Capabilities and limitations of style transfer with CycleGANs for ring design automatic generation

Tomas Cabezon Pedroso
Master's in Computational Engineering and Mathematics

Natalia Díaz Rodríguez
Carles Ventura Royo

INDEX CARD OF THE FINAL MASTER PROJECT

Title of the FMP:	Capabilities and limitations of style transfer with CycleGANs for ring design automatic generation
Name of the author:	Tomas Cabezon Pedroso
Name of the tutor:	Natalia Díaz Rodríguez
Name of the PRA:	Carles Ventura Royo
Date of delivery (mm/yyyy):	06/2021
Degree:	Master's in Computational Engineering and Mathematics
Area of the Final Work:	Artificial Intelligence
Language of the Work:	English
Keywords:	GAN, style, design

Summary of the Work:

Rendering programs have changed the design process completely as they permit to see how the products will look like before they are fabricated. However, the rendering process is complicated and takes a lot of time not only in the rendering itself but in the setting of the scene as well. Materials, lights and cameras need to be set in order to get the best quality results, nevertheless, the optimal output may not be obtained in the first render. This all makes the rendering process a tedious process.

Since Zhu et al. introduced Generative Adversarial Networks (GANs) in 2014, they have been used to obtain computer-generated data. From non-existing human faces to medical data analysis or image style transfer. GANs have been used to transfer image textures from one domain to another, but paired data was needed. When this same group introduced the CycleGANs, this all changed. CycleGANs allow transforming one image from one domain to another, without the need of paired data.

This Work studies the possibilities of CycleGANs on style transfer from an initial sketch to a final render. A process that is crucial in the automatic generation of ring designs as allows the costumer to see the final products before buying.

The present Work sets a basis for future research, showing the possibilities of GANs in design and establishing a starting point for new applications.

Resumen:

Los programas de renderizado han cambiado el proceso de diseño por completo, ya que permiten ver cómo se verán los productos antes de ser fabricados. Sin embargo, el proceso de renderizado es complicado y lleva mucho tiempo, no solo en el renderizado en sí, sino también en la configuración de la escena. Es necesario configurar los materiales, las luces y las cámaras para obtener los mejores resultados de calidad; sin embargo, el resultado óptimo no suele conseguirse en el primer renderizado. Todo esto hace que el proceso de renderizado sea un proceso tedioso.

Desde que Zhu et al. introdujeron las Redes Generativas Antagónicas (RGAs) en 2014, estas se han utilizado para obtener datos generados por computadora. Desde rostros humanos inexistentes, hasta el análisis de datos médicos o transferencias de estilos de imágenes. Las RGAs se han utilizado para transferir texturas de imágenes de un dominio a otro, pero se necesitaban datos emparejados. Cuando este mismo grupo introdujo las CycleGAN, todo esto cambió. Las CycleGANs permiten transformar una imagen de un dominio a otro, sin la necesidad de datos emparejados.

Este Trabajo estudia las posibilidades de las CycleGANs en la transferencia de estilo desde un boceto inicial a un render final. Un proceso que es crucial en la generación automática de modelos de anillos ya que permite al cliente ver los productos finales antes de comprarlos.

El presente Trabajo sienta las bases para futuras investigaciones, mostrando las posibilidades de las RGAs en el diseño y estableciendo un punto de partida para nuevas aplicaciones.

First and foremost, I would like to express my sincere gratitude to my supervisor Natalia Díaz Rodríguez for the support on my Final Master Thesis. Her guidance has not only helped me complete this work but has also motivated me to give my best during this difficult course. I would also like to thank her for having encouraged and allowed me to this work on the intersection of computation and design.

Secondly, I would like to thank my sister and friends who have helped me during this course, if it wasn't for them, I would not be writing these lines now.

Contents

1 INTRODUCTION ... 13

1.1 CONTEXT AND JUSTIFICATION OF THE WORK ... 13
1.2 AIMS OF THE WORK ... 15
1.3 APPROACH AND METHOD FOLLOWED ... 15
1.4 PLANNING OF THE WORK ... 15
1.5 BRIEF SUMMARY OF THE OBJECTIVES ... 17
1.6 BRIEF DESCRIPTION OF THE OTHER CHAPTERS OF THE MEMORY ... 18

2 PERSONAL REFLECTIONS ... 19

3 STATE OF ART ... 22

3.1 PRELIMINARY NOTIONS ... 23
3.1.1 Autoencoders ... 23
3.1.2 Latent space ... 24
3.2 GENERATIVE ADVERSARIAL NETWORKS ... 26
3.2.1 Generator and discriminator ... 27
3.2.2 BCE Cost Function ... 29
3.2.3 Activation functions ... 30
3.2.4 Convolutions ... 33
3.3 TYPES OF GAN ... 35
3.3.1 Conditional GAN ... 35
3.3.2 Pix2pix ... 36
3.4 CYCLEGAN ... 37
3.4.1 Cycle consistency loss ... 37
3.4.2 Adversarial loss ... 38
3.4.3 Identity loss ... 39
3.4.4 Generator total loss ... 39
3.4.5 Generator architecture ... 41
3.4.6 Discriminator architecture ... 42

4 PRACTICAL CASE DEVELOPMENT ... 43

4.1 CASE DESCRIPTION ... 43
4.1.1 XYU ring ... 43
4.1.2 Actual model state ... 45
4.1.3 New model proposal description ... 48
4.2 CYCLEGAN DEVELOPMENT ... 49
4.2.1 CycleGAN description ... 49
4.2.2 Building the CycleGAN ... 50
4.2.3 Building the network ... 52
4.2.4 Training the CycleGAN ... 55

5 RESULTS ... 59

5.1.1 Sketch2object3D ... 59
5.1.2 Sketch2rendering ... 60

6 DISCUSSION AND FUTURE WORK 64

6.1 DETECTED ARTIFACTS 64

6.1.1 White spots 64

6.1.2 Continuity loss in the lines in the sketch transformation 65

6.1.3 Aureole around the ring 65

6.1.4 Checkerboard pattern 66

6.2 MODEL LIMITATIONS 67

6.2.1 Lighting settings 67

6.2.2 2D perspective 68

7 CONCLUSIONS 69

8 GLOSSARY OF TERMS 71

9 BIBLIOGRAPHY 72

10 ATTACHMENTS 75

10.1 DATABASES 75

10.1.1 Sketch dataset (first version, thin wire sketch) 75

10.1.2 Sketch dataset (second version, thick wire sketch) 76

10.1.3 Object3D dataset (first version, white background) 77

10.1.4 Object3D dataset (second version, blue background) 78

10.1.5 Rendering dataset 79

10.2 CODE 80

List of figures

Figure 1: Gantt diagram for project structure 16
Figure 2: Personal project prototypes. 20
Figure 3: Simple autoencoder architecture. [17] 23
Figure 4: s-SNE projection of the latent space of the MNIST dataset of handwritten digits. [19] 24
Figure 5: The two GAN subnetworks, their inputs and outputs, and their interactions. 27
Figure 6: Schema of the generator. 28
Figure 7: Schema of the generator's parameters update, training of the generator. 28
Figure 8: Schema of discriminator's parameters update, training of the generator. 29
Figure 9: Schema of activation functions. 31
Figure 10: ReLu activation function [26] 31
Figure 11: Leaky ReLu activation function [26]. 32
Figure 12: Sigmoid activation function [26]. 33
Figure 13: tanh activation function [26] 33
Figure 14: Convolution operation on a 7x7 matrix with a 3x3 kernel [27]. 34
Figure 15: Spatial hierarchy of visual modules that combine into the 'cat' concept [16]. 35
Figure 16: Image-to-Image Translation with Conditional Adversarial Networks [10]. 36
Figure 17: Schema of the Cycle consistency loss. 38
Figure 18: Output image comparation with and without Identity loss [33]. 39
Figure 19: Loss function plot for the first 1400th steps of the training, corresponding to 14 epochs 40
Figure 20: Architecture of the generator [33]. Example of the *orange2apple* CycleGAN. 41
Figure 21: Example the classification of a portion of the image in the PatchGAN architecture [36]. 42
Figure 22: Original concept schema 43
Figure 23: Mood board (visualization of concepts and ideas) of the XYU ring and rendered images of different XYU rings created. 44
Figure 24: Actual model's program and steps followed. On top Matlab program screenshot where the algorithm is run. In the middle, Maya program with the 3D object of the ring. On the bottom, blender program and the rendered image. 45

Figure 25: Schema of how each of the brands of the ring are 3D modelled on Maya program. The points that compose the curve as well as the circle that will be extruded along this curve are passed to Maya program using Maya MEL coding language. 46
Figure 26: Rendering setting and image rendering times for 1000x1000 pixel size simple images....................47
Figure 27: Actual model schema: the algorithm run in Matlab, the 3D object modelled in Maya, the rendered image in Blender and the final product after being made by the jewel. 48
Figure 28: Proposed model idea. The algorithm in this case, apart from creating the ring is also responsible of generating the rendered images of it. A CycleGAN is added to the algorithm for this last part....................48
Figure 29: Proposed CycleGANs, *sketch2object3D* and sketch2rendering. 49
Figure 30: CycleGAN architecture schema. 54
Figure 31: Result of the CycleGAN *sketch2object3D* (epoch 4, step 420). On top, the input images and bellow the images in the other domain generated by the CycleGAN. 55
Figure 32: Influence of the background color on the images of the first 60 steps of the training of the CycleGAN with white and blue backgrounds. 56
Figure 33: Training process steps 420 and 2800 with both background colors. The input images are followed by the images generated by the Cyclegan with the domain changed. 57
Figure 34: Continuity problems found during sketch2rendering CycleGAN training with the thin lines sketch....................58
Figure 35: Collection of some results of the *sketch2object3D* CycleGAN. 59
Figure 36: Collection of some results of the first trained *sketch2rendering*. 60
Figure 37: Collection of some results of the second trained *sketch2rendering*. 60
Figure 38: Different rings in the sketch domain and the rendered domain. The expected rendering has been generated with Blender and the other using the *sketch2rendering* CycleGAN....................61
Figure 39: Different views of the same object in the sketch domain and the rendered domain. The expected rendering has been generated with Blender and the other has been generated by the *sketch2rendering* CycleGAN. 62
Figure 40: Different views of the same object in the sketch domain and the rendered domain. The expected rendering has been generated with Blender and the other has been generated by the *sketch2rendering* CycleGAN. 63
Figure 41: Examples of the white spots artifact....................65
Figure 42: Examples of the continuity loss in the sketch domain transformation..65
Figure 43: Examples of the aureole artifact. 66
Figure 44: Examples of the checkerboard artifact. 66
Figure 45: Examples of the influence of the lightning settings on the final render. 67

Figure 46: Example of two lines intersecting in the 2D image, and two different 3D examples of the actually intersecting and non intersecting cases. 68
Figure 47: Random images of the thin wire sketch dataset. 75
Figure 48: Random images of the thick wire sketch dataset. 76
Figure 49: Random images of the white background model dataset. 77
Figure 50: Random images of the blue color background model dataset. 78
Figure 51: Random images from the rendering dataset. 79

¨Turing believes machines think
Turing lies with men
Therefore machines do not think¨

Letter from Alan Turing to Norman Routledge,
February 1952 [1]

1 Introduction

1.1 Context and justification of the Work

Despite of his short life, Alan Turing was one of the most influential scientists o the 20th Century. He died at the age of 41 in 1594 after being prosecuted for homosexual acts. However, he is considered to be the father of computer science and artificial intelligence. Since then, computation has experienced a surge and now it is applied to ease our everyday life but also to radically change it. From the way we interact with each other, to the way we go shopping, or the way we work or leisure. Computation is here to stay.

In this project I want to see the possibilities and capabilities of computation in the field of design. If a machine can only do what it is programmed to do, how can it be creative? Not all computer theorists agree on this [2], and actually, the examples of the intersection of design and computation are growing and are more relevant than ever. In the last years, engineers, researchers or artists have begun to explore the possibilities of artificial intelligence for creative tasks that can vary from the AI generated music of Arca that sounds in the MOMA´s lobby [3] to the drawings by AARON computer program that can be visited at TATE Museum [4].

This Work rises in this same intersection design and technology, design and engineering, design and computation. The aim of this Work is to explore new areas and applications in which computers will change the way we consider design and the role that computers have on it. When this statement is made, often the fear of computers stealing people's jobs arises, nevertheless, this is not how this intersection of computers and design is conceived by the author. While algorithms will spend time doing repetitive work, designers will be able to focus on what really matters: the users, innovations, needs... This Work's objective is to make the most of the agents taking part in the process of design, the computer and the designer. The tools at our disposal cannot determine what we are capable of creating, the tools should serve for new ideas and not to limit the ones that the designer already has.

This work is organized in two parts, a theoretical one and a practical one, both complementary. The first one is motivated by the recent arrival of Generative Adversarial Networks (GAN) that since they were first introduced few years ago, in 2014 [5], have experimented and exponential growth and development. This research will be focus on the study and comprehension of the theory that supports GANs and their components. In the second part of the Work, taking into account all of the above, a new tool of image generation is applied to an actual design problem, in this case, the rendering of an example of the XYU ring (finger, jewelry area) [1]. This tool will consist on a CycleGAN that taking as an input the sketch of the shape of the ring will generate a 3D object representation or a rendered image of it.

The steps of the process of design can differ among authors, however, all of this process consist on going form a virtual concept or idea to the materialization in a concrete product [6]. This process stars with an initial brainstorming and later some of the concepts are developed, prototyped and after evaluation the final product is selected. Computers have become fundamental in these last steps allowing designers not only materialize their ideas with 3D objects and renders but also show the clients how the final products look like. Actually, the famous furniture seller Ikea reaches their clients with the yearly catalogs, full of not real images but renders [7]. This Work aims to input new tools for this last part of the design process

[1] The XYU ring is key to understand this Work and this project will lately be introduced. More information about this project previously developed by the auhor can be found on https://tomascabezon.com/

1.2 Aims of the Work

There are two main aims for this work. The first one is the state of the art of GAN image generation and how it can be applied for design purposes. The second one, applying this technology on a concrete example, as it can be the rendering of a sketch of the XYU ring example. To approach this issue, how GANs create images will be needed to be understood.

1.3 Approach and method followed

In this Work, GANs will be used to generate realistic images to see the scope of this technology in design. Although realistic images could be generated by other means, such as rendering, mocking up or photographing, in this work, the objective are not the output images themselves, but seeing the possibilities and limitations of GANs.

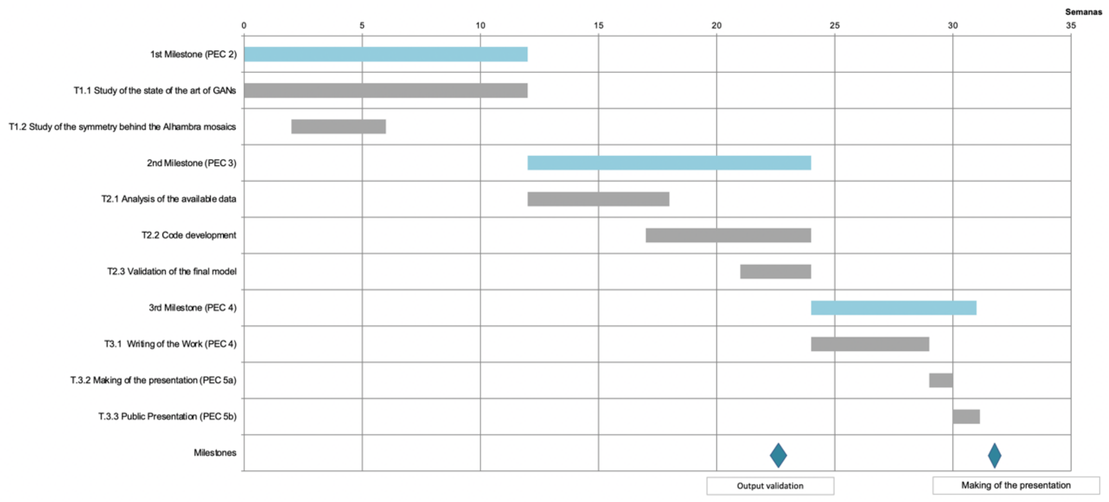
1.4 Planning of the Work

The project will follow the following scheme that has been divided taking into account the continuous evaluation PECs:

1st Milestone PEC 1 – PEC 2
For this milestone, what to do, how and the reach has been planned. Investigation of the state of the art in deep learning and GAN technology has been carried out. Familiarizing with this new field is crucial to develop a practical project in the following milestones.

2nd Milestone PEC 3
In this second phase, what has been learnt in the first phase is put into practice. The project transitions from a theoretical approach to the practical one, the code starts to be written and the images generated.

3rd Milestone PEC 4
The main output of this milestone is the finished documentation. After this milestone, the remaining work to do is to write and record the video presentation.



Phases	Start	Duration	End
1st Milestone (PEC 2)	**14/11/2020**	**12**	**05/02/2021**
T1.1 Study of the state of the art of GANs	14/11/2020	**12**	**05/02/2021**
T1.2 Study of the symmetry behind the Alhambra mosaics	06/02/2021	**4**	**05/03/2021**
2nd Milestone (PEC 3)	**06/02/2021**	**12**	**30/04/2021**
T2.1 Analysis of the available data	06/02/2021	**6**	**19/03/2021**
T2.2 Code development	20/03/2021	**7**	**07/05/2021**
T2.3 Validation of the final model	08/05/2021	**3**	**28/05/2021**
3rd Milestone (PEC 4)	**01/05/2021**	**7**	**19/06/2021**
T3.1 Writing of the Work (PEC 4)	01/05/2021	**5**	**04/06/2021**
T.3.2 Making of the presentation (PEC 5a)	05/06/2021	**1**	**11/06/2021**
T.3.3 Public Presentation (PEC 5b)	14/06/2021	**1**	**20/06/2021**

Figure 1: Gantt diagram for project structure

1.5 Brief summary of the objectives

In the following lines the general objectives (GE) as well as the specific objectives will be described.

GO 1. Bibliographic review of the different available Artificial Intelligent techniques that will be used for this Work, more concretely, the study of the state of the art of GANs. To achieve the objective, the following specific objectives need to be met:

EO 1.1. Obtain information from different sources and research articles.
EO 1.2. Define the different components that form a GAN.
EO 1.3. Establish the algorithms that best adapt to the requirements of the project.

GO 2. Obtaining the models. To achieve this general objective, the following specific objectives need to be met:

EO 2.1. Code the GAN.
EO 2.2. Create the databases to train the GANs.
EO 2.3. Train the GANs and compare the result obtained.

GO 3. Presenting the results obtained. To achieve this general objective, the specific objectives need to be met:

EO 3.1. Write the final report.
EO 3.2. Elaborate the presentation.
EO 3.3 Public defense of the master's Thesis.

1.6 Brief description of the other chapters of the memory

The Work begins with some personal reflections, that although not being necessary to understand it, they serve to illuminate the perspective from which it has been developed.

In the second chapter, a state of the art of the GANs can be found. The purpose of this section is to understand and develop the components necessary to perform the later implementation of a GAN in the practical case. GANs are nowadays a tools used in a wide range of area that go from dataset generation [5], realistic image generation [8], text to image translation [9] or image translation [10]. Understanding how these tools are used is key to not make the most on them. One of the intentions of the Work is to help the designers not to depend on the tools but to make the tools available to them.

Finally, in the fourth chapter, the XYU ring concept is presented, an algorithm designed to randomly generate different ring examples. The initial concept is explained, and the actual state of the project is presented. The process that is carried out every time that a XYU ring example is created is shown to understand the limitations that are found on the generation of realistic images, renderings, of the final products.

Afterwards, the new proposal is explained, in which the implementation of the CycleGAN not only simplifies the process but allows its automatization. This new tool restructures the process in a way in which no interaction of the designer is needed. This proposal permits to design an algorithm in which the interaction of the user and the machine ends up in a completely personalized and unique ring examples. The tools used for its development are explained as well as the problems encountered during the elaboration of this Work. Although the ultimate goal has been the design of this completely automated model, issues of great relevance are addressed along the way. The possibilities of algorithms in design, of the automation of certain creative tasks and the personalization of products by the user.

2 Personal reflections

Prior to doing this Master's in Computational Engineering and Mathematics, I studied a Double Degree in Industrial Design Engineering and Mechanical Engineering. During those university years one learns many things. But above all, one realizes which things he should have learnt but he hasn't. Or even better, one realizes those things that he did not know existed, but about which he wanted to learn more. This project is mainly the result of the latter, but also a bit of the former.

This Final Master Project (FMP) is in its the most part the sum of two disciplines that not only enrich oneself as an engineer but also as a person. The first one, Design Engineering, which is essential to understand what surrounds us. The second, computing, because digitization is an increasingly present and powerful reality. Digitization together with automatization, allows that machine are dedicated to everything that we do not like to do so that we can dedicate ourselves to what we are really passionate about. In my case, I would have loved an artificial intelligence to be so advanced that it could have helped me write these lines. If the Instagram algorithm already knows me so well that it suggests better posts than the ones I follow, why won't one day a machine write this FMP better than I do with the information it had about me.

The reason for this FMP is also the sum of two events mainly. The first, my first design course during my bachelor's degree, in the subject of Workshop I. In that course we were asked to redesign a travel brush. What I learned from that design is the importance of the tools available to the designer. In that particular case, the influence of the CAD (Computer Aided Design) programs that I knew how to use.

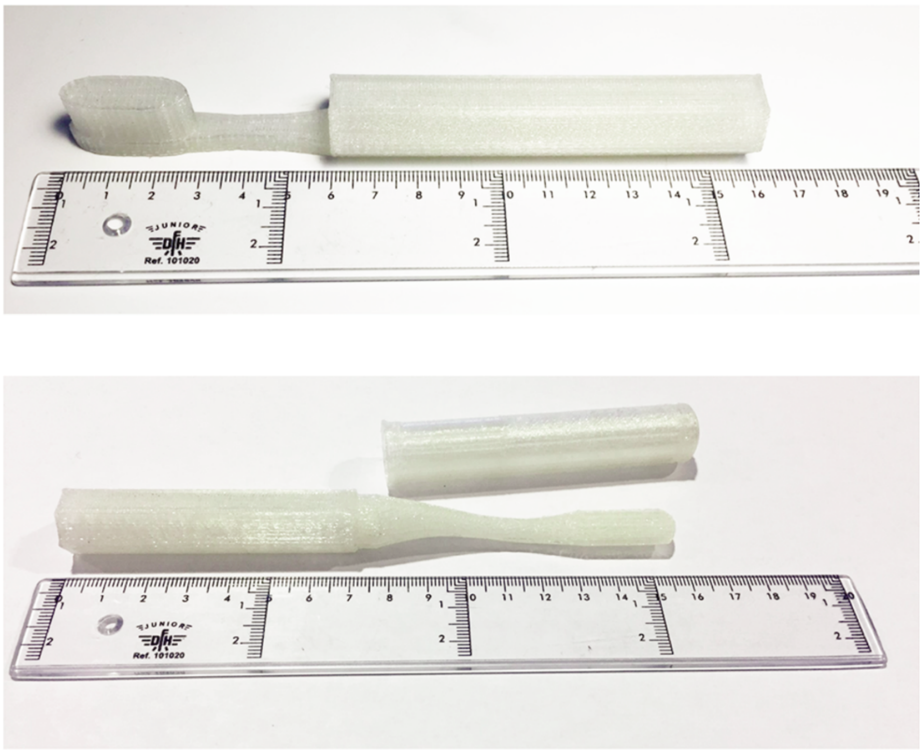


Figure 2: Personal project prototypes.
Evolution of a project with the same idea but using different modelling tools such as AutoCAD or SolidWorks.

The influence of the tools the designer has available is very clear in his work, even limitating, as the designer is not able to materialize his ideas as a consequence of not having the necessary tools to do it. Therefore, this project tries to understand how some artificial intelligence (AI), more concretely GANs, work so we do not depend on them, but we make these tools available for the designer. Tools should serve for new ideas and not to limit the ones that one already has.

The second reason for this FMP is my exchange study in Tongji University in Shanghai, China. Before that exchange, I thought I was going to go to a less developed country than mine, nevertheless, I couldn't be more wrong. My experience in China was a trip to the future, which made me think a lot about which would be the next technological advances, which changes would they bring in our ways of life, in our jobs. I remember the day one of my professors, Dean Lou Yongji, explained how the advertising banners that appeared in the Taobao[2] application were completely personalized for each of the buyers and that these banners were not designed by people, but by algorithms[11]. This made me think about the graphic designers who had lost their jobs or about how AI could affect me as an industrial designer. But then I understood the true power of AI. While algorithms do the repetitive work, designers would be able to focus on what is really important, the people and their needs.

In this project I wanted to delve into the field of AI in design and understand how AI can be introduced for designing products. Explore the customization possibilities offered by the automatization of the design process. Instead of designing an object, designing an algorithm that can generate different objects, different designs.

