UNIVERSITY OF BUCHAREST

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

Bachelor's Degree Thesis

Generating Paintings with Generative Adversarial Networks

Graduate:

Dumitrescu Teodor

Scientific coordinator:

Prof. Dr. Păun Andrei

Bucharest, June 2021

Abstract

The aim of this work is to propose a way of creating paintings that resemble particular art styles with historic relevance. The technology used in order to accomplish this goal is Generative Adversarial Networks (GANs), a relatively new generative method introduced in 2014 which has attracted a lot of interest from the Deep Learning research community since it was first proposed. GANs have been successful for a variety of tasks including data augmentation, image-to-image translation or text-to-image translation. Perhaps the most extraordinary success in the field of GANs has been generating human faces of such high quality that they would fool most people into thinking they are photos of real people.

The task of generating paintings with GANs is not one of the widely pursued directions of research in the field and as a result I wasn't able to find much relevant work on the topic, especially on painting generation based on art style. Two of the most obvious reasons for this lack of interest and results in this direction are the small number of paintings available from any historic art style (3,000 - 17,000) and the high variety of paintings inside a single style, as they are not limited to depicting only portraits or landscapes, but also more abstract artworks very different from the others in their respective art style. Painting generation that does not focus on only one of the main genres like portrait or landscape is a very different (and difficult!) task compared to human faces generation, as the paintings do not share much in terms of structure. The most relevant work on this topic that I was able to find focused on generating paintings at 256 x 256 pixels resolution but did not use style based generation.

However, the limitations mentioned above didn't deter me from dedicating my thesis to this goal, which I believe I have accomplished, maybe not in a revolutionary way but in an innovative one. The final results of this work are 128 x 128 resolution paintings generated for 10 relevant historic art styles: abstract expressionism, baroque, cubism, expressionism, impressionism, post-impressionism, realism, romanticism, surrealism and symbolism as well as a web application that not only showcases my results but also allows any user to generate paintings from any of those 10 styles by simply clicking on a button.

I also talk into detail about the experiments I went through with different GAN architectures, including DCGAN, WGAN-GP, ProGAN and Conditional GAN, what worked and what didn't work, the hardware limitations that slowed me down and the decisions I took in order to speed up the networks training.

I believe that the results of my work as well as my experiments and decisions taken are relevant for the direction of painting generation based on style and that continuing my work without the hardware limitations I had could result in considerable advancements in the field.

Rezumat

Scopul acestei lucrări este de a propune un mod de a crea picturi care aparțin anumitor stiluri de artă cu relevanță istorică. Tehnologia utilizată pentru a atinge acest obiectiv este: Rețele Generativ Adversative (GAN), o metodă generativă introdusă în 2014, care a atras mult interes din partea comunității Deep Learning de când a fost propusă pentru prima dată. GAN-urile au avut succes pentru o varietate de sarcini, inclusiv augmentarea datelor, traducerea image-to-image sau traducerea text-to-image. Poate că cel mai extraordinar succes în domeniul GAN-urilor a fost generarea unor fețe umane de o calitate atât de înaltă încât majoritatea oamenilor le-ar considera ca fiind fotografii ale unor oameni reali.

Sarcina de a genera picturi cu GAN-uri nu este una dintre direcțiile de cercetare populare în domeniu și, in consecință, nu am reușit să găsesc multe lucrări relevante pe această temă, în special despre generarea de picturi bazată pe stilul artistic. Două dintre cele mai evidente motive pentru această lipsă de interes și de rezultate în această direcție sunt numărul mic de picturi disponibile din orice stil de artă istoric (3.000 - 17.000) și varietatea mare de picturi în cadrul unui singur stil, deoarece acestea nu se limitează la a reprezenta doar portrete sau peisaje, dar și opere de artă mai abstracte, foarte diferite de celelalte din stilul lor de artă. Generarea de picturi care nu se concentrează doar pe unul dintre genurile principale, cum ar fi portretul sau peisajul, este o sarcină foarte diferită (și dificilă!) în comparație cu generarea fețelor umane, deoarece picturile diferă foarte mult în ceea ce privește structura. Cea mai relevantă lucrare pe acest subiect pe care am reușit să o găsesc s-a concentrat pe generarea de picturi la rezoluție de 256 x 256 pixeli, dar nu a folosit generarea bazată pe stil.

Cu toate acestea, limitările menționate mai sus nu m-au împiedicat să-mi dedic teza acestui scop, pe care consider că l-am realizat, poate nu într-un mod revoluționar, dar într-un mod inovator. Rezultatele finale ale acestei lucrări sunt picturi cu rezoluție de 128 x 128 generate pentru 10 stiluri de artă istorice relevante: expresionism abstract, baroc, cubism, expresionism, impresionism, postimpresionism, realism, romantism, suprarealism și simbolism, precum și o aplicație web care nu doar prezintă rezultatele mele, ci permite oricărui utilizator să genereze picturi din fiecare dintre cele 10 stiluri, făcând clic pe un buton.

De asemenea, in cadrul lucrării vorbesc în detaliu despre experimentele pe care le-am realizat cu diferite arhitecturi GAN, inclusiv DCGAN, WGAN-GP, ProGAN și GAN condițional, ce a funcționat și ce nu a funcționat, limitările hardware care m-au încetinit și deciziile pe care le-am luat pentru a accelera antrenamentul rețelelor.

Consider că rezultatele expuse in lucrarea mea, precum și experimentele și deciziile luate sunt relevante pentru direcția generării picturilor bazată pe stil și că o continuare a muncii mele fără limitările hardware pe care le-am avut ar putea duce la progrese considerabile în domeniu.

Contents

Chapter 1

Introduction

For thousands of years humans have been captivated by art. Art, in its many forms, makes us contemplate, dream with our eyes open, it awakens a wide variety of emotions within us. Art helps us leave behind the agitated, stressful and perhaps monotonous world that surrounds us daily and makes us transcend to a higher consciousness, to a state of contemplation of the beauty that is unconstrained by time. It makes as feel as being part of a greater whole, by living the same emotions as people from different parts of the globe do, or people long gone did. Art is a bridge between ages, between people. It is a universal language.

I have always loved art, be it music, literature or painting. However, my talents lean towards rationality rather than creativity therefore I have mostly played the role of a passive observer while others created the subjects of my contemplation. Luckily, in the past year (and several months) I have been introduced to Artificial Intelligence, Machine Learning and in particular to Deep Learning, and I have discovered that nowadays one can create art with the help of mathematics and computer science, without having to be an artist. This encouraged me to dedicate this thesis to one of my favorite areas of art: painting, by aiming to generate unique paintings resembling historic art styles like expressionism, surrealism or romanticism.

The tools I used for this task are Generative Adversarial Networks (GAN)[1], a relatively new system of neural networks proposed in 2014 which attracted a lot of interest from the research community since then. GANs are used in order to generate artificial images that are indistinguishable from the images in a real data set. In the past few years, GANs have improved so much that they are now able to generate realistic human faces that would fool most people into thinking they are photos of real people.[2]

Figure 1.1: Human faces generated by Nvidia's StyleGAN. The faces are not only very realistic but also diverse.

1.1 State of the art

Generation of art has not attracted so much research, compared to generation of human faces, which has achieved a very high level of quality and diversity.

The most relevant work with the purpose of generating paintings that I was able to find is that of A. Elgammal, B. Liu, M. Elhoseiny and M. Mazzone titled *CAN: Creative Adversarial Networks Generating "Art" by Learning About Styles and Deviating from Style Norms*[3]. They used a resolution of 256 x 256 pixels for the generated images. However, they did not focus on generating paintings for a particular art style (like I do).

Figure 1.2: CAN generated images, top ranked by human subjects.

This paper served as a proof that what I wanted to do was manageable (style based

generation), although the path I took was very different from that of the authors.

Figure 1.3: CAN generated images, the closest the authors get to style based generation. The authors used style information for computing the loss of their networks. However no aimed style for the generated images is mentioned in the paper, just that the images resemble portraits, landscapes etc.

1.2 Overview of my work

I have experimented with different GAN architectures and resolutions for the paintings generated, as well as different data sets. I have settled on an architecture similar to DCGAN[4] but with some additional components and techniques specific to other GAN architectures. The resolution of the generated images is 128 x 128 as I wasn't successful in generating good looking paintings for higher resolutions because I was constrained by the hardware and time available as well as the small number of paintings available for a single art style, sometimes even less than 4,000. The end of my experiments resulted in 10 networks able to generate unique paintings, each for one of the following art styles: abstract expressionism, baroque, cubism, expressionism, impressionism, post-impressionism, realism, romanticism, surrealism and symbolism.

I have also developed a web application called **GenerateArt** that allows any user to generate paintings for any of the 10 styles mentioned above, using the 10 networks trained. By simply pressing a button, the user generates a grid of 16 unique paintings. The application also showcases a collage of 20 paintings generated for each style, handpicked by me in order to give the user a sense of how some of the best aspect generated images can look like. These collages can be seen in the **Paintings generated with the final architecture** chapter.

Figure 1.4: Paintings I have generated for expressionism with the final GAN architecture.

1.3 Description of the chapters

1. **Introduction**. I talk about my perspective on the purpose of art as well as the motivation for this thesis. I mention the technology used, namely GANs, and I show some state of the art results in face generation with GANs. I also talk about the most relevant work in painting generation that takes into account the style of paintings but doesn't display paintings generated for particular art styles. I also give a brief description of the final results of my experiments for the task of generating paintings based on art style.

2. **Generative Adversarial Networks**. I describe the Generator and Discriminator networks that are the components of a GAN. I talk about their structure, goal, input, output and training philosophy. I also mention that training a GAN is very difficult and give several reasons that back up my affirmation.

3. **Data**. I talk in detail about the source of data used for my project, namely the website WikiArt, what it has to offer and why I believe it is currently the best source of paintings for a project similar to mine. I describe in detail the process used in order to obtain all the paintings available on the website for any desired art style. I also compare alternatives for several important decisions regarding data storage that greatly influence the speed of training, therefore playing a big role in how much experimenting one can do in a given amount of time. Finally, I talk about the preprocessing done on the data from the point of downloading to feeding it to the networks.

4. **Networks Architecture**. This chapter is dedicated to the final GAN architecture used in this project, the one that gave the best results and helped me accomplish my goal of generating paintings based on style. I talk in detail about the layers used for the networks, the training process, including hyperparameter choices. I also show the evolution of six paintings from different art styles along their respective training session.

5. **Technologies Used**. A short chapter where I mention the technologies used for my project as well as the role they played.

6. **Experiments**. In this chapter I talk about the different GAN architectures I used while experimenting as well as the results I was able to obtain with them. I also describe the problems I encountered while training and the impact of hardware limitations.

7. **Paintings generated with the final architecture**. This chapter showcases the best paintings I was able to generate for each art style, with the final architecture. I have dedicated a collage of 20 images to each style.

8. **Web application - GenerateArt**. I talk about the web application I developed in order to showcase my art and to allow any user to generate paintings by simply pressing a button. The structure, logic and aspect of the application are discussed. I also display various elements that a user sees on the application's website.

9. **Conclusion**. I state my final thoughts on this project as well as the future work that I have in mind for improving it.

10. **Appendix - Random selection of images generated for each style**. Finally, in the appendix, I display 16 images generated for each art style. These images are not handpicked. They represent batches of images generated at once, that any user could have generated. They give the viewer a sense of what to expect from a single generation for each style.

Chapter 2

Generative Adversarial Networks

2.1 Description

Generative Adversarial Networks (GANs) are a type of neural networks invented by Ian Goodfellow and his colleagues in 2014. A GAN is actually a system of two neural networks that fight against each other in a minimax game, in which the advantage of one means the disadvantage of the other.[1; 5]

The first network is called the **Generator**. Given a training data set, the purpose of the Generator is to learn the distribution of the data and to generate new data that would integrate perfectly into the real data set, so that an observer would not be able to say which data is real and which is generated with a probability greater than 50%. The second network is called the **Discriminator**. The purpose of the Discriminator is to distinguish between the real data, from the training set, and the data created by the Generator. Thus, the Discriminator's task is a real/generated classification one. This system of two adversarial networks is trained with the help of backpropagation, no additional components are needed.[5; 6]

In the following, I will often replace *data* with *images*, because generative adversarial networks are predominantly used in the field of Computer Vision. In addition, the explanations will become more intuitive.

2.2 Input and Output of the Networks

The generator can be thought of as a function from an n-dimensional vector space (n = 128, for example), to the image space with a certain resolution and a number of channels (128 x 128 and 3 RGB channels in our case). This function associates each vector in the domain with an image. Ideally, the Generator can create a wide variety of high quality images.

However, this depends on the amount of training data, the distribution of the training data, as well as the architectures of the Generator and the Discriminator.[6]

The Discriminator models a function from the space of images with the resolution and the number of channels of the training images (and of the images created by the Generator), to the interval [0, 1]. An output closer to 0 is associated with a generated image, while an output closer to 1 is associated with a real image. The Discriminator's structure is similar to a convolutional network used in a binary classification task.

2.3 How the networks work

Intuitively, the Generator is guided by the Discriminator to improve its generation function. The generator seeks to fool the Discriminator. Each time the Discriminator correctly classifies an image as generated, it provides feedback to the Generator (through a loss function). The generator then updates its parameters in the direction indicated by this feedback. This process keeps repeating itself, and the images created by the Generator become more and more plausible.[1; 5]

In order for the Discriminator to evolve as well, he has to classify batches that contain only real images, respectively only generated images. And it receives feedback through a loss function, and thus changes its parameters.

A training step for the Generator looks like this: a batch of images is generated (with the Generator). The images are sent to the Discriminator, who processes them and assigns to each image a number in the range [0, 1], which represents the probability that the image is real. The loss function applied is binary cross entropy, with the target label being 1, because we want to penalize the Generator to a large extent when the Discriminator associates a generated image with a small probability to be real, respectively to a small extent when the Discriminator is tricked and associates a generated image with a high probability of being real. The gradients of the loss function are propagated throught the network and the parameters of the Generator are changed. The Discriminator does not change its parameters at this step, even though the loss function has been computed with its predictions.[1; 5]

A training step for the Discriminator looks like this:

1. Get a batch of real images from the training data set (which is shuffled). These images are processed by the Discriminator, who assigns to each of them a number in the range [0, 1], representing the probability of the image being real (and not generated). Similar to the step for the Generator, the loss function is binary cross entropy, and the target label is 1, because we want to penalize the Discriminator harder when associating a

probability close to 0 (generated) with a real image, respectively more superficially when associating a probability close to 1 (real).

2. A batch of images is generated with the Generator. These images are also processed by the Discriminator in a similar way, with the same loss function, but the target label is 0 this time. We want to penalize the Discriminator more when it associates a probability close to 1 with a generated image, respectively less when it associates a probability close to 0.[1]

The two values obtained are then added, and the result represents the value of the total loss function for the Discriminator. The gradients of this function, with respect to the parameters of the Discriminator (not of the Generator, even though in step 2 it generates the images!) are propagated through the network, and the parameters of the Discriminator are modified. The Generator does not change its parameters at this step.

Note that a training step for one network does not change the parameters of the other network. For this to happen, within the code, when the Discriminator is trained with images created by the Generator, the computational graph based on which the images are generated must be detached from the Discriminator's computational graph. In Pytorch, this is done by calling the *detach* method on the tensor that contains the generated images, before the tensor is used by the Discriminator to compute the real/generated probabilities.

During training, at every step, the performance of each network should increase. However, if the performance of one network grows too fast and the other lags behind, training gets stuck at one point. Due to the fact that the task of the Discriminator is simpler than that of the Generator, it often happens that he gets to classify the images perfectly. In this case, the Discriminator loss function (which provides feedback to the Generator) will have a value very close to 0, and the gradients that propagate through the Generator will also have the value 0, so the Generator's parameters do not evolve at all. It can also happen that the Generator far exceeds the performance of the Discriminator (less often). In this case, the Discriminator's feedback is no longer useful to the Generator, because the loss function no longer reflects reality. The result is the same, the generated images remain in the same stage and no network evolves.[7; 8]

Training a GAN is very difficult and requires the existence of a balance between the two networks throughout. Achieving this balance is one of the main research directions in the field of Generative Adversarial Networks.[9; 10; 11]

2.4 Problems while training

1. **Mode Collapse.** The **Generator** produces a very small variety of images.[7; 12] The distributions of real data sets are multi-modal. One mode is an area of a data distribution in which many observations are concentrated. In other words, any peak of the density function of a distribution can be considered a mode. For example, a normal distribution has only one mode, namely its mean. In the MNIST data set we are dealing with 10 modes, one for each digit, each mode being an image with the respective digit (a kind of average image for each digit). Mode collapse occurs when the Discriminator reaches a local minimum (of the loss function). For the MNIST set such a local minimum can be when the Discriminator has a good accuracy in classifying the digits from 1 to 8, but not for the digits 0 and 9. Thus the Generator is encouraged to create only digits of 0 and 9, because they will most likely fool the Discriminator. In my experiments I often had the problem of Mode Collapse, in two variants: all the generated images looked the same or the images in a subset of considerable size looked almost the same. The second option happens especially when I used a single conditional GAN in order to train with all artistic styles together and generate images based on a chosen style.

2. **Vanishing Gradients**: The discriminator far exceeds the performance of the Generator, obtaining a probability of 1 in detecting the generated images. Thus the loss function becomes 0, as do the propagated gradients. Therefore, the Generator no longer changes its parameters and remains stuck in the current stage (most of the time in a stage in which the images it generates do not look satisfactorily).[13; 14]

3. **Non-Convergence**: The images produced by the Generator, for the same vectors in the input vector space, change significantly from one training epoch to another, even in advanced stages of training.[7] Ideally, the parameters of both networks should converge to values for which the Nash equilibrium is achieved, and the probability with which the Discriminator correctly associates the real/generated tag is 0.5. In practice, this does not happen in most GAN applications[8], but there are different methods designed to encourage the convergence of the networks parameters[9]. In my experiments I failed to achieve convergence, so I stopped training at a stage where I was satisfied with the appearance of the generated images. It is remarkable that many times, even after almost 100 epochs of training, the results can look very good after the 99th epoch, then become less satisfactory for the next 3 epochs, and then look good again, after the 103rd epoch, for example.

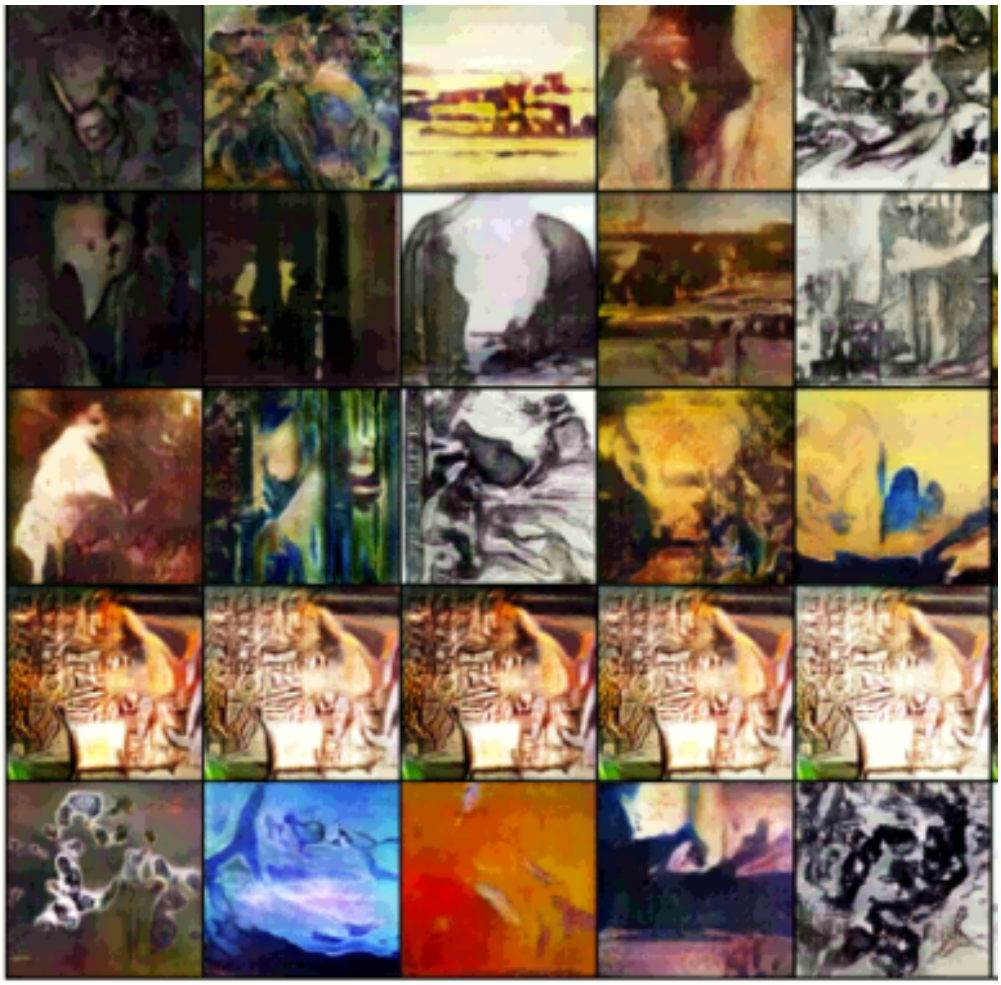

Figure 2.1: Mode Collapse for surrealism (4th row). Each row represents an artistic style.

