X,D_Y) = \mathcal{L}_{GAN}(G,D_Y) + \mathcal{L}_{GAN}(F,D_X) + \lambda\mathcal{L}_{cycle}(G,F) + \lambda\gamma\mathcal{L}_{identity}(G,F)$$

Our objective is to solve

$$G^*, F^* = \underset{G,F}{\text{argmin}} \max_{D_X,D_Y} \mathcal{L}(G,F,D_X,D_Y)$$

Thus, the CycleGAN training step algorithm is: 4.1

Train step algorithm:

```
1:  function TRAIN STEP($r_x, r_y$):
2:      Generate fake images
            $f_y = G(r_x) \ , \ f_x = F(r_y)$
3:      Generate cycled images
            $c_x = F(f_y) \ , \ c_y = G(f_x)$
4:      Generate identity images
            $i_x = F(r_x) \ , \ i_y = G(r_x)$
5:      Classify real images
            $D_X(r_x) \ , \ D_Y(r_y)$
6:      Classify fake images
            $D_X(f_x) \ , \ D_Y(f_y)$
7:      Compute losses (as described in 4.1)
            $\mathcal{L}_T(G, F, D_X, D_Y)$
8:      Compute gradients
            $\nabla_\theta \mathcal{L}$
9:      Update network weights
            $w^{t+1} = \text{Optimizer}(\nabla_\theta \mathcal{L}, \theta)$
10:     return losses
11: end function
```

CHAPTER 5

Experimental Framework

This chapter explains the technical choices made for the implementation of the Cycle-GAN, starting with the used technologies and chosen dataset, as well as the essential functions for the model operation and training. In addition, we discuss the proposed architecture for the generators and discriminators, and we finish with some details about the hardware available for the development.

5.1 Technologies

As for the technologies, the project is written using the following software, frameworks and libraries. The code is available in this [1] GitHub repository.

Python

Python is a high-level interpreted programming language. It was designed with an emphasis on code readability. Python is dynamically typed and supports multiple programming paradigms, including structured, object-oriented and functional programming. Python has become very popular in AI and Data Science, as it combines its ease of use with highly efficient AI libraries.

Version: 3.9.12

TensorFlow

TensorFlow[16] is an open source end-to-end platform for artificial intelligence and machine learning, but is focused on the development of DNNs.

It can be used in many programming languages, such as Python, JavaScript, and Java. It was written in Python and C++. Tensorflow was developed by Google and was first released in 2015. A few years later, in 2019, an updated version named TensorFlow 2.0 was released.

It has achieved popularity due to its highly flexible ecosystem coupled with high performance derived from its integration with cuDNN and CUDA[2] that allows for GPU accelerated ML models, datasets and optimizers.

Version 2.9.1

[1] https://github.com/Alefuag/domainnet-cyclegan

[2] https://developer.nvidia.com/cuda-toolkit

Keras

Keras [3] is another open-source Python library built on top of TensorFlow. The main creator and maintainer of Keras is François Chollet, a software engineer working at Google.

Keras provides a high-level architecture API, as well as a low-level training API. This allows for a high degree of customization of the training loop, while at the same time facilitates network architecture development. It enables a fast development and experimentation framework for state-of-the-art research.

From TensorFlow v2.0, Keras is included within the tensorflow package.

Version 2.9.1

NumPy

NumPy [17] is a Python library that adds support for multi-dimensional arrays, in addition to many useful mathematical functions. Numpy arrays have a great speed-up compared to regular Python lists and are the basis of TensorFlow Tensors.

Version 1.23.2

Pandas

Pandas is a Python library for data manipulation and analysis. It allows to create dataframes from CVS files and has a useful set of Python-like syntax tools.

Version 1.4.4

Seaborn

Seaborn is a Python data visualization library that integrates with Pandas. It provides a high-level interface for visualizing statistical data. Most of the graphs shown in this memory were created using Seaborn.

Version 0.12.0

Tensorboard

Tensorboard [4] is a tool that provides the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics such as loss and accuracy, displaying images, text, and audio, visualizing the model graph, and much more.

Version 2.2.0

Jupyter Notebook

Jupyter[18] is a web-based interactive computing platform. The notebook combines code, markdown text, and image visualizations. It works on top of the Ipython kernel to provide an interactive execution based on input-output cell pairs. It has been used to test the model, as well as to perform all the experiments for an easy and fast development flow.

[3]https://keras.io
[4]https://www.tensorflow.org/tensorboard

Docker

Docker[19] is a set of PaaS products that use OS-level virtualization to deliver software in packages called containers. These containers are isolated from one another and have their own software, libraries and configuration files. In this way, it is possible for different users to share a machine without interfering with each other's installation.

Version 20.10.3

5.2 Dataset

Given the nature of the task we are trying to solve, it is necessary to obtain a dataset that meets the following requirements.

- The data collection needs to contain a few domains. It is desirable to have at least 3, so that the transfer between domains can be compared.
- Each domain should have a decent amount of samples, for the NN to be able to map the translation function properly without overfitting due to a lack of data
- Each image must have enough resolution, since a low image quality could lead to undesired behavior due to non-representation of important details.

After intensive research, we have found a dataset that fits all the previous points, presented in the DomainNet paper[20]

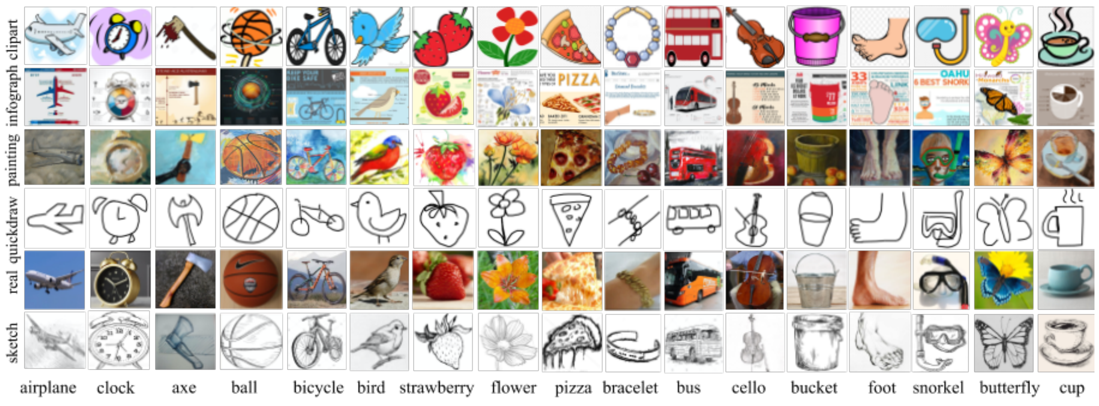


Figure 5.1: DomainNet dataset examples. Source [20]

This dataset consists on nearly **0.6 million images** of common objects in six different domains.

It has a wide range of classes, including *bracelet*, *plane* and *strawberry*, among others.

The DomainNet dataset if formed by the following domains:

- Quickdraw (172,500 samples): drawings of the players of the game 'Quick Draw'.
- Sketch (69,128): sketches of specific objects, usually black and white
- Clipart (48,129 samples): collection of clipart images
- Infograph (51,605 samples): infographic images with specific object
- Painting (72,266 samples): artistic depictions of objects in the form of paintings

- Real (172,947 samples): photos and real-world images

These domains have been sorted by level of detail as follows:

$$\text{quickdraw} \rightarrow \text{sketch} \rightarrow \text{clipart} \rightarrow \text{infograph} \rightarrow \text{painting} \rightarrow \text{real}$$

When evaluating results, we will pay more attention to the domain that has more detail, as it is more difficult to add detail and color than to remove it.