[2] **Taobao:** A online shopping website owned by the Alibaba group. Its division tmall.com is the third most visited website in the world.

3 State of Art

The aim of this chapter is to serve as a theoretical base of how GANs work as well as explain the different components of them so this tool for image generation can later be put into practice in the following part of this Work.

Machine learning is a discipline in the field of Artificial Intelligence that serves to recognize patterns in existing massive data and use this information to make predictions, these can be

- *Classification,* assigning the correct label or category to an example or group of found features.

- *Regression,* use the know data as an input to predict or estimate a numerical values.

Computers can classify bank customers according to their probability of non-payment [12], win a chess champion [13] or predict which Instagram post we are likely to like [14]. However, when asked to generate new data, computers have historically struggled.

When in 2014, Ian Goodfellow, then a PhD student at the University of Montreal, introduced GANs, this changed [15]. Using not one but two separate and opposite neural networks, this new tool could be used by computers to create new data. This technique is not specific to generate images, actually, other data types such as sound, music or even text can be generated. In this Work, though, we will focus on the image generation as GANs will later be used with this purpose.

3.1 Preliminary notions

3.1.1 Autoencoders

Image generation with deep learning is done by learning the latent spaces that capture the statistical information about the data we are training on, in our case, a dataset of images. Each of the pictures will be a point in the latent space and sampling and decoding these points in the latent space, new data can be created. Two main tools are used for this task: Variational Autoencoders (VAEs) and GANs [16].

Understanding the first one, is a good starting point to understand how GANs work. Autoencoders, as their name suggests, automatically encode data. Autoencoders are a family of neural networks for which the input is the same as the output, as we can see in the following figure, although altered versions can be achieved.

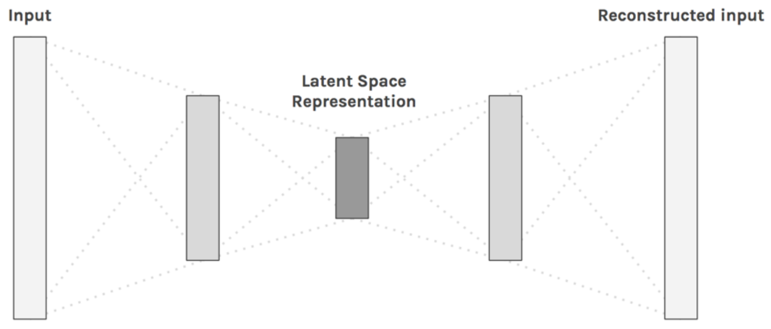


Figure 3: Simple autoencoder architecture. [17]

A simple image autoencoder would take an image, and would map it into the latent space, encode it; and then decompress it with the same size as the input, decode it. To do so, they are trained having the same image as input and output. Therefore, we can think of autoencoders as a tool to compress the input data into fewer bits of information.

To train the autoencoder the input image (*x*) is passed to it and the output image (x^*) is compared whit it to see if there is any change. The differences between the input and the output are measured to calculate the error which is called the reconstruction loss ($||x-x^*||$).

3.1.2 Latent space

As it has been seen in the previous section the encoder of the autoencoder brings the data from a higher dimension space into a bottleneck layer reducing its dimension. The space in which this lower dimension data lies is the latent space. The latent space is the hidden representation of the compressed data and contains all the important information needed to represent the original image, this is, contains the principal features of the data. When the autoencoder is trained on the images, the model learns to represent the features of the original data and simplifies its representation [18].

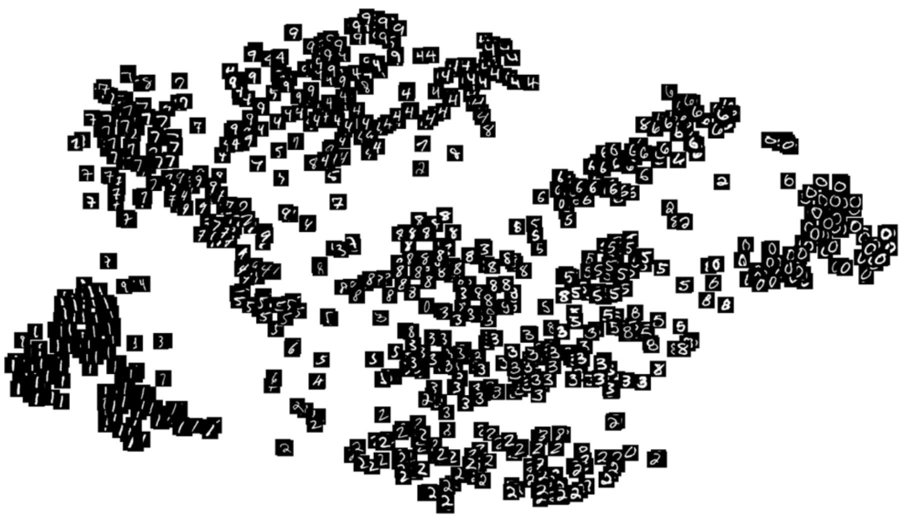


Figure 4: s-SNE projection of the latent space of the MNIST dataset of handwritten digits. [19]

Autoencoders use one only loss function, the reconstruction loss to train the network, however, as it has been previously mentioned, the objective of the GANs cannot be written in one function, two distinct functions are needed to train the model and represent the competing objectives of the generator and the discriminator. Formally, the generator and discriminator are represented by to different neural networks, each with its own objective, each own cost function.

As we saw in figure 3, it is the output of the discriminator, the discriminator's estimation of whether the picture is fake or real, what is used to train the model. If the generator manages to fool the discriminator, which means that the discriminator estimates the image is real, will be useful for the generator as it will know how to keep improving. On the other side, the discriminator, will have to learn to and improve so next time is not fooled again. Both of the networks are trained by backpropagation of the discriminator's loss.

Traditional neural networks, as it is the case of VAEs, are defined exclusively in terms of their own trainable parameter, the reconstruction loss, however, GANs differ from conventional neural networks in having two different networks whose cost functions are dependent in both of the networks' parameters. During training, each of the networks is able to change only its own weights and biases, so each of the networks has control over only a part of what determines its loss. Training a traditional neural network is optimizing its cost function, on the contrary, training a GAN is more like a competition in which the training will stop when neither of the opponents can beat the other. The GAN will ideally be trained when the discriminator cannot differentiate the real images (x) coming from the dataset and fake images (x^*) from the generator. The discriminator will only be able to make a guess having a probability of 50% to be right. In this point, the generator won't be able to improve, as its output is already indistinguishable, so no improvements are possible and any change on the way the fakes are generated my give a clue to the discriminator to know which is the fake image. This equilibrium achieved is called the Nash equilibrium. Nevertheless, in practice, it is nearly impossible to reach this point and still remains as an open question in GAN research [20].

3.2 Generative Adversarial Networks

Generative adversarial networks (GANs) generate realistic looking synthetic images by forcing the generated images to be statistically almost indistinguishable from real ones. They consist of two neural networks simultaneously trained, the generator, trained to create fake data based on a dataset; and the discriminator, trained to distinguish fake images from real ones. As their name indicates, this is a generative tool, this is, it is capable of production or reproduction. This ability to generate new data will depend on how the GAN is trained, in the case of this Work, it will be trained on a XYU ring example rings, so it will learn to reproduce other examples of this ring.

The work adversarial refers to the competitive dynamic between the generator and the discriminator, the two models that compose the GAN. Finally, the term network refers to the kind of machine learning technique that it is compose of, both the discriminator and the generator, are neural networks. The complexity of this networks depends on the complexity of the implementation. They can go from simple feed-forward neural network to a more complex convolutional network to even a U-Net as the one in our practical case.

An intuitive way to understand how GANs work, is using the counterfeiters and police metaphor, the one Ian Goodfellow himself used to explain this new tool [5]. The criminals (the generator) make fake money while police (the discriminator) tries to discern between counterfeit money and real money. This competition leads to more realistic money generated by the criminals until perfect fake money that fools the police is created. In more technical terms, the generator tries to generate images with the same characteristics of those in the training data, so they look identical to those in the dataset. On the other hand, the discriminators' goal is to determine if a particular example is real, this is, it comes from the dataset or if it is fake, it has been created by the generator.

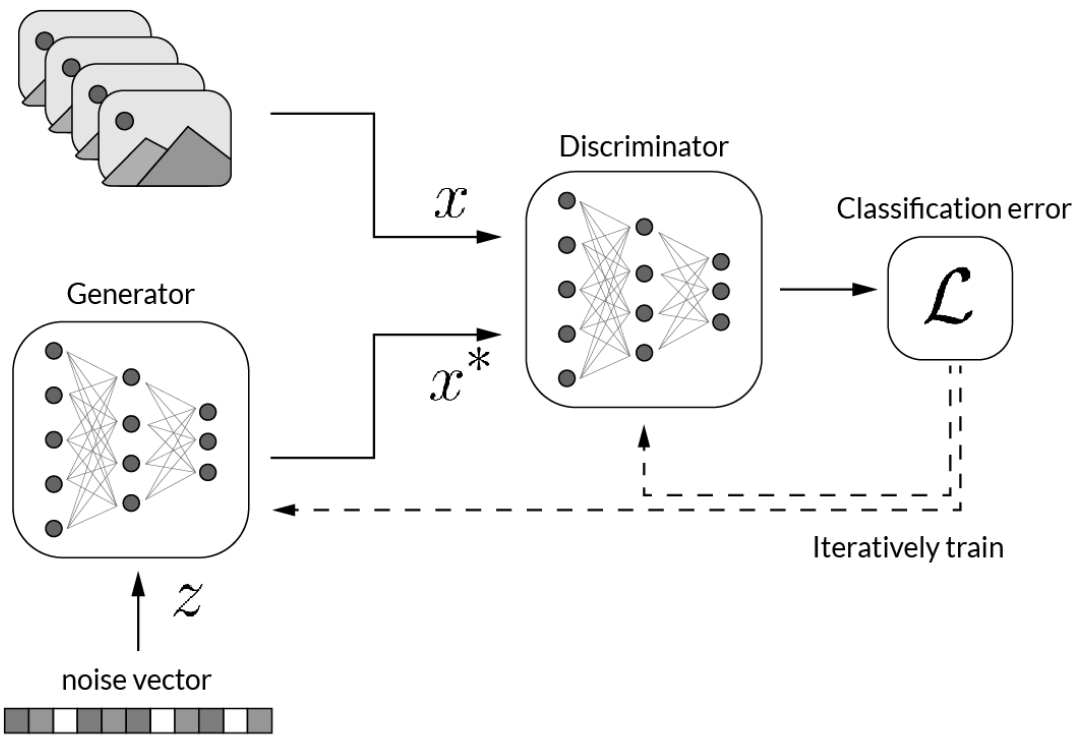


Figure 5: The two GAN subnetworks, their inputs and outputs, and their interactions.

3.2.1 Generator and discriminator

The generator as we have seen, is the model that is used to generate the outputs of the GAN, the synthetic images. The generator (*G*) takes a random noise vector *z* and produces a fake example *x** of the same class as the ones in the dataset. *G(z)=x*.* The generator's final goal is to produce examples of a certain class that fool the discriminator. This set of features that compose the fake image of the class is fed into the discriminator (*D*) that will determine how real or fake this image is based on its inspection of it. The generator wants the prediction of the discriminator, *D(x*)* as close as 1 as possible, indicating the image is real. While the discriminator wants this prediction to be as close to 0 as possible, indicating the image is fake.

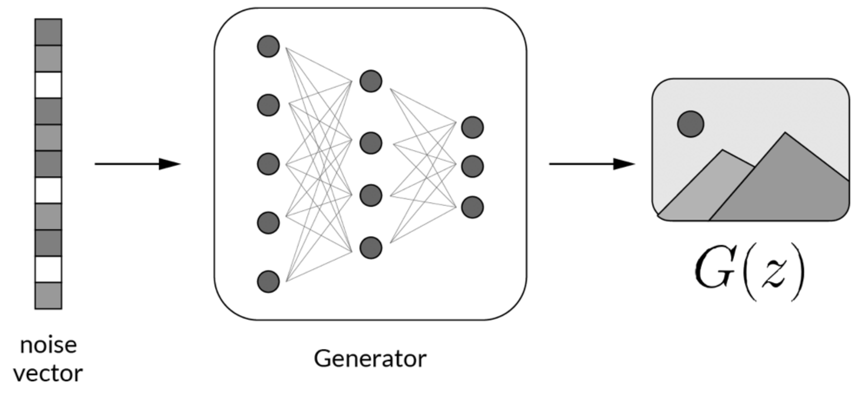


Figure 6: Schema of the generator.

The discriminators goal is to accurately differentiate real data from fake data, this is, the discriminator wants *G(x)=1* and *G(x*)=0,* but the generators goal is the opposite, wants to create a really good fake example *x** that strives to get *D(x*)* as close as 1 as possible. After the discriminator's predictions, a cost function can be computed that measures how far the examples produced by the generator are being considered real by the discriminator, because the generator wants to seem as real as possible. This function is used to update the parameters (θ_G) of the generator so it improves over time knowing in which direction to move it's parameters so the generated images, G(z)=x*, will be more likely to fool the discriminator, D(x*)=1 [21].

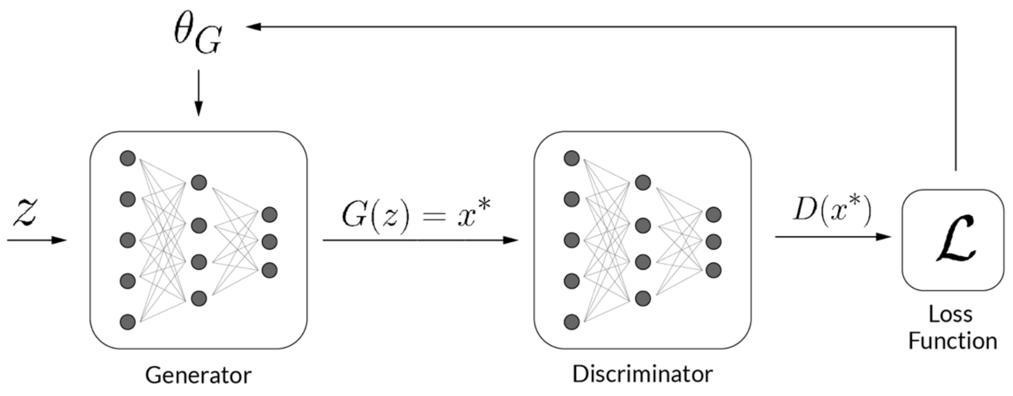


Figure 7: Schema of the generator's parameters update, training of the generator.

The discriminator works as a classifier, that will only have two classes, real and fake. To do so, the neural network will take the features of the dataset and train its parameters (θ_D) to learn how to map these features to each of the classes and learn how to make better predictions. The goal is to reach a point where the difference between the real values (0 for fake and 1 for real) and the predictions is minimized. The gradient of cost function will be used to improve the discriminative model updating its parameters (θ_D) as it indicates the direction in which those parameters should go to make a better prediction the next time [22].

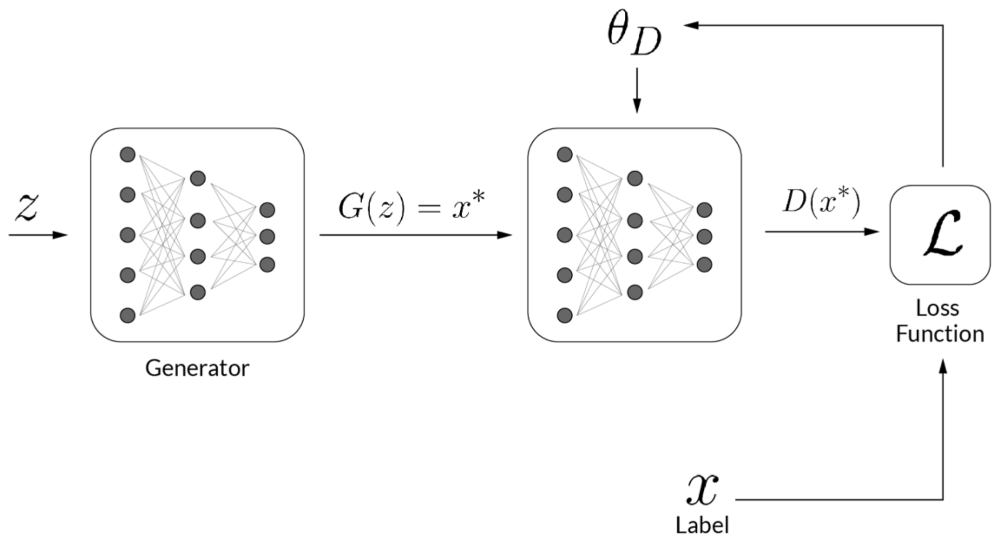


Figure 8: Schema of discriminator's parameters update, training of the generator.

3.2.2 BCE Cost Function

The Binary Cross Entropy (BCE) cost function is used for training GANs as it is a measure of the difference between computed probabilities and actual probabilities for predictions with only two possible class. The greater the cross-entropy loss, the further away our predictions are form the true labels.

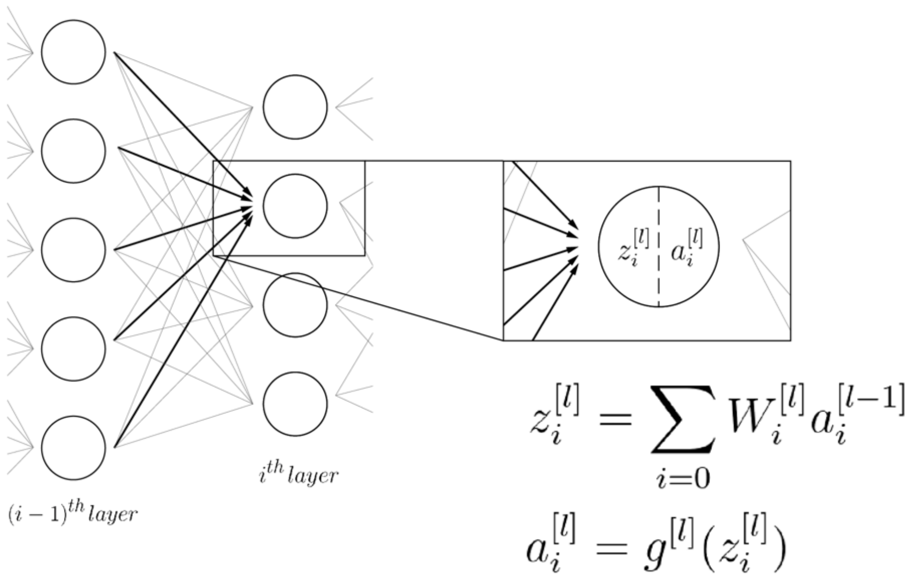
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[logh(x^{(i)},\theta) + (1-y^{(i)})log(1-h(x^{(i)},\theta))]$$

Being m the number of examples in the batch; h are the predictions made by the model; y are the labels of the different examples (true labels of if an image is real or fake); x are the features that are passed in through the prediction, the images and θ are the parameters of whatever is computing that prediction, in our case, the discriminator [23]. For this Work, what is needed to be taken into account about the BCE cost function is that it has two parts, one for each of the classes (true or fake). This function will be close to zero when the predictions and the real labels are similar, and it will approach the infinity if the predictions and the labels are different [24].

3.2.3 Activation functions

Activations are functions that take any real number as input, also known as its domain, and outputs a number in a certain range using a non-linear differentiable function. In deep neural networks, and more specifically in GANs, this activation functions are used for classification. These functions have to be non-linear and differentiable, the first condition, non-linearity prevents the hidden layers and neurons from collapsing in a simple regression. On the other hand, activation functions have to be differentiable as backpropagation is used to train the network, so it needs to provide a gradient to the previous layer to update its parameters.

To understand these activation functions, we need to understand what each of the individual nodes in the neural network do. A node takes the information from the previous layer a_i^{l-1} and predicts two things. The first one, z_i^l , is the sum of the product of various weights W_i^l on the outputs products from the previous layer a_i^{l-1}. Therefore, z is composed by the sum of the weighted outputs from the previous layer. On the other side of the node, we have the outputs of this node, a_i^l, this is the output of the activation function G^l of this node that takes the previously calculated z_i^l as input. In the following picture and scheme of this process can be found [25].



$(i-1)^{th}$ layer
i^{th} layer
$z_i^{[l]}$
$a_i^{[l]}$
$z_i^{[l]} = \sum_{i=0} W_i^{[l]} a_i^{[l-1]}$
$a_i^{[l]} = g^{[l]}(z_i^{[l]})$

Figure 9: Schema of activation functions.

These activation functions G^l, as we have mentioned, need to be non-linear and differentiable. The most common activation functions for deep learning models are:

3.2.3.1 ReLu:

Rectified Linear Unit (ReLu) takes the max value between 0 and the input value z^l.

$$g^{[l]}(z^{[l]}) = max(0, z^{[l]})$$

This means, that this function eliminates the negative values and makes them 0. Graphically, the ReLu activation looks like this

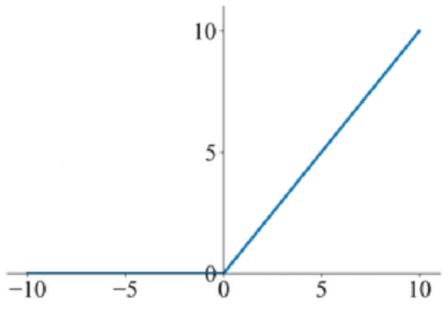


Figure 10: ReLu activation function [26].

As it can be seen in the graph, for values smaller than 0, the negative numbers, the derivative is equal to 0, so this function gives no backpropagation information to the network, that gets stuck on the same value and the weights stop learning. This is known as the dying ReLu problem, that makes the network stop learning. A variation of this function, the Leaky ReLu has been proposed to solve this problem.

3.2.3.2 Leaky ReLu

The Leaky ReLu is the same as the ReLu for the positive values, however, for the negatives, it adds a little leak or slop in the line. Now this function has a non-zero derivative, that is intended to be smaller than one so it doesn't form a line with the positive side, therefore, *0<a<1*.

$$g^{[l]}(z^{[l]}) = max(az^{[l]}, z^{[l]})$$

Graphically the Leaky ReLu function looks like the following graph:

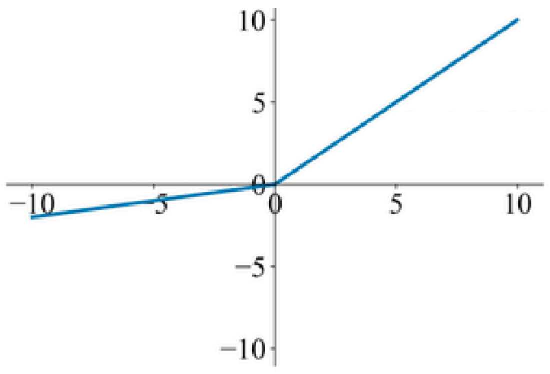


Figure 11: Leaky ReLu activation function [26].

3.2.3.3 Sigmoid

This activation function values the outputs between 0 and 1. If the input is positive the value will be between 0,5 and 1, while if the input is negative, the value will be between 0 and 0,5. This function is not very often used in hidden layers as it produces what is called the vanishing gradient problems, which means that in the tails of the function the derivative approaches to zero.

$$g^{[l]}(z^{[l]}) = \frac{1}{1 + e^{-z^{[l]}}}$$

Graphically the sigmoid function looks like the following graph:

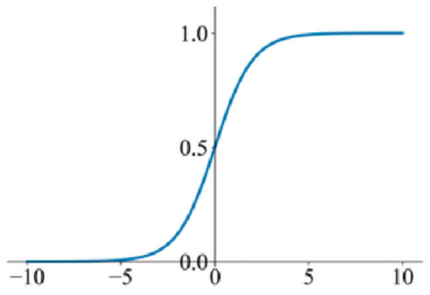


Figure 12: Sigmoid activation function [26].