4. **High sensitivity to hyperparameter choices and other decisions**: The learning rate for each network (because it is not mandatory to update their parameters with the same learning rate, actually many studies encourage a higher learning rate for the Discriminator[10]), the size of the batch of images, the hyperparameters of the optimization algorithm used (Beta values for Adam, in this case), the number of filters in each convolutional or transpose convolutional layer, the size of the filters, etc. are hyperparameters that greatly influence the results obtained after training. Very often, an uninspired choice for a hyperparameter results in an obvious failure (the loss function reaches the maximum value for one of the networks) after only one training epoch (or even after a few batches). Other times, this failure occurs only in the middle of the training, and restarting the training from scratch is required.

   There are also several canonical variants of GANs (DCGAN, WGAN-GP, ProGAN, etc.), which differ greatly in architecture and/or loss function. If you want to switch

from one architecture to another, experiments performed with the previous architecture become useless for the new architecture, as well as most hyperparameter settings. I say this from my experience (and from articles I read[15; 16]), doing many experiments with the ProGAN architecture, which in theory had the potential to give better results than DCGAN or WGAN-GP, but my results weren't better at all.

Chapter 3

Data

3.1 WikiArt

I used probably the most complete source of images dedicated to paintings (and sculptures, ceramics, etc.), namely the WikiArt website[1]. WikiArt contains from the most famous paintings, such as Vincent van Gogh's The Starry Night or Pablo Picasso's Harlequin, to lesser-known paintings belonging to obscure styles such as synchronism: Synchromy, by Morgan Russell. The website creators claim that WikiArt displays over 250,000 artworks from more than 3,000 artists. Not all of them are paintings though, there are also different types of art, like sculptures or ceramics, as I previously stated, but for the purpose of this work, we will focus mostly on paintings. I will briefly mention the number of artworks available from other art forms, as WikiArt also divides them in different categories based on materials and techniques used.

Before taking a more detailed look at what WikiArt has to offer, we should also check some other sources of data, and see whether they are worthy competitors to WikiArt. One such source is Web Gallery of Art[2], a website created in 1996, which contains 50,575 artworks by 5,456 artists. Most of them are paintings (63%). Two notable subsets of the database are sculptures (14%) and architecture (9%). While the number of artists covered by this website is higher than that of WikiArt, the total number of artworks is much smaller. Also, an important thing to note, as we are going to focus on different styles of paintings in this work, is that Web Gallery of Art contains no artworks from the 20th and 21st centuries. This means that very significant painting styles like expressionism, cubism or surrealism, are not present on the website, which is a big drawback.

[1] `https://www.wikiart.org/`. Last accessed 6 June 2021.

[2] `https://www.wga.hu/index1.html`. Last accessed 6 June 2021.

Another source of images to consider is the website Artsy[3]. It contains more than one million artworks from over 100,000 artists. The main purpose of this website is acting as a market for buying and selling art, but it also displays art that is not for sale, like Leonardo da Vinci's The Last Supper. One of the reasons for Artsy containing much more artworks compared to WikiArt or Web Gallery of Art is that Artsy is much less restrictive in terms of what can and cannot be displayed on the website. It does not focus on covering historical art, even tough it is pretty good at that too, and we can find most of the famous paintings on Artsy too. But the majority of artworks displayed are modern, created by more or less unknown artists. If we take a look at some relevant historic art styles, we will see that WikiArt is a better source of art for our work. WikiArt contains 7,245 baroque, 7,469 surrealism and 3,332 cubism artworks, while Artsy contains close to 1,200 baroque, less than 5,000 surrealism and only 180 cubism artworks (May 2021). We could look at other important art styles too but I believe the analysis above should be enough to decide that WikiArt is a better option for this project, as I am interested in generating artwork based on historical painting styles.

Let us take a deeper look at what WikiArt has to offer. The site continues to gather new artworks from year to year. An article that used WikiArt as a source of paintings [3] claims that in 2015, the site provided 10,733 paintings in the style of realism, 7,019 paintings in the style of romanticism and 6,736 paintings in the style of expressionism. In 2021, the number of paintings from the aforementioned styles increased to 15,771, 14,291, and 13,125 respectively, some styles thus containing twice as many paintings. The number of artworks displayed on the website grows continuously which is a very encouraging fact. Projects similar to my work, or simply my work with an updated database of pictures, will produce more and more convincing results by simply updating the data sets used every once in a while. From 10th to 25th May 2021, close to 550 new artworks have been added on the website and contrary to what one might think, most of these artworks are not modern at all. They are from different time periods: 21st century, 20th century, 19th century (perhaps most of them) and even 17th century!

[3] `https://www.artsy.net/`. Last accessed 8 June 2021.

Figure 3.1: *Assumption of the Virgin* by Orazio Gentileschi, 1608. Added on WikiArt on 14 May 2021.

On WikiArt we have two main options of searching: for artists or for artworks. If we are interested in the pictures of a particular set of artists, say Italian artists, we can use filters to get only those artists. Then we can get all the artworks for each of those artists.

The artists are split in categories on the following criteria:

1. **Art movements** (pretty much the same as style, for artworks). If we sort them by the artist count, the first five art movements are the following: *romanticism* with 275 artists, *impressionism* with 229 artists, *expressionism* with 224 artists, *realism* with 205 and *surrealism* with 180 artists. These first five art movements are all used in my project, because they are all defining moments in the history of art. As my project focuses on generating art by style (or art movement), I will also enumerate the number of artists in the other five movements that I used: *abstract expressionism* with 159 artists, ranked 6th in the art movement list sorted by artist count, *baroque* with 156 artists (ranked 7th), *post-impressionism* with 131 artists (ranked 10th), *symbolism* with 90 artists (ranked 15th) and *cubism* with 72 artists (ranked 20th).

2. **Schools and groups**. First 5: *École de Paris* with 74 artists, *New York School* with 52 artists, *Degenerate Art* with 37 artists, *Peredvizhniki (Society for Traveling Art Exhibitions)* with 30 artists and *Mir Iskusstva (World of Art)* with 28 artists.

3. **Genres**. Most artists did not limit themselves to only one genre, but many artists are considered more prolific in a particular genre, in which case they have been gathered

together under the same genre. The top five genres, by artist count are: *abstract* with 444 artists, *landscape* with 265 artists, *portrait* with 260 artists, *sculpture* with 173 artists and *genre painting* with 102 artists.

4. **Field**. This category classifies artists based on techniques or materials used for creating their artworks. The top five fields are: *painting* with 2,946 artists, *sculpture* with 608 artists, *drawing* with 447 artists, *printmaking* with 437 artists and *installation* with 306 artists.

5. **Nationality**. The top five nationalities in terms of artist count are the following: Americans (684), French (466), British (326), Italians (316) and Germans (234). Romanians occupy the 7th place in the list, with a total artist count of 119, which is not bad at all.

6. **Century**. The time frame of the activity of the artists is considered (not necessarily when they were born). If an artist was active during two centuries, they are attributed to both. I will not sort the centuries in terms of artist count as it might be a little hard to read/confusing. I will rather list them in reverse chronological order, in a "century - artist count" manner:

   - 21st century - 352
   - 20st century - 2,686
   - 19th century - 1,344
   - 18th century - 229
   - 17th century - 188
   - 16th century - 182
   - 15th century - 121
   - 14th century - 34
   - 13th century - 16
   - 12th century - 8
   - 11th century - 5
   - 10th century - 4
   - 9th century - 1
   - 8th century - 5
   - 7th century - 3
   - 4th century - 2
   - 3rd century - 1
   - 1st century - 1
   - 1st century BC - 2
   - 4th century BC - 1
   - 7th century BC - 1
   - 9th century BC - 1
   - 32nd century BC - 1

   Note: each item represents the start of a period, so 32nd century BC refers to the time period between 32nd and 9th century BC. Also many artists counted (especially for the older time frames) are not actually individual artists, as it is very hard or even

impossible to know the name of the author for older artworks. These are multiple artists but counted as only one artist, under a generic name. Some examples are: *Aztec Art* , from the 16th century, *Viking art*, from the 11th century, *Orthodox Icons*, from the 4th century or even *Ancient Egypt*, from 32nd century BC.

7. Art institution. The artists are split based on the art institution that they were teaching or studying in. The first five art institutions by artist count are: *École des Beaux-Arts, Paris, France*, with 87 artists, *Académie Julian, Paris, France* with 58 artists, *Art Students League of New York, New York City, NY, US* with 55 artists, *Akademie der Bildenden Künste München (Munich Academy), Munich, Germany* with 37 students and *Imperial Academy of Arts, Saint Petersburg, Russia* with 36 artists.

Note1: an artist might be in more than one category, for most of these criteria.

Note2: there is also an option located in the footer of the WikiArt website, namely **Advanced Search** (for artists). This searching option allows the user to filter the artists by the following five criteria: **Time Frame**, which is different from **Century** from the previous list of criteria, because here the 20th century is divided into decades, the 19th and 18th centuries are divided into halves, centuries 11-15 are merged into one time frame, centuries 1-10 as well and the other 3 time frames available are 1000 BC - 0, 3000 BC - 1000 BC and pre 3000 BC. The other four criteria available for filtering are **Art Movements**, **Nation**, **School or group** and **Field of Art**. They are the same as in the previous list of criteria. With this **Advanced Search** option, we can select multiple criteria for filtering, and for each criteria (but one) we can set multiple values if we are interested, for example, in both French and Italian artists. The exception is the **Time Frame**, as we can only pick a contiguous time interval.

The other option provided by WikiArt is to search for artworks (and not artists). This option is useful if we are not particularly interested in grouping pictures by artist, but by style, genre or materials/techniques used. These are the three criteria available for the option of searching for artworks:

1. **Style**. **This is the filtering method I have used in order to get the pictures I was interested in.** There are nearly 200 art styles available on the website. The top five styles, ordered by the number of artworks available are as follows: *impressionism* with 16,627 artworks, *realism* with 15,872 artworks, *romanticism* with 14,575 artworks, expressionism with 13,140 artworks and post-impressionism with 8,428 artworks available. All five of the previously mentioned styles have been used for art generation in my project. The other five styles used in the project are: *surrealism* with 7,469 artworks, ranked 7th, *baroque* with 7,245 artworks, ranked 8th, *symbolism*

with 5,471 artworks, ranked 9th, *abstract expressionism* with 4,682 artworks, ranked 10th and *cubism* with 3,332 artworks, ranked 14th.

Filtering artworks by style is similar to filtering artists by art movement, as they share the same names. However, we should use filtering by style, for artworks, if we do not want to get all the artworks of an artist whose work focuses mainly on our target style but contains artworks from different styles too. In other words, if we chose to generate art based on art movement, we would get a list of all the artists under that art movement, but after selecting one of them, we would see all of his artworks displayed and not only those that are related to the target art movement. For example, if we search for artists from the surrealism movement, we will find Pablo Picasso. If we click on him, we will see 1,167 artworks that he created, but not all of them can be classified as part of the surrealism style. The same thing happens if we search for artists from the cubism movement. We find Pablo Picasso with all his 1,167 artworks.

2. **Genre**. This method of searching is similar to the one with the same name in the artists section. The genres are the same, but the rankings are a bit different, because the genre with the most artists is not necessarily the genre with the most artworks. The top five genres by artwork count are: *portrait* with 27,029 artworks, *landscape* with 23,150 artworks, *genre painting* with 22,438 artworks, *abstract* with 16,873 artworks and *religious painting* with 11,482 artworks. If *sculpture* was ranked 4th in the artists section, here it is ranked 13th with only 4,086 artworks. As was the case for the **style** criteria, searching for artwork by genre instead of for artists by genre is the way of making sure we do not collect artworks that have nothing to do with our target genre, but were created by an artists that focused on the target genre for most of his work.

3. **Media**. Art media refers to the materials and techniques used by the artists for creating their art. Here, two categories are by far the most well represented: *oil* with 41,972 artworks and *canvas* with 32,122 artworks. Other notable categories are the next three in the list sorted by artwork count: *paper* with 9,187 artworks, *watercolor* with 3,923 artworks and *panel* with 3,094 artworks.

   There are artworks which fit into more than one media category, like *Coffin of Nesmin*, from the Ancient Egypt, dated circa 200 - 30 BC, which belongs to four different categories: *wood*, *plaster*, *oil* and *gold*. As a matter of fact, the coffin also belongs to two different genres: *portrait* and *symbolic painting* and is labeled with the *Ptolemaic* style.

Figure 3.2: Coffin of Nesmin

Note: as in the artists section, there is an **Advanced Search** option in the artwork section too, in the footer of the website. Here, the available filters are: **Time Frame**, **Style**, **Genre**, **Media** and **Tag**. **Time Frame** is the same as in the artists section. The next three filters aren't different from the three artwork search criteria presented in the previous list. The last one, **Tag**, refers to keywords such as *Forest*, *Acrobatics*, *pilgrimage*, *Theseus* or *Moses* that may be used to describe artworks. We can click on **view all** in order to see the whole list of keywords available. For each of the criteria available in the **Advanced Search**, we can set one value, multiple values or no value at all. As in the artists section, the time frame has to be a contiguous interval.

3.2 Obtaining the paintings from a style

For my project, I was interested in getting as many **paintings** as possible from a particular **art style**, as I wanted to generate paintings that resemble that particular style. As I mentioned in the previous section, WikiArt allows the user to see all the paintings (available on the website) from a particular art style. Here's where the things get a bit tricky. WikiArt doesn't provide an easy way of downloading a large number of images. There is no option of downloading a collection of pictures as an archive. If we want to get all the artworks in the style of **surrealism** for example, we have two options provided by the website graphic interface.

- The first one is going to the **Artworks by style** section and selecting **surrealism**. 60 pictures would be displayed on the page. We would have to download each image

individually and scroll to the bottom of the page to click on **Load more** in order to see the next 60 pictures. Also, the limit of pictures displayed this way is 3,600 (even though surrealism has 7,469 pictures on the website) so we wouldn't be able to access the other pictures. Obviously, this approach is not convenient as we have to click on each picture and we can't even access all of them!

- The second one is going to the **Advanced search** section for artworks and selecting **surrealism** for the style filter. If we do just that, the search will return only 1,800 images (the upper limit of a search result). Also, only the first 60 images will be loaded and we need to scroll to the bottom of the page and click on **Load more** to get the next 60 images and so on. We can do a little trick and add a **Time Frame** filter with a smaller interval of time set. This way, instead of searching for all surrealism artworks at once, and getting only 1,800 images, we can search for one decade (or century) at a time and chances are that the number of images assigned to that time frame is less than 1,800, so we won't hit the upper limit of a search result. Doing this for each decade/century, we would have access to all the surrealism artworks, supposing that each non-divisible time interval doesn't have more than 1,800 artworks assigned to itself. As the reader might have noted, this approach does not remove the most important drawback: **We still have to download one picture at a time, by clicking on it**. In order to get past this obstacle, I had to come up with a more creative solution that I will describe shortly. The solution is built upon this second option presented above and involves web scraping.

I will further describe the method used to obtain all the paintings on the site, from a certain style (valid for any style).

I wrote a python code that sends requests to the WikiArt server and receives the images I am interested in. In order to see how these requests should look like, I used the second option presented above as a starting point. More exactly, I went to **Advanced Search**, on the website, selected a small time interval and **surrealism** as the art style and inspected the page (using Google Chrome as a browser). After clicking on the **Search** button, a couple of XHR requests appear in the Networks tab. We are interested in the one whose name starts with *?json=*. If we click on that request, we can see it's URL. If we make a request with the same URL with Postman (or another application that can send requests), we see that the response is a JSON which contains URLs for 60 images. We can send a new request for each of those 60 URLs and download all the 60 images. Now, if we look at the URL for the first request, we see that there is a parameter named **page** set to 1. Increasing the value of this parameter before sending the request results in a JSON which contains a different set

of URLs for 60 images. Two other important parameters for the request are the beginning and the end of the time frame set as a filter. We can easily change those parameters in order to get images from another time interval. These observations are all we needed, we are now able to download all the images for any target art style.

I will briefly describe the steps followed by the web scraping algorithm designed to download all the artworks for one style:

1. Make a list of all possible time intervals
2. For each interval do:
    2.1. Set **page** to 1
    2.2. Loop
        2.2.1. Send request to the WikiArt server with URL parameters: **page**, **beginning** and **end** of current time interval and **dictIdsJson** (style specific, more on it soon). Other parameters used have fixed values
        2.2.2. If the JSON received as response contains no URLs for images, end loop
        2.2.3. Else send a request for each URL in the JSON, receive the image in the response's content and write it to the disk (at most 60 images are processed at this step)
        2.2.4. Increment **page**

The parameter **dictIdsJson** has a specific value for each particular style, and in order to get the value for let's say cubism, I go to the WikiArt website, **Advanced Search** for artworks, set a random time frame and cubism for the style filter, inspect the page and click on **Search**. In the **Network** tab, the XHR request starting with **?json=** contains the value for **dictIdsJson** that I need for cubism. I simply copy that value in the code and run it for cubism.

Once the presented method works for one artistic style, to apply it to another style, only one parameter must be modified, **dictIdsJson**, so it is easily reusable.

3.3 Data storage

I will list some key decisions regarding data storage, as well as the advantages or disadvantages of the possible options. They had a fundamental contribution, by significantly reducing the training time. Many of them are not immediate/obvious. I didn't find any source that gathers a large subset of them. Most of them are the results of discussions I watched on the

Pytorch forum or on StackOverflow. It should be noted that I wasn't limited by the hard disk memory for the project, as I had almost 1 TB of free space. If someone would try to do my experiments on Google Colab, or a machine with not too much disk space available, making a different set of choices (from the list below) could be a better approach, especially doing the data augmentation at runtime. Also, I was able to keep different versions of the data sets on the same hard disk at the same time. I needed them as I was switching between different data preprocessing pipelines before settling on one pipeline. I also experimented with a **GAN** architecture that utilizes multiple resolutions for the data set (4 x 4, 8 x 8, 16 x 16, ... 128 x 128) called **ProGAN** and kept a different copy of the data set for each of those resolutions.[17]

For each decision I will state two options and why I chose one of them:

- **Loading the whole data set in GPU memory before starting training (and keeping it there for the whole training)** *VS* **Loading the data one batch at a time to GPU memory during training**.

  The **first** option greatly increases the training speed and it is good for a small data set. However, it must be taken into account that the networks parameters are also stored in the GPU memory, so the space available for the data set is smaller. While working on Google Colab, in the early stages of my experiments, I preferred this approach as I didn't use data augmentation and used a data set with four different art styles mixed, containing close to 30,000 images. This approach is not very popular (and impossible to use for large data sets). Most projects use the **second** approach as it works for data sets of any size.

  The **second** option is slower but does not impose a limit on the size of the data set. This advantage was especially useful as I decided to augment the data and each art style grew 6 times larger in terms of image count. Impressionism for example has 98,268 images after augmentation. Let's see how much memory those images need: each image is loaded into memory as a 3 x 128 x 128 tensor, where 3 is the number of channels (RGB), width and height are equal to 128 and 4 bytes of memory are used to store each channel value of a pixel, because they are stored with *float64* type. The whole impressionism data set requires $\frac{98{,}268 \times 3 \times 128 \times 128 \times 4}{2^{30}} = 17.99$ GB of memory. That is a lot of memory and most GPUs do not even have that much memory available (including the one I used). Also, we should keep in mind that the parameters of both networks are being kept in GPU memory for the whole training, which reduces the available GPU memory even more.