5.2.1. Important decisions regarding the dataset

Since the infograph domain has naturally text and the clipart domain has many watermarks, which do not produce good results in GAN, we will drop these two in our experiments.

Given that each domain has a different amount of samples, the amount of training will be expressed in number of batches to be domain independent.

5.2.2. Preprocessing the dataset

We need to process the dataset to be able to feed it to the network. As designed, the CycleGAN accepts RGB images with size 256x256 – i.e. (256, 256, 3)

As it is, each domain of the DomainNet dataset contains images of different classes with different image sizes.

First of all, as this is an unsupervised task, we drop the labels of all the samples. After that, we do a random square crop followed by a resizing to 256x256. Lastly, and only to improve the performance of the network, the images will be normalized so that each value range is $[-1, +1]$.

Since each RGB pixel is represented by values that range from 0 to 255, after applying the normalization the minimum value will be -1 and the maximum will be $+1$.

$$\text{Normalized image} = \frac{\text{original image}}{127.5} - 1 \tag{5.1}$$

Before training, we only need to batch each dataset with the desired batch size and merge the two domains so that each batch of the combined dataset is $(batch\_dom\_X, batch\_dom\_Y)$

5.2.3. Partitions

Finally, we have decided to split each domain into 3 sets.

- 80 % training
- 10 % validation
- 10 % test

For the experimentation phase, we will use the training split to train the model and the validation split to check the performance.

For the results phase, we will train the model with a set consisting of both test and validation splits and evaluate it with the test split.

5.3 Optimizer

In chapter 2.3.5 we talk about the learning rate and its vital importance in achieving good results when training neural networks. Optimizers are closely related to the learning rate, but offer superior convergence results than plain learning rates.

An optimizer is an algorithm that indicates how the model's learnable parameters should be updated. They have better performance compared to just using a learning rate and gradient descend.

5.3.1. Adam

The optimizer that we will use is called Adam. Adam optimization is a SGD method based on the adaptive estimation of first-order and second-order gradient moments and provides learning rates for each parameter. Adam can be seen as a combination of RMSprop [21] and SGD with momentum [22].

The updates for each training step are:

$$m_\theta^{(t+1)} \leftarrow \beta_1 m_\theta^{(t)} + (1-\beta_1)\nabla_\theta \mathcal{L}^{(t)} \tag{5.2}$$

$$v_\theta^{(t+1)} \leftarrow \beta_2 v_\theta^{(t)} + (1-\beta_2)(\nabla_\theta \mathcal{L}^{(t)})^2 \tag{5.3}$$

$$\hat{m}_\theta = \frac{m_\theta^{(t+1)}}{1-\beta_1^t} \tag{5.4}$$

$$\hat{v}_\theta = \frac{v_\theta^{(t+1)}}{1-\beta_2^t} \tag{5.5}$$

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta} + \epsilon} \tag{5.6}$$

Where θ are the network's trainable parameters, β_1 and β_2 are the decay rates for the first and second moments, η is the learning rate, ϵ is a small value to prevent a division by zero, and $\nabla_\theta \mathcal{L}^{(t)}$ are the gradients of the weights in step t.

*Squaring and square root apply element-wise.

Adam has proven to be one of the best optimizers, matching or improving the results of all other methods in almost all cases. This was the reason why it has been chosen as the optimizer for our neural network.

5.4 Loss function

As discussed in section 2.3.4, we need a function to evaluate how well our model is performing. The adversarial loss function is calculated by the discriminators, but we need to evaluate the difference between images for the cycle consistency and identity losses.

5.4.1. Mean Square Error

We use Mean Squared Error (MSE) [4] to calculate this distance.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

5.5 Evaluation metric

Once we have the optimizer and the loss function, we can start to train the model, but an evaluation metric is used to assess the quality of the trained network.

5.5.1. Frechet Inception Distance

The Frechet Inception Distance Score is a metric used to assure the quality of the images generated by an artificial neural network.

Given two normal distributions $\mathcal{N}(\mu_X, \Sigma_X), \mathcal{N}(\mu_Y, \Sigma_Y)$, the Frechet distance between them is

$$d_F(\mathcal{N}(\mu_X, \Sigma_X), \mathcal{N}(\mu_Y, \Sigma_Y))^2 = |\mu_X - \mu_Y|^2 + tr\left(\Sigma_X + \Sigma_Y - 2\left(\Sigma_X \cdot \Sigma_Y\right)^{\frac{1}{2}}\right)$$

For a given set of real images, we use the pre-trained InceptionV3 network[12] (without the top set of classification layers) to extract the features of that set. Based on these features, we calculate the parameters $(\mu_{real}, \Sigma_{real})$, which will be the reference against which we will compare. We can do this for each domain.

When we want to assess the quality of our generator, we need to generate a batch of images of the desired size (the bigger the better), generate its feature map with the inception network, and estimate its μ and Σ parameters to compute the FID between the real distribution and the generated one.

5.5.2. Activations

In Section 2.3.1, we talk about the activation functions. In practice, we can discern between two main types of activations, those that add nonlinearity, usually used in the hidden layers of the network, and those that normalize the output.

Nonlinearity

To add nonlinearity we have two main functions, ReLU and LeakyReLU:

$$\text{ReLU}(x) = \begin{cases} 0 & if \quad x < 0 \\ x & otherwise \end{cases} \quad (5.7) \qquad \text{Leaky ReLU}(x) = \begin{cases} \alpha x & if \quad x < 0 \\ x & otherwise \end{cases} \quad (5.8)$$

LeakyReLU activation helps to avoid sparcity (a matrix that has a lot of zeros), since this is detrimental to GAN models.

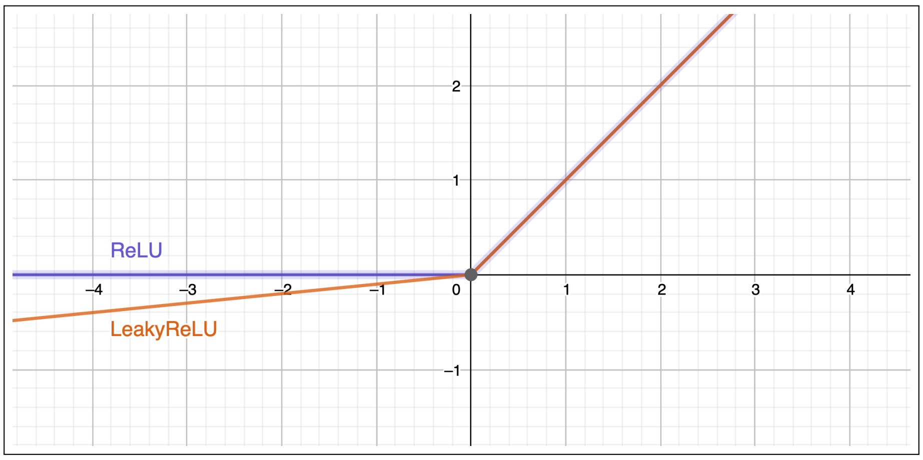


Figure 5.2: ReLU vs LeakyReLU

Normalize

To normalize the output of the networks, the sigmoid function is widely used due to its low slope, but since we are trying to generate normalized values within the range $[-1, 1]$, tanh is the right choice.

$$sigmoid(x) = \frac{1}{1+e^{-x}} \quad (5.9) \qquad tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (5.10)$$