3.2.3.4 *Hyperbolic tangent*

The hyperbolic tangent (tanh) activation function is similar to the previous sigmoid function, however, this function outputs values between -1 and 1. This allows this function to maintain the sign of the input which can be useful for some applications.

$$g^{[l]}(z^{[l]}) = tanh(z^{[l]})$$

Graphically the sigmoid function looks like the following graph:

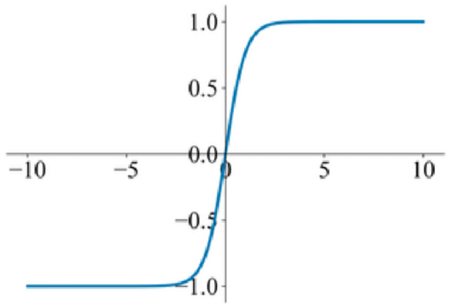
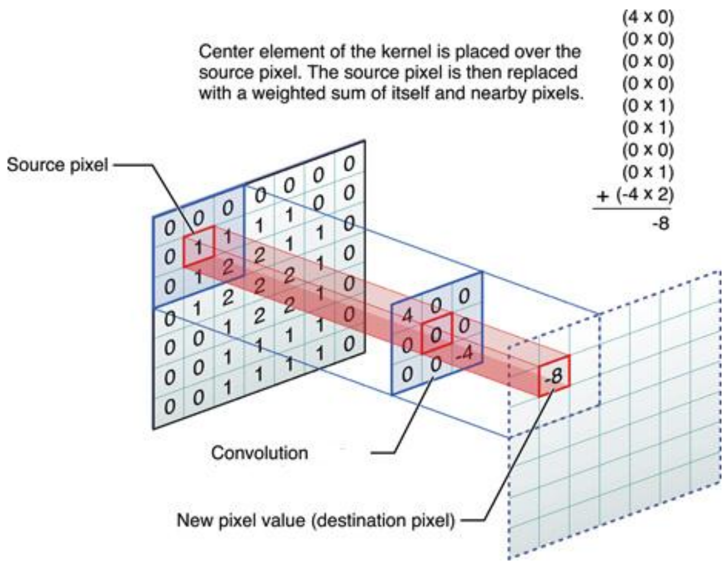


Figure 13: tanh activation function [26].

3.2.4 *Convolutions*

In mathematics convolution is a mathematical operation on two functions (*f* and *g*) that produces a third function (*f***g*) that expresses how the shape of one is modified by the other. In image processing, it is the process of transforming the image by applying a kernel over each of the pixels and its local neighbors across the entire image. In other words, convolutions are performed by sliding one or more filters over the input layer. Each filter has a relatively small receptive field (width x height) but always extends through the entire depth of the input volume. At every step the dot product between the input values and the filter entries is calculated as it can be seen in the following image.



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution
New pixel value (destination pixel)

Figure 14: Convolution operation on a 7x7 matrix with a 3x3 kernel [27].

The parameters of the filter will be applied to all the input values to the given filters. Therefore, these parameters will be shared across the image and allow us to learn the visual features and shapes found on the input image. Convolutions layers learn the patterns of the image, in the case of images, they learn the patterns found in 2D. This is key to understand some of their characteristics. The patterns they learn are translation invariant, this is, it doesn't matter which part of the image they are found in. This makes these tools data efficient when processing images. On the other hand, convolutions allow to learn the spatial hierarchies of patterns. The first convolution layer will learn local patterns while the second convolution will learn larger patterns made of the local patterns found in the first operation. This allows these convolutions to be really efficient in image processing as they can learn complex and abstract visual concepts, which is key to feature classification [28].

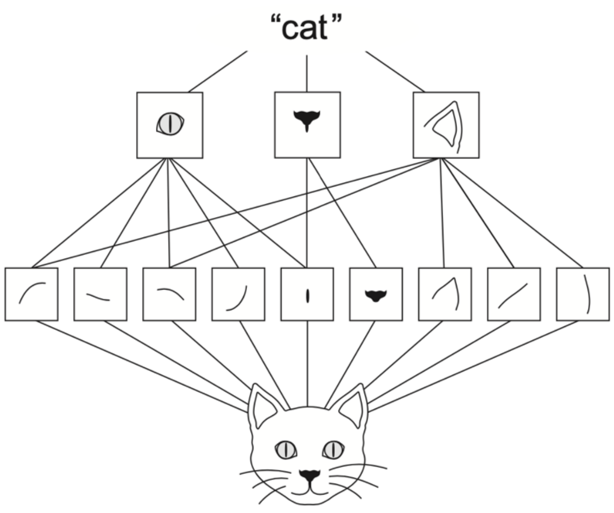


Figure 15: Spatial hierarchy of visual modules that combine into the 'cat' concept [16].

3.3 Types of GAN

After having explained how GANs work and how to build them, some concrete GAN types will be explained. Although the final type of GAN selected for the XYU ring rendering style transfer has been a CycleGAN, other alternatives are studied in the following lines in order to better understand what a CycleGAN is.

3.3.1 Conditional GAN

Until now, in the previous chapters, the outputs of the GANs were random examples that mimicked the dataset, this is, unconditional generation. However, if we had conditional image generation, we could ask for an example of an specific class to be generated by the GAN [29]. Remembering the metaphor of the counterfeiters, maybe we don't want to make a fake 5€ bill to fool the police, but we rather make a 500€ bill. To do so, in conditional GANs, apart from the random noise vector that the generator has as input, we will also provide a class and the generator will have to provide an example of that class. To be able to do so, we will need that our dataset is also labeled so the GAN during training learns what the features of each of the classes is. For example, in the counterfeit money metaphor, blue color would be a feature of the 5€ bill, while purple, would be a feature for the 500€ bill.

3.3.2 *Pix2pix*

In the previous section, we have seen how giving a certain class to the conditional GAN to generate a certain type of image. In section, the input for the generator, won't be a class but a picture. As we can imagine from its name, *pix* comes from image, this type of GAN will output the same image as the input but with a certain change. Therefore, this is an *image-to-image translation* kind of GAN, an special case of conditional GAN, whose condition is the whole image rather than a class [30]. In the following figure a famous image translation example from University of California, Berkeley can be seen, in this image, it can be seen how this style[3] transfer works on different domains.

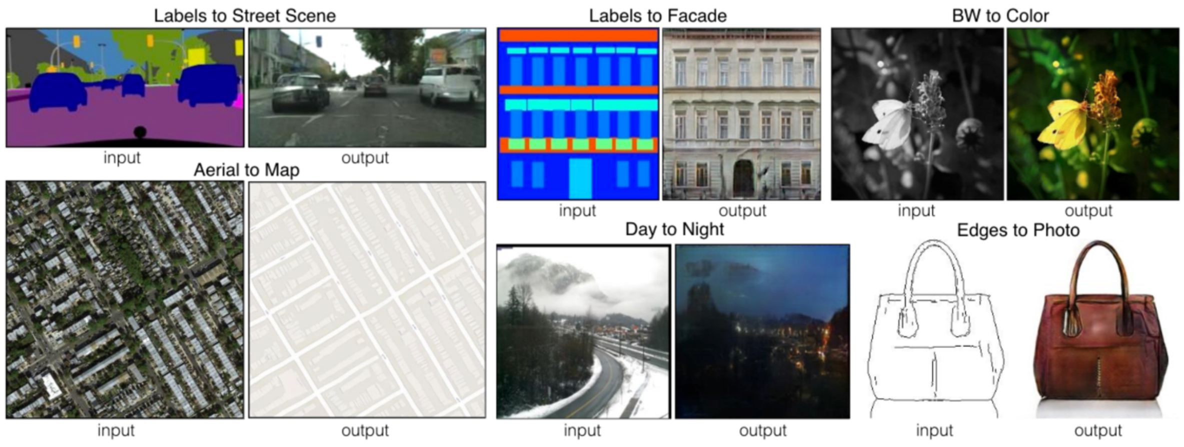


Figure 16: Image-to-Image Translation with Conditional Adversarial Networks [10].

As it can be seen pix2pix is a really powerful tool that can be applied in a wide range of applications. However, as in conditional GANs we need labeled data. In the case of image-to-image translation, we need that the data is composed of paired images. Our dataset should consist of the same exact image but with the different style. For example, in the case of the Day to Night style transfer in the previous figure, the day picture and the night picture should be taken from the same point and position [31].

[3] In this context, style refers to the textures, colors, and visual patterns in the image, at various spatial scales; while the content is the higher-level macrostructure of the image [16].

3.4 CycleGAN

The problem with the previous need of paired data was solved by the same UC Berkeley group that realized that we do not need perfect pairs, in fact, it is enough with just closing the cycle by translating from one domain to another and then back again [32]. This close cycle gave the name to this kind of GANs, the CycleGAN. This is the GAN type that will be later be used in the practical part of this Work. The CycleGAN is a technique that involves the automatic training of image-to-image translation models without paired examples that makes it perfect for the later practical application as no need of paired image dataset is needed.

To achieve this cycle consistency two generators are needed, the fist generator (G_{AB}) will translate from the domain A to B and the second generator (G_{BA}) that will translate from the domain B back to A. Therefore, there will be two losses, one forward cycle-consistency loss and another backward cycle consistency loss. But as all they mean is x*=G_{AB}(G_{BA}(x)) and y*=G_{BA}(G_{AB}(y)), we can think of them as essentially the same, but off by one [32].

3.4.1 *Cycle consistency loss*

The cycle consistency loss, that can be expressed as *//x-x*//* or *//y-y*//* depending on which of the styles we consider as the starting point, ensures that the original image and the output image after completing the cycle, the twice-translated image, are the same. Apart from this loss function, the total loss function in CycleGANs have other terms.

$$\begin{aligned} Lcyc(G, F) = {} & E_{x-p_{data(x)}}[\|G_{BA}(G_{AB}(x)) - x\|_1] \\ & + E_{y-p_{data(y)}}[\|G_{BA}(G_{AB}(y)) - y\|_1] \end{aligned}$$

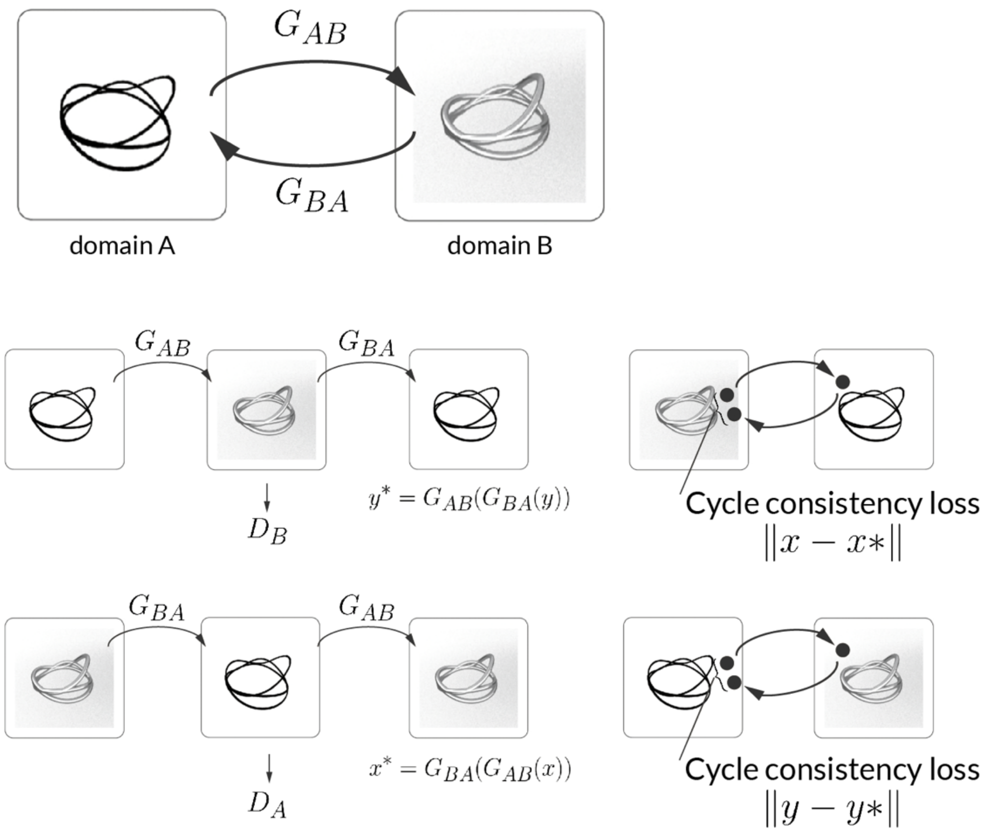


Figure 17: Schema of the Cycle consistency loss.

3.4.2 Adversarial loss

Apart from the cycle consistency loss mentioned before, we still have the adversarial loss that should be taken into account. Every translation by the G_{AB} generator will be checked by the D_B discriminator and the generator G_{BA} will be controlled by the D_A discriminator. Every time we translate from one domain to another, the discriminator will test if the output of the generator looks real.

This component of the generator's loss that we are later going to implement in the practical part is similar to the previously mentioned loss for GANs, however, it is important to note that the criterion now is based on least squares loss, rather than binary cross entropy loss or W-loss used in general GANs. This loss function will ensure that the outputs of this CycleGAN look real, so it is key for the good functioning of the model.

$$L_{GAN}(G, D_B, x, y) = E_{x-p(y)}[logD_B(y)] + E_{y-p(x)}[(1 - D_B(G_{AB}(x))]$$

This function has two terms, the first is the probability of the given image to be the real one rather than the fake one, the translated one. The second term is where the generator may get to fool the discriminator. The previous formula is only the formula for the D_B discriminator, in the final loss there is an equivalent formula for D_A.

As in the general GANs, the adversarial loss measures if the generated images look real, if they are indistinguishable to the ones in the training set [33].

3.4.3 *Identity loss*

The identity loss measures if the output of the CycleGAN preserves the overall color temperature or structure of the picture. Although it is an optional term for the total loss, it will be considered in this Work for color preservation. Pixel distance is used to ensure that ideally there is no difference between the output and the input, this ensures that the CycleGAN only changes the parts of the image when it needs to. In the following figure, the effects of considering this Identity loss can be seen in two different examples [34].

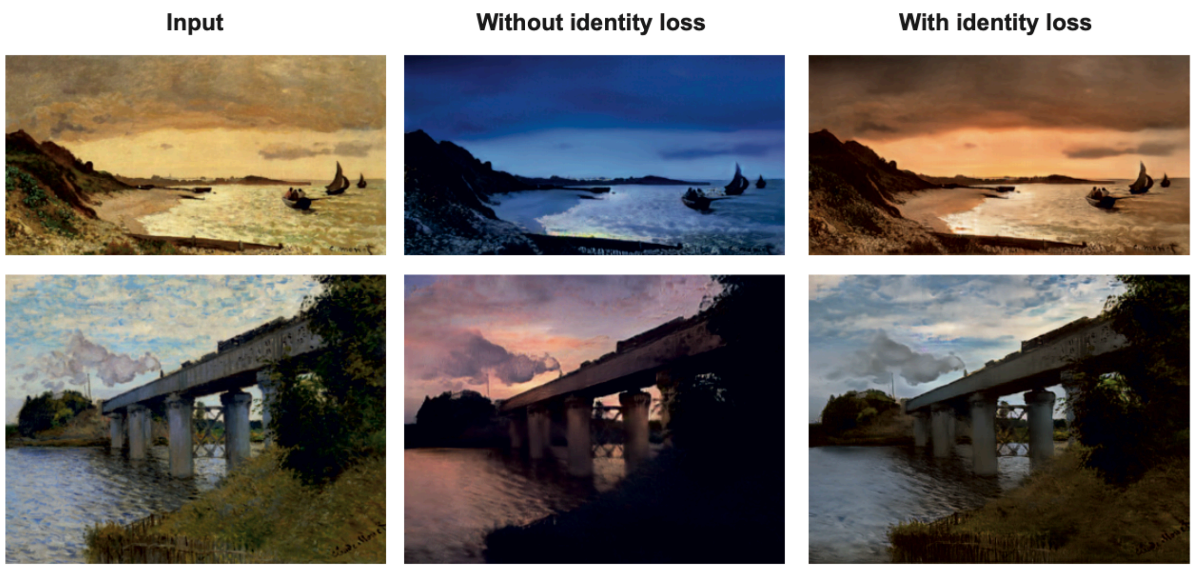


Figure 18: Output image comparation with and without Identity loss [33].

3.4.4 *Generator total loss*

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, x, y) + L_{GAN}(F, D_X, y, x) \\ + \lambda_{cyc} L_{cyc}(G, F) + \lambda_{ident} L_{ident}(G, F)$$

As explained by Zhu et al. [32], the full objective of the CycleGAN must be reducing this three loss functions. Actually, Zhu et al. in their article show that training the networks with only one of the functions doesn't arrive to high-quality results. In the Previous formula, we can see that both the identity loss and cycle consistency functions are weighted by λ_{ident} and λ_{cyc}, respectively. These scalars control the importance of each of the losses in the training. In our case, following the values for these parameters proposed in the article, λ_{cyc} will be 10, and λ_{ident} will be 0.1, as this last function only controls the tint of the input and output images, in our case, as the dataset is composed of the same colors, it doesn't suppose any big problems.

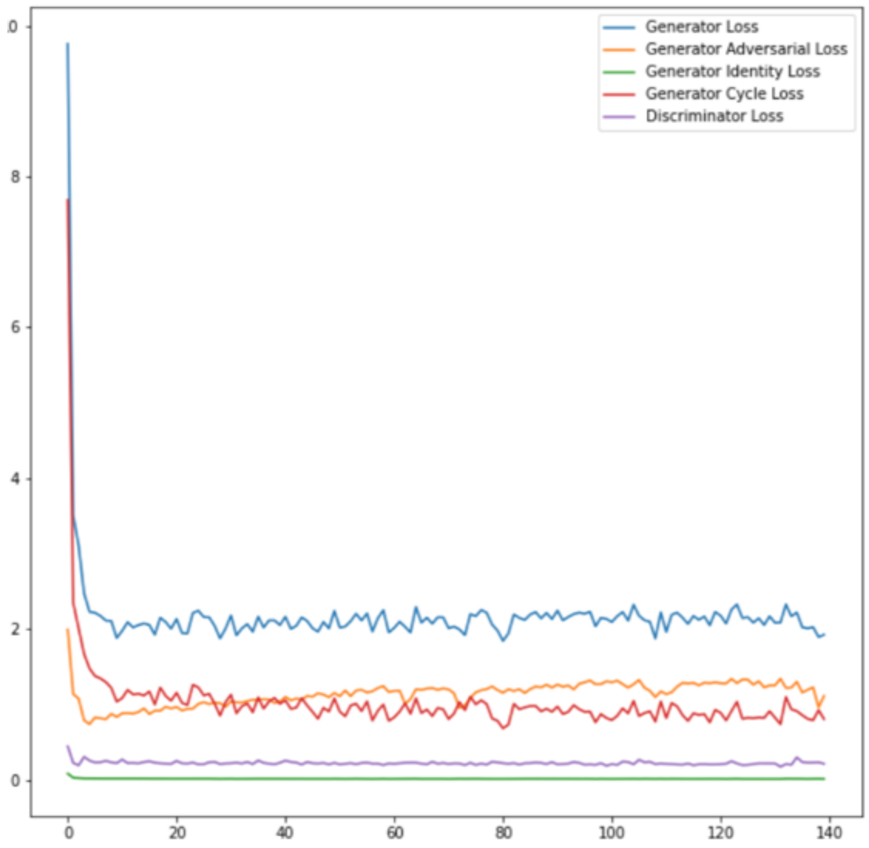


Figure 19: Loss function plot for the first 1400th steps of the training, corresponding to 14 epochs.

In the previous image, the different components of the generator loss, as well as the generator total loss and discriminator total loss can be seen in a chart. This chart corresponds to the training of one of the CycleGANs of this work and we can see how the cycle consistency loss has decreased during training, however, the adversarial loss has been increasing in the first steps but at the end of the plot, we can see that it will start decreasing.

3.4.5 Generator architecture

The generators in the CycleGAN are updated generators compared to the traditional GANs, this upgraded generators are based in U-NET. This architecture framework it's an encoder-decoder model that uses skip connections. Compared with the traditional model, that takes a noise vector as input, the U-net takes in an entire image and uses convolution layers in the encoder and can be thought as a classification model that finds the classes in the input, and outputs a value or some compressed data. Those important features are decoded to the output, another image. They can be then imagined as an autoencoder; however, we don't want the output x^* to be as close as possible to x, we want it to be conditioned with a certain style.

Nevertheless, to prevent these layers from losing some information during encoding, the U-Net also introduces some skip connections between the decoder and encoder layers, allowing certain details that may have been lost during the encoding still be present on the later layers.

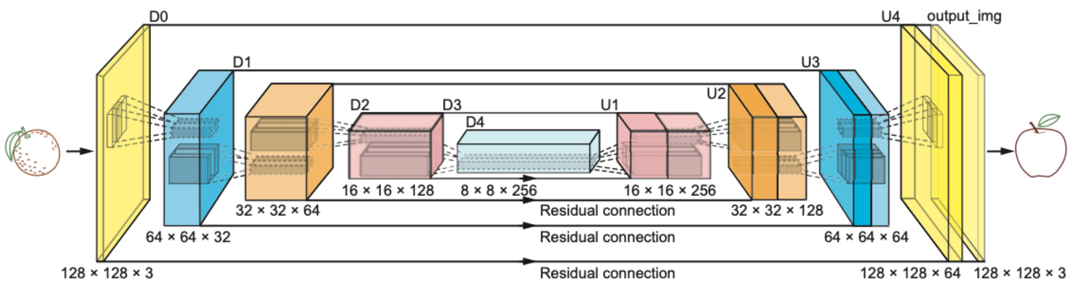


Figure 20: Architecture of the generator [33]. Example of the *orange2apple* CycleGAN.

The encoder as it can be seen in the previous image is made of convolutional layers that reduce the resolution while the decoder is made of deconvolutional layers, transposed deconvolutions, that upscale the information back to an image with the same size as the input. In-between skip connections are created to permit that the information has an easier way to propagate through the network [35].

3.4.6 Discriminator architecture

The discriminator of the CycleGANs is based in the PatchGAN architecture. The difference between this architecture and the previous generator, is that in this CycleGAN instead of having a single float as an output, is outputting a matrix of values. A PatchGAN architecture will output a matrix of values, each of them between 0 (fake) and 1 (real), classifying the corresponding portions of the image. The activation function for the training of the model in the practical part of this Work is the LReLu (Leaky Rectified Linear Activation)[4].

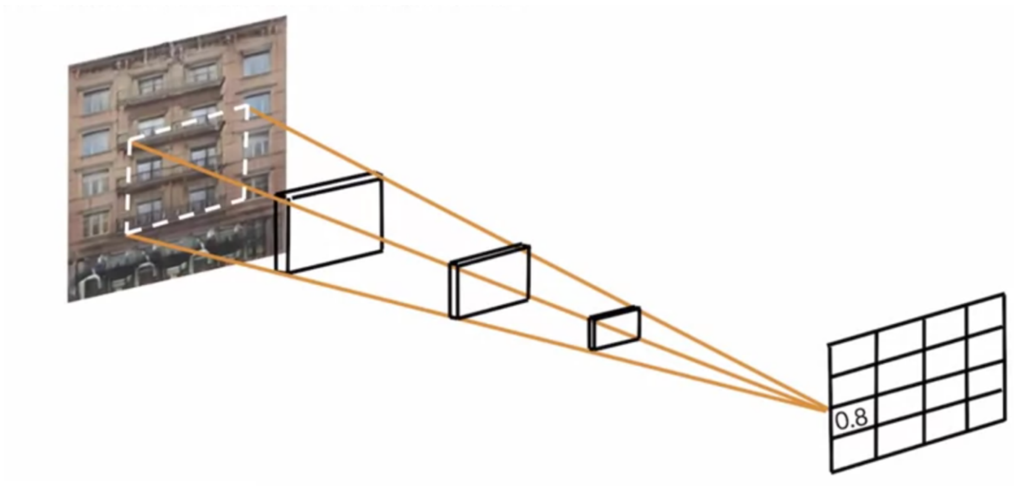


Figure 21: Example the classification of a portion of the image in the PatchGAN architecture [36].

In summary, if a fake image is passed to the discriminator, this should output a matrix of all zeros, on the contrary, if a real image is passed, it should output a matrix of only ones [36].

[4] This function is different from the previously mentioned function as a new term is added: $f(x)=\max(0,x)+\beta*\min(0,x)$

4 Practical case development

4.1 Case description

4.1.1 XYU ring[5]

XYU is not only a ring but an algorithm to create them. This algorithm uses splines to generate infinite ring possibilities. The starting point is set by the user, who specifies the number of splines and the length and thickness of the ring. The control points of the splines are randomly selected and adapted to make them continuous on the ring. If the user does not like the result, the algorithm can be run again until an aesthetic shape is achieved.