  As I decided to use data augmentation (which had a huge impact on the final results),

I transitioned from the **first** to the **second** option.

- **Resize followed by crop, once, before training, and saving the results to the disk** VS **Resize followed by crop for each image in the current batch, during runtime**.

  I will describe the pipeline for the **first** option. The input is a set of pictures in .jpg, .png or other image formats, with different resolutions. These images are resized and cropped to 128 x 128 resolution, the resolution that our networks will work with. The images are processed one by one and saved on the disk with their new resolution. This whole process is being done only once. For any future training session, the data set used is this resized and cropped data set.

  The **second** option is a very popular one in tutorials and on the Pytorch forums. It keeps the data set on the disk in the initial state, where the pictures have different resolutions. It uses a Pytorch **data set** object to which we can specify a series of transformations that we want to apply to the data, resize and crop in our case, and those transformations are being applied on each batch of images when that batch is being loaded into memory. In other words, the resize and crop preprocessing step is being applied at each training session, as we always keep the images with their initial resolutions on the disk.

  In the early stages of my experiments, I have used the **second** option as I kept seeing it in almost every tutorial/project/article. After realizing that the **first** option could reduce the training time significantly, as the preprocessing is being done only once, and not during training, I switched to it and only used the **first** option from then on.

- **Storing images on disk as .jpg, .png etc.** *VS* **Storing images as normalized tensors with .pt file extension**.

  The **first** option is more frequently seen in projects/tutorials and it is perhaps the first one that anyone would try while designing a project similar to mine. The information in a .jpg file for example (most images have been downloaded in .jpg format) is compressed. After being resized to 128 x 128 resolution, the size of the images in surrealism range from 2.45 KB to 43.4 KB, depending on the content of the image. Those are pretty small sizes compared to the one of a 128 x 128 image saved as a Pytorch tensor with .pt extension, which is $\frac{3\times128\times128\times4}{2^{10}} = 192$ KB. In conclusion, storing the images on disk in image specific formats requires a lot less memory than storing them as .pt tensor files. However, there are some disadvantages to this approach. When the images are loaded in Pytorch, they need to be decoded, converted into tensors and

then normalized (more on this in the following section). These operations are being done while training and at each epoch and naturally they will slow down the training.

The **second** option involves another preprocessing step for the images (aside from resize and crop): converting them into tensors, normalizing the tensors and saving the images as .pt files on the hard disk. This step is done only once, before training. This option requires much more disk space but it increases the training speed as the decoding, tensor conversion and normalization operations are no longer done for each image every time it is loaded for training (each epoch).

I have experimented with both options but settled on the **second** one as I didn't mind trading disk space for training speed.

- **Data augmentation only once and disk storage** *VS* **Data augmentation during each training, at runtime**.

  The **first** option means augmenting the data set for each particular art style and saving the new *artificial* images on the disk for being used in future training sessions, along with the *original* images. This process is being done only once, before any training occurs. The process is entirely deterministic so whether we run it once, or for each training, we would get the same exact *artificial* images. The exact method of augmentation used will be described in the next section, in the context of the whole data preprocessing pipeline. In short, the result is a 6 times larger data set for each art style. The main disadvantage of this approach is that we need a lot more (6 times more) disk space in order to keep all this data on the hard disk. This approach combined with keeping the images as tensor .pt files requires 106GB disk space for $\sim$ 560,000 images (all 10 art styles combined). The advantage is that training is faster as we do not need to perform the augmentation for each training session. Most projects I've seen do not use this method. They use the one I'm going to explain in the next lines.

  The **second** option takes advantage of the Pytorch **Dataset** object, which allows us to perform transformations like resize, crop and even cropping multiple different pieces out of the original image, which is basically the augmentation method I was interested in. This augmentation is being done whenever a batch of images is being loaded into memory so each image goes through the augmentation process once each epoch. This will result in a slower training compared to the **first** method. There is also a more subtle difference between the two methods. The **first** one keeps a bigger data set on disk, and for each training epoch the data set is shuffled before being divided into batches. The **second** option also shuffles the data set before each epoch, but the

augmentation is done **after** picking the images for the current batch. In other words, if a batch contains 16 original images, after augmentation it will contain $16 \cdot 6 = 96$ images. For each of the 16 original images there will be 5 other artificial ones, that are very similar to their original counterparts. The result is that there is much less diversity within this batch of images compared to a batch of 96 images from the 6 times larger data set created with the **first** approach. This low diversity would interfere with the work of a layer in the Discriminator network that has precisely the purpose of capturing the diversity within the batches of real images fed to the Discriminator in order to encourage the Generator network to output a wide variety of generated images. The layer is called Minibatch Standard Deviation layer and I talk about it in detail in the **Layers Used** section from the **Networks Architecture** chapter.

After deciding to augment the data set, I picked the **first** option as I thought that paying the price of disk space for faster training and more image diversity within the batches was worth it. I did not experiment with the **second** option.

3.4 The pipeline used to obtain and preprocess the data

I will explain the steps followed to prepare the training data for an artistic style: **cubism**. For any other style, the steps are identical, only the number of images used differs.

1. I downloaded all the cubism style paintings from WikiArt. The algorithm used in order to get the paintings has already been explained in the section **Obtaining the paintings from a style**.

2. In order to feed the images to the neural networks later, I needed them to have the same format. In other words, I wanted each image to be in RGB format as I wanted to generate RGB paintings. There were a couple of troublesome image types:

    - A couple of images where in grayscale format, with a single channel. I used OpenCV in order to convert them to RGB, by simply appending 2 additional channels with the same value for each pixel.
    - Other images had 4 channels, most likely because they were in RGBA format. After trying to convert them to RGB by simply removing the last channel, I noticed that the results were not satisfying. At that time I was working only with paintings from the style of surrealism and only 9 images out of $\sim$ 7,400 were in

RGBA format, so I decided to simply remove them as they were not a significant part of the data set. I have kept the same approach for the other art styles.

- Another type of problematic images were the **truncated** ones as they caused runtime errors while training. The truncated images were very rare and I chose to remove them from the data set. Cubism had 5 truncated images and romanticism had 1. The other styles had no truncated images, so I didn't put the number of truncated images per style in the following table.
- While loading images with OpenCV in order to check their number of channels, some of the images couldn't be loaded as their name contained non-unicode characters (mostly french specific characters). I renamed these files so that they can be loaded, but a small amount of them still couldn't be loaded so I removed them.

The exact numbers of files of each problematic type are covered in the next table, as well as the remaining total image count for each style. The initial number of images downloaded for each style is also provided for comparison.

Number of images					
Style	Download	RGBA (removed)	Grayscale (converted)	Couldn't load (removed)	Total left
Impressionism	16,440	62	26	0	16,378
Realism	16,099	51	102	2	16,046
Romanticism	14,810	48	695	1	14,760
Expressionism	12,859	31	129	1	12,827
Post-Impressionism	8,305	12	3	0	8,293
Surrealism	7,421	9	86	3	7,409
Baroque	7,125	38	42	4	7,083
Symbolism	5,463	2	8	2	5,459
Abstract Expressionism	4,422	24	3	0	4,398
Cubism	3,453	3	17	1	3,444

Note: the number of downloaded images for each style differ from the number of

available images on the website (from **section 3.1**) because some images are assigned to two consecutive time frames, as their year of appearance is not clear but estimated in a time interval. Also, for cubism and romanticism, 5 and 1 truncated images have also been removed from the data sets and that is why **Total left**, **RGBA** and **Couldn't load** don't sum up to **Download** for those two art styles.

3. I resized each image to have 128 pixels on the smallest side. The other dimension was scaled so as to maintain the width-height ratio. I cropped the central piece of size 128 x 128 of the image and converted it into a tensor: the pixels no longer have natural values in the range [0, 255], but have float values in the range [0, 1]. I normalized the tensor as follows: from the value of each channel of each pixel I subtracted 0.5 and divided the result by 0.5. Finally, the pixels take values in the range [-1, 1]. These two values of 0.5 take the place of mean and dispersion. These are canonical values, they are not obtained as statistics of the data set I use. Finally, the normalized tensor is saved as a .pt file.

   Note: I had to transform each image into a tensor so that I could manipulate it in Pytorch. By default, Pytorch converts [0, 255] unsigned integer pixel range to [0, 1] float pixel range, while converting the image into a tensor. This was a necessary step anyway, because the Generator network's output is a tensor resulted from an activation function (Tanh), and naturally activation functions give float outputs. Most GAN applications use a range of [-1, 1] for the pixel values of tensor images, instead of [0, 1], and Tanh as the activation function in the last layer of the Generator, in order to have the same pixel range for the generated images as for the real images. I have used the same approach, and in order to get the pixel values in range [-1, 1], I subtracted 0.5 from them and divided the result by 0.5 (for each channel). This approach can also be thought of as a normalization step, although those 0.5 values are generally good values for GAN applications and not the mean and the standard deviation computed on the data set. This *normalization* step not only brings the pixel values in the [-1, 1] range that we want, but also helps to speed up the training by (almost) zero-centering the data set.

4. I applied the following method of augmenting the data, for each image (separate from the previous step; the input for this augmentation step is also an image in .jpg, .png. etc. format before being resized, cropped or transformed into a tensor): I resized the image so that it had 143 pixels on the smallest side, analogously the other side was scaled accordingly, in order to keep the aspect ratio of the image. Next I cropped the central area measuring 143 x 143 pixels. Then I used a Pytorch transformation,

namely *FiveCrop*, which crops 5 areas of 128 x 128 pixels from the image received as input as follows: the four corners and the center area. I chose the value 143 for the previous resizing because 128 is almost 90% of 143. I had previously read an article[3] that also aimed to generate art with generative adversarial networks, and the authors were successful with the same division into 5 areas of 90% of the surface of the initial image. The 5 pieces obtained do not differ much, but they do differ enough to improve the paintings generated by a large extent. Each cropped area is then transformed into a tensor and normalized, as in the previous point.

It should be noted that the piece cropped from the center of the image after resizing is different from the one cropped at the previous point (**3**). It contains less of the original painting and can be viewed as a zoom-in. Let's say we are dealing with a painting measuring 400 x 500 pixels (width x height). At the previous point, after resizing we have an image of size 128 x 160. After cropping we have a 128 x 128 image. 16 pixel rows above and 16 below the cropped area were lost. In the second case (FiveCrop), after the first resizing, the image has a size of 143 x 179 pixels. After cropping the 143 x 143 center area, the first 18 pixel rows are lost, as well as the last 18. After cropping the 128 x 128 center area (the center crop from FiveCrop, which extracts 5 different areas from the image), the first 7 and the last 8 rows of pixels, but also the first 7 and the last 8 columns of pixels are lost. The difference can be seen in figure 1.

(a) Center crop. (b) Five crop: center.

Figure 3.3: Cropped image at step 3 vs step 4.

Observation: The resizing, cropping, tensor conversion and normalization were done using an object from the Dataset class, from Pytorch. I had previously used OpenCV for these operations. With the OpenCV preprocessing, the generated images were darker than with the one in Pytorch (which uses the PIL library for working with images).

5. Finally, for the cubism style, I've got from 3,444 paintings saved in jpg., png., etc.

formats and dimensions that even reached 4000 x 4000 pixels, to 20,664 images (6 times more) with 128 x 128 resolution, in .pt format, transformed into tensors and normalized, all in one directory.

6. For training I used a DataLoader object, which loads the data into memory in batches of 64 images and contains a TensorDataset object. The data is already saved in the final form on disk, the TensorDataset object must read them with the **torch.load** function, no other transformations are required at runtime. Each batch of 64 images is actually a tensor with shape (64, 3, 128, 128), where 3 represents the number of channels. The batch is first loaded in the CPU memory and then transferred to the GPU memory for being utilized by the networks.[18]

Chapter 4

Networks Architecture

4.1 Layers Used

Some of the types of layers used are well known, being used in many other types of networks, such as convolutional or feedforward, and are not specific to GANs only. Others are less known, especially to readers who are unfamiliar with generative methods. I will briefly present each one of them to ensure a good understanding of the final architecture that I will talk about in the next section.

- **Convolution**: it is the base layer in most convolutional networks and has the role of distinguishing features in an image.[19; 20] 6 convolutional layers are used in the Discriminator to reduce the size of the image received as input, from 128 x 128 resolution to 1 x 1, i.e. a single value in the range [0, 1], representing the probability that the image is real. No pooling layers are used, each convolution halves the dimensions of the feature map from the current step, except for the last one, which makes the transition from 4 x 4 to 1 x 1. In Pytorch, the corresponding class is *Conv2d*.

- **Transpose Convolution:** intuitively, it is the inverse of the convolution operation, the size of the output being larger than that of the input.[21; 22] In the case of a convolution, a filter of size k x k is applied over an area of size k x k in the image, and the result is the scalar product between the filter and the area in the image, i.e. a single value. In the case of a transposed convolution, a filter of size k x k is also used, but each element in the filter is multiplied by a single element (the same) in the image, the result being of size k x k. Six layers of transposed convolutions are used within the Generator, to increase the input dimensions from 1 x 1 to 128 x 128 (final image). In Pytorch, the corresponding class is *ConvTranspose2d*.

Figure 4.1: Transposed Convolution[1]

- **Batch Normalization:** has the role of making the training faster and more stable, normalizing the data from the current batch[23], according to the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta \tag{4.1}$$

  The mean and standard deviation are computed for the current batch, on each channel. γ and β are parameters that change their value during training, each of them having the same size as the number of channels. Initially, γ is a vector full of 1, and β is a vector full of 0.[24]

  For the Generator, I used Batch Normalization in each transposed convolutional block, except the last one. The position within the block is immediately after the transposed convolutional layer and before the activation function. Similarly for the Discriminator, I used Batch Normalization within each convolutional block, except for the last. The position in the block was between the convolutional layer and the activation function. In Pytorch, the corresponding class is BatchNorm2d.

- **Minibatch Standard Deviation**: is a type of layer made popular by the researchers at Nvidia when they introduced the ProGAN architecture.[17; 11] I did a lot of my experiments with the complete ProGAN architecture, but the results were not satisfactory. Nevertheless, I kept this type of layer because I noticed that it works in other architectures too. How the layer works: divide the current batch (64 examples) into mini-batches (4 examples). Compute the standard deviation for each feature in each space location, within the mini-batch. For each mini-batch, a single value is then obtained by averaging all the features and all the spatial locations. This value is then

[1]Source: `https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8`. Last accessed 10th June 2021.

replicated to have the dimensions of the current feature map (width x height) and concatenated to each example as an additional channel. Thus, each example ends up containing an additional channel, with a single replicated value, which is identical for all examples in the same mini-batch. That value indicates how similar the examples in a mini-batch are.

This layer is used only once in the Discriminator, before the last convolutional block. The purpose of the layer is to encourage the Generator to produce as varied images as possible, and to prevent Mode Collapse. How the layer does this: The standard deviation of a mini-batch of real images will be large almost every time and will have different values for each mini-batch. For a Generator that has started to generate very similar images (possibly Mode Collapse), the standard deviation calculated for images generated from a mini-batch will be very low and almost the same from one mini-batch to another. Thus, the Discriminator is helped to separate the generated images from the real ones, and the Generator, in order to keep up, is guided towards a more varied generation. For the Minibatch Standard Deviation layer to make sense, a batch of images sent to the Discriminator must contain either real images or generated images only. This layer is not implemented by the Pytorch developers.

The following figures contain images generated with two GAN architectures, for the romanticism art style. The only major difference between the architectures is that the Discriminator of the **second** one uses a Minibatch Standard Deviation layer.

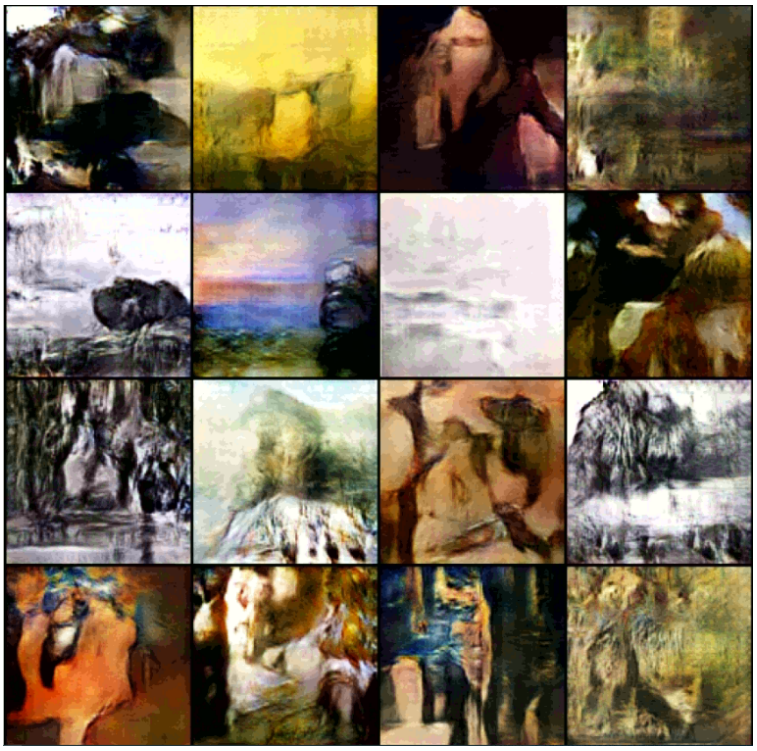

Figure 4.2: Romanticism generated images without Minibatch Standard Deviation.

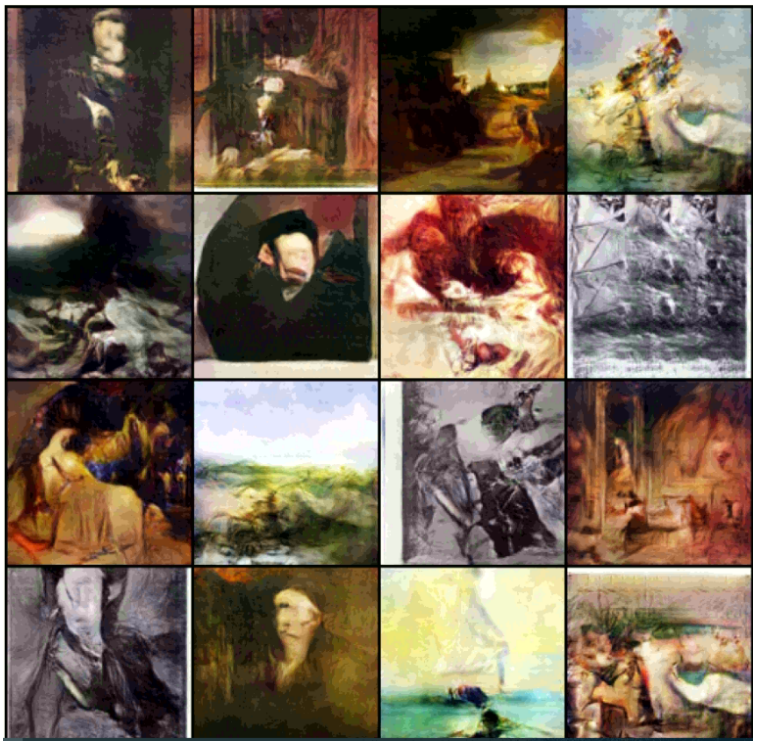

Figure 4.3: Romanticism generated images with Minibatch Standard Deviation.

- **Spectral Normalization**: it is not exactly a network layer, but a normalization method for the weights of a layer.[25; 18] I have used it for each convolutional layer in the Discriminator. It's purpose is to stabilize the training of the Discriminator by rescaling the weight tensor with spectral norm σ of the weight matrix computed using power iteration method. The formula used is the following:

$$\mathbf{W}_{SN} = \frac{\mathbf{W}}{\sigma(\mathbf{W})}, \sigma(\mathbf{W}) = \max_{\mathbf{h}:\mathbf{h}\neq 0} \frac{\|\mathbf{W}\mathbf{h}\|_2}{\|\mathbf{h}\|_2} \tag{4.2}$$

  Where $\mathbf{W}$ is the weight matrix of a layer and $\mathbf{h}$ is the input for the layer.