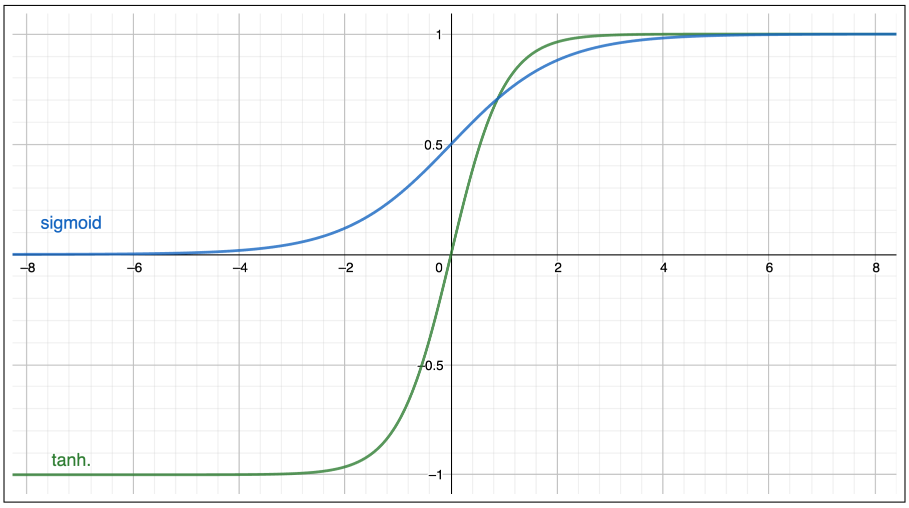


Figure 5.3: Sigmoid vs tanh

Following the common practice and as suggested in the DCGAN paper [23]:

- For the generator, the activation for all layers is ReLU, except for the final layer, that uses tanh
- Fot the discriminator, we use the LeakyReLU as the activation for all the layers.

5.6 Discriminators

Regarding the discriminator, we need an appropriate approach to the nature of the problem. Since we are not interested in the "big picture" of each image, but only in its domain, a good choice is PatchGAN [14].

A PatchGAN network is a CNN that acts as a discriminator but only penalizes images at a local scale, i.e. patches of the original image. This network classifies the images guessing if each of the $N \times N$ patches is real or fake.

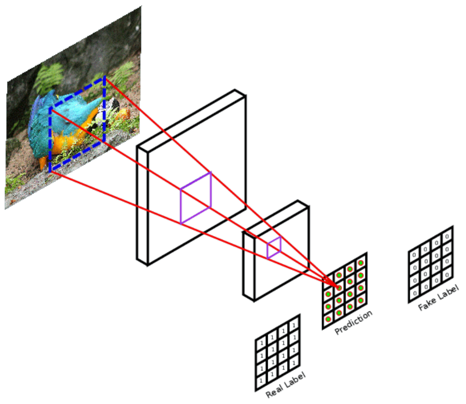


Figure 5.4: PatchGAN behaviour [24]

In our experiment, the discriminator uses a set of convolutions and activation layers, as well as normalization layers after the first downsample block.

Therefore, each block after the first is as follows:

- Downsample Layer: Conv2D
- Normalization Layer: InstanceNormalization
- Activation: LeakyReLU

The architecture of the discriminator network is shown in Table 5.1.

Layer	Output Shape
Input	256x256x3
Conv2D, 64 filters, kernel 4x4, stride 2, pad 1	128x128x64
LeakyReLU	128x128x64
Conv2D, 128 filters, kernel 4x4, stride 2, pad 1	64x64x128
Instance Normalization	64x64x128
LeakyReLU	64x64x128
Conv2D, 256 filters, kernel 4x4, stride 2, pad 1	32x32x256
Instance Normalization	32x32x256
LeakyReLU	32x32x256
Conv2D, 512 filters, kernel 4x4, stride 2, pad 1	16x16x512
Instance Normalization	16x16x512
LeakyReLU	16x16x512
Conv2D, 1 filters, kernel 4x4, stride 2, pad 1	16x16x1

Table 5.1: Discriminator layer structure (16x16 PatchGAN)

5.7 Generators

Generators are the key piece of the CycleGAN architecture, since they are the part that must learn to map one domain to another. Therefore, it is important to pay special attention to the design of this component.

We propose two generator architectures ResNet and U-Net. But first, we explain some preliminary ideas.

5.7.1. Building blocks

First of all, there are some basic building blocks that both generators share; this is due to its usefulness and wide use in the most leading-edge models.

Downsample

The downsample block consists of a convolutional layer with stride 2, followed by a normalization and activation layers, which output an image of half height and width compared to the original.

Downsample block(filters f)	
Layer	**Output Shape**
Input	dim_x, dim_y, dim_z
Conv2D, f filters, kernel 3x3, pad 1, stride 2	dim_x/2, dim_y/2, f
InstanceNormalization	dim_x/2, dim_y/2, f
ReLU	dim_x/2, dim_y/2, f

Table 5.2: Downsample Block

The other basic block is the opposite operation, the upsample block. As opposed to the downsample block, it uses a convolutional layer with fractional stride $(\frac{1}{2})$, also called deconvolution. Therefore, instead of halving the image size, it doubles it.

Upsamble block(filters f)	
Layer	**Output Shape**
Input	dim_x, dim_y, dim_z
Conv2DTranspose, f filters, kernel 3x3, pad 1, stride 2	dim_x*2, dim_y*2, f
InstanceNormalization	dim_x*2, dim_y*2, f
ReLU	dim_x*2, dim_y*2, f

Table 5.3: Upsample Block

5.7.2. ResNet

The first generator is a ResNet [25], a network architecture that has an original way of training acceleration with the use of residual connections.

A residual connection is made when the output of a layer is added to the output of another layer several steps ahead of the latter. With this approach, the layers between the first one and the residual connection do not need to map the desired image $\mathcal{D}(x)$, but the residual data $\mathcal{R}(x)$ that is added to the image to resemble the desired mapping, as can be seen in Figure 5.5.

$$\mathcal{D}(x) = \mathcal{F}(x) + x \tag{5.11}$$

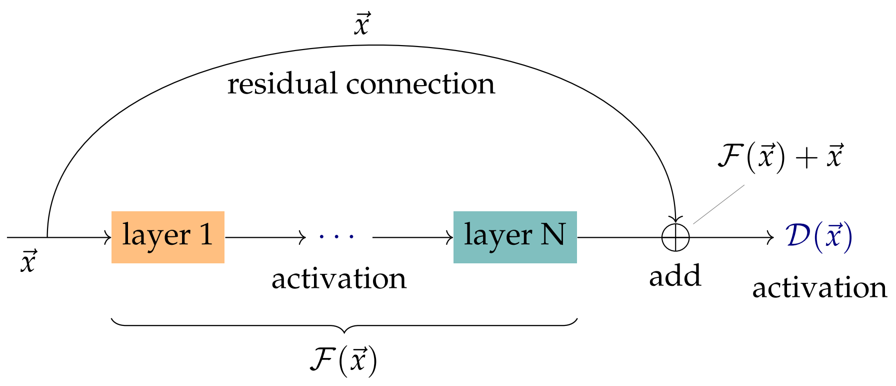


Figure 5.5: Residual connection block. Adapted from [5]

Each residual block consists of two convolutions with normalizations, with an activation between them, and a residual connection from the start to the end of the block.