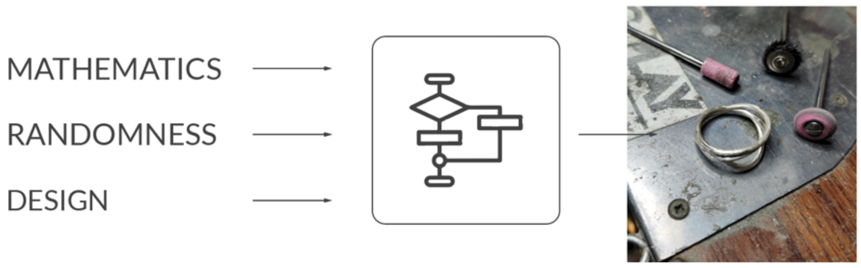


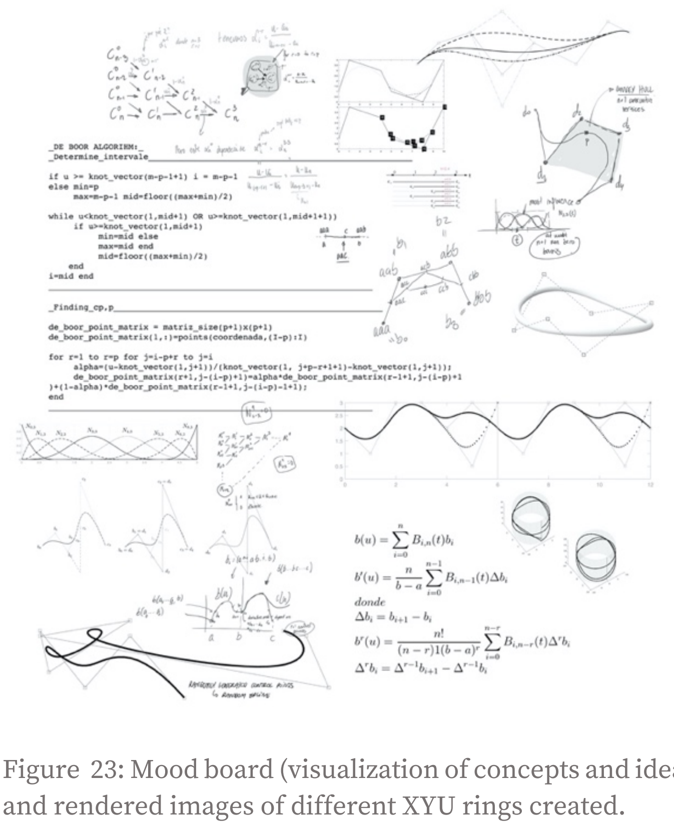
Figure 22: Original concept schema.

[5] XYU is the name of this ring project, it is not an acronym, but the name of this ring composed of 3 randomly chosen letters.

Inspiration can be quite a tricky topic, because I do believe every single random encounter I have had in my life has led me to this unique position where I am right now, writing to you about what it means to design, to do maths and to integrate both.

There are infinite possibilities and universes in which this decision did not happen at all, as well as I am sure there are many others in which it also occurred. But if I had to point down a single event it would probably be how I realized AI was quickly taking over designers' jobs, creating ads, banners... So I thought: how will designers face this new world where thousands of variants can be created with only one click?

Maths can be the answer to this, the one that gives the input to generate these infinite possibilities if we integrate algorithms with the power of randomness. If creativity is imagining the impossible, why not have the infinite as the starting point? XYU ring is not only an item but an infinite. This is not just a jewel, but as many as you want. This is not just a ring, it is an algorithm that designs them.



```
_DE BOOR ALGORIHM:_
_Determine_intervale_

if u >= knot_vector(m-p-1+1) i = m-p-1
else min=p
     max=m-p-1 mid=floor((max+min)/2)

while u<knot_vector(1,mid+1) OR u>=knot_vector(1,mid+1+1))
     if u>=knot_vector(1,mid+1)
          min=mid else
          max=mid end
          mid=floor((max+min)/2)
    end
i=mid end

_Finding_cp,p_

de_boor_point_matrix = matriz_size(p+1)x(p+1)
de_boor_point_matrix(1,:)=points(coordenada,(I-p):I)

for r=1 to r=p for j=i-p+r to j=i
     alpha=(u-knot_vector(1,j+1))/(knot_vector(1, j+p-r+1+1)-knot_vector(1,j+1));
     de_boor_point_matrix(r+1,j-(i-p)+1)=alpha*de_boor_point_matrix(r-1+1,j-(i-p)+1
)+(1-alpha)*de_boor_point_matrix(r-1+1,j-(i-p)-1+1);
end
```

$$b(u) = \sum_{i=0}^{n} B_{i,n}(t) b_i$$

$$b'(u) = \frac{n}{b-a} \sum_{i=0}^{n-1} B_{i,n-1}(t) \Delta b_i$$

donde

$$\Delta b_i = b_{i+1} - b_i$$

$$b^r(u) = \frac{n!}{(n-r)1(b-a)^r} \sum_{i=0}^{n-r} B_{i,n-r}(t) \Delta^r b_i$$

$$\Delta^r b_i = \Delta^{r-1} b_{i+1} - \Delta^{r-1} b_i$$

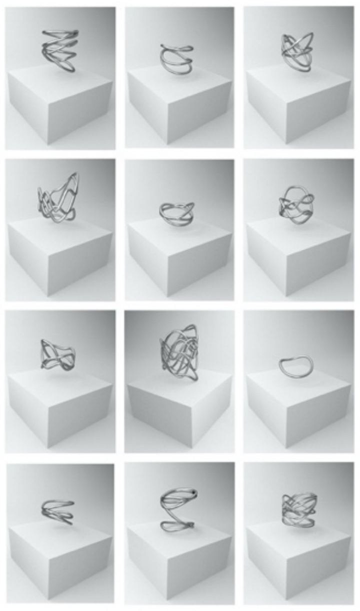


Figure 23: Mood board (visualization of concepts and ideas) of the XYU ring and rendered images of different XYU rings created.

4.1.2 Actual model state

XYU ring is not merely an ornament, but a proof of the infinite possibilities of computational design. This algorithm allows the users to design their own 3D ting example based on their preferences which are used by the algorithm to generate random rings. Each time the code is run, a different and unique ring is produced. This procedure permits to personalize each of the XYU ring examples.

Once the users find the ring that they like, this is automatically modelled on Maya and the 3D object is sent to the jeweler. Each of the rings is 3D printed and cast, so each of the pieces are unique.

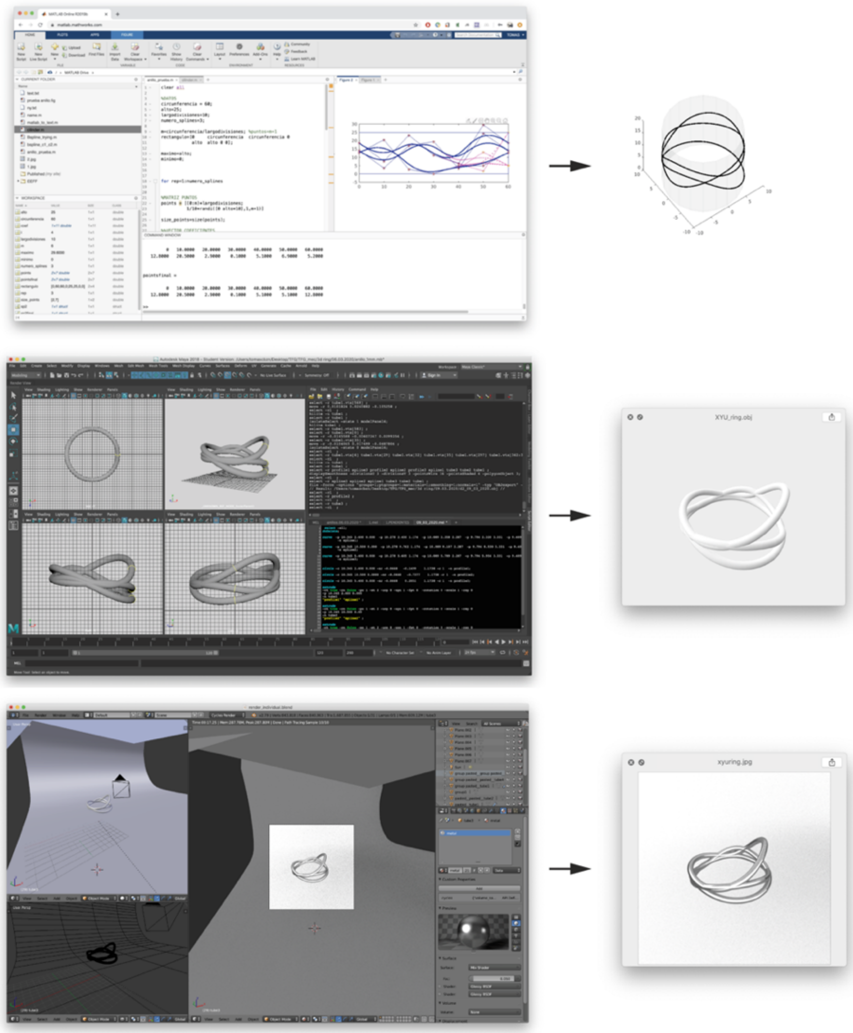


Figure 24: Actual model's program and steps followed. On top Matlab program screenshot where the algorithm is run. In the middle, Maya program with the 3D object of the ring. On the bottom, blender program and the rendered image.

Description of the actually used programs and different steps followed to go from the initial starting data to the final design of the ring are shown in the previous figure. The algorithm is run on Matlab, the ring is later automatically translated into Maya MEL coding language where the ring is 3D modelled and .obj file is created. Finally, the realistic images of the ring are rendered using Blender program.

To generate the 3D object and send the .obj file to the jeweler who will 3D print and cast it, Maya 3D computer graphics application is used. To do so, the information of the ring is passed to Maya using the Maya MEL coding language. This is, the output of the Matlab algorithm is a .txt file with the instructions of the curves that generate the different brands of the ring, the splines, as well as the circles that will be extruded along the splines to form the 3D object. Therefore, the information to 3D generate the ring is passed as coded instructions to Maya.

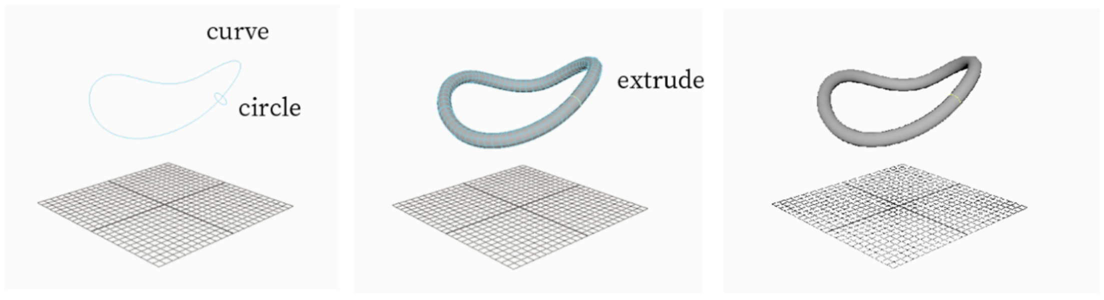
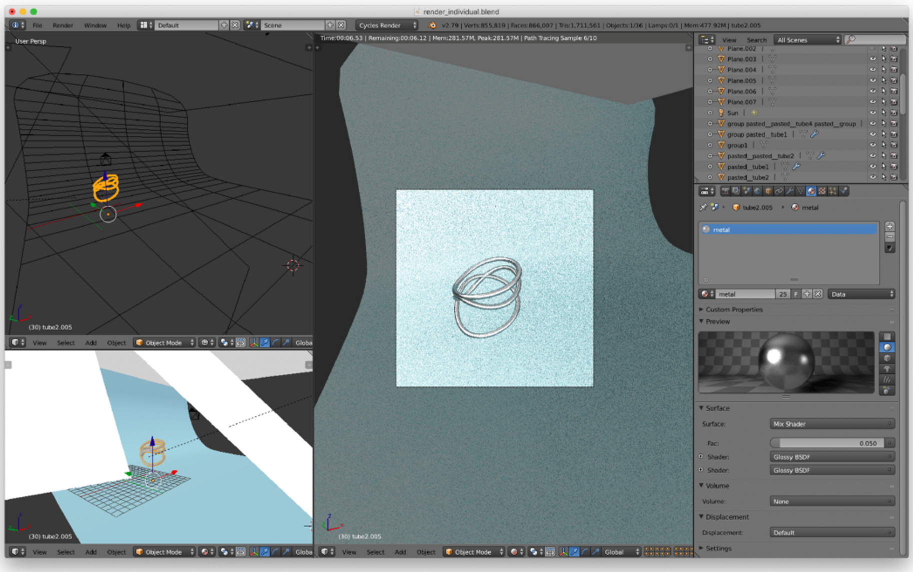


Figure 25: Schema of how each of the brands of the ring are 3D modelled on Maya program. The points that compose the curve as well as the circle that will be extruded along this curve are passed to Maya program using Maya MEL coding language.

The last stage of the process corresponds to the rendering of the final product. In this last step, Blender rendering program is used to generate realistic images on the ring and to show the final product to the costumer.

Rendering is the process of turning a 3D scene into a 2D image [37]. A 3D scene is composed of various elements apart from the object we want to render, such as the background, the camera, the materials and the light. This step is the most tedious part of the XYU ring generation. The rendering of images not only takes a long time to be calculated, but scenes need to be arranged and the images not always render as expected the first time. Actually, companies like Pixar that create whole animation films by rendering each of the video frames of their films, have rendering directors to optimize this process [38].

To calculate how long it takes to render an image, apart from the time needed to calculate the color of each of the pixels, that is not the biggest one of the process; the scene setting time, the lightning configuration and the material generation and selection times should be added. A good rendered image of one of the XYU rings would take in total around an hour in the making.



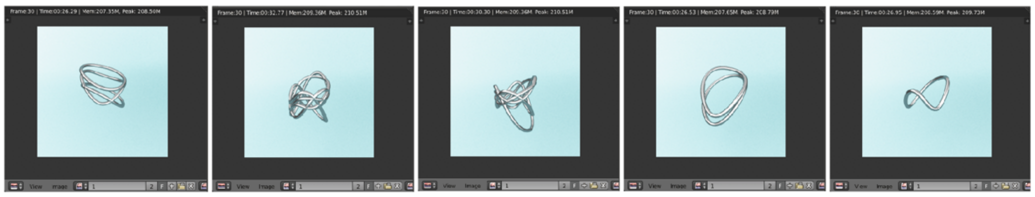


Figure 26: Rendering setting and image rendering times for 1000x1000 pixel size simple images.

As it has been seen, the original idea was not achieved as intermediate external programs need to be used in the process. In the following picture the actual process is shown in which intermediate steps for the 3D object and rendering generation are needed.

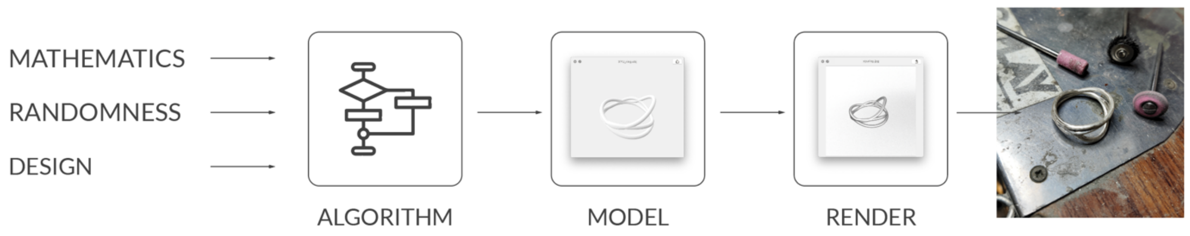


Figure 27: Actual model schema: the algorithm run in Matlab, the 3D object modelled in Maya, the rendered image in Blender and the final product after being made by the jewel.

4.1.3 *New model proposal description*

The initial process in which the generation of rings was completely automatized has not been achieved yet. This limitation was the starting point of this Work. Therefore, finding a new approach for this ring design generation algorithm in which no need of the designer has been tackled in this Work which has resulted in this proposal of including a CycleGAN in the process that allows the algorithm to generate the rendered images of the ring and show them to the user.

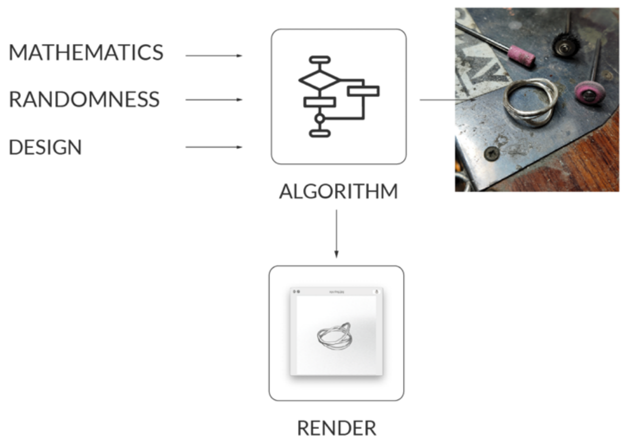


Figure 28: Proposed model idea. The algorithm in this case, apart from creating the ring is also responsible of generating the rendered images of it. A CycleGAN is added to the algorithm for this last part.

4.2 CycleGAN development

4.2.1 CycleGAN description

CycleGANs are used to translate image style such as *horse2zebra, apple2orange, photo2Cezanne, winter2summer*… and vice versa [39]. In this Work CycleGANs will be used for the last parts of the process of the design of the XYU rings, the presentation. To achieve this, two CycleGANs will be trained: a first *sketch2object3D* that will take a starting point a sketch of the shape of the ring (an image of the 3D view of the splines of which is composed) and will apply a style transfer to generate an of the actual 3D volume of the ring. This will be used to see the possibilities of CycleGAN, as the *sketch2object3D* is a simpler model, it can be trained with a simpler image dataset, as no renderings are needed to be done. This model's training will serve as the starting point for the practical part of this Work.

The second CycleGAN, *sketch2rendering*, will also start with the sketch and will apply a style transfer to generate a rendered image of the ring. In the following figure these two CycleGAN can be seen. Actually, the second CycleGAN, the *sketch2rendering* is the final objective of this Work, as renders are used in design to present final products, but long time and preparation are needed to obtain them. Training a CycleGAN that would render the generated rings would not only ease this process and reduce the process time but also automatize the process.

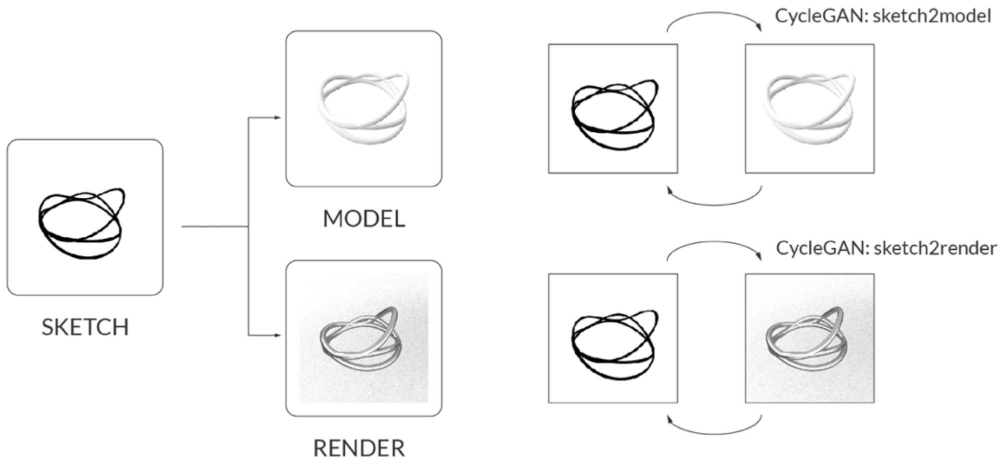


Figure 29: Proposed CycleGANs, *sketch2object3D* and sketch2rendering.

4.2.2 *Building the CycleGAN*

In this part of the Work the code used for training the CycleGAN will be introduced. This code is based on the generative model proposed by Zhu et al. in the paper *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* [32]. In the following lines the different elements of this code will be discussed and explained. The code is attached at the end of this Work. The CycleGAN will be implemented on Pytorch.

Before starting with the CycleGAN code, a visualization function will be defined, and the image dataset imported. The first one, *show_tensor_images* will be the visualization function that will plot and print the tensor of images. The second, *ImageDataset,* that is inspired by the *dataset.py* by aitorzip [40] will serve to setup the dataset with the following directory structure:

```
.
├── datasets
|   ├── <dataset_name>          # i.e. sketch2render
|   |   ├── train A             # Training .jpg images of domain A
|   |   ├── train B             # Training .jpg images of domain B
|   |   ├── test A              # Testing .jpg images of domain A
|   |   └── test B              # Testing .jpg images of domain B
```

4.2.2.1 *Building the generator*

As explained in the theoretical part of this Work, the generator is based on the U-Net architecture, which uses skip connections as described before. Between the encoding (contracting) and decoding (expanding) the residual blocks have been added.

- Residual Block

In CycleGANs after the expanding blocks, there are convolutional layers where the output is added to the original input to ensure that the network cannot completely change the image. This can be thought as a kind of skip connections that also allows the network to be deeper, as they help with the vanishing gradients issues that occur when the neural network gets too deep, and the gradients multiply in backpropagation become very small.

The *ResidualBlock* will perform two deconvolutions and an instance normalization. Afterwards, the input will be added to this output to form the residual block output.

- Contracting and Expanding Block

This is the decoding and encoding section of U-Net. The *ContractingBlock* will be the encoding that will perform a convolution followed by an instance normalization followed by a optional instance norm. The activation function will be the ReLu function.

The *ExpandingBlock*will be the decoding that will perform a convolutional transpose operation in order to upsample, with an optional instance norm.

The *FeatureMapBlock* is the last and first layer of the generator. It maps the input and output to the desired number of channels.

After defining the block classes, we can define the actual generator. This will be composed of 2 contracting blocks, 9 residual blocks and 2 expanding blocks to transform the input image into an image of the other domain. This generator will have an upfeature layer at the start and a downfeature layer at the end.

4.2.2.2 Building the discriminator

The discriminator in the CycleGAN is based on the PatchGAN architecture as it has been previously explained. The discriminator will output a matrix of values classifying corresponding portions of the image as real or fake. The discriminator's final layer will simply map from the final number of hidden channels to a single prediction for every pixel of the layer before it. The discriminator will be based on the contracting path of the U-Net. It will be composed of a series of 3 contracting blocks and a final convolutional layer.

4.2.2.3 Training parameters

In this part of the code, the different parameters are defined, and the images of the dataset are loaded (an horizontal random flip is added to introduce some data augmentation).

4.2.3 *Building the network*

Once the basic elements are defined, we can initialize the network. Our CycleGAN is composed of two generators and two discriminators:

- gen_AB: generator for domain A to domain B
- gen_BA: generator for domain B to domain B
- disc_A: discriminator for domain A
- dics_B: discriminator for domain B

There is an option to load a pretrained model, which is useful to continue with the training of the CycleGAN loading a pretrained model but adding some modifications to the parameters.

4.2.3.1 *Discriminator Loss*

As explained before, the discriminator's loss function is an adversarial loss function. This function takes the discriminators predictions and the target labels and returns the adversarial loss. In our case, following the Zhu er al. recommendations [32], the adversarial loss calculation criterion is the mean squared error of this two elements.

4.2.3.2 *Generator Loss*

The generator loss in CycleGAN is compassed of 3 losses as it has been explained in the theoretical part.

- Adversarial Loss
- Identity Loss
- Cycle Consistency Loss

After calculating this losses, they are put together into the generator total loss. To sum these three components, *lambda_identity* and *lambda_cycle* will be used to weight the importance of this two components in the final loss. In our case the *lambda_identity* will be small because this is not an important component for our application.

4.2.3.3 *CycleGAN training algorithm*

After all the components are coded, the training loop is created, the details of each iteration are as follows:

```
for training_step do
    Dataloader returns random images batch (real_A, real_B)
    Use the generator for domain translation
           fake_B=gen_AB(real_A) and viceversa
    Compute discriminator losses disc_A_loss, disc_B_loss
    Update discriminator gradients and optimizers
    Compute generator total loss gen_loss
    Update generators gradients and optimizers
    if iteration_step % display_step == 0 then
         Visualize the results
    if iteration_step % save _step == 0 then
         Save the model
    iteration_step += 1
```

4.2.3.4 *CycleGAN testing*

Part of the image dataset has been set aside to be used with testing purposes. These images of both of the domains are saved in the testA/ and testB/ folders respectively. In this part of the code these images are loaded, and they are used as input for the model. The testing is used to make sure that when our model gets new images, images that it has not seen before, the output has the same conditions as the ones in the training set. Therefore, the testing algorithm is similar to the training algorithm but no loops are needed as no parameters are changed. The images are loaded and transferred to the other domain by the generators, then the loss is calculated and printed to be compared with the loss obtained in the training.