  Spectral Normalization was introduced in 2018[25] and it has proven useful in many GAN applications, including mine.

A technique I used that is worth mentioning is **One-sided label smoothing**. The labels of the real images shown to the Discriminator are set to 0.9 instead of 1. This way we penalize the network when it is too confident that an image is real i.e it outputs a probability between 0.9 and 1. This method has shown good results in practice as well as for my project.[11]

4.2 Architecture overview

The final architecture uses as a foundation the DCGAN architecture, proposed in 2015.[4] The DCGAN architecture, invented only 1 year after the whole GAN movement started, is a naive architecture compared to those proposed in the following years in the literature, such as WGAN-GP[26], ProGAN[17] or StyleGAN[2], but it has given good results in many applications, and the training time and the volume of data required for convincing results are much smaller, unlike the other networks mentioned. I also experimented with WGAN-GP or ProGAN, but I decided to focus my time and efforts towards improving the DCGAN architecture, for generating paintings from an artistic style, incorporating some techniques proposed in more recent articles and presented in the previous section.

The Generator receives as input a batch of 64 vectors with 128 elements (thought of as channels, in the code). The vectors are generated from a standard normal distribution. This input passes through a transposed convolutional block in order to be scaled from 1 x 1 to 4 x 4 size. There are 5 more blocks of the same type, each of them doubling the width and height of the input. The output has the size 128 x 128, with 3 RGB channels. The first 5 blocks used have the following structure: transposed convolution, batch normalization, followed by the Leaky ReLU activation function, with a slope of 0.2. The last block does

not use normalization, and the activation function is Tanh, which brings the output in the range [-1, 1], just as the pixels of real images are scaled.

The Discriminator receives as input a batch of 64 images of size 128 x 128, with 3 RGB channels. This input is passed through 5 convolutional blocks, the dimensions (width x height) being halved by each block. Next is the **Minibatch Standard Deviation** layer, which receives 4 x 4 size feature maps as input. This layer adds an additional channel, but does not change the width or height of the input. Then follows the final convolutional block, whose output is a real number in the range [0, 1]. Convolutional blocks 2-5 have the following structure: convolution, batch normalization, followed by the activation function Leaky ReLU, with slope 0.2. The first and last block do not use normalization, and the last block uses Sigmoid as an activation function so that the output is a probability. The size of the minibatch within the Minibatch Standard Deviation layer is 4. All 6 convolutional blocks use **spectral normalization** for the weights of the convolutional layer.

Pytorch allows us to print the architecture of a model in a pretty readable format, by simply *printing* the model. However, the blocks are displayed in the order that they have been attributed to the model in the constructor, and not in the order that they are applied on the data, which might be different. In the current case, the only difference is in the **Discriminator**, which uses the **MiniBatchStd** block (displayed first) right before the last (6th) convolutional block. For each convolutional and transpose convolutional layer, the first argument is the number of channels in the input and the second argument is the number of filters used in the layer, which is the same as the number of channels that we want in the output of the layer.

The **Generator** has the following structure:

```
Generator(
  (block_1): GeneratorBlock(
    (deconv): ConvTranspose2d(128, 1024, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_2): GeneratorBlock(
    (deconv): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_3): GeneratorBlock(
    (deconv): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_4): GeneratorBlock(
    (deconv): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_5): GeneratorBlock(
    (deconv): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_6): GeneratorFinalBlock(
    (deconv): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (activation): Tanh()
  )
)
```

Figure 4.4: Generator structure

The **Discriminator** has the following structure:

```
Discriminator(
  (minibatchStd): MiniBatchStd()
  (block_1): DiscriminatorBlock(
    (conv): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_2): DiscriminatorBlock(
    (conv): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_3): DiscriminatorBlock(
    (conv): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_4): DiscriminatorBlock(
    (conv): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_5): DiscriminatorBlock(
    (conv): Conv2d(512, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (activation): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (block_6): DiscriminatorFinalBlock(
    (conv): Conv2d(1025, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (activation): Sigmoid()
  )
)
```

Figure 4.5: Discriminator structure

During training, I used a learning rate of 0.0001 for each network and I trained each GAN for approximately 100 epochs. I kept track of the training progress by visualizing the paintings generated after each epoch for a fixed set of 36 input vectors. If I noticed a decrease in quality after the 100th epoch, I would train for a few more (1-5) until I was satisfied with the images generated for this set of fixed vectors. The **Generator** receives input vectors from a 128 dimensional space, generated from a standard normal distribution. Each training step processes a batch of 64 images. Both networks use the Adam[27] optimizer for updating their parameters during training. The β_1 and β_2 parameters of the optimizer are set to 0.5 and 0.999 respectively, for both networks.

Note: after the **Generator** has been trained and I want to use it for generating images, I do **NOT** set it to evaluation mode, I just leave it as it is. Neural networks that use Batch Normalization layers are usually switched to evaluation mode for inference. This way, those layers use statistics computed during training and not statistics computed on the current batch that is being fed. However, using this evaluation mode approach results in worse looking generated images than the ones displayed during training.

As a consequence of using training mode during inference too, generating one image at

a time also results in lower quality images, as the Batch Normalization Layers use statistics for that single image, while they used statistics computed on a batch of 64 images during training. The optimal approach is to generate batches of a decent number of images (comparable to 64). I use 25 in the web application, even though only 16 images are displayed at a time.

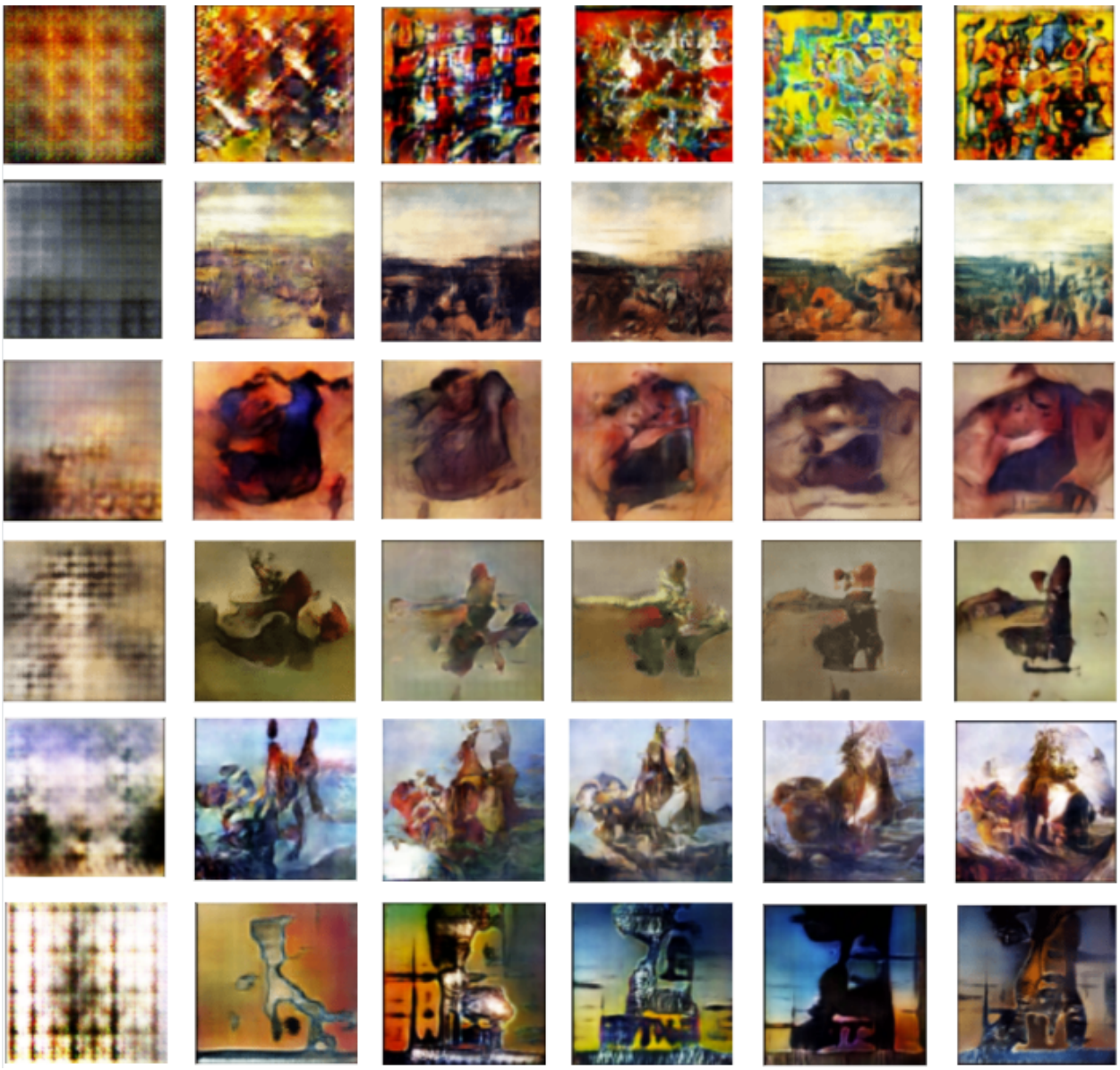

Figure 4.6: The evolution of a painting from each of six different art styles, in order from top to bottom: abstract expressionism, baroque, impressionism, realism, romanticism, surrealism. Each painting is displayed in six states, after the following epochs of training: 1, 20, 40, 60, 80 and 100.

Chapter 5

Technologies Used

5.1 Python

Python is a universally loved programming language because it makes writing/reading code easy and it has a lot of packages for a wide variety of tasks, like Networking, Web Development, Statistics and especially Machine Learning and Deep Learning. These days, whenever someone thinks about writing code for implementing Machine Learning pipelines, Python is the 1st language choice in most cases. It was the same for me. For my project I've used Python along with a couple of useful libraries. I'm going to enumerate the most important ones:

1. **Pytorch**: Deep Learning
2. **OpenCV**: processing images
3. **Matplotlib**: displaying images and graphics
4. **Flask**: developing the web application
5. **Requests**: downloading the images using web requests
6. **Tqdm**: displaying progress bars for time consuming processes (very useful!)
7. **Os**: writing and modifying files on disk

5.2 Pytorch

The bread-and-butter for this project. I used Pytorch in order to convert the data that I downloaded in the form of images, to resized, cropped and normalized tensors, saved on the

disk as .pt files. I also used it to define the architecture of the networks, to train the networks taking advantage of GPU memory (which greatly speeds up the training process[28]) and to generate the images, which was the main goal of my project. In order words, I did everything related to Deep Learning through Pytorch.[18]

For the networks implementation, I used some of Pytorch's predefined layers like *Conv2d*, *ConvTranspose2d* and *BatchNorm2d*, the *spectral_norm* normalization technique, the *BCE* loss function or the *Adam* optimizer for the networks parameters.

5.3 Jupyter Notebook

I used Jupyter Notebook as the coding environment mainly because it provides two very useful features:

- The first one is dividing the code into cells and running any cell I wanted in any order. This is very useful as many times I wanted to run only one of the last cells, for example a cell in which the training function for the networks is called in order to train for 20 epochs, without restarting the training from scratch.
- The second feature is that Jupyter uses a kernel for each .ipynb file (Jupyter specific extension). The kernel starts when we open the file and keeps running until we stop it, by simply shutting it down or stoping the whole Jupyter process. As long as the kernel is running, the variables and functions declared are kept into the RAM memory. This was very useful for scenarios when I wanted to modify a small part of the code and run that part, but without having to run all the necessary code before that part: for example, when printing the shapes and content of tensors for debugging, i didn't have to declare and initialize the tensors multiple times as they were kept into RAM memory since they were first created. Also, if I wanted to train for let's say 70 epochs and then go to sleep without shutting down the kernel, I could continue the training for another 30 epochs the next day, if I wanted.[29; 30]

5.4 OpenCV

I used OpenCV in order to convert the grayscale images to RGB format and to remove the RGBA and truncated images from the data set. OpenCV uses Numpy arrays, so the OpenCV operations I performed are also tied to Numpy. In some of my experiments, when I was working with less images (no data augmentation), I used OpenCV to read images and

create a file containing the whole data set stored as a huge Numpy array. This was useful as I was loading the whole data set into GPU memory.[31]

5.5 Matplotlib

The most often used Python library for visualizing graphics or images. I used Matplotlib in order to visualize the evolution of the training loss for both networks but most importantly to display generated images while training in order to see how well the training progresses. This was incredibly useful because for GANs, unlike other Deep Learning applications, the networks losses are not a very good indicative of how well the training is going. Monitoring the losses is useful especially when the training goes really bad and one of the networks gets far behind the other in terms of performance. When this happens, the loss for one network approaches 0 and the loss for the other network gets stuck at a very large value. But in most cases, when the training is in a more balanced state, I had to look at the images generated after each epoch in order to see if there is progress or not and whether the current session has potential or it would be a better idea to change some hyperparameters and start from scratch. Also, I used Matplotlib to display the final images, after the networks have reached a state that I am happy with (some of these images are also displayed in this document).[32]

5.6 Flask

Flask is a lightweight web application framework written in Python.[33] I used Flask in order to build the final web application which displays images generated by the networks I trained. Flask uses templates (html files) to display information on the website and it allows the developer to write python code inside the html files with the help of Jinja2[1], a templating engine. The logic of a Flask application is implemented through functions with assigned routes. I chose Flask as the web developing framework because it is easy to work with and well suited for small applications. I do not use a database, logging or forms so I didn't need the extra features of a more complex framework, say Django. Also, choosing a more complex framework with multiple layers of abstraction would require writing more code for the same output, in my case.

The main purpose of my project was generating paintings from a particular art style and not the web application itself. The application is only a way of allowing anyone who has access to a computer or phone to generate paintings from the art style he chooses (from the ones available). The generated paintings are unique and so even I, the creator, may

[1] https://jinja.palletsprojects.com/en/3.0.x/

be surprised of what other users might generate. The web application is also a way for me to showcase my art to a wide audience, non-tech people that might not be interested in reading a paper in order to see the images I generated, but would enjoy generating images themselves, by simply going on a website, choosing an art style and pressing a button.

Chapter 6

Experiments

Before settling on the final GAN architecture, data processing pipeline and hyperparameters, I have experimented a lot with different options, changing the code from scratch on multiple occasions. I have also trained networks on three different hardware environments. In this chapter I want to talk about the experiments that I did in order to get to the final state of my project, the fails and successes that helped me gain intuition on how to reach my goal of generating paintings and the trial and error I went through in order to reach that goal.

6.1 Hardware used

The hardware used has a huge impact on a Deep Learning project.[34; 35] Using a modern GPU can reduce the time of training for a neural network from a matter of hours to a matter of minutes.[36; 37] However, not all GPUs are created equal. For example, on Google Colab, where the GPU assigned to a user is almost random (from a pool of available GPU's), getting a Tesla T4 GPU results in a training 2 times slower than getting a Tesla P100.[38; 39] Obviously, this limits the number of training sessions a user can do in a given amount of time, therefore the user cannot experiment as much as he could if he had a better GPU. Another important thing is the amount of disk space available. A lot of disk space means the option of gathering lots of data for the project and even doing data augmentation and saving the augmented examples on disk, instead of doing it at runtime for each training session. Since I started working on this project and until I finished it, I have used three different hardware environments:

1. **My personal laptop**. It has 800 GB of disk space, which was more than enough for keeping a very large data set of pictures, as well performing data augmentation and storing the augmented pictures on the harddisk. However, my GPU is a Nvidia

GeForce GTX 1050 Ti with 4GB memory and 768 CUDA Cores. The GPU is pretty slow, at least compared to the ones provided by Google Colab, which are at least twice as fast.

2. **Google Colab**. Most of the time, Colab assigns the Tesla T4 GPU to the user, but sometimes I was lucky and received Tesla P100, which is 2 times faster (almost 4 times faster than my laptop's GPU). However, there are some pretty big downsides when working with Colab, that I fully experienced.

   The first one is that it only allowed me to use the GPU for like 8 hours in a row, in every 24 hours. At least that was the case in theory. In reality, sometimes Colab would kick me out after only 2 hours, or would suddenly shutdown the kernel in the middle of training (all progress being lost). From my experience, Colab is unreliable for projects that require long training sessions (more than 3-4 hours) and training day after day. This unreliability was by far the most important drawback while using Google Colab, and it has slowed down my progress considerably.

   The second downside is that there are only two available options of storing data, neither of which are satisfactory:

   - The first one is keeping the data on Google Drive and mounting the drive. This is the method I used for a lot of time, as I had unlimited drive space on my institutional Google account. However, reading the images from the drive is slow in Google Colab, as it has to access the drive for each image.
   - The second option is keeping the data in an archive on drive, downloading the archive in the Colab workspace and extracting the files on the disk available in the Colab workspace. However, there are two problems with this approach: we have to download and extract the files from the archive every time we start Colab, so at least once a day. This operation may take even an hour, depending on the size of the archive. The second problem is that the disk space available in the Colab workspace is almost 30 GB, which is not much, especially if we want to perform data augmentation and store the examples on the disk.

3. **A computer that my coordinating teacher made available to me**. I have accessed this computer remotely with the AnyDesk application (similar to TeamViewer). It has an Nvidia Geforce GTX 1080 Ti GPU with 11 GB memory and 3584 CUDA Cores, much faster than my personal laptop and comparable in speed with Colab. It also had 1 TB of disk space available and no limitations in terms of hours of training per day. While training with a huge data set with all art styles at the same time,

over 560,000 images, it took like 40 minutes for a single training epoch, and I would let it run over night and see the results after a bit more than 10 epochs, and let it continue training if the results were promising. On Colab, this kind of experiment would not have been possible, as the kernel might have been shut down after a few epochs. The progress I did on Colab in weeks is comparable to the progress I did on this computer in days, thanks to the reliability (not being kicked out after a seemingly random amount of time), huge disk space and fast data reading (not having to access the drive in order to read an image).

6.2 DCGAN 64 x 64

The first architecture that I tried was the DCGAN architecture for generating images with resolution 64 x 64. I saw that someone else had some success in generating paintings with this architecture.[40] However, they used a Kaggle data set[1] which contains 17,000 paintings from 50 of the most famous artists of all time. I used this architecture as a starting point, but my data set contained only $\sim$ 7,400 paintings from the style of surrealism. My results were pretty good and similar to the ones of the author.

[1] `https://www.kaggle.com/ikarus777/best-artworks-of-all-time`. Last accessed 6th June 2021.

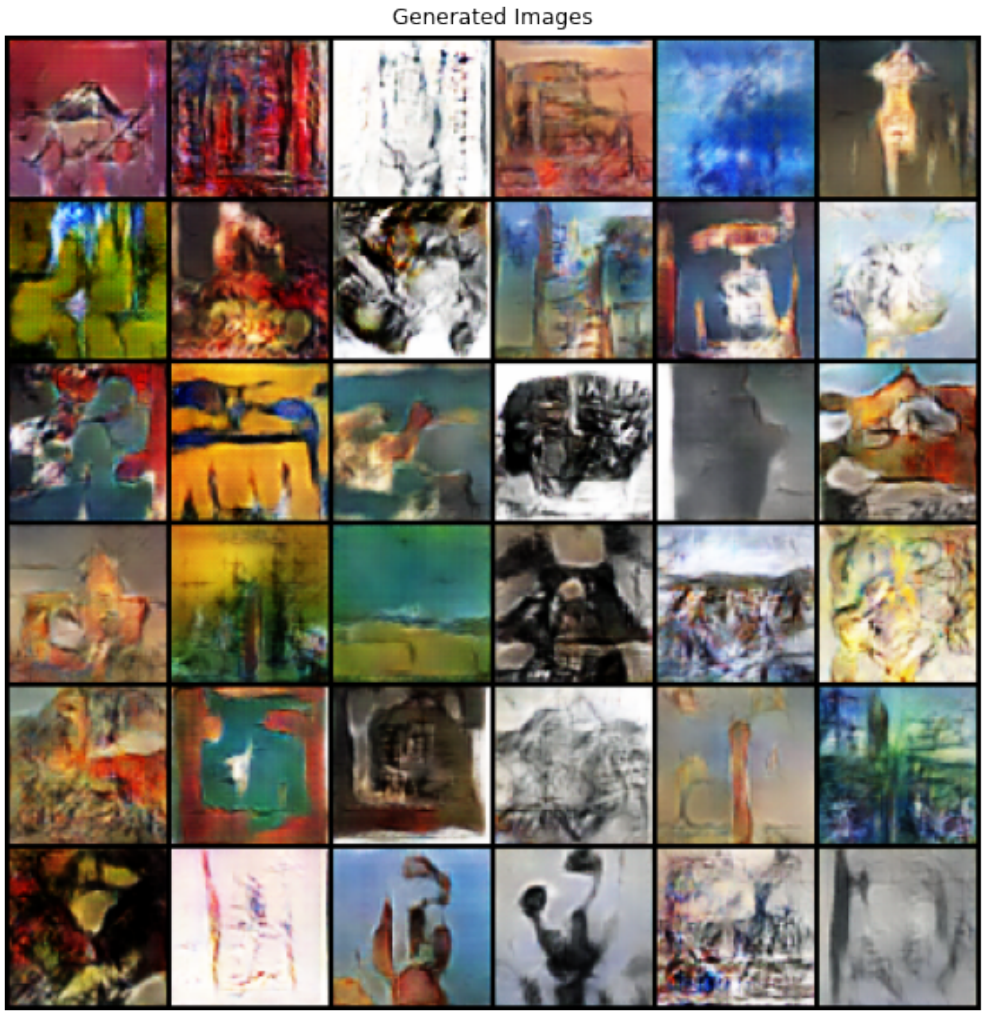

Figure 6.1: DCGAN 64 x 64 generated images for surrealism, no data augmentation.