Layer ID	Layer	Output Shape
[1]	Input	dim_x, dim_y, dim_z
[2]	ReflectionPadding2D, pad 1	dim_x+2, dim_y+2, dim_z
[3]	Conv2D, 3 filters, kernel 3x3	dim_x, dim_y, dim_z
[4]	InstanceNormalization	dim_x, dim_y, dim_z
[5]	ReLU	dim_x, dim_y, dim_z
[5]	ReflectionPadding2D, pad 1	dim_x+2, dim_y+2, dim_z
[6]	Conv2D, 3 filters, kernel 3x3	dim_x, dim_y, dim_z
[7]	InstanceNormalization	dim_x, dim_y, dim_z
[8]	Add([1], [7])	dim_x, dim_y, dim_z

Table 5.4: Residual block structure

Our ResNet will start with a first convolutional layer, then two downsample blocks as explained in 5.2, which will downsample the input to a $64x64x256$ feature map, then apply some residual blocks and upsample back to the original size with two upsample blocks as shown in 5.3. Lastly, we will top with a convolution that maps back the feature map to an image of 3 channels. The structure of our ResNet is shown in Table 5.5.

[5]https://tikz.net/neural_networks/

Layer	Output Shape
Input	256, 256, 3
ReflectionPadding2D, pad 3	262, 262, 3
Conv2D, 64 filters, kernel 7x7	256, 256, 64
InstanceNormalization	256, 256, 64
ReLU	256, 256, 64
Downsample block, 128 filters	128, 128, 128
Downsample block, 256 filters	64, 64, 256
Residual block	64, 64, 256
⋮	⋮
Residual block	64, 64, 256
Upsample block, 128 filters	128, 128, 128
Upsample block, 64 filters	256, 256, 64
ReflectionPadding2D, pad 3	262, 262, 64
Conv2D, 3 filters, kernel 7x7	256, 256, 3
tanh	256, 256, 3

Table 5.5: ResNet structure

5.7.3. U-Net

The second generator is a U-Net, although originally proposed for biomedical image segmentation [26], it has been used for different generative tasks outside of the medical field.

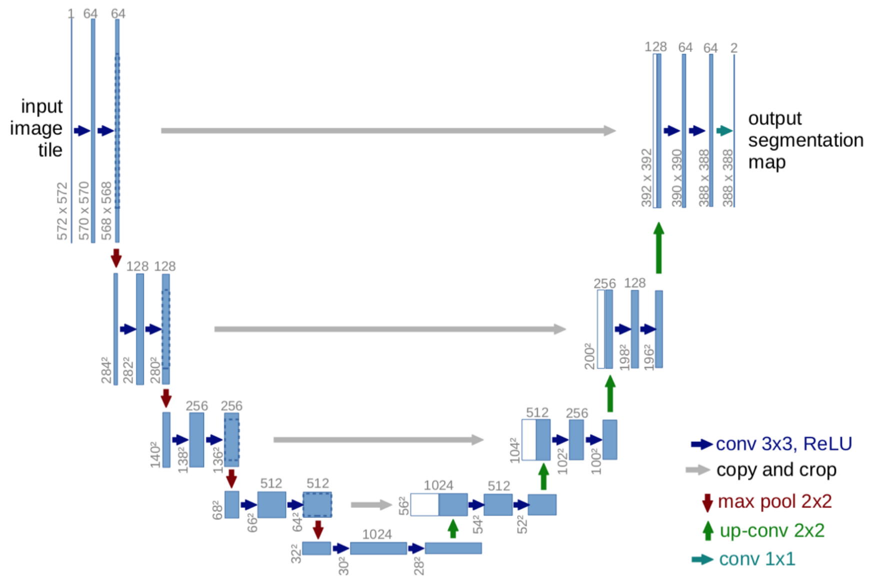


Figure 5.6: U-Net concept as shown in the original paper [26]

A characteristic building block of this architecture is the skip connection (Figure 5.6)

As the name suggests, a skip connection consists of a connection of the output of one layer to another ahead. The difference between this connection and the residual seen in the last section is that the contents of the skip connections concatenate to the outputs of a given layer. This remembers the network the values of the first layers on subsequent ones, helping in task of reconstruction or detail adding.

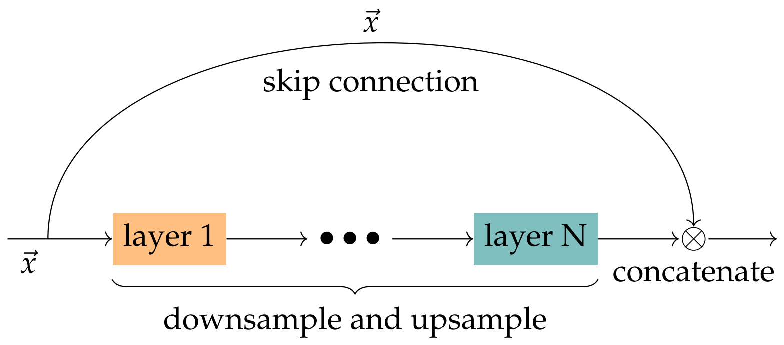


Figure 5.7: Skip connection. Adapted from [6]

The U-Net architecture works by downsampling an image a number of times and upsampling it back to the original size, but after each upsampling, we add a skip connection to help reconstruct the generated image with details of the original one. The structure of our U-Net with eight downsamples and upsamples is shown in 5.6.

Layer ID	Layer	Output Shape
[1]	Input	256, 256, 3
[2]	Downsample block, 64 filters	128, 128, 64
[3]	Downsample block, 128 filters	64, 64, 128
[4]	Downsample block, 256 filters	32, 32, 256
[5]	Downsample block, 512 filters	16, 16, 512
[6]	Downsample block, 512 filters	8, 8, 512
[7]	Downsample block, 512 filters	4, 4, 512
[8]	Downsample block, 512 filters	2, 2, 512
[9]	Downsample block, 512 filters	1, 1, 512
[10]	Upsample block, 512 filters	2, 2, 512
[11]	Concatenate([8],[10])	2, 2, 1024
[12]	Upsample block, 512 filters	4, 4, 512
[13]	Concatenate([7],[12])	4, 4, 1024
[14]	Upsample block, 512 filters	8, 8, 512
[15]	Concatenate([6],[14])	8, 8, 1024
[16]	Upsample block, 512 filters	16, 16, 512
[17]	Concatenate([5],[16])	16, 16, 1024
[18]	Upsample block, 256 filters	32, 32, 256
[19]	Concatenate([4],[18])	32, 32, 512
[20]	Upsample block, 128 filters	64, 64, 128
[21]	Concatenate([3],[20])	64, 64, 256
[22]	Upsample block, 64 filters	128, 128, 64
[23]	Concatenate([2],[22])	128, 128, 128
[24]	Conv2DTranspose, 3 filters	256, 256, 3
[25]	tanh	256, 256, 3

Table 5.6: U-Net structure with depth 8

[6]https://tikz.net/neural_networks/

5.8 Hardware

A dedicated server has been used to perform all the hyperparameter exploration tests, network training, and result inference. This server has the following specifications:

- Processor: Intel(R) Core(TM) i9-10940X, 14 cores @ 3.30GHz
- Graphics: NVIDIA RTX A6000 48 GB VRAM x2
- RAM: 128 GB

Docker containers with CUDA functionality have been deployed to perform the training and visualization processes. Thanks to the large RAM and VRAM memory capacity, it has been possible to handle the entire image dataset and network workflow, thus reducing bottlenecks in the training process.