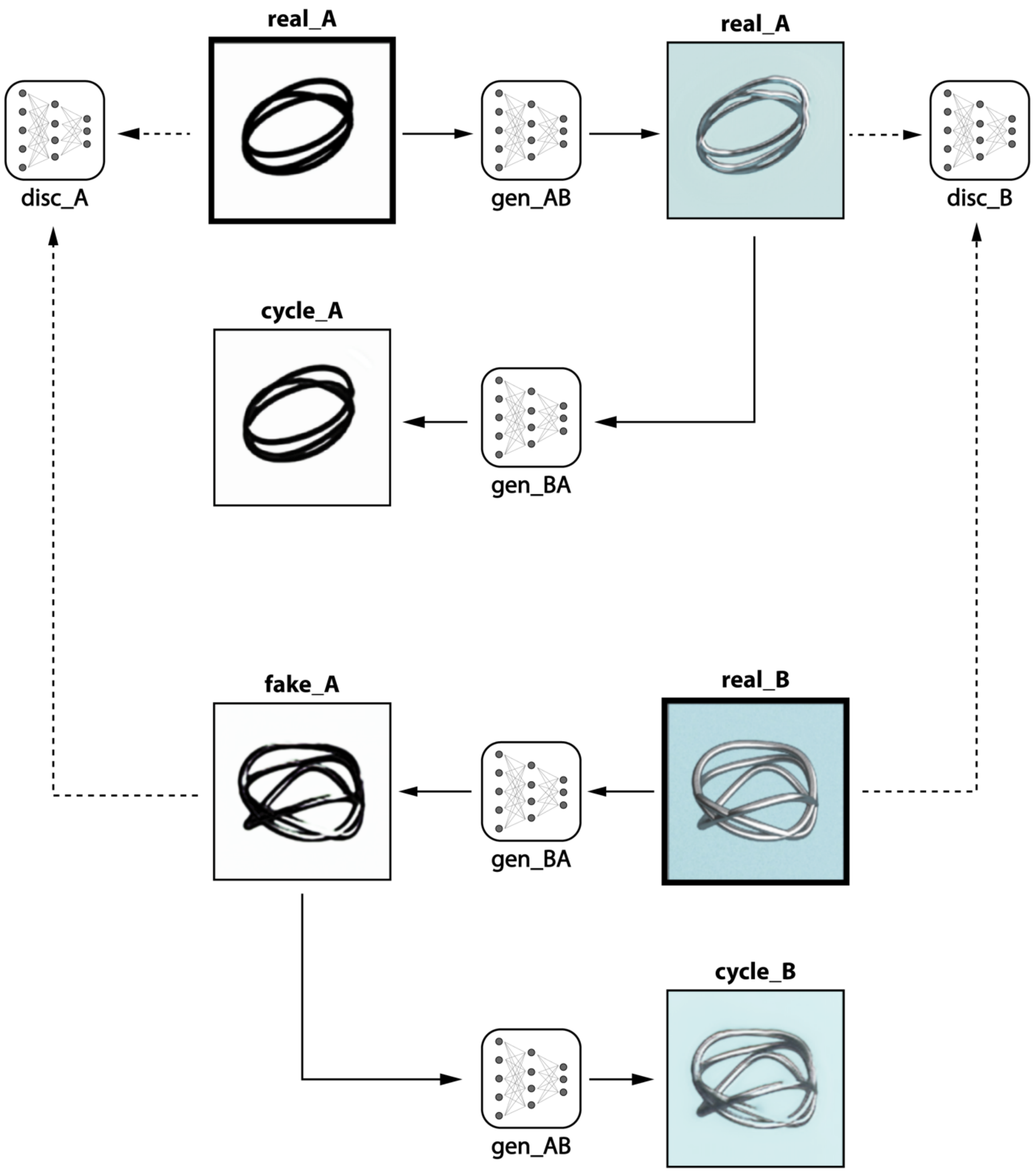


Figure 30: CycleGAN architecture schema.

4.2.4 *Training the CycleGAN*

Once the algorithm to train and test the CycleGANs is explained the actual training process followed in this Work will be explained. First the *sketch2object3D* was trained, as a simple dataset was needed as an starting point, so in case any changes in the training image dataset should have been made, these would be easier to be done. During the training of these CycleGANs some needed changes on the initial idea and dataset have been found. On the following lines the process followed for training both of the CycleGANs will be explained as well as the problems encountered.

4.2.4.1 *Sketch2object3D: white background problem*

During the training of this *sketch2object3D* CycleGAN, a problem on the database images has been found. The white background images used for the training of the CycleGAN supposed a problem for the development and learning of the system that was uncappable of differentiating where the ring ends and the background starts.

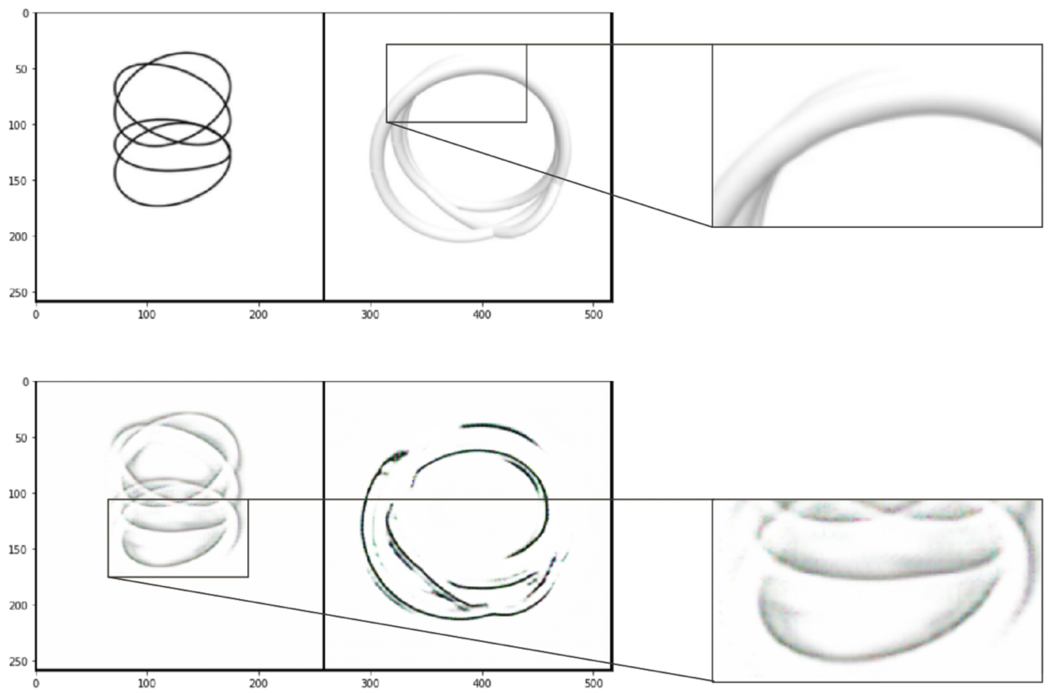
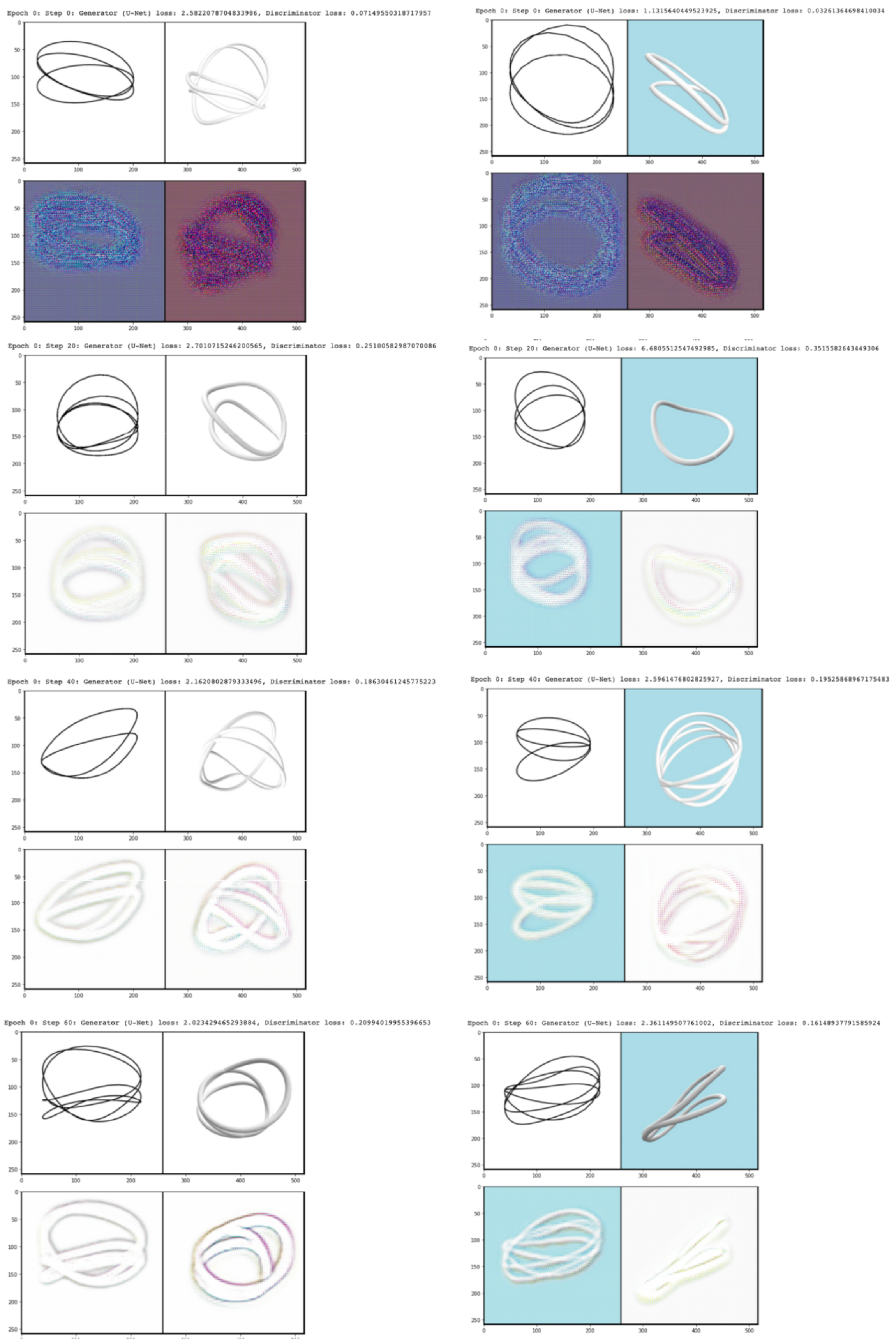


Figure 31: Result of the CycleGAN *sketch2object3D* (epoch 4, step 420). On top, the input images and bellow the images in the other domain generated by the CycleGAN.



```
Epoch 0: Step 0: Generator (U-Net) loss: 2.5822078704833986, Discriminator loss: 0.07149550318717957
```

```
Epoch 0: Step 0: Generator (U-Net) loss: 1.1315640449523925, Discriminator loss: 0.03261364698410034
```

```
Epoch 0: Step 20: Generator (U-Net) loss: 2.7010715246200565, Discriminator loss: 0.25100582987070086
```

```
Epoch 0: Step 20: Generator (U-Net) loss: 6.6805512547492985, Discriminator loss: 0.3515582643449306
```

```
Epoch 0: Step 40: Generator (U-Net) loss: 2.1620802879333496, Discriminator loss: 0.18630461245775223
```

```
Epoch 0: Step 40: Generator (U-Net) loss: 2.5961476802825927, Discriminator loss: 0.19525868967175483
```

```
Epoch 0: Step 60: Generator (U-Net) loss: 2.023429465293884, Discriminator loss: 0.20994019955396653
```

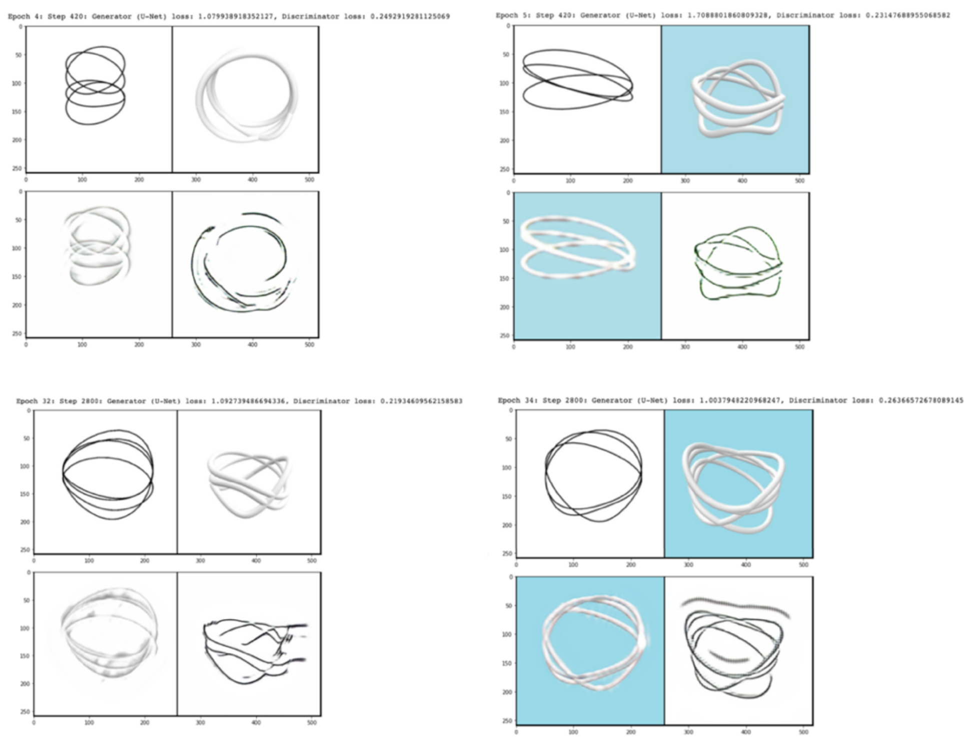
```
Epoch 0: Step 60: Generator (U-Net) loss: 2.361149507761002, Discriminator loss: 0.16148937791585924
```

Figure 32: Influence of the background color on the images of the first 60 steps of the training of the CycleGAN with white and blue backgrounds.

To solve this problem a new database with a colored background (blue color background images dataset) was created and the influence of the background on the development of the CycleGAN has been observed. Actually, in the previous picture, even in the first 60 steps of the training of GAN this influence can be seen.

Solving this problem with the background has been fundamental for the training, as no learning could be achieved with the white background. Indeed, if we look closely to the generated images, they were closer to the form of the shadows that objects standing over some surfaces create. Actually, if we looked closely to the images with the white background, it would also be impossible for a person to differentiate the line in which the ring ends, and the background starts.

This change on the color of the background allows the CycleGAN to be trained. In the following image, we can compare the training process of this CycleGAN with both of the databases and see how the first database, the one with the white background wouldn't permit the CycleGAN to reach a good style transfer level.



```
Epoch 4: Step 420: Generator (U-Net) loss: 1.079938918352127, Discriminator loss: 0.2492919281125069
Epoch 5: Step 420: Generator (U-Net) loss: 1.7088801860809328, Discriminator loss: 0.23147688955068582
Epoch 32: Step 2800: Generator (U-Net) loss: 1.092739486694336, Discriminator loss: 0.21934609562158583
Epoch 34: Step 2800: Generator (U-Net) loss: 1.0037948220968247, Discriminator loss: 0.26366572678089145
```

Figure 33: Training process steps 420 and 2800 with both background colors. The input images are followed by the images generated by the Cyclegan with the domain changed.

In the following parts of this Work, blue will be used in order to avoid white backgrounds. However, different blue tones will be found in the images, although all the backgrounds have the same blue color, #B9E2EA, the lighting will influence how the background is rendered.

4.2.4.2 *Sketch2rendering: wireframe thinckness problem*

Training the *sketch2rendering* CycleGAN some continuity problems on the sketches were found when the CycleGANs had to transfer the image from the sketch domain to the rendering domain. To solve this problem, it was decided to use thicker lines in the sketches and a new dataset was created. As it will be seen in future lines, going from thin lines to thicker ones permitted the CycleGAN to produce continuous rings

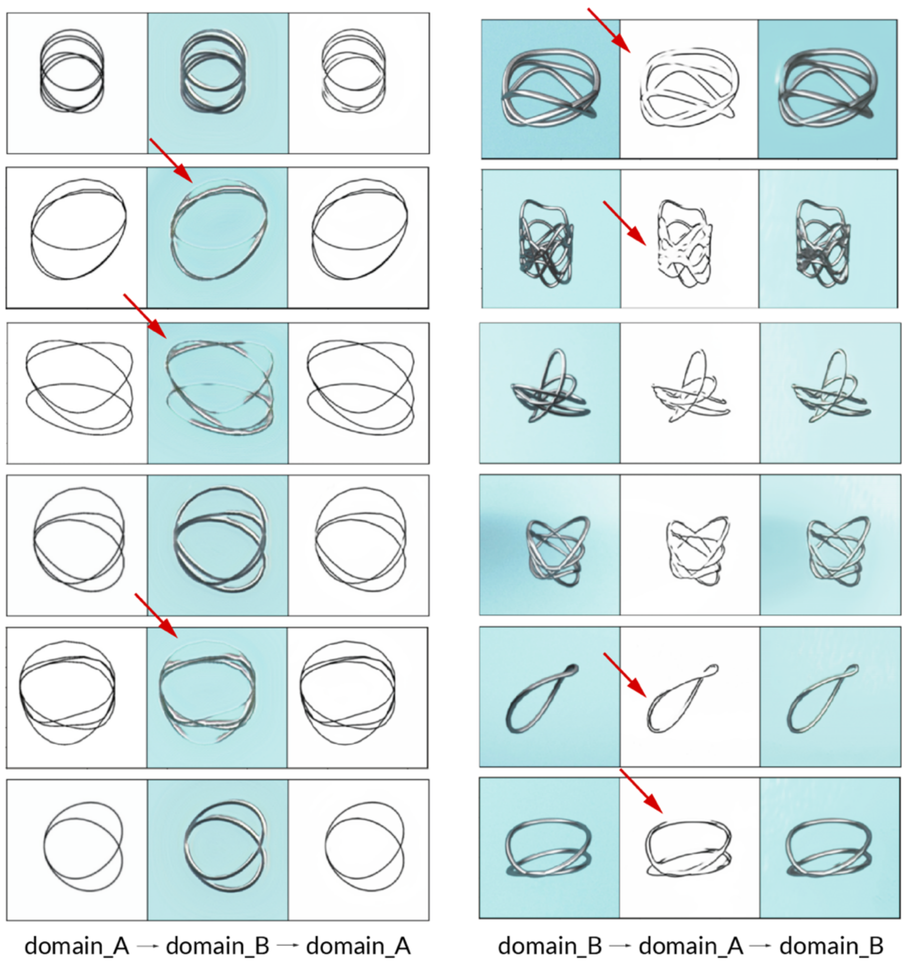

domain_A → domain_B → domain_A
domain_B → domain_A → domain_B

Figure 34: Continuity problems found during sketch2rendering CycleGAN training with the thin lines sketch.

5 Results

5.1.1 *Sketch2object3D*

Although the *sketch2object3D* CycleGAN is not the final objective of this work, training this CycleGAN has served for learning about the possibilities and limitations of style transfer. Training this first CycleGAN showed the importance of the background color. The results of this *sketch2object3D* CycleGAN can be seen in the following image. Cyclegan trained after

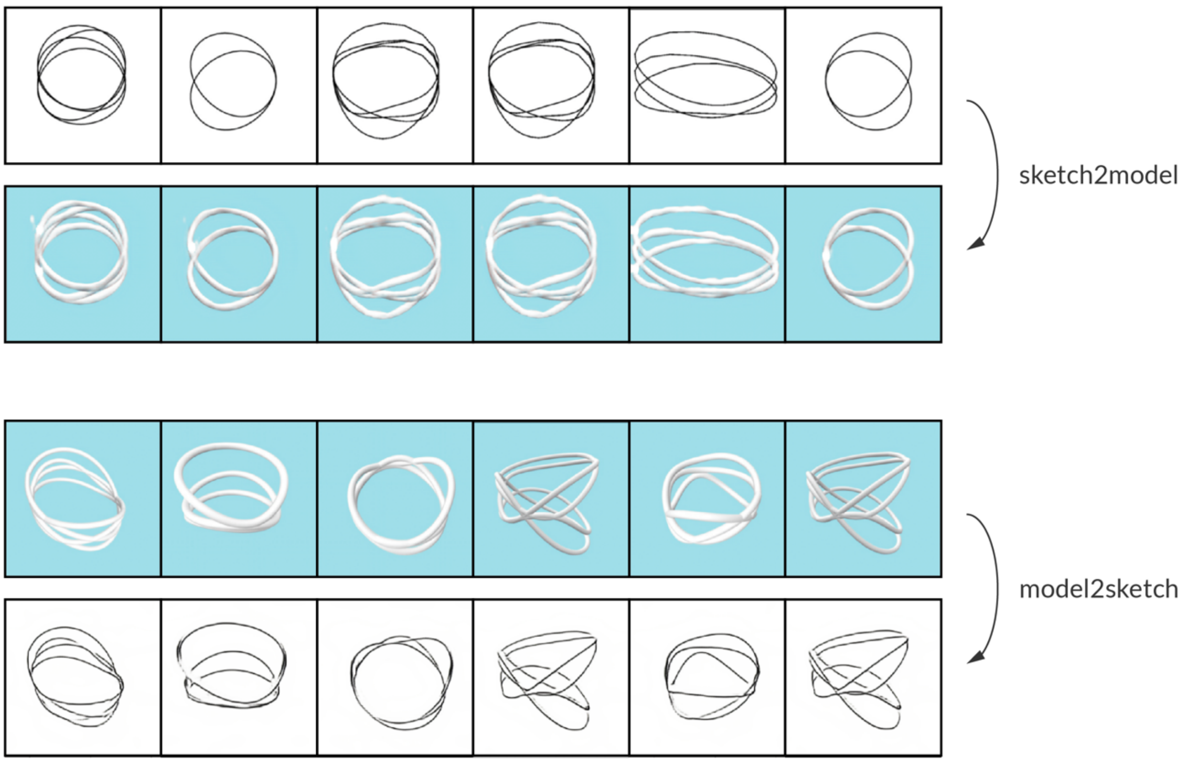


Figure 35: Collection of some results of the *sketch2object3D* CycleGAN.

5.1.2 *Sketch2rendering*

In the following image some examples of the style transfer by the *sketch2rendering* CycleGAN trained in this work are shown, in the first image, some of the results of the first trained CycleGAN are shown, the one trained with thin line sketches of the different rings. On the second image the actual CycleGAN outputs are shown.

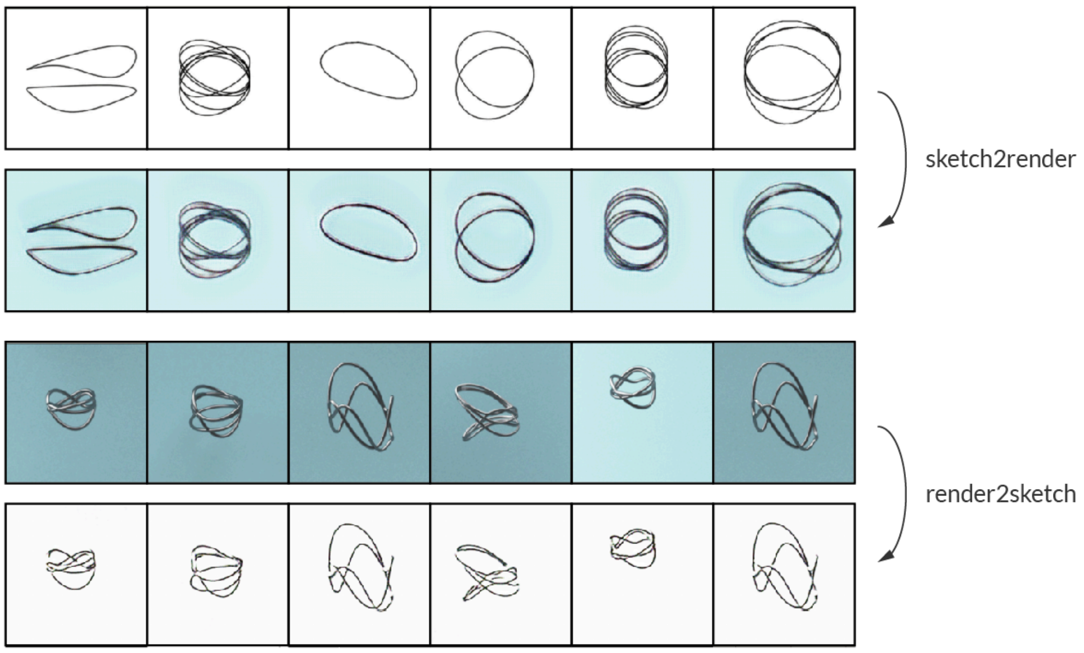


Figure 36: Collection of some results of the first trained *sketch2rendering.*

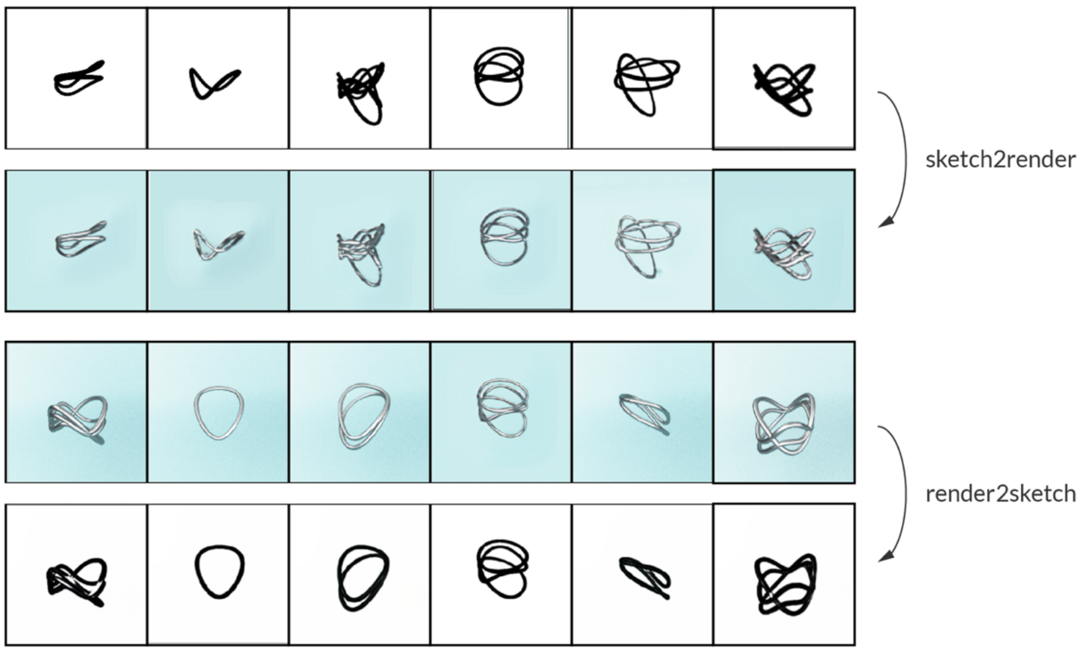


Figure 37: Collection of some results of the second trained *sketch2rendering*.