The **Generator** used had the following structure:

1. Transpose Convolution: 512 filters with 100 channels (the dimension of the input vector space), kernel size 4, stride 1, padding 0; followed by Batch Normalization and ReLU activation function.

2. Transpose Convolution: 256 filters with 512 channels, kernel size 4, stride 2, padding 1; followed by Batch Normalization and ReLU activation function.

3. Transpose Convolution: 128 filters with 256 channels, kernel size 4, stride 2, padding 1; followed by Batch Normalization and ReLU activation function.

4. Transpose Convolution: 64 filters with 128 channels, kernel size 4, stride 2, padding 1; followed by Batch Normalization and ReLU activation function.

5. Transpose Convolution: 3 filters (the number of channels in a RGB image) with 64 channels, kernel size 4, stride 2, padding 1 and Tanh activation function.

The **Discriminator** used had the following structure:

1. Convolution: 64 filters with 3 channels (the number of channels in a RGB image), kernel size 4, stride 2, padding 1; followed by Batch Normalization and Leaky ReLU activation function, with negative slope 0.2.

2. Convolution: 128 filters with 64 channels, kernel size 4, stride 2, padding 1; followed by Batch Normalization and Leaky ReLU activation function, with negative slope 0.2.

3. Convolution: 256 filters with 128 channels, kernel size 4, stride 2, padding 1; followed by Batch Normalization and Leaky ReLU activation function, with negative slope 0.2.

4. Convolution: 512 filters with 256 channels, kernel size 4, stride 2, padding 1; followed by Batch Normalization and Leaky ReLU activation function, with negative slope 0.2.

5. Convolution: 1 filter (for a real number, a probability) with 512 channels, kernel size 4, stride 1, padding 0 and Sigmoid activation function.

6.3 WGAN-GP

After experimenting a bit with the DCGAN architecture for 64 x 64 resolution, I tried it with 128 x 128 resolution with some small changes that proved not so successful. Not being satisfied with DCGAN and knowing that there are other GAN architectures proposed more recently, I implemented the next architecture in the hierarchy of complexity of GANS, the WGAN-GP[26]. Actually, the WGAN[15] is the next step, but when WGAN-GP was introduced, it quickly put its older brother in its shadow.

The WGAN-GP architecture is similar to DCGAN, but it uses a different method in computing the loss function of each network. DCGAN uses binary cross entropy and a sigmoid activation function after the last convolutional layer. A problem with this approach is that when the Discriminator becomes too good, it will output probabilities close to 0 for the realness of images generated by the Generator, and only probabilities close to 1 for the realness of actually real images. This means that its loss approaches 0 if it gets ahead of the Generator and the vanishing gradients problem will appear, so the Generator won't receive any useful feedback and its parameters will be stuck with their current values.[11] However, this problem might not appear during training at all, with carefully tuned hyperparameters

and a balance between the Discriminator and the Generator, although it is true that this balance is very hard to achieve and requires much experimenting.[16]

WGAN-GP aims to avoid the vanishing gradients problem (and also the Mode Collapse problem, that might happen when the Discriminator gets stuck in a local optima) by using a different cost function, called **Wasserstein Loss**, which tries to approximate the **Earth Mover's Distance** between the distribution of real images and the distribution of fake images. The Earth Mover's Distance measures how different the two distributions are by estimating the amount of effort it takes in order to make the generated images distribution equal to the real images distribution.[15]

Wasserstein Loss aproximates the Earth Mover's Distance with the following formula:

$$\min_{g} \max_{c} E(c(x)) - E(c(g(z))) \tag{6.1}$$

Where g is for Generator, c is for **Critic**, the name for the Discriminator used in the WGAN architecture[15], x is the real images and z the input vectors used by the Generator in order to produce fake images. In other words, the Generator aims to minimize the distance between the average of the Critic's predictions for the real images and the predictions for the fake images, while the Critic aims to maximize this distance.

In the WGAN-GP architecture, the Critic (Discriminator) does not use an activation function after the last convolutional layer, and so the output is no longer a probability, but an unrestrained real number. This way, the Wasserstein Loss is not limited between 0 and 100 (the maximum value for the BCE loss in Pytorch). It can range from huge negative values to huge positive values, and it doesn't get stuck at 0 if the Critic gets ahead of the Generator, so we don't encounter the vanishing gradients problem.

For now, I have only talked about the first half of the name WGAN-GP, which stands for Wasserstein GAN. The second half, namely **GP**, stands for **Gradient Penalty**. Wasserstein Loss, as I have defined it above, is not necessarily **valid**. What this means is that it may grow too fast and it may not maintain stability during training. In order to ensure that the loss function is valid, we restrict it to be **1-Lipschitz continuous**, which means that we want its gradients norm to be at most 1 in all of its points.[26] In practice, 1-Lipschitz continuity for the Wasserstein Loss function may be achieved in two ways:

1. **Weight Clipping.**[15] After the weights of the Critic have been updated with gradient descent, we simply replace any weight that has a value larger than **max_value** with **max_value** and any weight that has a value smaller than **min_value** with **min_value**. These values are hyperparameters that we need to tune. However, this is not a very good approach as we impose a hard limit on the freedom and learning

potential of the Critic.

2. **Gradient Penalty** is the preferred option nowadays.[26] It adds a regularization term to the loss function in order to encourage the norm of the gradients to be less than 1 for any point on the function. However, we cannot practically check every point on the function. Instead, we create a set of artificial images by interpolating between the batch of real images and the batch of generated images and then compute the gradients of the predictions of the Critic with respect to the features of the artificial images (each channel of each pixel). The interpolation parameter is chosen from a standard uniform distribution and it is different for each pair of real-fake images. The final formula for the WGAN-GP loss is as follows:

$$\min_g \max_c E(c(x)) - E(c(g(z))) + \lambda E(\|\nabla c(\hat{x})\|_2 - 1)^2 \tag{6.2}$$

   Where $\hat{x}$ is the artificial images and λ is a hyperparameter, typically set to 10.

In Pytorch, I computed the Wasserstein Loss for each network separately with different formulas:

- **Critic (Discriminator)**: mean of the predictions for fake images - mean of the predictions for real images + gradient penalty. This way, I encourage the network to output small values for fake images and big values for real images, so that the loss is as small as possible (a huge negative number if possible).
- **Generator**: - mean of the predictions of the Critic for fake images (generated by the Generator). This way the the Generator is punished harder when the Critic outputs small values for the images generated.

I didn't have much success with WGAN-GP, but I also didn't experiment much with it. The reason for the last statement is that there was already another famous GAN architecture that I couldn't wait to implement and experiment with, an architecture that actually uses the same loss as WGAN-GP, even the Gradient Penalty term, so all that I said in this section also applies to the next architecture: **ProGAN**.

6.4 ProGAN

This architecture was introduced by researchers at Nvidia in 2017 and it is the first from a series of highly successful GANs that advanced the state of the art in the domain: ProGAN[17],

StyleGAN[2], StyleGAN2[41]. Until ProGAN, GANs were able to generate convincing images for resolutions up to 128 x 128 resolution (or maybe even 256 x 256).[42] ProGAN pushed the limits and was able to generate human faces in 1024 x 1024 resolution that would fool most people into thinking they are images of real people. The images in the following figure were generated with ProGAN and have been taken from the original paper.[17]

Figure 6.2: ProGAN generated human faces.

ProGAN used a couple of interesting features in order to reach this degree of quality:

- **Progressive Growing**. The GAN does not produce images with the resolution that we are interested in right from the start. It divides training into several steps, each step working on a different resolution, starting from 4 x 4 and increasing by a factor of 2 after each step. Working with one resolution at a time and starting from a very small one (4 x 4) means that the networks start with an easy task that gets harder as time goes by[43]. When the resolution is changed from let's say 32 x 32 to 64 x 64, the networks do not start from scratch. They have already been trained for resolutions 4 x 4 through 32 x 32. This approach should result in a more stable training and a potential of generating high quality images even for high resolutions (1024 x 1024 for human faces) as the complexity of the task increases only after the networks perform well enough at the current level.

  This change in resolution from one step to another isn't sudden. After a given number of training steps, a new block of layers is appended to each network in order to double the resolution at which the networks operate. However, the output of this new block is interpolated with the upsampled output of the previous block. The interpolation parameter α goes from 0 to 1 over a couple of training epochs so that the new block is slowly introduced into the architecture and the transition from a resolution to the

next one is as smooth as possible. This process is illustrated in the next figure, for resolutions 16 x 16 and 32 x 32.[17]

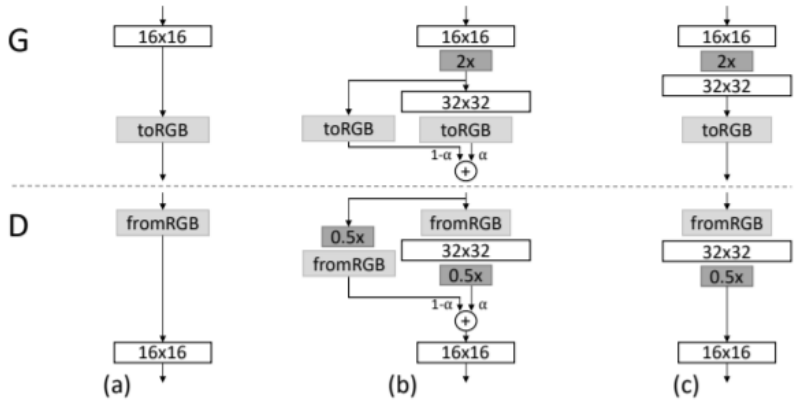

Figure 6.3: Transition from 16 x 16 to 32 x 32 resolution in each network. (a) only 16 x 16 resolution (b) fading in the new block (c) the new block has fully faded in and it is no longer interpolated with the upsampled output in the previous block.[17]

- **Equalized Learning Rate**. The weights of the convolutional layers are initialized from a standard normal distribution as opposed to a carefully thought initialization, like from a uniform distribution of mean 0 and standard deviation 0.02. To account for this rough initialization, the authors of ProGAN scale the weights at runtime using the per-layer normalization constant from He's initializer (c)[44] as follows: $\hat{w}_i = w_i/c$. Modern optimizers such as Adam[27] or RMSProp[45] normalize the update of a gradient by its estimated standard deviation, so the update is independent of the scale of the parameter. This means that parameters with larger range take longer to converge than those with smaller range, therefore the learning rate can be considered too small for some parameters and too large for others. Equalized Learning Rate scales the weights in order to have the same range, resulting in the same speed of convergence.[46]

- **Pixelwise Feature Vector Normalization in Generator**. After each convolutional layer, the feature vector corresponding to each pixel in the feature map is normalized to unit length with the following formula:

$$b_{x,y} = a_{x,y} / \sqrt{\frac{1}{N} \sum_{j=0}^{N-1} (a_{x,y}^{j})^2 + \epsilon} \tag{6.3}$$

where $a_{x,y}$ and $b_{x,y}$ are the pixel at position x, y in the feature map before and af-

ter normalization, N represents the number of channels and $\epsilon = 10^{-8}$. The aim of this technique is to prevent the out of control escalation of signal magnitude in both networks. It is applied only after the convolutional layers of the Generator.[17]

- **Minibatch Standard Deviation**. This is a layer used only once in the Discriminator, right before the last convolutional block. Its purpose is to increase the variety of the generated images and to prevent Mode Collapse. It does that by incorporating statistics related to the variation of images inside a batch to the feature map, as an additional channel.[11; 17] This layer is explained in more detail in the **Layers Used** section from the **Networks Architecture** chapter.

My implementation of ProGAN is faithful to the original paper[17] and the structure of both networks can be seen in the following figure. However, as I was aiming for 128 x 128 resolution and not 1024 x 1024, my Generator didn't contain the last three blocks in the figure, and the Discriminator the first 3.

Generator	Act.	Output shape	Params
Latent vector	–	512 × 1 × 1	–
Conv 4 × 4	LReLU	512 × 4 × 4	4.2M
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Upsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Upsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Upsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Upsample	–	512 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	1.2M
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Upsample	–	256 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	295k
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Upsample	–	128 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	74k
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Upsample	–	64 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	18k
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Upsample	–	32 × 1024 × 1024	–
Conv 3 × 3	LReLU	16 × 1024 × 1024	4.6k
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 1 × 1	linear	3 × 1024 × 1024	51
Total trainable parameters			**23.1M**

Discriminator	Act.	Output shape	Params
Input image	–	3 × 1024 × 1024	–
Conv 1 × 1	LReLU	16 × 1024 × 1024	64
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 3 × 3	LReLU	32 × 1024 × 1024	4.6k
Downsample	–	32 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Conv 3 × 3	LReLU	64 × 512 × 512	18k
Downsample	–	64 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Conv 3 × 3	LReLU	128 × 256 × 256	74k
Downsample	–	128 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Conv 3 × 3	LReLU	256 × 128 × 128	295k
Downsample	–	256 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Conv 3 × 3	LReLU	512 × 64 × 64	1.2M
Downsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Downsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Downsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Downsample	–	512 × 4 × 4	–
Minibatch stddev	–	513 × 4 × 4	–
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
Fully-connected	linear	1 × 1 × 1	513
Total trainable parameters			**23.1M**

Figure 6.4: Structure of Generator and Discriminator of ProGAN.[17]

I have dedicated perhaps more than half of the time spent experimenting with GAN architectures to ProGAN. I had high hopes as it seemed much more complex and with a

huge potential in any GAN project. However, I didn't achieve much with it. At first I was using Google Colab and did not use any data augmentation. I was using a data set of 27,000 images comprised of four different styles gathered together: surrealism, cubism, abstract expressionism and expressionism. Given the time limitations of Colab, I was only able to train once or twice a day and didn't even reach 128 x 128 resolution because at 64 x 64 resolution the images were looking really bad and showed no sign of improvement from one epoch to another. Also, even though I was keeping the whole data set for each resolution in GPU memory (one resolution at a time), a training epoch for 64 x 64 resolution took about 20 minutes and for 128 x 128 about twice as much. If I was lucky to get the Tesla P100 GPU that waiting time would be halved, but this happened like once every four days. Given the unreliability of Colab and the small data set used, my results didn't improve much and I was stuck for a while.

After I received access to a good computer that allowed me to train for as long as I wanted, I decided to try the same ProGAN architecture but with 166,000 images. This data set is the same one comprised of only four art styles but each style was augmented in order to have 6 times as many images. I did not use all 10 art styles because I already expected it to train slowly with only those 4 styles. I knew that ProGAN is much more data hungry compared to DCGAN or WGAN-GP so I didn't focus on training with only one art style. I wanted to see decent results with a much larger data set (because those results should be easier to achieve), before trying with a data set of only one style. However, after multiple experiments, some of them using less convolutional filters in later layers, in order to reduce the training time (which was huge, more than an hour for an epoch with 64 x 64 resolution), I didn't manage to obtain satisfying generated images.

An example of one of my last experiments with ProGAN is as follows: data set of 166,000 images, number of filters per block (in both networks): 512, 512, 256, 128, 64, 32 instead of 512, 512, 512, 512, 256, 128 as in the paper and a training of 10 epochs per resolution scale. An epoch took 1:49 minutes for scale 4 x 4, 3:47 minutes for scale 8 x 8, 8:19 minutes for scale 16 x 16, 14:23 minutes for scale 32 x 32, 26:45 minutes for scale 64 x 64 and about 52 minutes for scale 128 x 128. I stopped the training before reaching 128 x 128 because since the 32 x 32 resolution, the images looked really bad and there was no sign of improvement during the training for resolution 64 x 64 either.

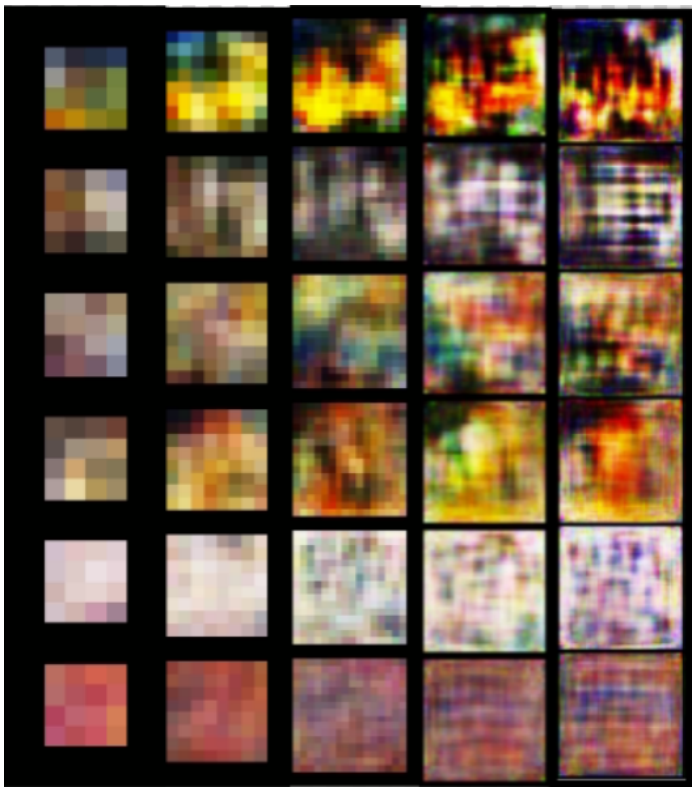

Figure 6.5: Evolution of a fixed set of images from a resolution to another. Each column represents a resolution from 4 x 4 to 64 x 64 and each row represents one image.

In order to see if an experiment with ProGAN had potential, I had to train for each resolution until I reached 64 x 64 and only then I was able to decide whether to continue training or start again from scratch with different settings, because for smaller resolutions, the evolution seemed decent in every training session.

After trying different sets of hyperparameters with ProGAN and seeing pretty much the same results over and over and having to wait even a whole day for a training session I decided to go back to DCGAN and dedicate my time to improving it. A key factor in this decision was discovering an article in which the authors stated that they had some success in generating art using a variant of DCGAN.[3] The authors used a data set of pictures from 25 different styles, but the total amount of pictures was 81,449. I am using a data set of 96,097 images from only 10 different styles. Their generated images had 256 x 256 resolution. However, they handpicked the generated images that they show in the article and their goal isn't generating paintings based on art style. The next architectures that I implemented and that I'm going to describe in the following sections are different in terms of structure from the one presented in the previously mentioned article. I didn't use the article in order to copy the architecture presented by the authors, but only to have a rough idea about how to make DCGAN work: learning rate, data augmentation method, whether the number of

layers I already used was a good choice and whether adding much complexity to the model is necessary. Actually. the architecture I ended up with is very different than the one in the article, but the article served an important purpose: it gave me confidence that spending my time and energy on **DCGAN** was a good choice.