Server provided by the Intelligent Informatics Laboratory, School of Information, Kochi University of Technology. [7]

[7] Intelligent Informatics Laboratory

CHAPTER 6
Experiments and results

In this chapter, we explain all the experimentation methods and decisions, including fixed parameters, hyperparameter tuning, and the final design of the network. In addition, we train a network with the optimized hyperparameters and show the achieved results through evaluation metrics and image examples.

First of all, some fixed configurations are as follows:

Adam is used as the optimizer with the following parameters:

$\eta = 2\text{e}{-04}$, $\beta_1 = 0.5$, $\beta_2 = 0.999$, $\epsilon = 1\text{e}{-07}$

A batch size of 16 samples is used, as it is a good balance between processing time and memory usage for these networks.

6.1 Hyperparameter tuning

Our two generators have a decisive factor to determine: the number of residual blocks in ResNet and the number of downsamples and upsamples in the U-Net. We will call these hyperparameters "depth" from now on.

Taking into account the computational load of training these models and given that the task is not that different between domains, the hyperparameter tuning has been done using the Sketch and Painting domains, assuming that the task has enough similarity across domains to be able to generalize the selection of hyperparameters across the different domains. .

For this purpose, we need to consider two key factors.

- How long does it take for the network to stabilize, based on the network loss
- Which hyperparameter values to use, taking into account the evaluation metric.

With a few experiments, we have concluded that the discriminator loss drops faster than the generator, reaching its stable minimum around the 50th batch. Figure 6.1

On the other hand, the generators take longer to stabilize, which is around the range of 150-200 batches. Even when "stabilized", it has significantly more variability than the discriminator. Figure 6.2

Both generators have been trained with different depth levels. We will check how the FID score changes with the depth of the network.

Regarding ResNet, the FID score decreases as we sum up residual layers, as shown in Figure 6.3, but it comes at the cost of a longer training time and more memory usage.

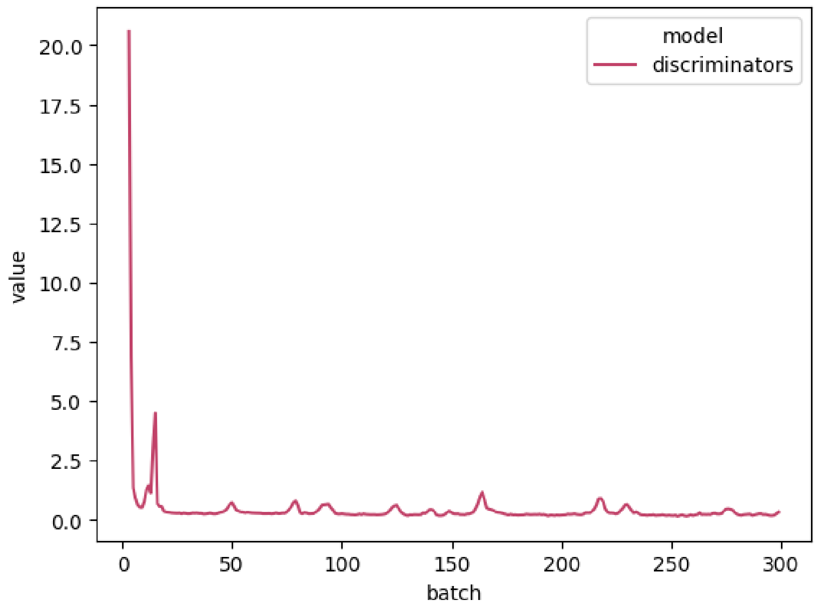


Figure 6.1: Discriminators loss

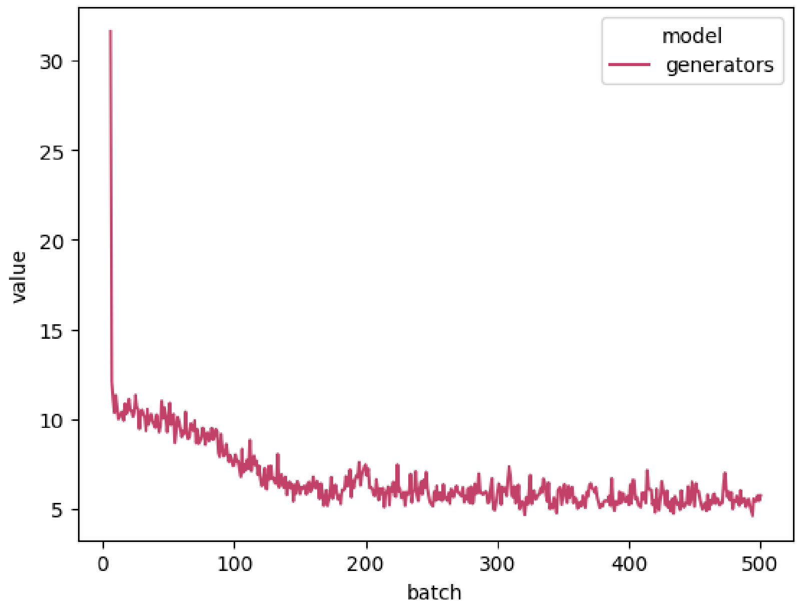


Figure 6.2: Generators loss

The gap between the scores is probably due to the intrinsic characteristics of each domain, in addition to the fact that translating to a less detailed domain is easier than doing the other way.

We have decided to use a ResNet with 9 residual blocks, as it achieves similar results with less training.

In the case of the U-Net, the best configuration is the 8-depth model, as shown in Figure 6.4. Going deeper than that for the U-Net would result impractical, since with an image size of 256x256 and 8 downsamples we already have a 1x1 size on the lowest layer. Taking that into account, we will so we will use the 8-depth version of the U-Net.