For comparison purposes, some of the ring sketches used to train the data have been modelled and rendered using the traditional procedure. In the following image, the outputs of the CycleGAN are show next to what they could be some expected rendered images using Maya modelling program and Blender rendering program. Although the data are unpaired images, in the following figure, the images are show as pairs of the input sketch and the generated image by the *sketch2rendering* CycleGAN and the rendered image using Blender.

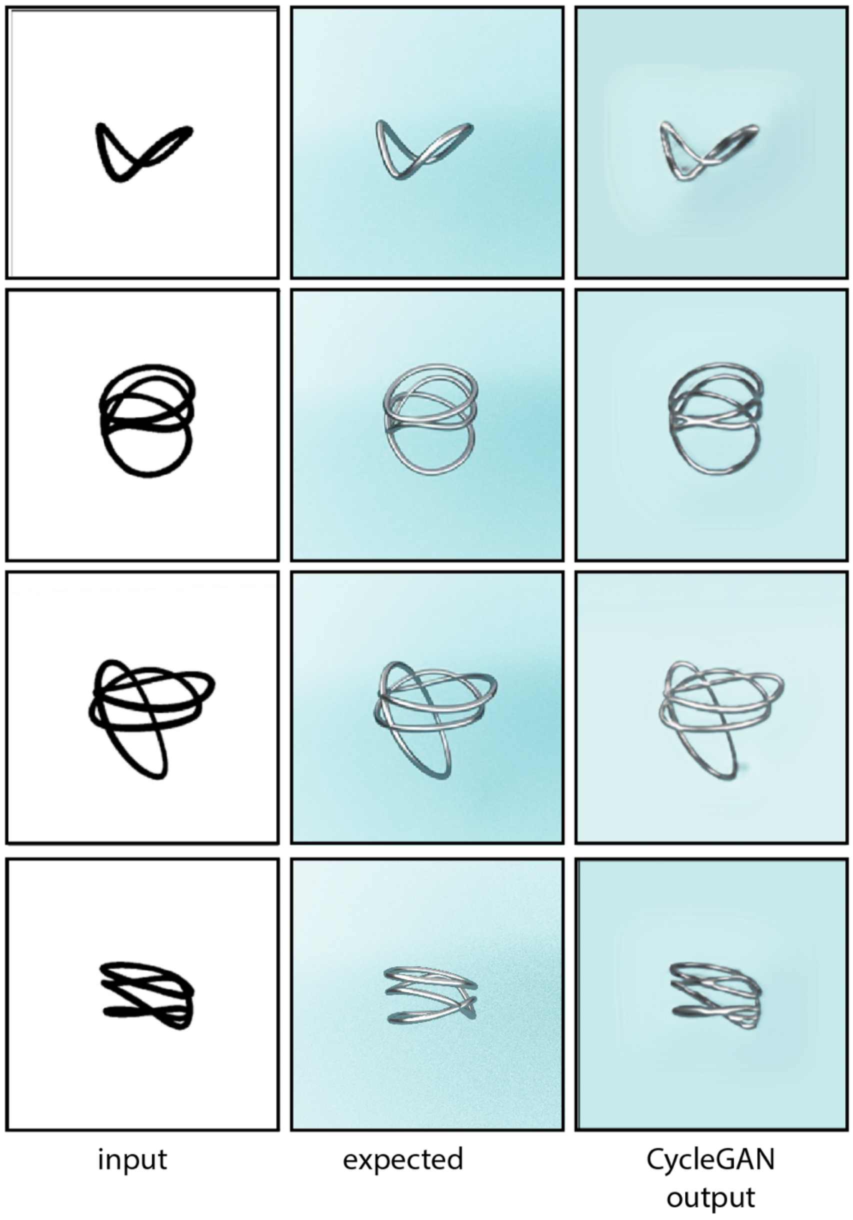


Figure 38: Different rings in the sketch domain and the rendered domain. The expected rendering has been generated with Blender and the other using the *sketch2rendering* CycleGAN.

In the following images 360 degrees of the same XYU ring can be seen. On the left, the input image (sketch image) is shown and next to it an expected rendered image of the 3D object using Blender can be seen and next to it, the output of the *sketch2rendering* CycleGAN.

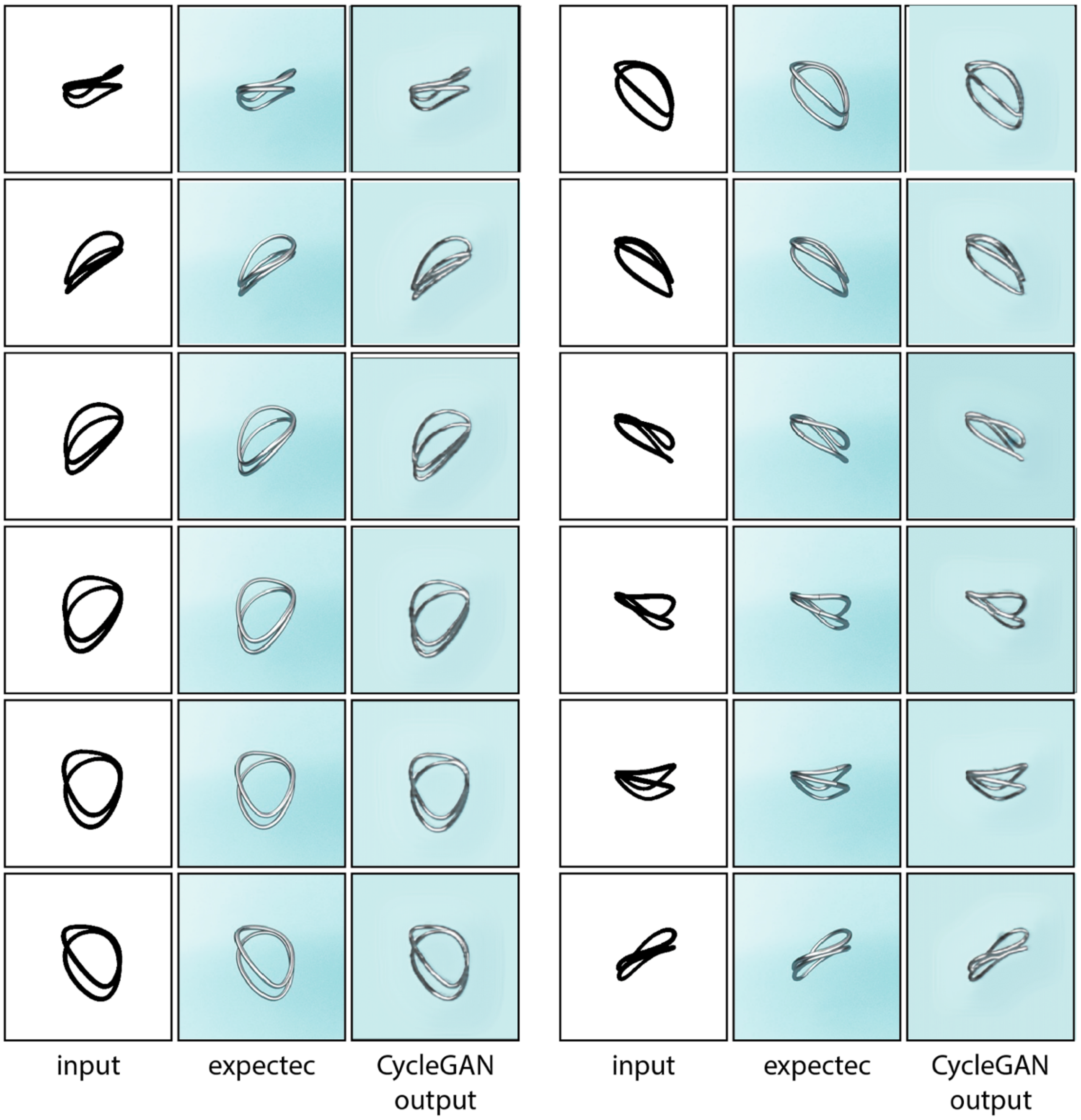


Figure 39: Different views of the same object in the sketch domain and the rendered domain. The expected rendering has been generated with Blender and the other has been generated by the *sketch2rendering* CycleGAN.

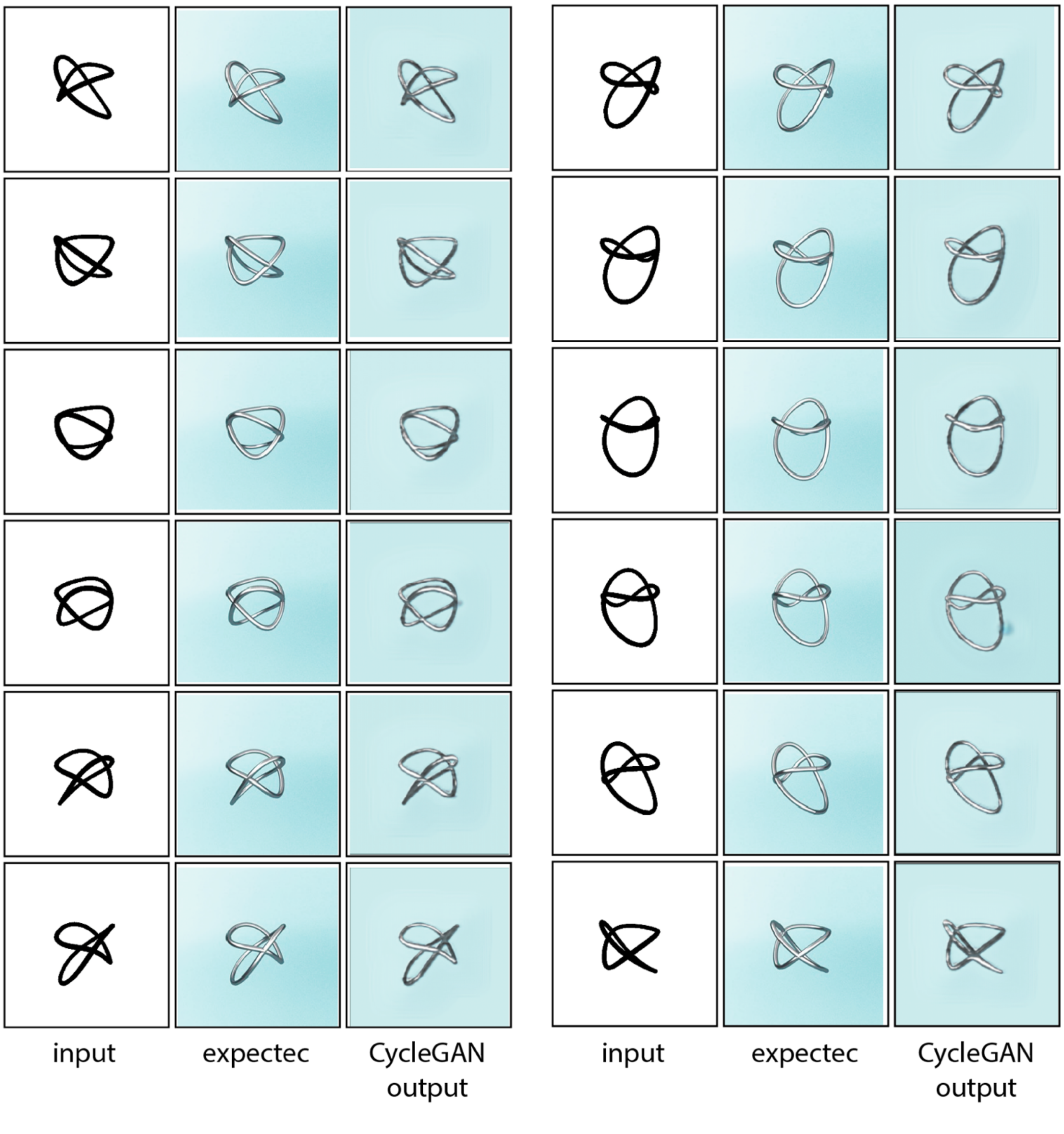


Figure 40: Different views of the same object in the sketch domain and the rendered domain. The expected rendering has been generated with Blender and the other has been generated by the *sketch2rendering* CycleGAN.

6 Discussion and future work

After having shown the possibilities of the applications of style transfers with CycleGANs with rendering purposes, in the following section, the artifacts found during the training and testing as well as some of the limitations found on the model will be considered.

Although the model can achieve reasonable results in some cases, there are areas for improvements in future works. As it can be seen in the following lines, the results are far from uniformly positive and there are still some challenges and improvements to be done before good quality realistic images of the rings are generated by the CycleGAN. The following artifacts have been found in both the training and the testing and solving them is important for future works.

6.1 Detected artifacts

6.1.1 White spots

During the training white blurry spots have been found on the output images. They have been usually found in the edge of the ring in areas where there is a strong shine on the ring or where the different bands of the ring intersected.

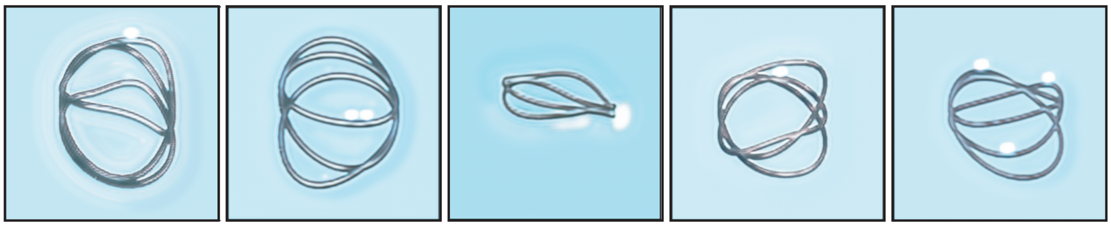


Figure 41: Examples of the white spots artifact.

6.1.2 *Continuity loss in the lines in the sketch transformation*

When the rendered images are transformed into the sketch domain, there is no continuity in the curves that form the ring due to the shiny parts of the rendered image. Although the domain change we are looking for in this Work is the sketch->rendering change, to train the CycleGAN the whole cycle is applied to the image, so solving this problem with the style transfer from the rendering domain to the sketch domain may be fundamental to obtain better results in the sketch->rendering transformation.

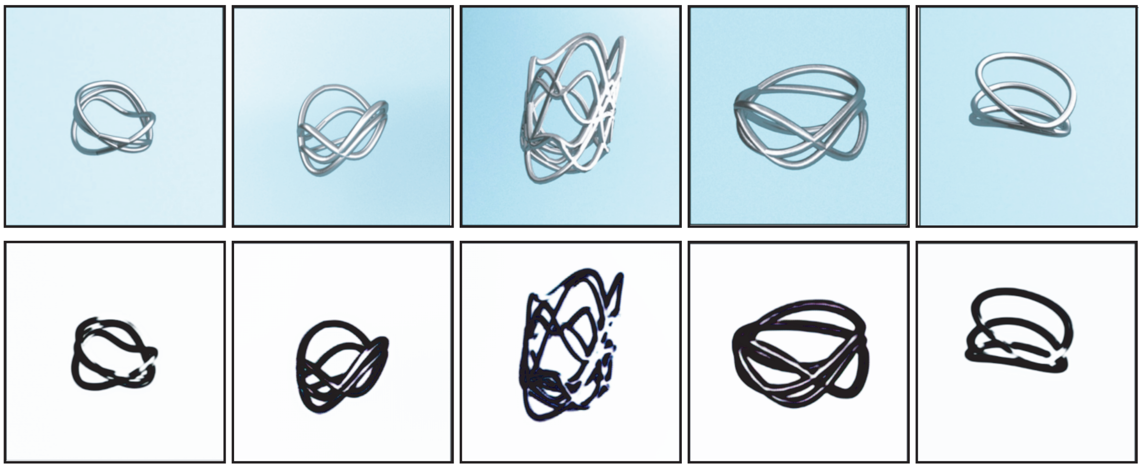


Figure 42: Examples of the continuity loss in the sketch domain transformation.

6.1.3 *Aureole around the ring*

This may be one of the most commonly found artifacts in the model. It is a gradient or aureole around the edges of the ring. Due to the different lighting settings, there is non uniform background color in the training dataset. Actually, the rendered images created using the Blender program show noise in the background, like if a photoshop *Film Grain* filter would have been applied to the background. This is due to the renderization parameters on Blender. In order to accelerate the renderization process, the number of calculation steps for the color of each pixel was reduced when the dataset was created. In order to see if this is the actual cause of this artifact, in future works a better-quality dataset should be created, not only for the rings themselves but also for the backgrounds.

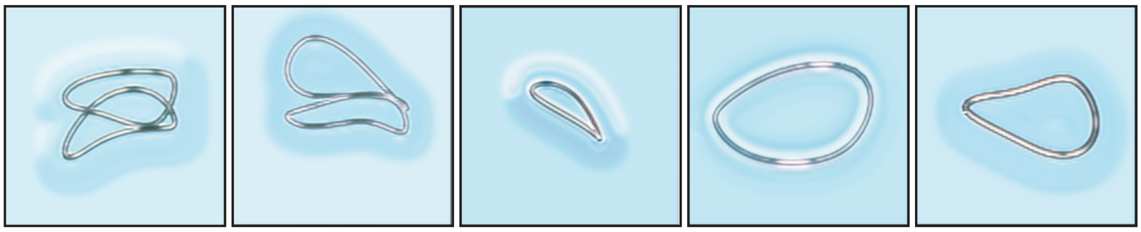


Figure 43: Examples of the aureole artifact.

6.1.4 Checkerboard pattern

This is one of the most typical artifacts in GANs, the reason for this checkboard like pattern in images is due to the upsampling process of the images from the latent space. This deconvolution “can easily have uneven overlap putting more of the metaphorical paint in some places than others” [41]. Solving this artifact may be on of the first problems to be tackled in future works.

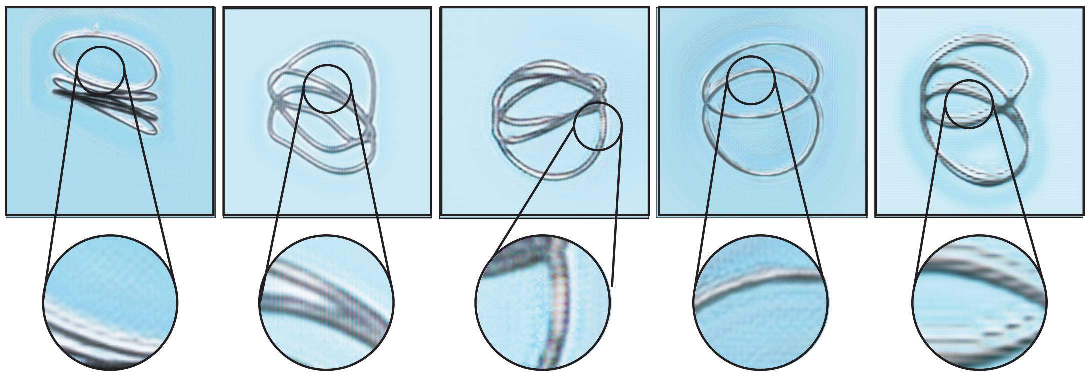


Figure 44: Examples of the checkerboard artifact.

6.2 Model limitations

Apart from the artifacts described in the previous section, some limitations of the actual model have been found, and solving them would need to change the model itself, for example using paired data for the lighting setting instruction to know where the light is coming form or change how the lines in the sketch intersect to show which one is on top of the others.

6.2.1 Lighting settings

In the following image, different lightning settings have been used in the rendering of the same object with the same materials. Therefore, different images have been created. In order to improve the training of the CycleGAN it would be better if always the same rendering settings are used so the CycleGAN is able to learn the rendering style. Another solution, as previously mentioned could be to used labeled data, adding information about the light position and direction, so the network can learn the differences. However, this would complicate the generation of the datasets and really precise information would be needed to make sure that all the information about the lighting settings is included in the labels.

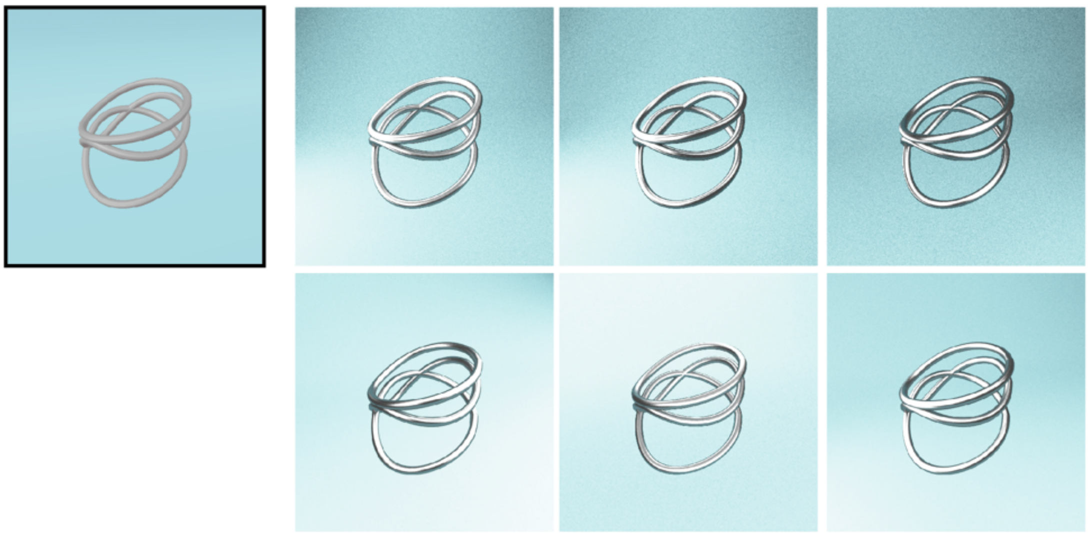


Figure 45: Examples of the influence of the lightning settings on the final render.

6.2.2 2D perspective

The sketch input image of the ring is an image of a 3D plot of the different splines that form the ring. Therefore, when in the 2D image two lines intersect; this may be because the lines actually intersect in the 3D space, or it may just be a consequence of the perspective. When created a plot of a 3D object, some of the information is lost, so there is no way for the CycleGAN to know which of the intersecting lines in the image is on top of which or if they are actually intersecting. A good way to solve this could be to a different representation for intersecting lines and the ones that aren't, for example, the diagrams used for knot representation in the study of mathematical knots [42] could be used.