6.5 DCGAN 128 x 128 (not final architecture)

Right after giving up on ProGAN, I tried the **DCGAN** architecture again but with a larger data set comprised of four different art styles (surrealism, cubism, expressionism, abstract expressionism) gathered together and augmented in order to be 6 times larger: a total of 166,000 images. This architecture was pretty much the same as DCGAN for 64 x 64 resolution but with an additional block in both the Discriminator and the Generator, with 1,024 filters for the convolutional or transpose convolutional layer. This experiment was different from the one I did right after setting a starting point with DCGAN 64 x 64 only by the size of the data set, which had only about 7,400 images from the style of surrealism and no data augmentation. The images that I generated looked a lot better than in the case of the previous experiment.

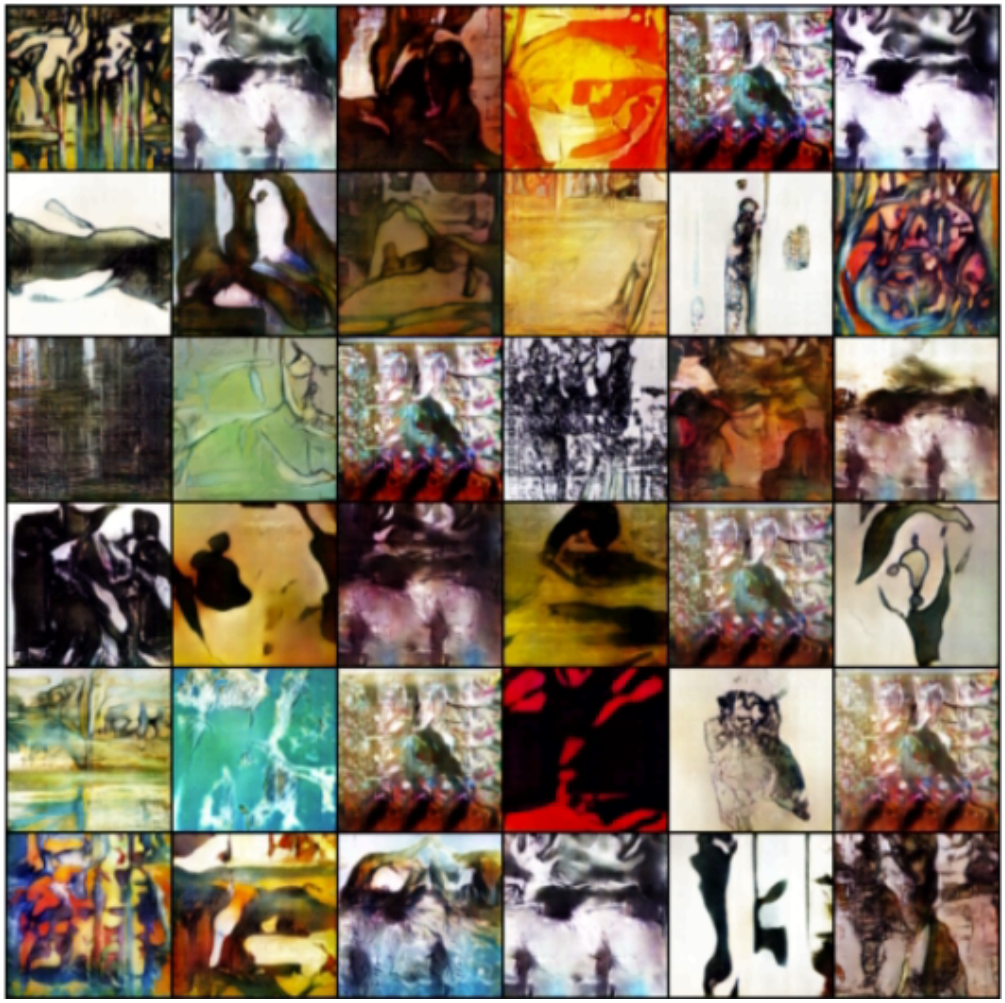

Figure 6.6: Images generated with DCGAN for 128 x 128 resolution with a data set of 166,000 images (after being augmented 6 times) comprised of four art styles.

Adding the other 6 styles to the data set for a total of more than 560,000 images brought some more variety as some images generated started to resemble landscapes.

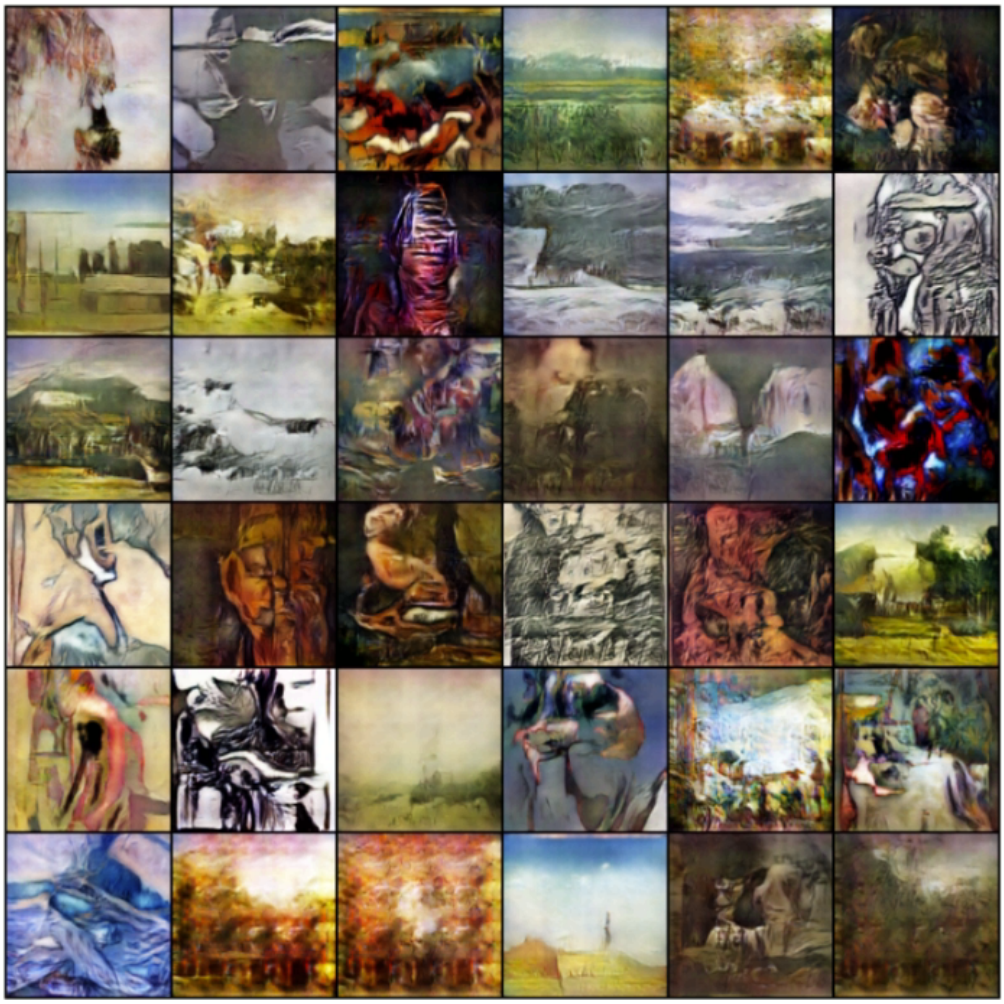

Figure 6.7: Images generated with DCGAN for 128 x 128 resolution with a data set of 560,000 images (after being augmented 6 times) comprised of all 10 art styles.

However, my original goal was not only generating paintings randomly but rather paintings that resemble a particular art style. In order to reach this goal, I saw two options:

1. Converting this DCGAN into a **Conditional GAN**[47] that still uses the whole 560,000 images data set but assigns each image a label from 0 to 9 representing the art style that it comes from. This new GAN would generate paintings given a label for the art style desired. This was the first option that I tried and I am going to talk about it more in the next section.

2. Training a DCGAN for each style separately, using only that style as the data set. This is actually the approach that I settle on as the best for my project.

6.6 Conditional GAN

The **Generator** of the **Conditional GAN**[47] is similar to the one of **DCGAN**, as it also has 6 convolutional blocks that operate with resolutions from 4 x 4, 8 x 8,... to 128 x 128, each block containing a Transpose Convolution, a Batch Normalization layer and a Leaky ReLU activation function with negative slope 0.2. The difference is that the input vector is concatenated to the embedding of the class label of the image, before being fed to the Generator network. This embedding is obtained by getting the class label (0-9) through a Pytorch **Embedding layer** that outputs a 10 channel representation. Each label has a different representation and two equal labels have the same representation. This Embedding layer has weights and is being trained just like the convolutional layers, so the representation of each label changes during training.

The **Discriminator** is also similar to the one in the **DCGAN** architecture. This network also needs the label of the image to be appended to the image in some way. I did that by getting the label through an **Embedding layer** as in the case of the Generator, but the 10 dimensional vector output is upsampled with nearest neighbors method into a 128^2 dimensional vector which is then reshaped into a 128 x 128 matrix and appended to the image as an additional channel, before being fed to the Discriminator.

The output of the Discriminator is still a probability, as I kept the Sigmoid activation function for the final block. However, now the Discriminator no longer answers to the question *Is this a real image?* but to the new question *Is this a real image from the style of x?*, where x is the style associated with the label of the image. The real images are obviously labeled based on the art style they belong to. This labeling is done by Pytorch if we store the data set in the following way: a directory containing 10 other directories, each of them containing all the images from one style. I used a **Dataloader** object for shuffling the data set and receiving one batch of images at a time. This object assigns a label from 0 to 9 to each image in a batch based on the index of the directory where the image is stored. For the fake images, the Generator received a random label for each input vector.

The results after training with 166,000 images from four styles (same ones as before) were okay but images generated for one style were not so different than the ones generated for the other styles, they didn't show any distinct characteristics related to their style. Many training sessions, after many epochs of training, ended with the phenomenon of **Mode Collapse**. The next two figures show some images generated in a training session before and after Mode Collapse, which occurred after 55 epochs of training, and in a pretty interesting way: the images do not look much alike as is the case for most Mode Collapse events, but many of them contain a checkerboard pattern. Also the loss of the Discriminator is constantly 100

while the loss of the Generator stays at 0.

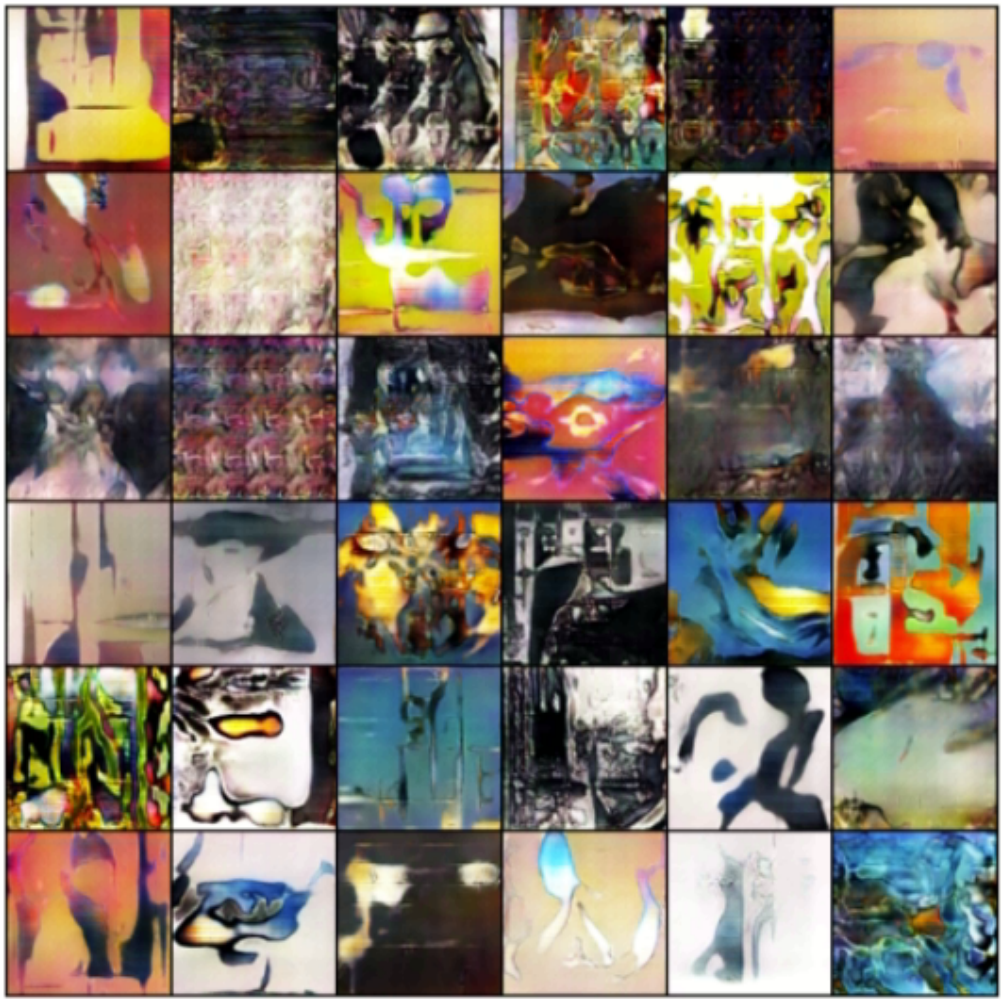

Figure 6.8: Images generated with Conditional GAN trained on a data set of 166,000 images from four styles. The 1st and 5th row contain images for abstract expressionism, the 2nd and 6th for cubism, the 3rd for expressionism and the 4th for surrealism.

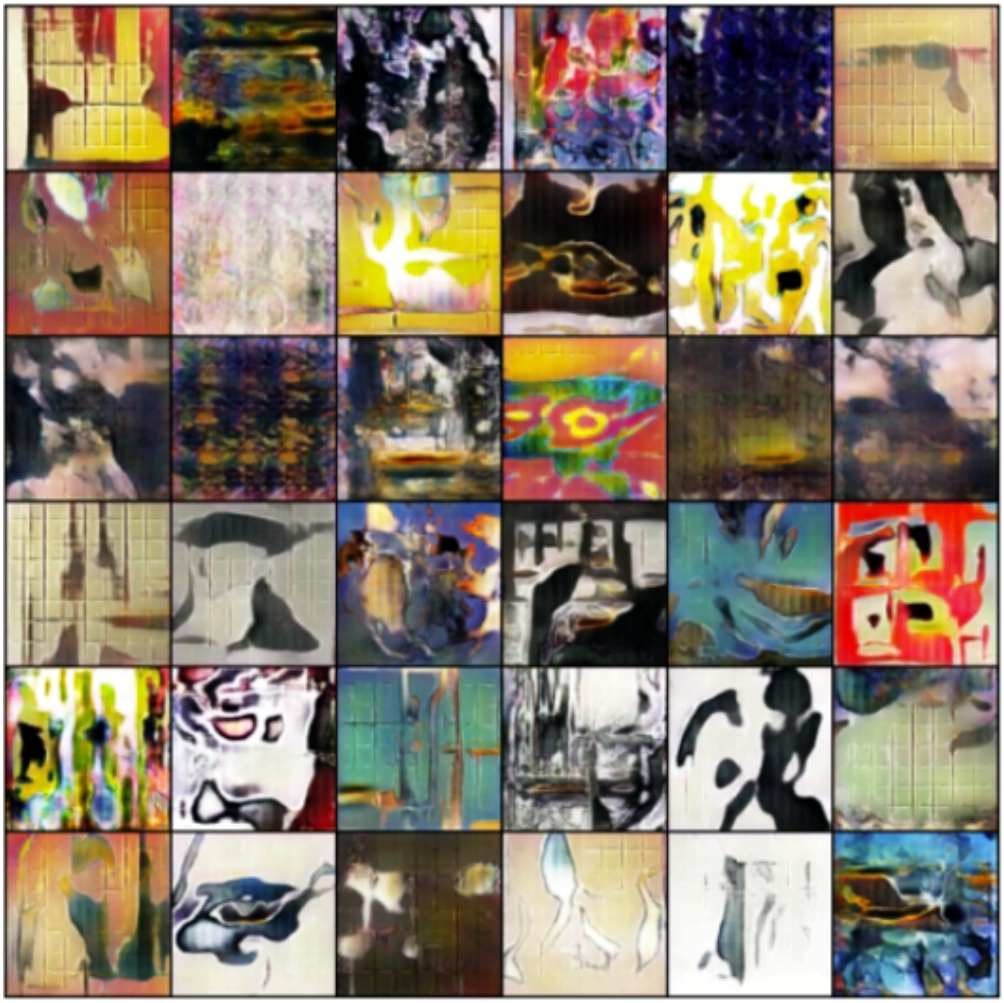

Figure 6.9: The same images as in the previous figure but after one more training epoch, in which Mode Collapse occurred.

After using the whole data set with 560,000 images I tried to generate paintings from each of the 10 art styles. However, every training session resulted in Mode Collapse for at least one art style at some point during training. One particular training session was special because the mode collapsed after like 5 epochs for one art style but I decided to let it train more in order to see if the other 9 styles will look good in the end. After like 10 other epochs, that style suddenly went out of Mode Collapse (first time I saw that happen) and was fine again (not generating the same image over and over, but actually different images), but after the next epoch, another style fell into Mode Collapse and didn't recover later so my happiness was quickly gone. Also, even tough for most experiments only one or two styles collapsed and started to produce the same image over and over, the Generator wasn't producing images that I was satisfied with for the other styles either. After adding a Minibatch Standard Deviation layer to the Discriminator and Spectral Normalization in the

convolutional layers the results improved a bit but not enough so I tried the other option: training a **DCGAN** for each style separately.

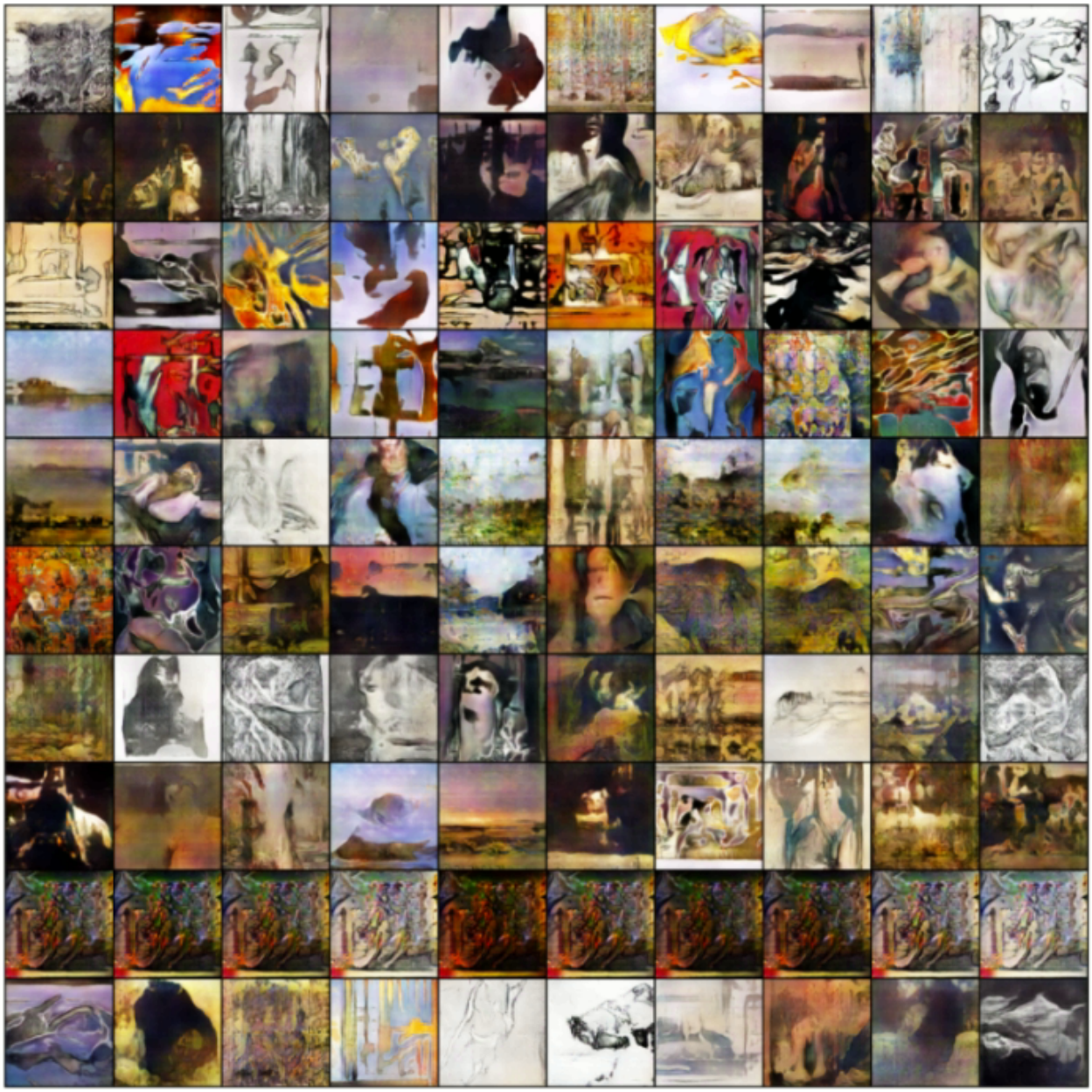

Figure 6.10: Images generated with Conditional GAN trained on a data set of 560,000 images from all 10 art styles. Each row represents an art style, in the following order: Abstract Expressionism, Baroque, Cubism, Expressionism, Impressionism, Post-Impressionism, Realism, Romanticism, Surrealism, Symbolism. The architecture used Minibatch Standard Deviation and Spectral Normalization but the mode still collapsed for one style: Surrealism

6.7 Experiments with 256 x 256 resolution

At some point during my experiments, I decided to take a break from generating 128 x 128 images and try some architectures for generating 256 x 256 resolution paintings. I wasn't confident that I would be able to generate convincing paintings from a particular style at this resolution because I had already seen how difficult the task of generating 128 x 128 paintings was, but I wanted to give it a try nonetheless.