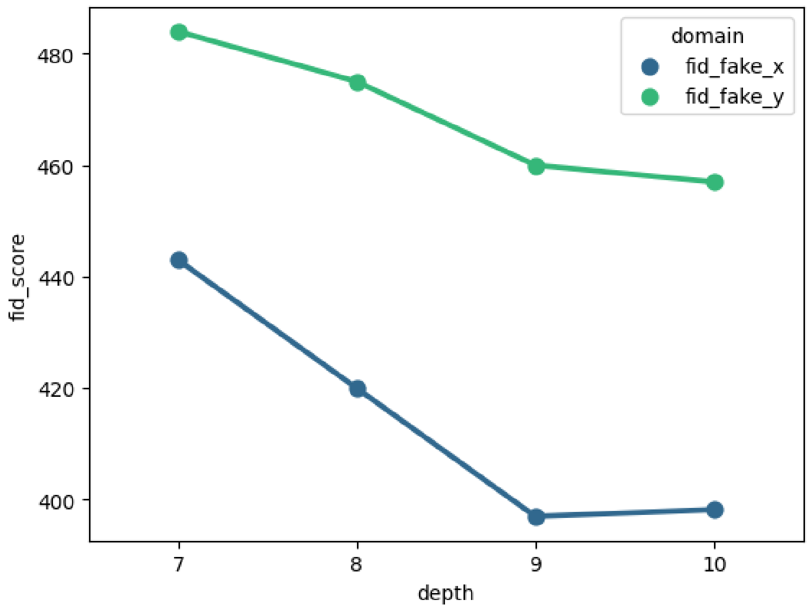


Figure 6.3: FID score for different depths. ResNet

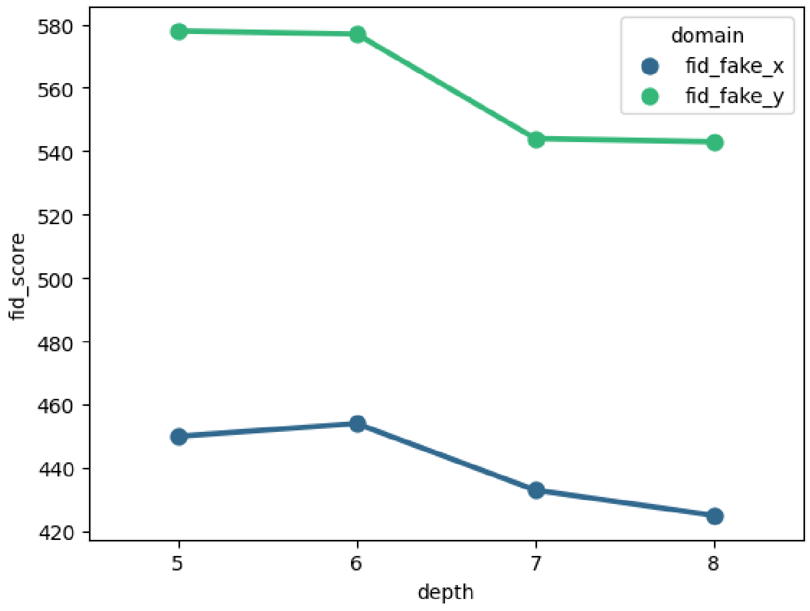


Figure 6.4: FID score for different depths. U-Net

6.2 Results

To evaluate the final architecture of our networks, we will train them with both training and validation splits and compute the FID score with the generated samples from the test split.

FID Score	quickdraw to sketch		sketch to painting		painting to real	
	quickrdraw	sketch	sketch	painting	painting	real
ResNet	445	441	397	472	415	450
U-Net	492	489	433	538	420	460

Table 6.1: FID Score for each domain pair. Each cell shows the FID score of each pair of generators

Since to our knowledge, no one has studied the image-to-image translation task applied to the DomainNet [20] dataset, we do not have any references to compare with. These numbers do not explain anything by themselves, but we can draw some conclusions from them. First, ResNet outperforms U-Net in all the tasks. Second, translations into a domain of lower quality usually have a better FID score.

6.3 Examples

With the experiments and the result finished, let us take a look at the generated images. In this section, we will show some interesting examples produced by our network.

6.3.1. Quickdraw to sketch

The model does not adapt to the combination of quickdraw and sketch. It produces images that look like they belong to the target domain, but do not retain the original content of the image. Figures 6.5 and 6.6.

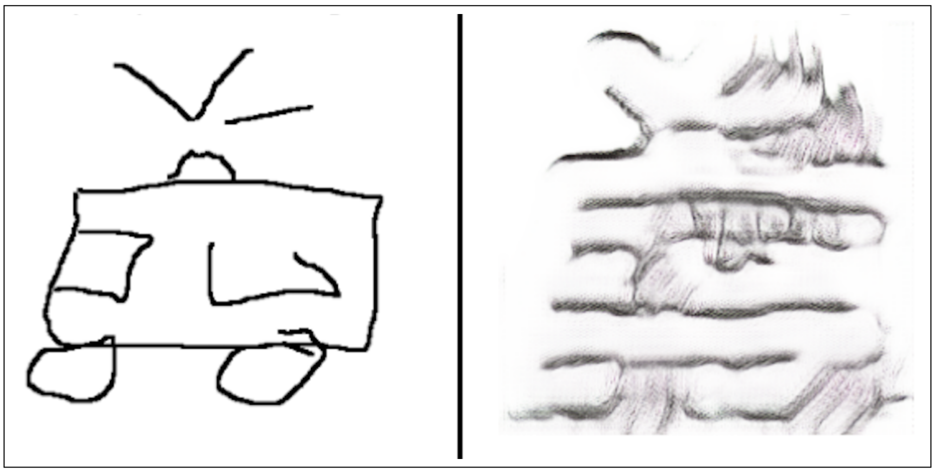


Figure 6.5: Quickdraw to sketch example

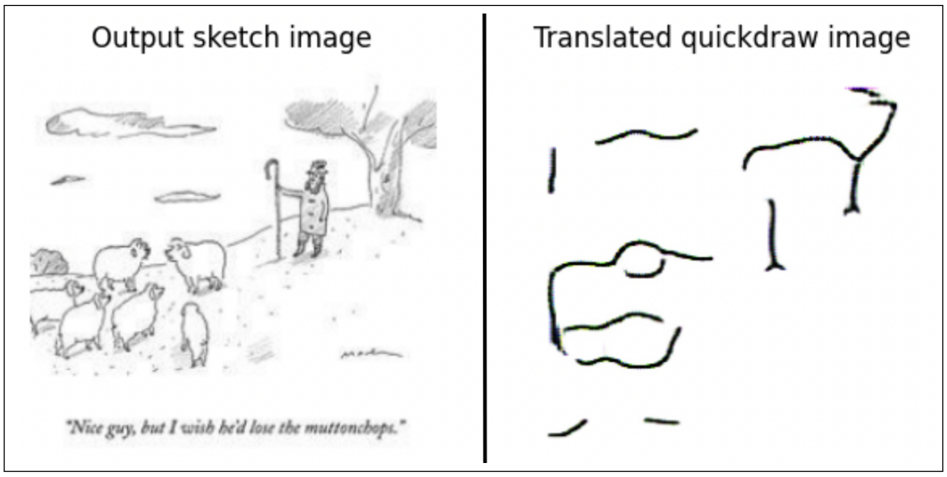

Output sketch image
Translated quickdraw image
"Nice guy, but I wish he'd lose the muttonchops."

Figure 6.6: Sketch to quickdraw example

6.3.2. Sketch to painting

The results with this domain combination are surprisingly good in some cases, as opposed to the previous case. Figures 6.7, 6.8 and 6.9.