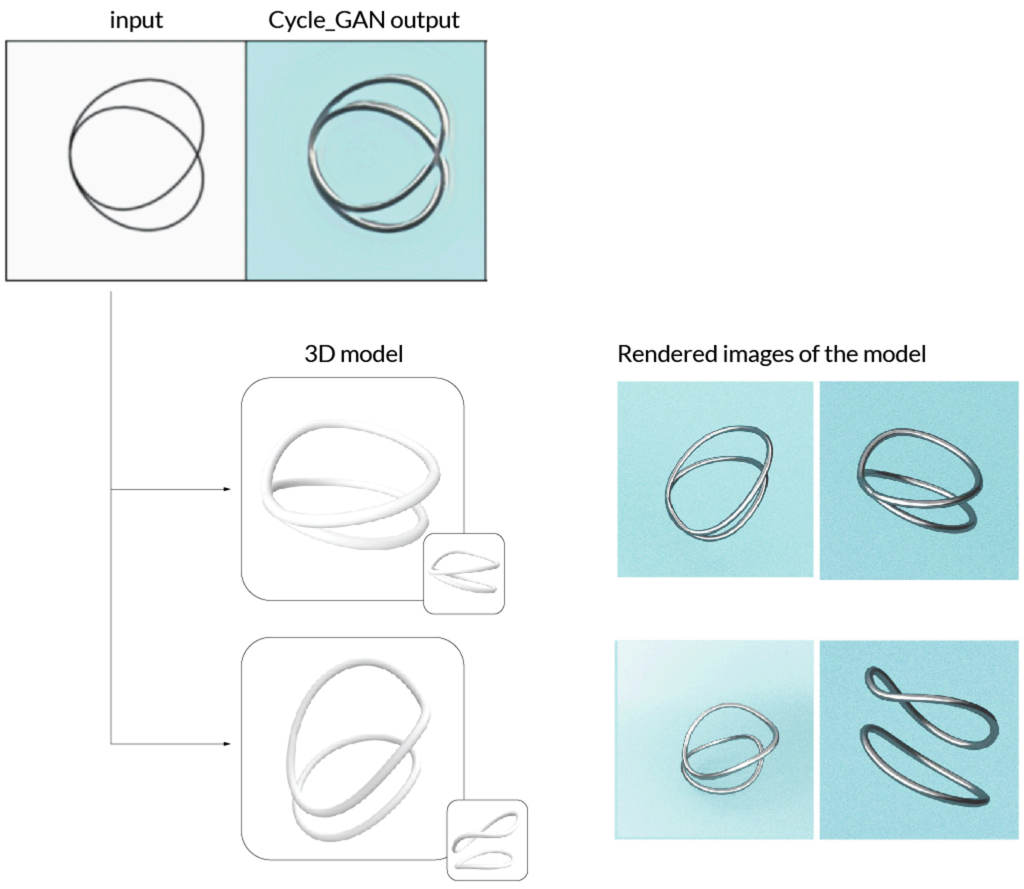


Figure 46: Example of two lines intersecting in the 2D image, and two different 3D examples of the actually intersecting and non intersecting cases.

7 Conclusions

After having presented the obtained results in the CycleGAN training and having discussed the problems encountered, some conclusions around the initially set objectives have been made. These conclusions are discussed in the following lines

First of all, it is concluded that the model can achieve compiling results for the rendering style transfer, that was the initially objective. Nevertheless, as it has been seen, there are areas for improvement in future works before high quality realistic images of the ring examples are generated by the CycleGAN. However, the results obtained exceed the initial expectations for this Work. Actually, this new model supposes a new approach for the XYU ring algorithm and even though perfect results have not been obtained, the reduction of the time and the allowance of a complete automatization of the different ring design generation, makes this Work the perfect starting point for future research and improvements.

Secondly, this Work shows the possibilities of the intersection of computation and design, an intersection that allows the designers to focus on what really matters while the algorithms do the repetitive work. The rendering style transfer supposes going from the rendering of images that could take up to one hour on the making, to renders generated by the CycleGAN in seconds.

Therefore, it can be concluded that the research objective has been achieved, having developed a software that is capable of transferring the rendering style to the initial sketches of the ring. The contribution of this work to the XYU ring design generation algorithm supposes an inflexion point for the way that the rings are shown to the costumer, who now would be able to see real time rendered images of the ring that is generating while interacting with the algorithm.

With regards to the initial planification, it can be concluded that is has been adequate and has managed to meet the milestones proposed. Some delay was suffered in the third milestone with the code development and the problems encountered with the databases, however, in the last part of the elaboration of this Work the delay has been overcome. Therefore, the initial objectives of this Work have been achieved on time.

Last of all, in the previous chapter the problems encountered during the making of this Work have been discussed and they suppose the challenges to be solved in future works. Apart from these, some quantitative results for this model should be done. In future work, other types of GANs could be trained like pix2pix, BiGAN or StyleGAN and train them with paired data to compare them with this model. This quantitative comparison with other methods would help decide if CycleGANs that do not need of paired data, as in this case, are the best approach for this problem, or preparing a paired data to train an image to images translation GAN is worth the time and the effort.

8 Glossary of terms

AI	Artificial Intelligence
CNN	Convolutional Neural Network
GAN	Generative Adversarial Network
VAE	Variational Autoencoders
FMP	Final Master Project
UOC	Universitat Oberta de Catalunya
BCE	The Binary Cross Entropy
ReLu	Rectified Linear Activation
LReLu	Leaky Rectified Linear Activation

9 Bibliography

[1] The Turing Digital Archive. (n.d.), from http://www.turingarchive.org/viewer/?id=167&title=1a, accessed 23-3-2021

[2] WHY PROGRAMMING IS A GOOD MEDIUM FOR EXPRESSING POORLY UNDERSTOOD AND SLOPPILY-FORMULATED IDEAS. (n.d.). *1967*, from https://web.media.mit.edu/~minsky/papers/Why programming is--.html, accessed 25-3-2021

[3] Arca will use AI to soundtrack NYC's Museum of Modern Art | Engadget. (n.d.), from https://www.engadget.com/2019-10-17-arca-ai-soundtrack-for-nyc-moma.html, accessed 25-3-2021

[4] 'Untitled Computer Drawing', Harold Cohen, 1982 | Tate. (n.d.), from https://www.tate.org.uk/art/artworks/cohen-untitled-computer-drawing-t04167, accessed 25-3-2021

[5] Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. (2020). Generative adversarial networks, *Communications of the ACM*, Vol. 63, No. 11, 139–144. doi:10.1145/3422622

[6] Design Thinking. (n.d.), from https://hbr.org/2008/06/design-thinking, accessed 24-5-2021

[7] Why IKEA Uses 3D Renders vs. Photography for Their Furniture Catalog | Cad Crowd. (n.d.), from https://www.cadcrowd.com/blog/why-ikea-uses-3d-renders-vs-photography-for-their-furniture-catalog/, accessed 22-5-2021

[8] Brock, A.; Donahue, J.; Simonyan, K. (2018). Large Scale GAN Training for High Fidelity Natural Image Synthesis, *ArXiv*

[9] Reed, S.; Akata, Z.; Yan, X.; Logeswaran, L.; Schiele, B.; Lee, H. (2016). Generative Adversarial Text to Image Synthesis, *33rd International Conference on Machine Learning, ICML 2016*, Vol. 3, 1681–1690

[10] Isola, P.; Zhu, J.-Y.; Zhou, T.; Efros, A. A.; Research, B. A. (n.d.). *Image-to-Image Translation with Conditional Adversarial Networks*

[11] Alibaba Luban: AI-based Graphic Design Tool | by Alibaba Cloud | Medium. (n.d.), from https://alibaba-cloud.medium.com/alibaba-luban-ai-based-graphic-design-tool-75dbb94a2115, accessed 25-5-2021

[12] We optimize recovery management with Artificial Intelligence. (n.d.), from https://the-cocktail.com/en/cases/we-optimize-recovery-management-with-artificial-intelligence, accessed 26-5-2021

[13] IBM100 - Deep Blue. (n.d.), from https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/, accessed 26-5-2021

[14] How does Instagram determine which posts appear in Suggested Posts? | Instagram Help Center. (n.d.), from https://www.facebook.com/help/instagram/381638392275939, accessed 26-5-2021

[15] IanGoodfellow PhD Defense Presentation - YouTube. (n.d.), from https://www.youtube.com/watch?v=ckoD_bE8Bhs, accessed 29-5-2021

[16] Chollet, F. (2017). Deep Learning with Python

[17] Autoencoders — Deep Learning bits #1 | Hacker Noon. (n.d.), from https://hackernoon.com/autoencoders-deep-learning-bits-1-11731e200694, accessed 25-3-2021

[18] Understanding Latent Space in Machine Learning | by Ekin Tiu | Towards Data Science. (n.d.), from https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d, accessed 25-3-2021

[19] Latent space visualization — Deep Learning bits #2 | Hacker Noon. (n.d.), from https://hackernoon.com/latent-space-visualization-deep-learning-bits-2-bd09a46920df, accessed 25-3-2021

[20] Farnia, F.; Ozdaglar, A. (2020). GANs May Have No Nash Equilibria

[21] Generator | Coursera. (n.d.), from https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/lecture/DkMQx/generator, accessed 28-5-2021

[22] Discriminator | Coursera. (n.d.), from https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/lecture/DdSCm/discriminator, accessed 28-5-2021

[23] BCE Cost Function | Coursera. (n.d.), from https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/lecture/2bF5q/bce-cost-function, accessed 28-5-2021

[24] BCE Cost Function | Coursera. (n.d.), from https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/lecture/2bF5q/bce-cost-function, accessed 29-5-2021

[25] Common Activation Functions | Coursera. (n.d.), from https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/lecture/WYsEO/common-activation-functions, accessed 29-5-2021

[26] Commonly used activation functions: (a) Sigmoid, (b) Tanh, (c) ReLU,... | Download Scientific Diagram. (n.d.), from https://www.researchgate.net/figure/Commonly-used-activation-functions-a-Sigmoid-b-Tanh-c-ReLU-and-d-LReLU_fig3_335845675, accessed 28-5-2021

[27] 6 basic things to know about Convolution | by Madhushree Basavarajaiah | Medium. (n.d.), from https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411, accessed 28-5-2021

[28] Review of Convolutions | Coursera. (n.d.), from https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/lecture/W6bPB/review-of-convolutions, accessed 29-5-2021
[29] Mirza, M.; Osindero, S. (2014). Conditional Generative Adversarial Nets
[30] Karras, T.; Laine, S.; Aila, T. (2019). A style-based generator architecture for generative adversarial networks, *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Vol. 2019-June), IEEE Computer Society, 4396–4405. doi:10.1109/CVPR.2019.00453
[31] Pix2Pix Overview | Coursera. (n.d.), from https://www.coursera.org/learn/apply-generative-adversarial-networks-gans/lecture/X9EOT/pix2pix-overview, accessed 29-5-2021
[32] Zhu, J.-Y.; Park, T.; Isola, P.; Efros, A. A.; Research, B. A. (n.d.). *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks Monet Photos*
[33] Langr, J.; Bok, V. (2019). *GANs in Action*
[34] CycleGAN: Identity Loss | Coursera. (n.d.), from https://www.coursera.org/learn/apply-generative-adversarial-networks-gans/lecture/RlHCn/cyclegan-identity-loss, accessed 29-5-2021
[35] Pix2Pix: U-Net | Coursera. (n.d.), from https://www.coursera.org/learn/apply-generative-adversarial-networks-gans/lecture/Xr0NK/pix2pix-u-net, accessed 29-5-2021
[36] Pix2Pix: PatchGAN | Coursera. (n.d.), from https://www.coursera.org/learn/apply-generative-adversarial-networks-gans/lecture/jSbor/pix2pix-patchgan, accessed 29-5-2021
[37] Introduction — Blender Manual. (n.d.), from https://docs.blender.org/manual/en/latest/render/introduction.html, accessed 1-6-2021
[38] Rendering | The Science Behind Pixar. (n.d.), from https://sciencebehindpixar.org/pipeline/rendering, accessed 1-6-2021
[39] CycleGAN Project Page. (n.d.), from https://junyanz.github.io/CycleGAN/, accessed 22-5-2021
[40] PyTorch-CycleGAN/datasets.py at master · aitorzip/PyTorch-CycleGAN. (n.d.), from https://github.com/aitorzip/PyTorch-CycleGAN/blob/master/datasets.py, accessed 22-5-2021
[41] Odena, A.; Dumoulin, V.; Olah, C. (2017). Deconvolution and Checkerboard Artifacts, *Distill*, Vol. 1, No. 10, e3. doi:10.23915/distill.00003
[42] Knot theory - Wikipedia. (n.d.), from https://en.wikipedia.org/wiki/Knot_theory, accessed 4-6-2021

10 Attachments

10.1 Databases

Some randomly selected .jpg images from the different datasets generated for this work are shown in this section. The aim of this section is to show the images that have been used in the training of the CycleGANs.

10.1.1 Sketch dataset (first version, thin wire sketch)

These .jpg images have been generated using Matlab program and the XYU ring algorithm. Actually, these images are a 3D plot of the splines that compose each of the rings, all with the same line thickness.

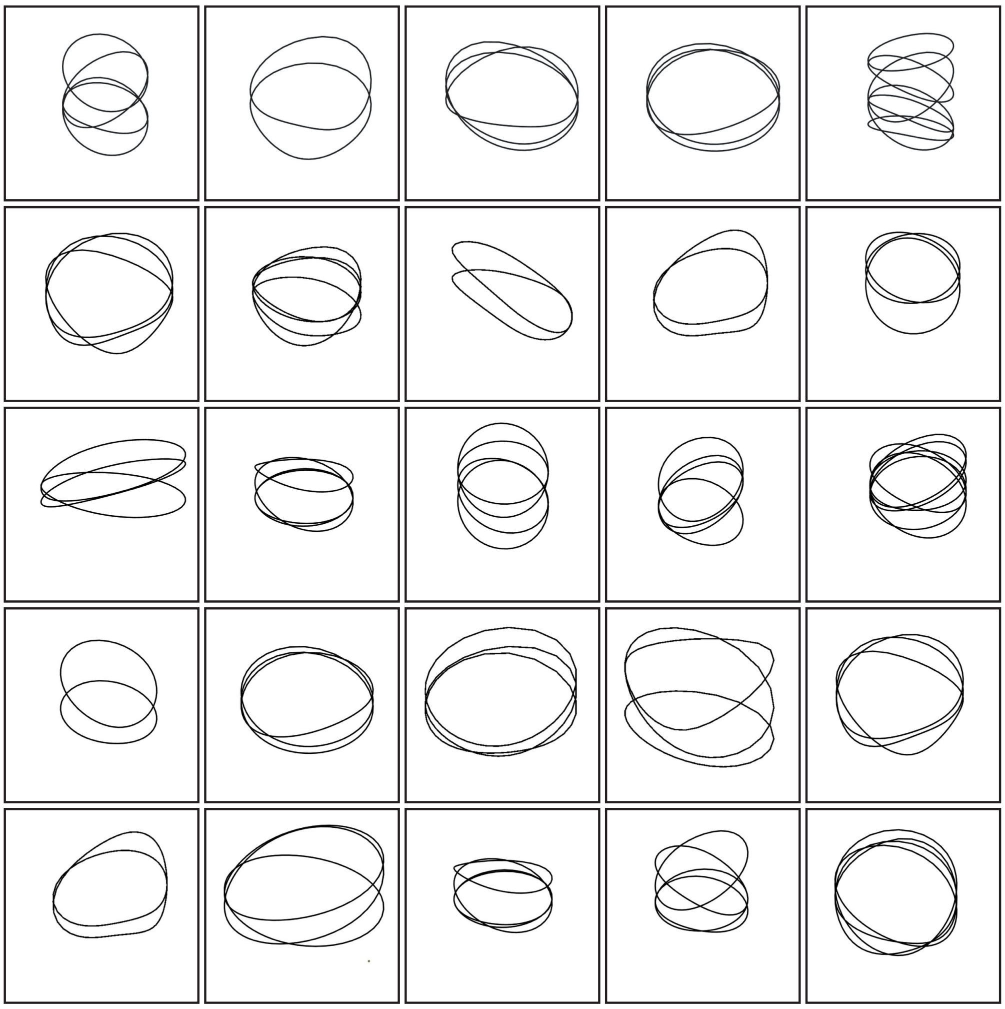


Figure 47: Random images of the thin wire sketch dataset.

10.1.2 Sketch dataset (second version, thick wire sketch)

These .jpg images have been created using the same algorithm as the previous dataset, however, when doing the 3D plot thicker lines have been used. The thickness has been varied to show different ring thicknesses.

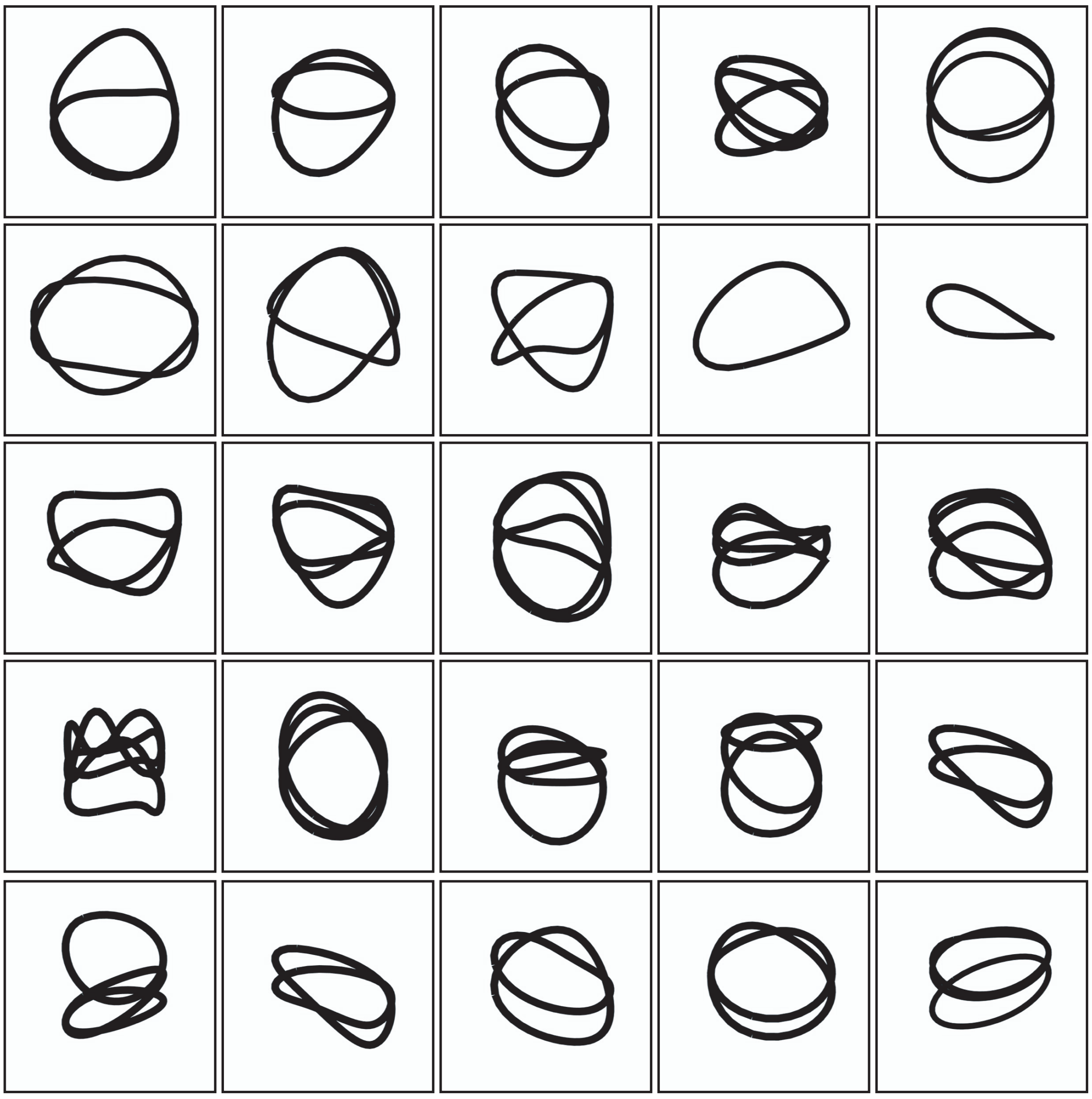


Figure 48: Random images of the thick wire sketch dataset.

10.1.3 Object3D dataset (first version, white background)

These images are screenshots of the of the 3D visualization of the ring 3D objects (.obj files) in the MacOS preview application.

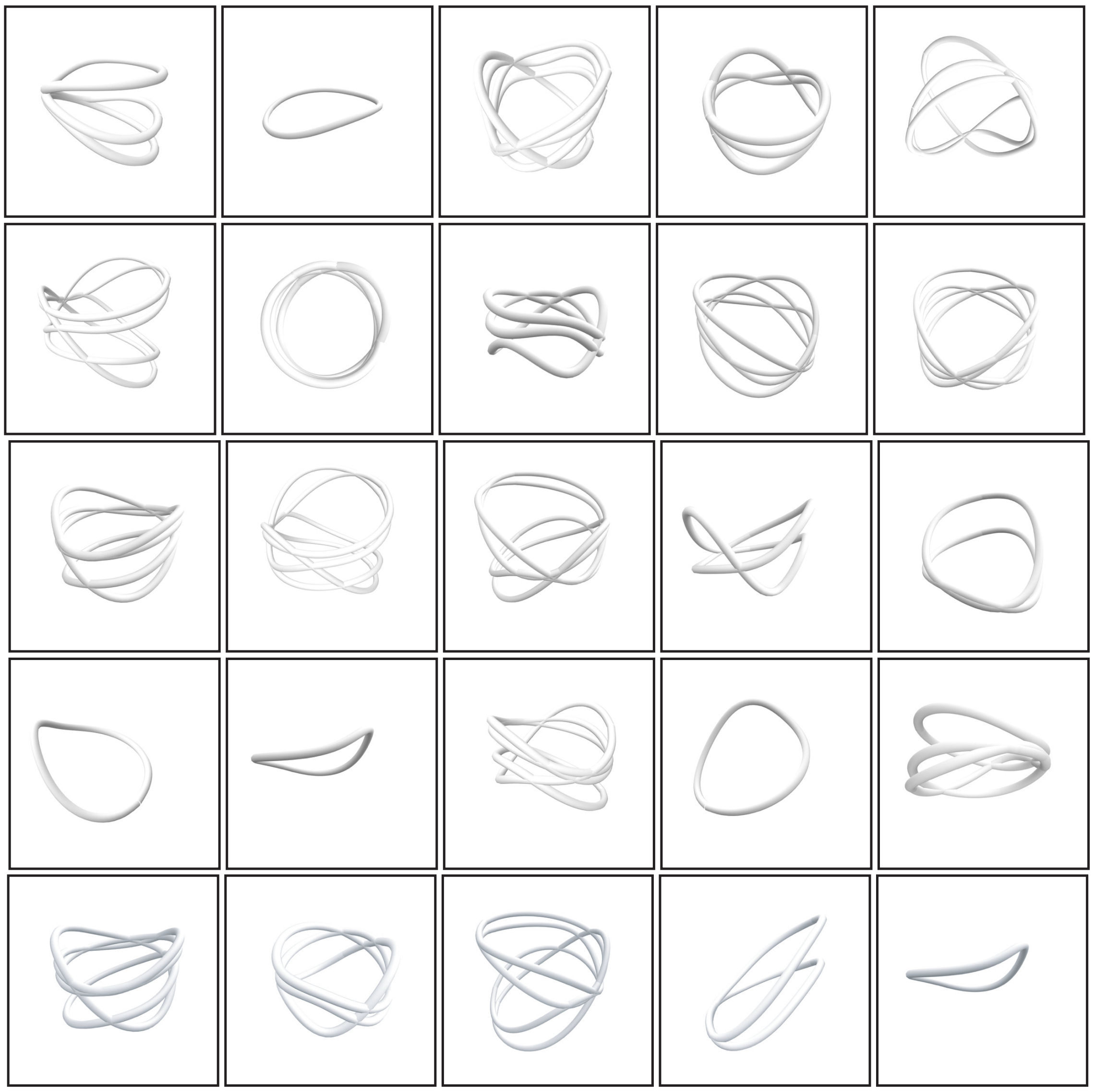


Figure 49: Random images of the white background model dataset.

10.1.4 Object3D dataset (second version, blue background)

As in the previous dataset, these images are screenshots of the 3D visualization of the XYU ring 3D objects, however, this time the background has been set to blue color, in order to have a non-white background.