- **ProGAN**. I did not try this architecture as I had really poor results even for 64 x 64 resolution. Also, given my available hardware (GPU mostly), a training session would have taken a couple of days so I quickly discarded this architecture option.
- **WGAN-GP**. As I said in the section where I talked about this architecture, which is basically DCGAN with the Wasserstein loss function, I didn't experiment much with this for 128 x 128 resolution and the experiments I did had poor results. I had much more experience with DCGAN and I knew that with a good set of hyperparameters (I wasn't able to find them for WGAN-GP), those two architectures have the same potential.[48; 16] I did not try this one for 256 x 256 resolution.
- **DCGAN**. I did two kinds of experiments with this architecture.

    1. Training with a data set comprised of four art styles augmented: abstract expressionism, cubism, expressionism and surrealism, with a total of 166,000 images. A training epoch took 43 minutes. A full day of training meant 33 epochs, so a thorough training would have taken 2-3 days. If I used the whole 560,000 images data set, the training would have been more than 3 times slower. The results I got with the data set of 166,000 images were not convincing. Mode Collapse appeared during every training session and the images generated where split into a couple of groups, the paintings in each group looking almost the same way.

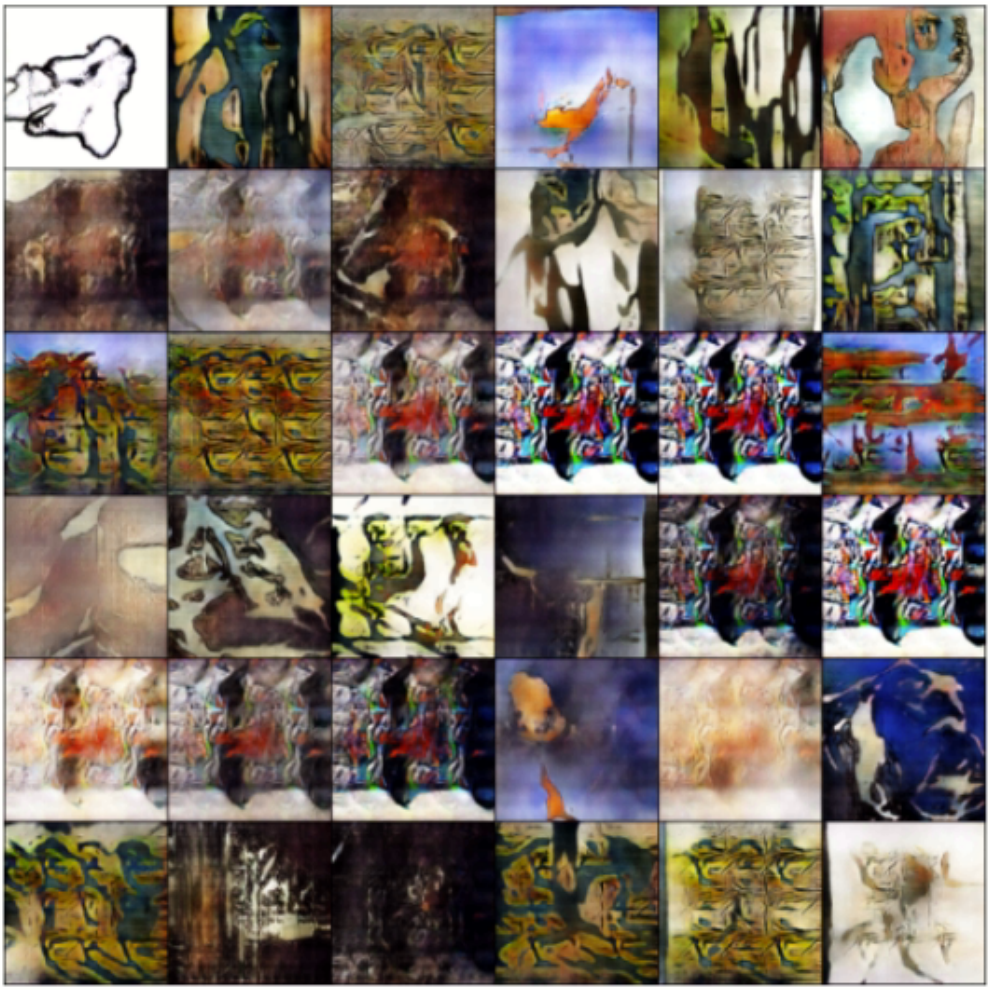

Figure 6.11: Images generated with DCGAN for 256 x 256 resolution with a 166,000 images data set. There is low variety as the mode collapsed.

2. Training with a data set comprised of a single art style. I have tried this option after I already had good results with the final architecture, described in the **Networks Architecture** chapter. Most of these experiments used expressionism as the art style, as it had a large number of paintings, 12,827 (76,962 augmented) and because I had already been able to generate good looking paintings for 128 x 128 resolution for expressionism. However, I wasn't able to prevent Mode Collapse and wasn't satisfied with the results.

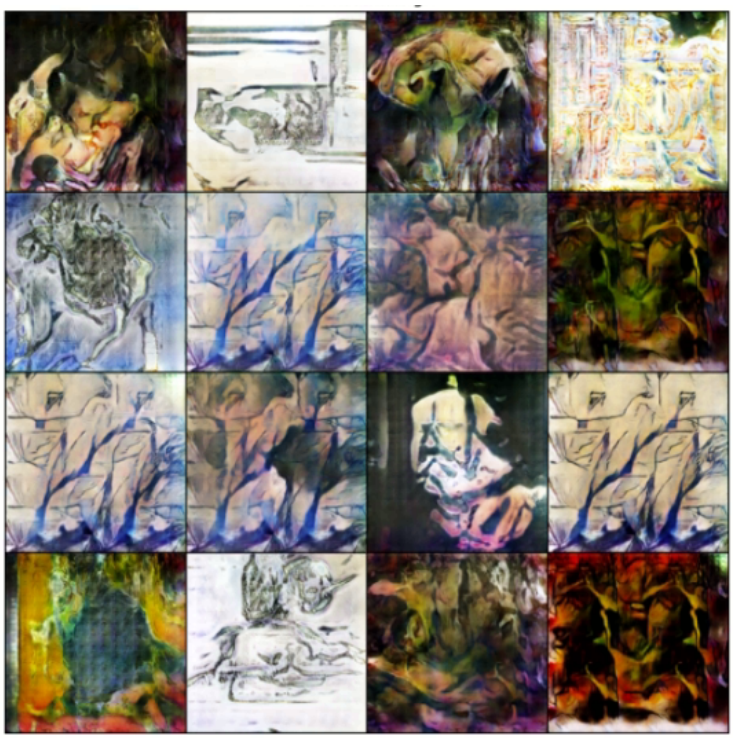

Figure 6.12: Images generated with DCGAN for 256 x 256 resolution with a 76,962 images data set (expressionism). Some images are very similar to one another, a sign of Mode Collapse.

- **Conditional GAN**. With this architecture I used the data set with 166,000 images from the four styles mentioned above, so I tried to generate images based on one of four styles, using labeled data. Even tough I was ready to invest the most effort into this architecture, compared to the ones mentioned previously in this list, I couldn't manage to get any further than generating images that are a bit better than random noise. The mode collapsed very early and given that an epoch took over 40 minutes, as was the case with DCGAN, I gave up on this architecture and did not experiment with the whole data set of 560,000 images.

At this point in time, I was already satisfied with the results at 128 x 128 resolution, and my nerves were giving in so I decided it was the time to renounce 256 x 256 and fully embrace 128 x 128 resolution.

Chapter 7

Paintings generated with the final architecture

The final architecture used is presented in the **Networks Architecture** chapter. In this chapter I want to display a collage of 20 of my favorite paintings generated for each of the 10 art styles. These images are handpicked.

Some of the images are highly complex. Even though what they depict is not obvious at a glance, in many cases this is exactly their appeal. I asked different human subjects about what they believe to be depicted in some of the images and I received a wide variety of answers. People seemed to prefer different pictures, which is quite encouraging, because I believe art is supposed to resonate differently with each person, based on their personality. Moreover, even tough I stared at each of those 200 pictures individually for a considerable amount of time, I was unable to notice shapes, symbols or even emotions that other people saw/felt. This means that this kind of art depends a lot on the perspective of the viewer.

Another important aspect is that the paintings generated for one art style are quite different than the ones generated for the other styles, which was one of my main goals from the very start of this project. Also, the images generated for one art style respect the widely accepted definition of that style, or at least have some of the defining characteristics of the style.

I believe that I have accomplished my goal of generating good looking paintings that resemble a particular art style.