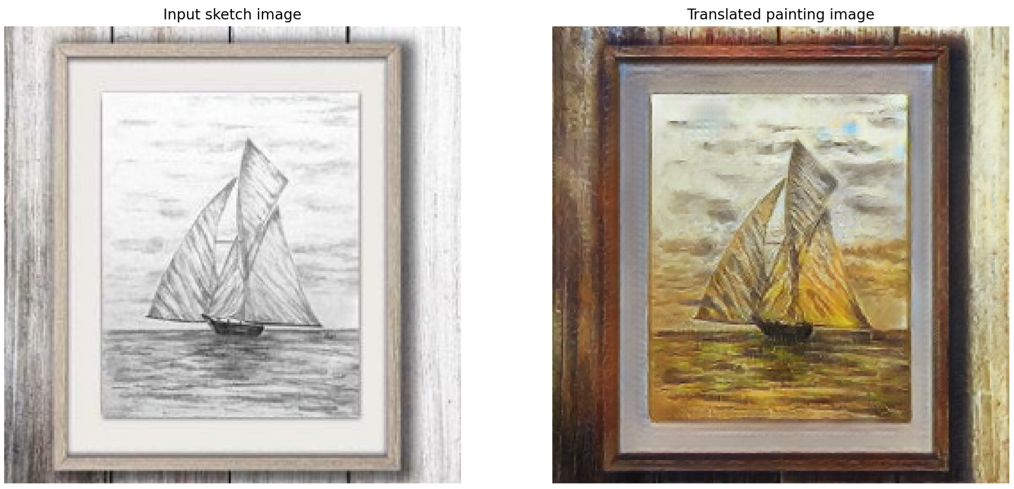


Figure 6.7: Sketch to painting example

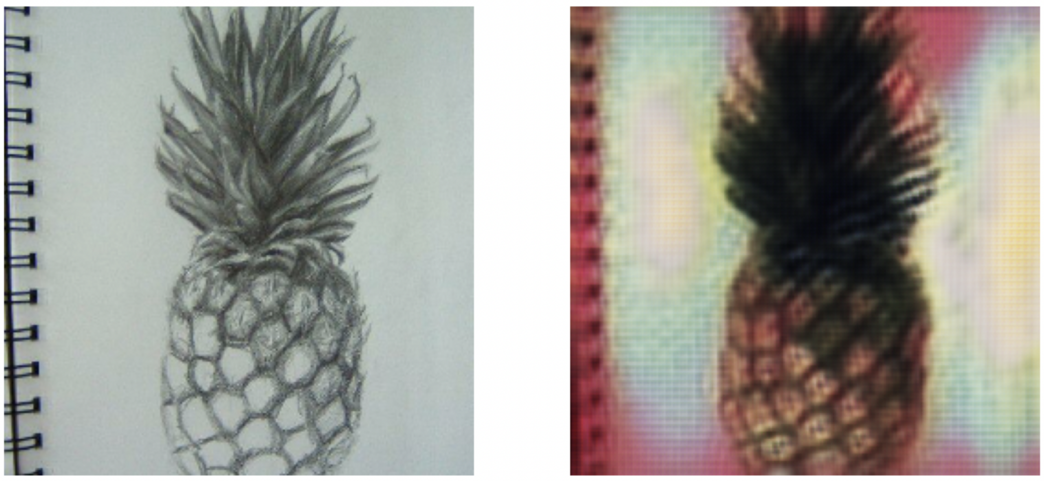


Figure 6.8: Sketch to painting. Example 2

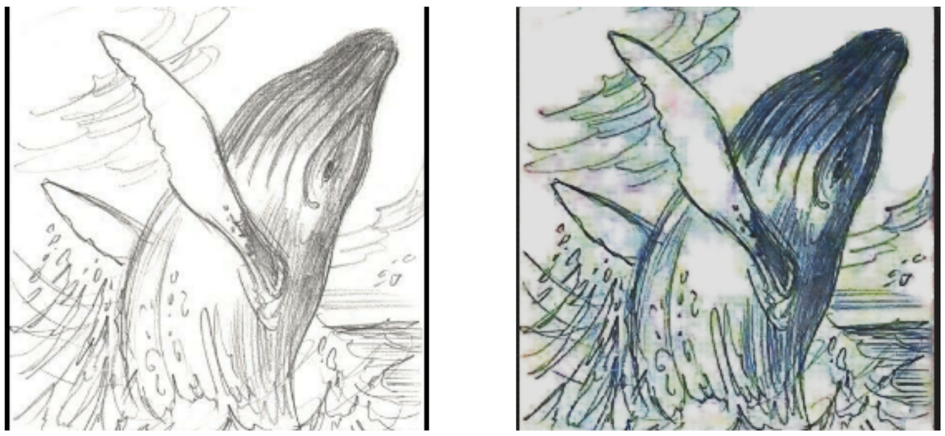


Figure 6.9: Sketch to painting. Example 3

6.3.3. Painting to real

This case deserves special attention; even though the generated images look realistic, it is difficult to discern with a naked eye which one pertains to each domain. Figures 6.10 and 6.11.

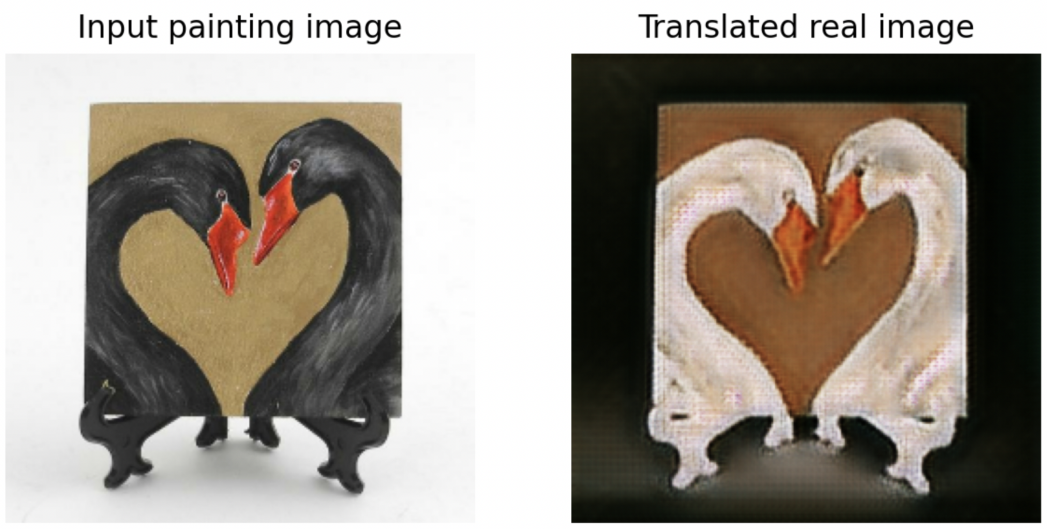


Figure 6.10: Painting to real example

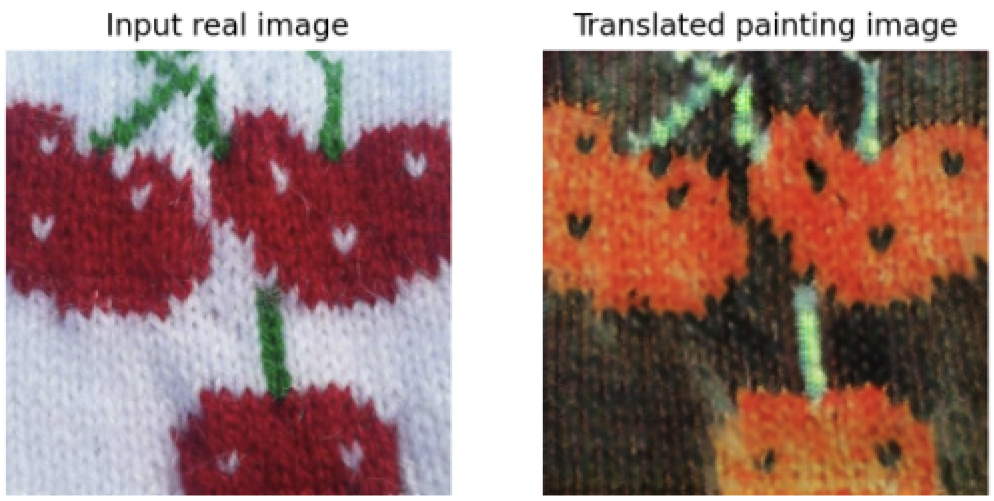


Figure 6.11: Real to painting example

6.4 Observations and comments

6.4.1. ResNet vs U-Net

Since two generator architectures have been proposed, we will compare the pros and cons of each one.

Performance

In terms of performance, ResNet is the reasonable option if you want a quality result. This could be because the ResNet approach is simply better suited for the task. As said in section 5.4, the residual blocks are learning the data that should be added to the original one. Since this task focuses on learning the translation of samples between two domains, the network can learn exactly what is needed to modify in each domain. If we take a random sample d_X from domain X and want the version of the same image translated into domain Y, represented as d_Y, the ResNet G can learn the specific modifications to apply so that:

$$d_Y = d_X + G(d_X) \tag{6.1}$$

Number of parameters

Our final ResNet model has 11,383,491 parameters, almost three times less than the U-Net, with 30,616,003. This is a good example of how a good design for a task works better than just brute force (more parameters).