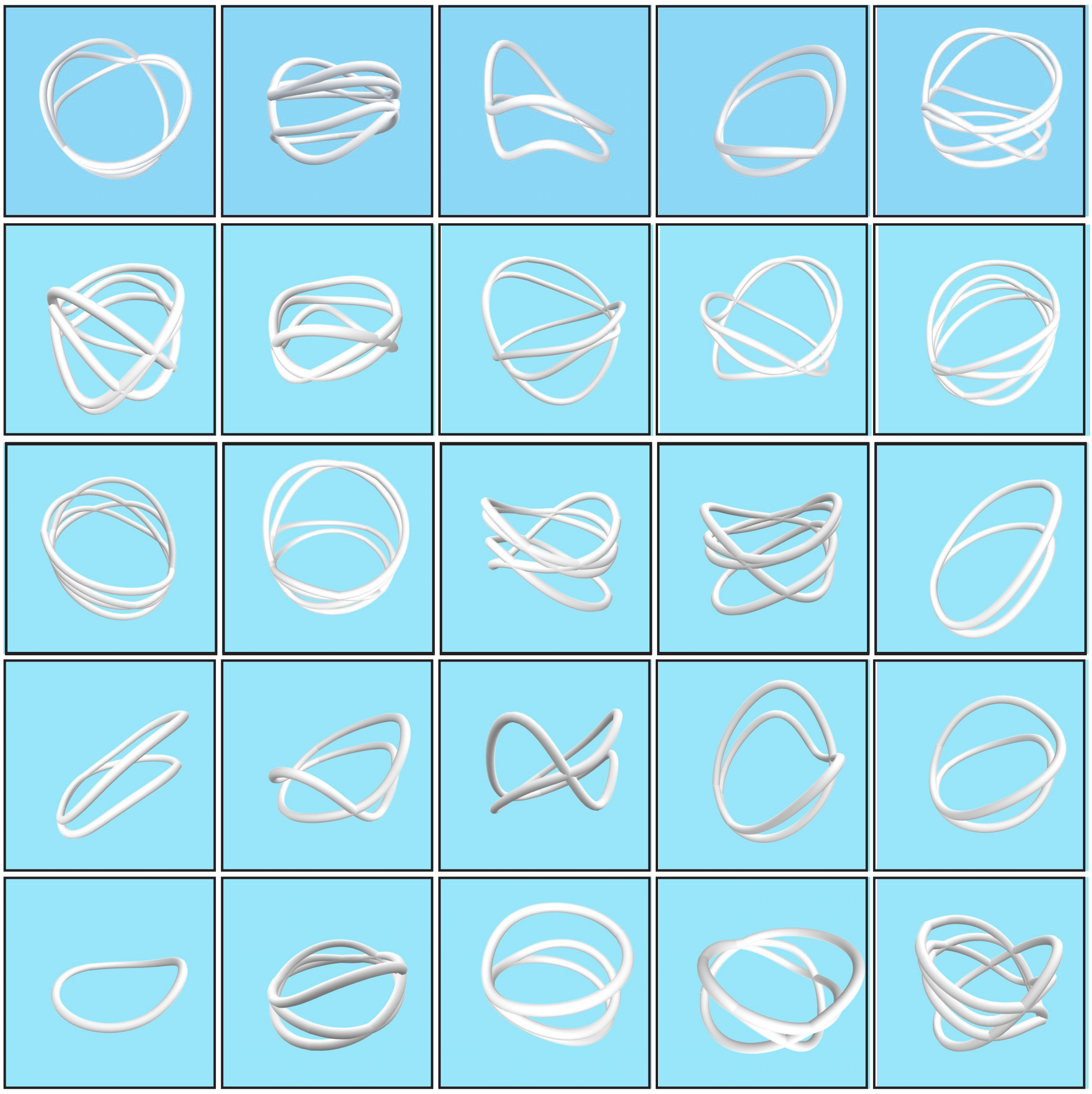


Figure 50: Random images of the blue color background model dataset.

10.1.5 Rendering dataset

These images have been created using the Blender rendering program. As it can be seen, although the background color has always been the same blue #B9E2EA, the lighting setting has changed as well as the camera position and orientation, therefore, different shadow and lights can be seen across the dataset.

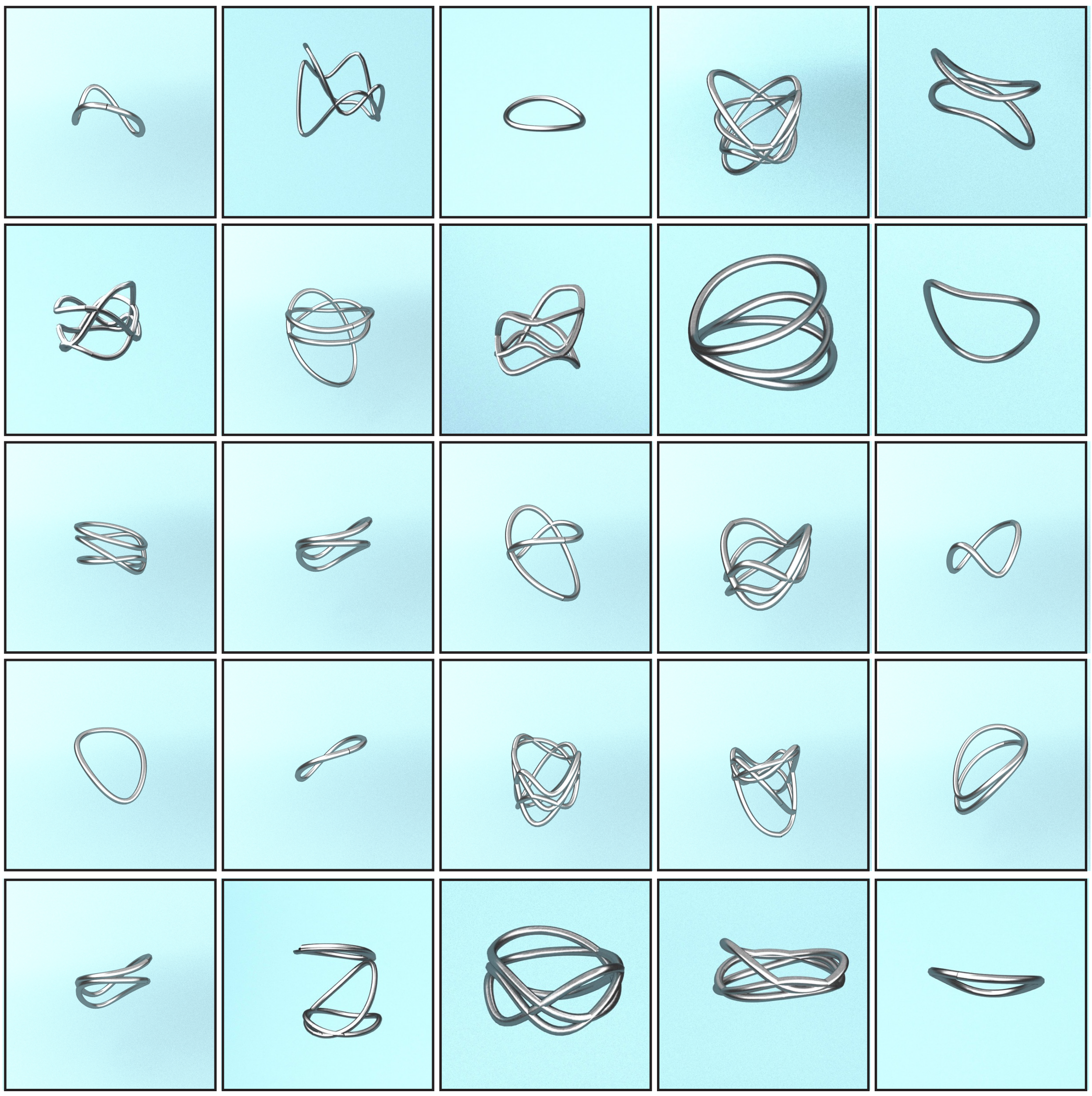


Figure 51: Random images from the rendering dataset.

10.2 Code

```
#Start by connecting gdrive into the google colab
from google.colab import drive
drive.mount('/content/gdrive')

#file in which the sketch2render folder is saved
%cd /content/gdrive/MyDrive/TFM /

import torch
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

def show_tensor_images(image_tensor, num_images=25, size=(1, 28, 28)):

    image_tensor = (image_tensor + 1) / 2
    image_shifted = image_tensor
    image_unflat = image_shifted.detach().cpu().view(-1, *size)
    image_grid = make_grid(image_unflat[:num_images], nrow=5)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.axis('off')
    plt.show()


import glob
import random
import os
from torch.utils.data import Dataset
from PIL import Image

class ImageDataset(Dataset):
    def __init__(self, root, transform=None, mode='train'):
        self.transform = transform
        self.files_A = sorted(glob.glob(os.path.join(root, '%sA' % mode) + '/*.*'))
        self.files_B = sorted(glob.glob(os.path.join(root, '%sB' % mode) + '/*.*'))
        self.new_perm()
        assert len(self.files_A) > 0, "Make sure you downloaded the images!"

    def new_perm(self):
        self.randperm = torch.randperm(len(self.files_B))[:len(self.files_A)]

    def __getitem__(self, index):
        item_A = self.transform(Image.open(self.files_A[index % len(self.files_A)]))
        item_B = self.transform(Image.open(self.files_B[self.randperm[index]]))
        if item_A.shape[0] != 3:
            item_A = item_A.repeat(3, 1, 1)
        if item_B.shape[0] != 3:
            item_B = item_B.repeat(3, 1, 1)
        if index == len(self) - 1:
            self.new_perm()
        return (item_A - 0.5) * 2, (item_B - 0.5) * 2

    def __len__(self):
        return min(len(self.files_A), len(self.files_B))
```

- CycleGAN generator

Residual Block

```
class ResidualBlock(nn.Module):
    def __init__(self, input_channels):
        super(ResidualBlock, self).__init__()
        self.conv1  =  nn.Conv2d(input_channels,  input_channels,  kernel_size=3,  padding=1,
        padding_mode='reflect')
        self.conv2  =  nn.Conv2d(input_channels,  input_channels,  kernel_size=3,  padding=1,
        padding_mode='reflect')
        self.instancenorm = nn.InstanceNorm2d(input_channels)
        self.activation = nn.ReLU()

    def forward(self, x):
        original_x = x.clone()
        x = self.conv1(x)
        x = self.instancenorm(x)
        x = self.activation(x)
        x = self.conv2(x)
        x = self.instancenorm(x)
        return original_x + x
```

Contracting and Expanding Blocks

```
class ContractingBlock(nn.Module):

    def __init__(self, input_channels, use_bn=True, kernel_size=3, activation='relu'):
        super(ContractingBlock, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, input_channels * 2, kernel_size=kernel_size,
        padding=1, stride=2, padding_mode='reflect')
        self.activation = nn.ReLU() if activation == 'relu' else nn.LeakyReLU(0.2)
        if use_bn:
            self.instancenorm = nn.InstanceNorm2d(input_channels * 2)
        self.use_bn = use_bn

    def forward(self, x):
        x = self.conv1(x)
        if self.use_bn:
            x = self.instancenorm(x)
        x = self.activation(x)
        return x

class ExpandingBlock(nn.Module):

    def __init__(self, input_channels, use_bn=True):
        super(ExpandingBlock, self).__init__()
        self.conv1 = nn.ConvTranspose2d(input_channels, input_channels // 2, kernel_size=3,
        stride=2, padding=1, output_padding=1)
        if use_bn:
            self.instancenorm = nn.InstanceNorm2d(input_channels // 2)
        self.use_bn = use_bn
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        if self.use_bn:
            x = self.instancenorm(x)
        x = self.activation(x)
        return x

class FeatureMapBlock(nn.Module):

    def __init__(self, input_channels, output_channels):
        super(FeatureMapBlock, self).__init__()
        self.conv = nn.Conv2d(input_channels, output_channels, kernel_size=7, padding=3,
        padding_mode='reflect')

    def forward(self, x):

        x = self.conv(x)
        return x
```

Building the generator

```
class Generator(nn.Module):

    def __init__(self, input_channels, output_channels, hidden_channels=64):
        super(Generator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels)
        self.contract2 = ContractingBlock(hidden_channels * 2)
        res_mult = 4
        self.res0 = ResidualBlock(hidden_channels * res_mult)
        self.res1 = ResidualBlock(hidden_channels * res_mult)
        self.res2 = ResidualBlock(hidden_channels * res_mult)
        self.res3 = ResidualBlock(hidden_channels * res_mult)
        self.res4 = ResidualBlock(hidden_channels * res_mult)
        self.res5 = ResidualBlock(hidden_channels * res_mult)
        self.res6 = ResidualBlock(hidden_channels * res_mult)
        self.res7 = ResidualBlock(hidden_channels * res_mult)
        self.res8 = ResidualBlock(hidden_channels * res_mult)
        self.expand2 = ExpandingBlock(hidden_channels * 4)
        self.expand3 = ExpandingBlock(hidden_channels * 2)
        self.downfeature = FeatureMapBlock(hidden_channels, output_channels)
        self.tanh = torch.nn.Tanh()

    def forward(self, x):
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.res0(x2)
        x4 = self.res1(x3)
        x5 = self.res2(x4)
        x6 = self.res3(x5)
        x7 = self.res4(x6)
        x8 = self.res5(x7)
        x9 = self.res6(x8)
        x10 = self.res7(x9)
        x11 = self.res8(x10)
        x12 = self.expand2(x11)
        x13 = self.expand3(x12)
        xn = self.downfeature(x13)
        return self.tanh(xn)
```

- CycleGAN Discriminator

```
class Discriminator(nn.Module):
    def __init__(self, input_channels, hidden_channels=64):
        super(Discriminator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1   =   ContractingBlock(hidden_channels,   use_bn=False,   kernel_size=4,
        activation='lrelu')
        self.contract2     =     ContractingBlock(hidden_channels     *     2,     kernel_size=4,
        activation='lrelu')
        self.contract3     =     ContractingBlock(hidden_channels     *     4,     kernel_size=4,
        activation='lrelu')
        self.final = nn.Conv2d(hidden_channels * 8, 1, kernel_size=1)

    def forward(self, x):
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.contract3(x2)
        xn = self.final(x3)
        return xn
```

- Training Parameters

```
import torch.nn.functional as F

adv_criterion = nn.MSELoss()
recon_criterion = nn.L1Loss()
n_epochs = 100
dim_A = 3
dim_B = 3
display_step = 50
save_step=50
step_bins = 10 #for plotting
batch_size = 1
lr = 0.0002
load_shape = 400
device = 'cuda'
```

```
transform = transforms.Compose([
    transforms.Resize(load_shape),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
])

import torchvision
dataset = ImageDataset("sketchThickness2renderColorBlue_REDUCED", transform=transform)
```

- Building the network

```
gen_AB = Generator(dim_A, dim_B).to(device)
gen_BA = Generator(dim_B, dim_A).to(device)
gen_opt   =   torch.optim.Adam(list(gen_AB.parameters())   +   list(gen_BA.parameters()),   lr=lr,
betas=(0.5, 0.999))
disc_A = Discriminator(dim_A).to(device)
disc_A_opt = torch.optim.Adam(disc_A.parameters(), lr=lr, betas=(0.5, 0.999))
disc_B = Discriminator(dim_B).to(device)
disc_B_opt = torch.optim.Adam(disc_B.parameters(), lr=lr, betas=(0.5, 0.999))

def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
    if isinstance(m, nn.BatchNorm2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
        torch.nn.init.constant_(m.bias, 0)

# Feel free to change pretrained to False if you're training the model from scratch
pretrained = False
if pretrained:
    pre_dict = torch.load('Models/lr_0.000002/cycleGAN_850.pth')
    gen_AB.load_state_dict(pre_dict['gen_AB'])
    gen_BA.load_state_dict(pre_dict['gen_BA'])
    gen_opt.load_state_dict(pre_dict['gen_opt'])
    disc_A.load_state_dict(pre_dict['disc_A'])
    disc_A_opt.load_state_dict(pre_dict['disc_A_opt'])
    disc_B.load_state_dict(pre_dict['disc_B'])
    disc_B_opt.load_state_dict(pre_dict['disc_B_opt'])
else:
    gen_AB = gen_AB.apply(weights_init)
    gen_BA = gen_BA.apply(weights_init)
    disc_A = disc_A.apply(weights_init)
    disc_B = disc_B.apply(weights_init)
```

- Discriminator Loss

```
def get_disc_loss(real_X, fake_X, disc_X, adv_criterion):

    disc_fake_X_hat = disc_X(fake_X.detach())
    disc_fake_X_loss = adv_criterion(disc_fake_X_hat, torch.zeros_like(disc_fake_X_hat))
    disc_real_X_hat = disc_X(real_X)
    disc_real_X_loss = adv_criterion(disc_real_X_hat, torch.ones_like(disc_real_X_hat))
    disc_loss = (disc_fake_X_loss + disc_real_X_loss) / 2
    return disc_loss
```

- Generator Loss

Adversarial Loss

```
def get_gen_adversarial_loss(real_X, disc_Y, gen_XY, adv_criterion):
    fake_Y = gen_XY(real_X)
    disc_fake_Y_hat = disc_Y(fake_Y)
    adversarial_loss = adv_criterion(disc_fake_Y_hat, torch.ones_like(disc_fake_Y_hat))

    return adversarial_loss, fake_Y
```

Identity Loss

```
def get_identity_loss(real_X, gen_YX, identity_criterion):
    identity_X = gen_YX(real_X)
    identity_loss = identity_criterion(identity_X, real_X)

    return identity_loss, identity_X
```

Cycle Consistency Loss

```
def get_cycle_consistency_loss(real_X, fake_Y, gen_YX, cycle_criterion):
    cycle_X = gen_YX(fake_Y)
    cycle_loss = cycle_criterion(cycle_X, real_X)

    return cycle_loss, cycle_X
```

Generator total Loss

```
def   get_gen_loss(real_A,   real_B,   gen_AB,   gen_BA,   disc_A,   disc_B,   adv_criterion,
identity_criterion, cycle_criterion, lambda_identity=0.1, lambda_cycle=5):

    # Adversarial Loss
    adv_loss_BA, fake_A = get_gen_adversarial_loss(real_B, disc_A, gen_BA, adv_criterion)
    adv_loss_AB, fake_B = get_gen_adversarial_loss(real_A, disc_B, gen_AB, adv_criterion)
    gen_adversarial_loss = adv_loss_BA + adv_loss_AB

    # Identity Loss
    identity_loss_A, identity_A = get_identity_loss(real_A, gen_BA, identity_criterion)
    identity_loss_B, identity_B = get_identity_loss(real_B, gen_AB, identity_criterion)
    gen_identity_loss = identity_loss_A + identity_loss_B

    # Cycle-consistency Loss
    cycle_loss_BA,    cycle_A    =    get_cycle_consistency_loss(real_A,    fake_B,    gen_BA,
     cycle_criterion)
    cycle_loss_AB,    cycle_B    =    get_cycle_consistency_loss(real_B,    fake_A,    gen_AB,
     cycle_criterion)
    gen_cycle_loss = cycle_loss_BA + cycle_loss_AB

    # Total loss
    gen_loss  =  lambda_identity  *  gen_identity_loss  +  lambda_cycle  *  gen_cycle_loss  +
     gen_adversarial_loss

    return            gen_loss,            gen_adversarial_loss,            lambda_identity
     *gen_identity_loss,lambda_cycle*gen_cycle_loss, fake_A, fake_B
```

- CycleGAN training

```
from skimage import color
import numpy as np
from torchvision.utils import save_image

def train(save_model=True):
    mean_generator_loss = 0
    mean_discriminator_loss = 0
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    cur_step = 0
    generator_loss = []
    generator_adversarial_loss = []
    generator_identity_loss = []
    generator_cycle_loss= []
    discriminator_loss = []

    for epoch in range(n_epochs):
        for real_A, real_B in tqdm(dataloader):
            real_A = nn.functional.interpolate(real_A, size=target_shape)
            real_B = nn.functional.interpolate(real_B, size=target_shape)
            cur_batch_size = len(real_A)
            real_A = real_A.to(device)
            real_B = real_B.to(device)

            ### Update discriminator A ###
            disc_A_opt.zero_grad()
            with torch.no_grad():
                fake_A = gen_BA(real_B)
            disc_A_loss = get_disc_loss(real_A, fake_A, disc_A, adv_criterion)
            disc_A_loss.backward(retain_graph=True)
            disc_A_opt.step()

            ### Update discriminator B ###
            disc_B_opt.zero_grad()
            with torch.no_grad():
                fake_B = gen_AB(real_A)
            disc_B_loss = get_disc_loss(real_B, fake_B, disc_B, adv_criterion)
            disc_B_loss.backward(retain_graph=True)
            disc_B_opt.step()

            ### Update generator ###
            gen_opt.zero_grad()
            gen_loss, gen_adversarial_loss, gen_identity_loss, gen_cycle_loss, fake_A, fake_B =
get_gen_loss(
                real_A, real_B, gen_AB, gen_BA, disc_A, disc_B, adv_criterion, recon_criterion,
recon_criterion
            )
            gen_loss.backward()
            gen_opt.step()

            # Keep track of the average discriminator loss
            mean_discriminator_loss += disc_A_loss.item() / display_step
            discriminator_loss += [disc_A_loss.item()]

            # Keep track of the average generator loss
            mean_generator_loss += gen_loss.item() / display_step
            generator_loss += [gen_loss.item()]
            generator_adversarial_loss += [gen_adversarial_loss.item()]
            generator_identity_loss += [gen_identity_loss.item()]
            generator_cycle_loss += [gen_cycle_loss.item()]

            ### Visualization code ###
            if cur_step % display_step == 0:
                print(f”Epoch  {epoch}:  Step  {cur_step}:  Generator  (U-Net)  loss  mean:
                {mean_generator_loss}, Discriminator loss mean: {mean_discriminator_loss}”)
                print(f”\t\t  Generator  (U-Net)  loss:  {gen_loss.item()},  Discriminator  loss:
                {disc_A_loss.item()}”)
                show_tensor_images(torch.cat([real_A,   real_B]),   size=(dim_A,   target_shape,
                target_shape))
                show_tensor_images(torch.cat([fake_B,   fake_A]),   size=(dim_B,   target_shape,
                target_shape))
                mean_generator_loss = 0
                mean_discriminator_loss = 0
                x_axis = sorted([i * step_bins for i in range(len(generator_loss) // step_bins)]
* step_bins)
                num_examples = (len(generator_loss) // step_bins) * step_bins
                plt.plot(
                    range(num_examples // step_bins),
                    torch.Tensor(generator_loss[:num_examples]).view(-1, step_bins).mean(1),
                    label="Generator Loss"
                )
                plt.plot(
                    range(num_examples // step_bins),
                    torch.Tensor(generator_adversarial_loss[:num_examples]).view(-1,
step_bins).mean(1),
                    label="Generator Adversarial Loss"
                )
                plt.plot(
                    range(num_examples // step_bins),
                    torch.Tensor(generator_identity_loss[:num_examples]).view(-1,
step_bins).mean(1),
                    label="Generator Identity Loss"
                )
                plt.plot(
                    range(num_examples // step_bins),
                    torch.Tensor(generator_cycle_loss[:num_examples]).view(-1,
step_bins).mean(1),
                    label="Generator Cycle Loss"
                )
                plt.plot(
                    range(num_examples // step_bins),
                    torch.Tensor(discriminator_loss[:num_examples]).view(-1,
step_bins).mean(1),
                    label="Discriminator Loss"
                )

                plt.legend()
                plt.show()

            if cur_step % save_step == 0:
                if save_model:
                    torch.save({
                        'gen_AB': gen_AB.state_dict(),
                        'gen_BA': gen_BA.state_dict(),
                        'gen_opt': gen_opt.state_dict(),
                        'disc_A': disc_A.state_dict(),
                        'disc_A_opt': disc_A_opt.state_dict(),
                        'disc_B': disc_B.state_dict(),
                        'disc_B_opt': disc_B_opt.state_dict()
                    }, f"Models/lambda_5_lr_0.0002/cycleGAN_{cur_step}.pth")

            cur_step += 1
train()
```

- CycleGAN testing

```
transform = transforms.Compose([
    transforms.ToTensor(),
])

#import the test imagedataset
import torchvision
dataset    =    ImageDataset("sketchThickness2renderColorBlue_REDUCED",    transform=transform,
mode='test')

from skimage import color
import numpy as np
from torchvision.utils import save_image
plt.rcParams["figure.figsize"] = (10, 10)

def test():
    #this testing function shows the whole cycle
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    mean_generator_loss = 0
    mean_discriminator_loss = 0
    generator_loss = []
    discriminator_loss = []

    for epoch in range(n_epochs):
      for real_A, real_B in tqdm(dataloader):
              real_A = nn.functional.interpolate(real_A, size=target_shape)
              real_B = nn.functional.interpolate(real_B, size=target_shape)
              cur_batch_size = len(real_A)
              real_A = real_A.to(device)
              real_B = real_B.to(device)

              with torch.no_grad():
                  fake_A = gen_BA(real_B)
                  fake_BAB = gen_AB(fake_A)
                  fake_B = gen_AB(real_A)
                  fake_ABA= gen_BA(fake_B)

              # Keep track of the average discriminator loss
              mean_discriminator_loss += disc_A_loss.item() / display_step
              discriminator_loss += [disc_A_loss.item()]

              # Keep track of the average generator loss
              mean_generator_loss += gen_loss.item() / display_step
              generator_loss += [gen_loss.item()]

              print(f"Epoch   {epoch}:   Step   {cur_step}:   Generator   (U-Net)   loss   mean:
{mean_generator_loss}, Discriminator loss mean: {mean_discriminator_loss}")
              print(f"\t\t  Generator  (U-Net)  loss:  {gen_loss.item()},  Discriminator  loss:
{disc_A_loss.item()}")
              show_tensor_images(torch.cat([real_A,     fake_B,     fake_ABA]),     size=(dim_A,
target_shape, target_shape))
              show_tensor_images(torch.cat([real_B,fake_A,fake_BAB]),             size=(dim_B,
target_shape, target_shape))

test()
```