Speed

Judging by the number of trainable parameters, one might say that ResNet would be faster to train, but this is not the case. Interestingly, the U-Net network is significantly faster than ResNet, around 5 times faster. For the same experiment, ResNet took around 40 hours to complete, while U-Net took only 8 hours. We suppose this is related to the sequential dependence of the ResNet (due to its greater layer depth) and highlights the importance of parallelization and block computing.

However, ResNet offers better results at the cost of being more computationally demanding.

6.4.2. Differences between domains

Quickdraw to sketch. Lack of detail of source images

As shown in the examples, the translation between quickdraw and sketch is not really good. This could be caused by the lack of detail of quickdraw images, since the network cannot discern even edges of the image and just tries to blur the image to make it look like pencil-drawn.

Paiting to real. Unrealistic color and lack of detail of resized images

As seen in the examples of the painting-to-real CycleGAN, it would seem that all the network is doing is changing colors.

There may be various reasons for this decrease.

One possible explanation is that the network discriminators are unable to distinguish the features of the two domains due to the details being too small in relation with the low image resolution (256, 256, 3). Therefore, each generator tries to change the color palette to resemble the predominant colors in the target domain, since it is its best option.

CHAPTER 7
Conclusions and future work

The image-to-image translation task has proven to be highly dependent on the domains to which it is applied.

As intended, we have succefully developed an integral system to train, evaluate and make use of an image-to-image translation model that is capable of translating images between two given domains of images without the need for paired images.

Given the modularity of the input pipeline, this project can be reused to retrain a new network applied to a custom dataset with minimal or even no changes.

Additionally, we have established a new baseline for image-to-image translation using the DomainNet dataset, with which future work could be compared and thus improve the project.

There are several unresolved questions as to whether the model will perform well with different architectures or whether there exists a better hyperparameter configuration than the one explored in this work. We have developed a solution that, although functional, still has room for improvement and expansion in order to achieve better results.

In conclusion, the development of this project has been an enriching and demanding experience and has led to the exploration of a problem in a previously unexplored dataset.

Glossary

batch The set of examples used in one step of a model training.. 1

batch size The number of examples in a batch. For example, the batch size of SGD is 1, while the batch size of a mini-batch is usually a power of 2 { 2, 4, 8, 16, 32 ... }.. 1

epoch A full pass over the entire dataset so that each sample has been seen once.. 1

hyperparameter a parameter whose value is used to control the learning process, such as the learning rate, the number of weights in a layer, or the number of layers . 1

overfitting a situation caused when a model matches the training data so closely that it fails to make correct predictions on new data. . 1

underfitting a situation caused when the model has poor predictive ability because it has not captured the complexity of the training data. . 1

Acronyms

CNN	Convolutional Neural Network. V, 1, 4, 8, 10, 28
CUDA	Compute Unified Device Architecture. 1, 21, 33
CVS	Comma-separated values. 1, 22
CycleGAN	Cycle consistent Generative Adversarial networks. 1, 17, 29
DCNN	Deep Convolutional Neural Network. 1
DNN	Deep Neural Network. 1, 10, 11, 21
FID	Frechet Inception Distance. 1, 26, 35, 37, 38
GAN	Generative Adversarial Network. 1, 4, 11, 14, 26
ML	Machine Learning. 1, 21
MLP	Multi-Layer Perceptron. 1, 6
MSE	Mean Squared Error. 1, 7, 26
NN	Neural Network. V, 1, 4, 8, 10, 23
OCR	Optical Character Recognition. 1, 4
PaaS	Platform as a Service. 1, 23
ReLU	Rectified Linear Unit. 1, 4, 26
RGB	Red Blue Green. 1
SGD	Stochastic gradient descent. 1, 8, 25, 45
VAE	Variational Autoencoder. 1, 14

Bibliography

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[2] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[3] I. J. Good, "Rational decisions," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 14, no. 1, pp. 107–114, 1952. (visited on 09/08/2022).

[4] "Mean squared error," in *Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2010, pp. 653–653.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2016.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] Y. V. R. Nagapawan, K. B. Prakash, and G. R. Kanagachidambaresan, "Convolutional neural network," in *Programming with TensorFlow: Solution for Edge Computing Applications*, K. B. Prakash and G. R. Kanagachidambaresan, Eds. Cham: Springer International Publishing, 2021, pp. 45–51.

[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[9] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015.

[10] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[11] A. Mordvintsev, C. Olah, and M. Tyka, "Inceptionism: Going deeper into neural networks," 2015.

[12] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[13] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, 2013.

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

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

[16] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

[17] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[18] T. Kluyver, B. Ragan-Kelley, F. Pérez, *et al.*, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds., IOS Press, 2016, pp. 87–90.

[19] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[20] X. Peng, Q. Bai, X. Xia, Z. Huang, K. Saenko, and B. Wang, "Moment matching for multi-source domain adaptation," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1406–1415.

[21] G. Hinton, *Neural networks for machine learning. lecture 6a. overview of mini-batch gradient descent*, 2012.

[22] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[23] A. Radford, L. Metz, and S. Chintala, *Unsupervised representation learning with deep convolutional generative adversarial networks*, 2015.

[24] U. Demir and G. Unal, "Patch-based image inpainting with generative adversarial networks," Mar. 2018.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[26] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds., Cham: Springer International Publishing, 2015, pp. 234–241.

APPENDIX A

Sustainable Development Goals

Sustainable Development Goals	**High**	**Medium**	**Low**	**N.A**
ODS 1. **No Poverty**				X
ODS 2. **Zero Hunger**				X
ODS 3. **Good Health and Well-being**				X
ODS 4. **Quality Education**				X
ODS 5. **Gender Equality**				X
ODS 6. **Clean Water and Sanitation**				X
ODS 7. **Affordable and Clean Energy**				X
ODS 8. **Decent Work and Economic Growth**	X			
ODS 9. **Industry, Innovation and Infrastructure**	X			
ODS 10. **Reduced Inequality**		X		
ODS 11. **Sustainable Cities and Communities**				X
ODS 12. **Responsible Consumption and Production**				X
ODS 13. **Climate Action**				X
ODS 14. **Life Below Water**				X
ODS 15. **Life On Land**				X
ODS 16. **Peace, Justice, and Strong Institutions**				X
ODS 17. **Partnerships for the Goals**				X

Explanation of the SDGs chosen

Since 2015, sustainable development goals have become of great importance. This project could contribute to some of them due to its close relationship with current technological advances.

This project is related with three SDGs: "Industry, Innovation and Infrastructure", "Decent Work and Economic Growth" and "Reduced Inequality".

- **Industry, Innovation and Infrastructure**: The first goal is closely related to the project insofar as it contributes to the field of machine learning, one of the most innovative fields today. The development of AI-based solutions greatly contributes to the emergence of innovative ideas. Additionally, these applications have a wide range of applications in a wide variety of industries.

- **Decent Work and Economic Growth**: In 2019, 14% of the world's workforce was employed in manufacturing. With the rise of the tech sector, this percentage is expected to fall even further, as we move to more skilled jobs. Thus, the research on this field contributes to the emergence of higher quality and therefore better paid jobs.

- **Reduced Inequality**: Lastly, the reduction of inequality is a consequence of the economic impact of the application of artificial intelligence in the economy, as anyone with an internet connection has access to this type of services and knowledge. In an increasingly globalized world thanks to technology and AI, the opportunities that arise affect everyone equally.