7.1 Abstract Expressionism

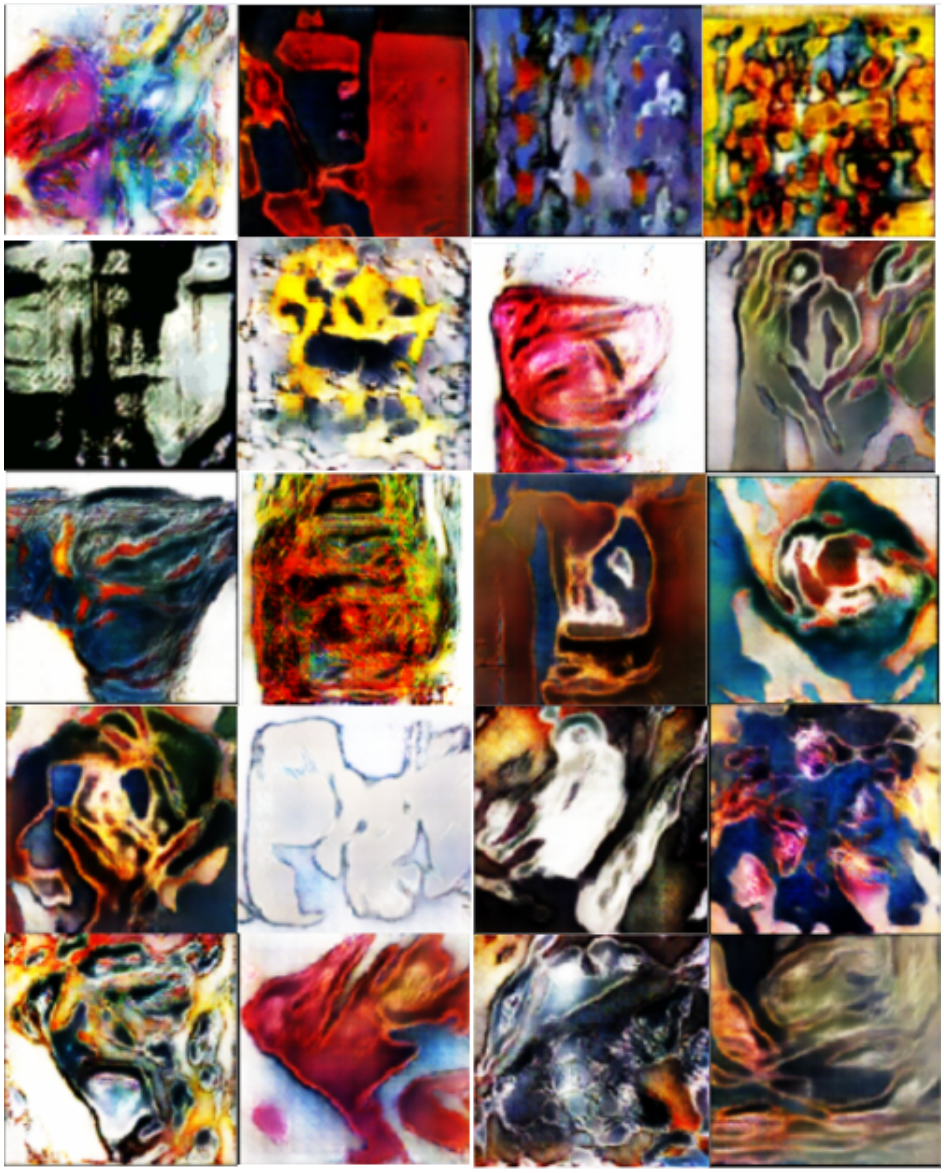

Figure 7.1: Abstract Expressionism

7.2 Baroque

Figure 7.2: Baroque

7.3 Cubism

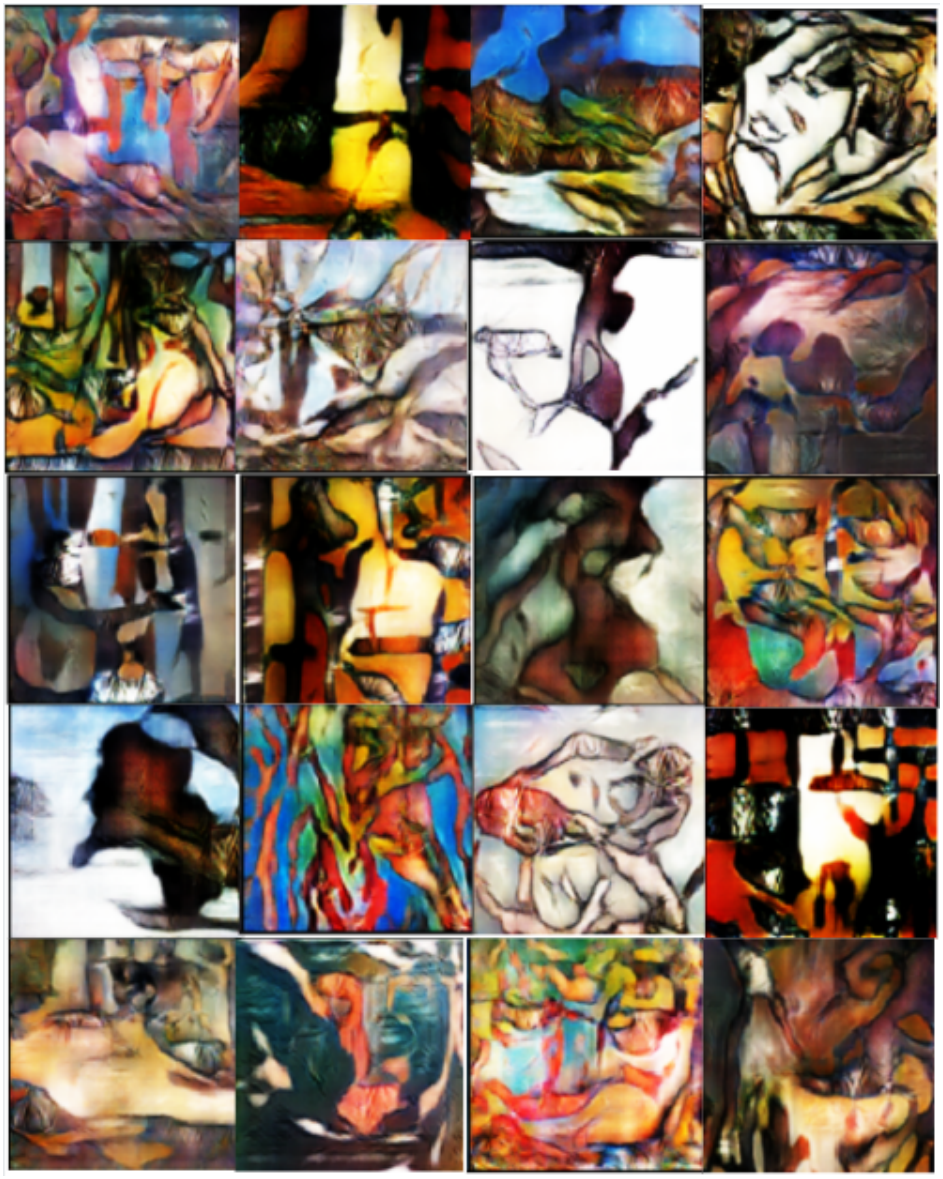

Figure 7.3: Cubism

7.4 Expressionism

Figure 7.4: Expressionism

7.5 Impressionism

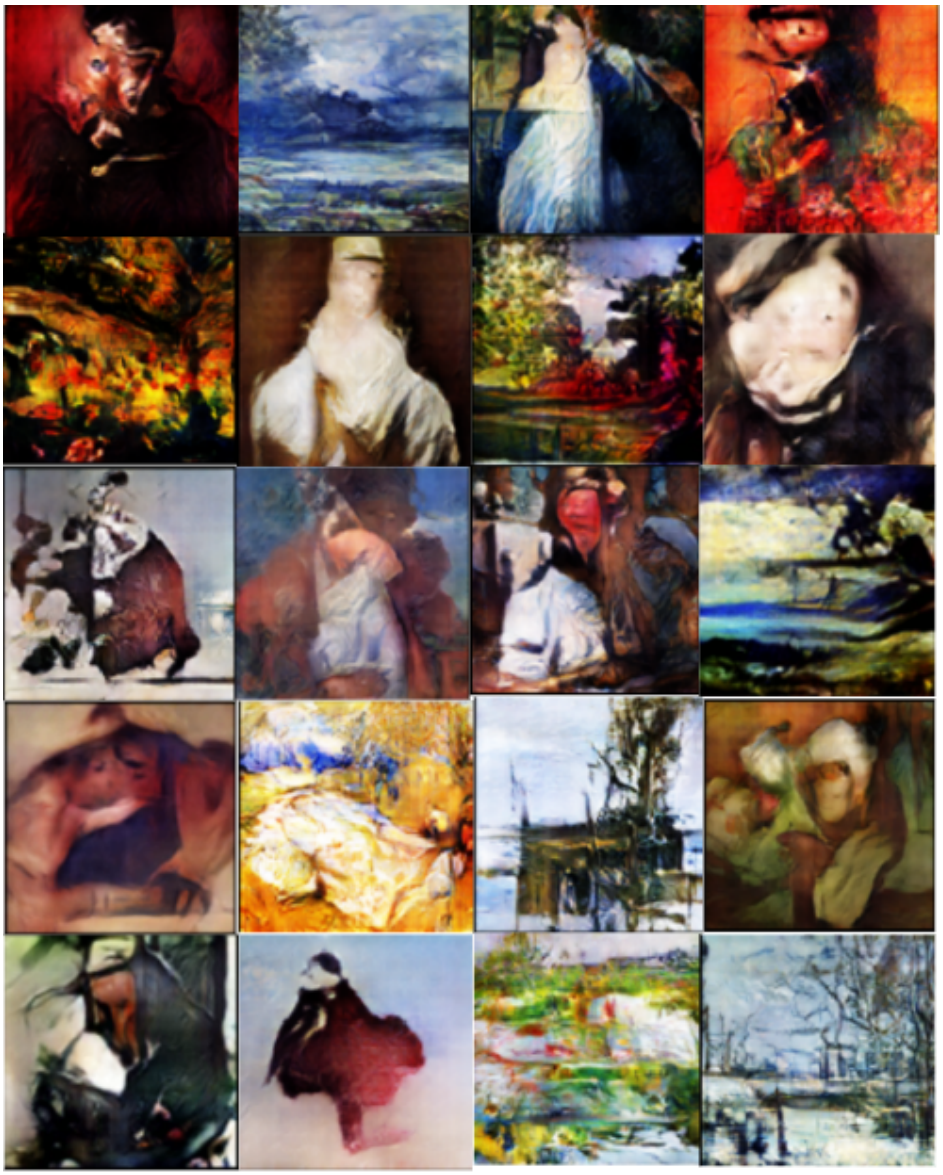

Figure 7.5: Impressionism

7.6 Post-Impressionism

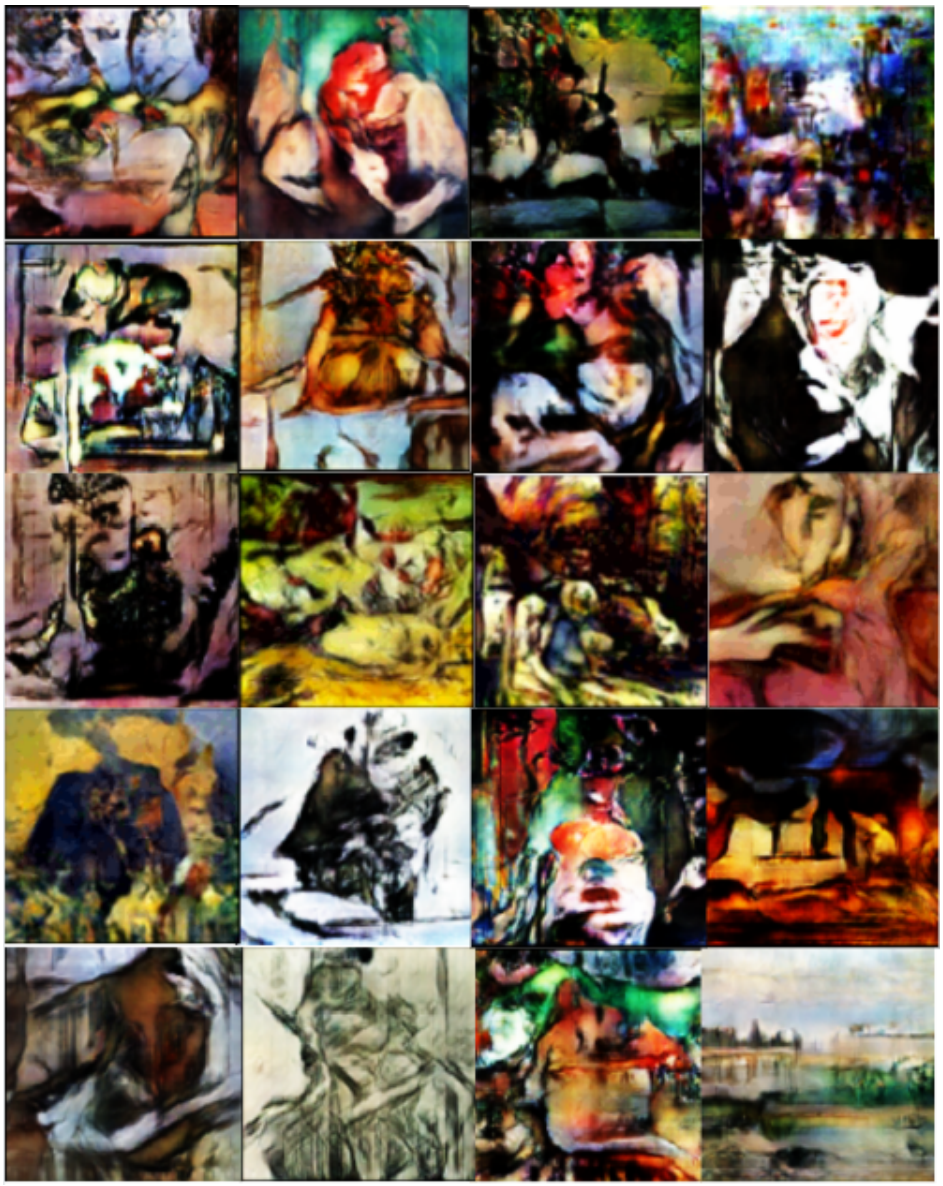

Figure 7.6: Post-Impressionism

7.7 Realism

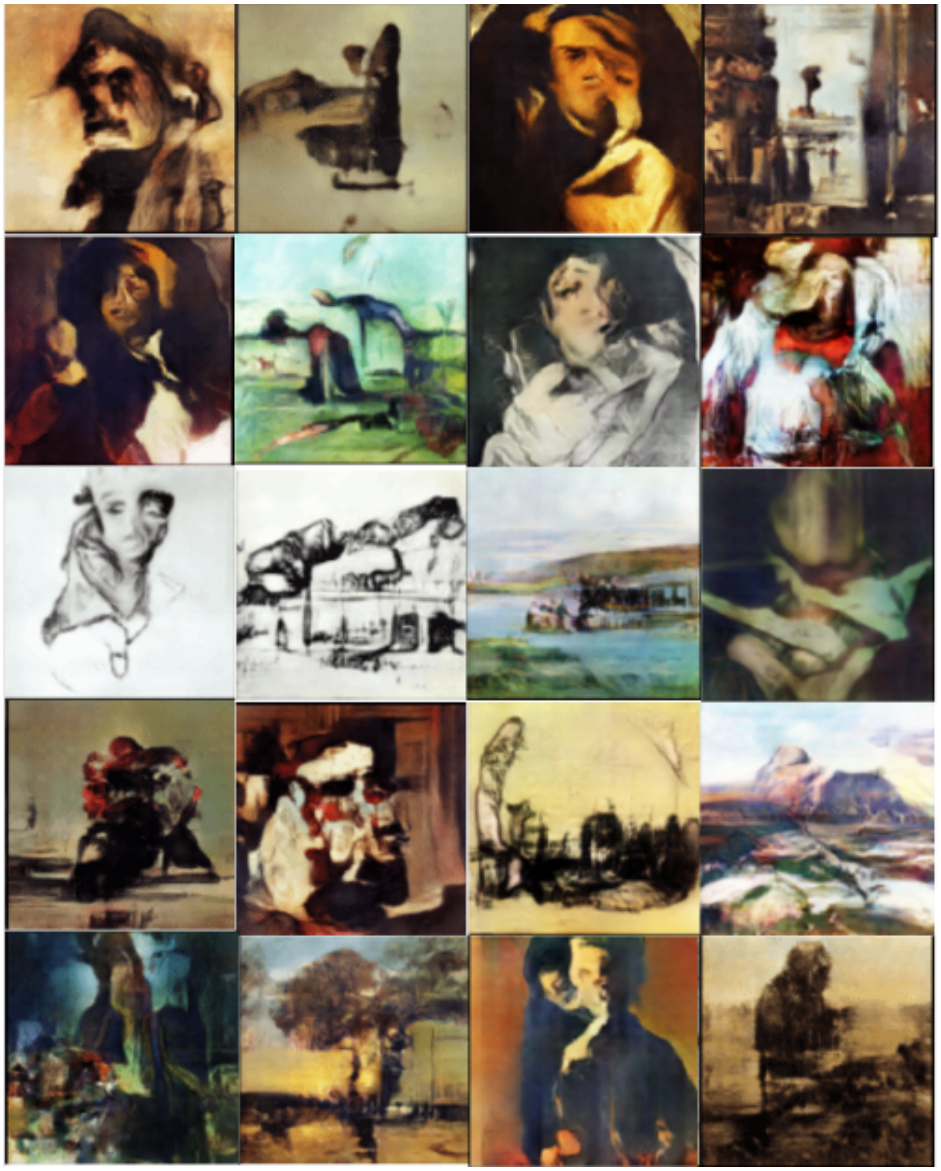

Figure 7.7: Realism

7.8 Romanticism

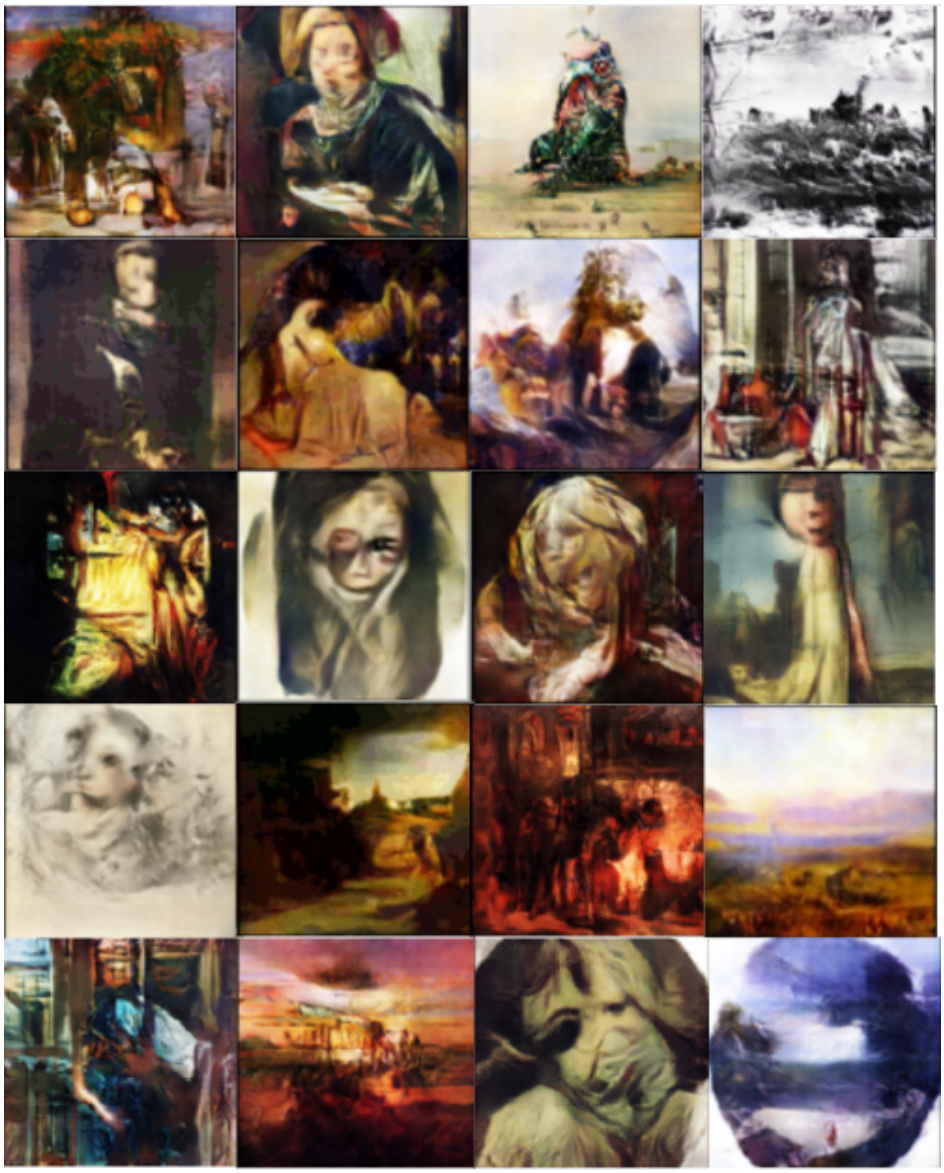

Figure 7.8: Romanticism

7.9 Surrealism

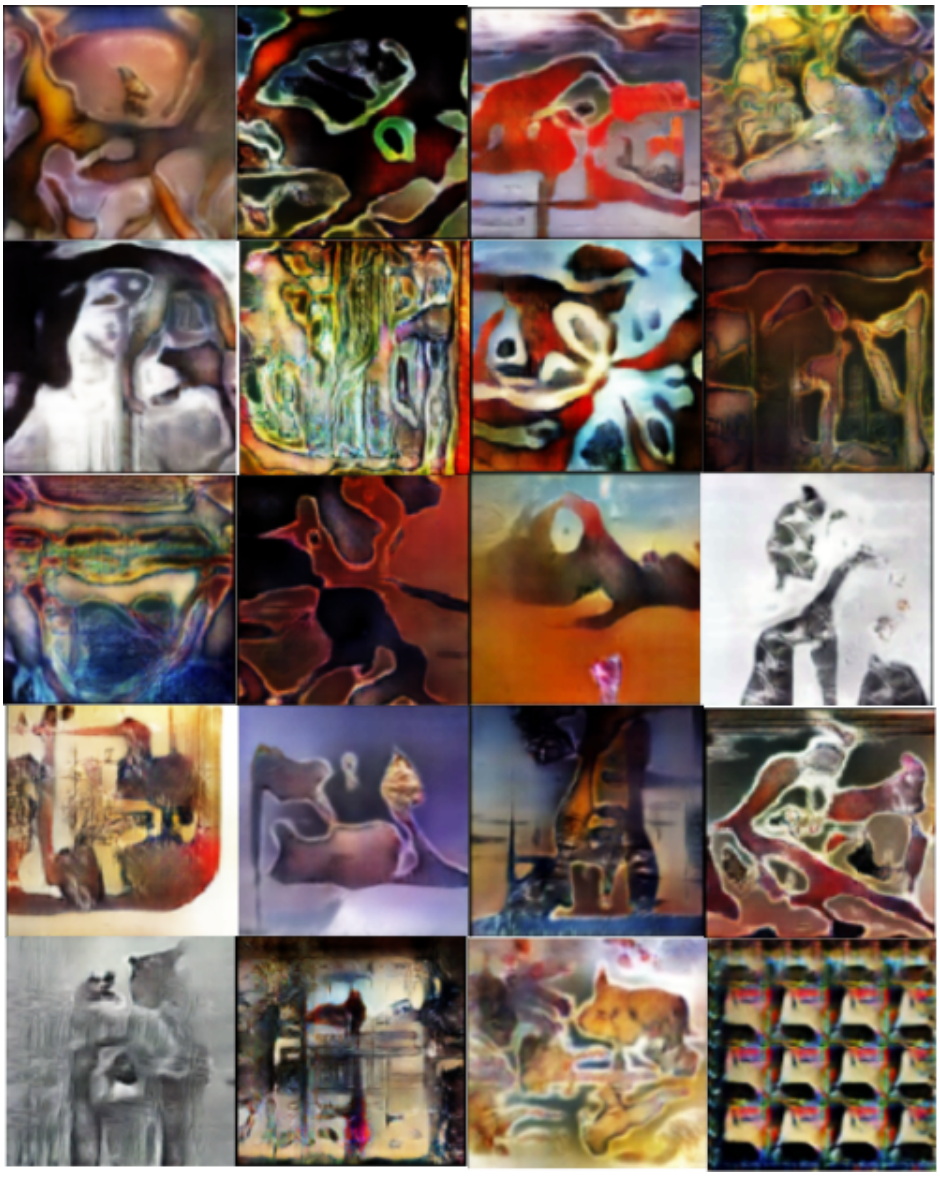

Figure 7.9: Surrealism

7.10 Symbolism

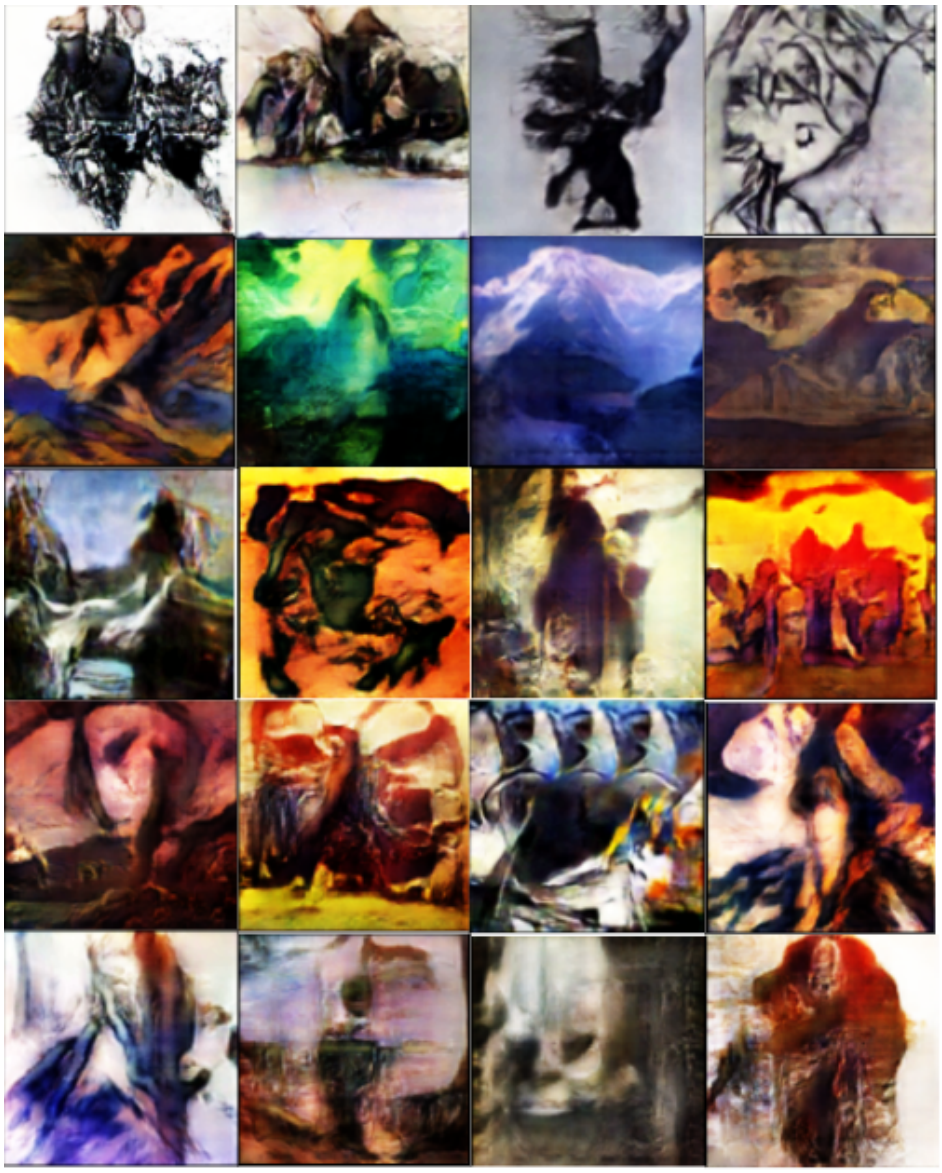

Figure 7.10: Symbolism

Chapter 8

Web application - GenerateArt

As a wrap up of this project, I decided to develop a web application for two purposes:

1. Allow any person with internet access to generate paintings the same way I do, with the final architecture and for any of the available art styles, but by simply going to a web site and clicking on a button.

2. Display those collages of 20 paintings generated for each style in a way that anyone can see them without much effort.

The name of the application is **GenerateArt** and it is a simple web application built with flask. It does not require logging in and does not use a database. Given that this application is not the main part of my work, and given that it has a low complexity, I will not describe it in as a detailed manner as one would if one's project would be mainly a web application. However, I will present the key points.

8.1 Structure

I have used **PyCharm** as the IDE for developing the application. The different components of the application can be divided into the following groups, based on their nature (images, python code files) and role (backend, frontend):

- **app.py**. This is the main python code file of the project. It contains the code used in order to start the app, all the logic of the app, namely the functions used by the app and their associated routes. Here, 10 (one for each style) **Generator** models are loaded when the application starts.

- **networks folder**. It contains a Pytorch .pt file for each **Generator** model. Each of these .pt files contains the parameters of a model after it had been trained. They are used by the application in order to load the models at their ready to generate paintings state.

- **networks.py**. This file contains the code for the **Generator** class. We do not need the **Discriminator** anymore because the **Generator** has already been trained.

- **static folder**. It contains all the images displayed on the web site (not the ones generated, just the fixed ones).

- **templates folder**. It contains all the .html files used by the application in order to display the content to the users.

- **bootstrap folder**. I used bootstrap in order to give the application a modern and pleasant look. This folder contains the css and js code downloaded from the Bootstrap website.

8.2 Logic

When the server starts, 10 **Generator** objects are created and each of them has its parameters loaded from a .pt file in the **networks** folder. Loading the models this way makes starting the server slower (< one minute) but I believe it is the best choice. If I chose to load the model for a particular art style when a user sends a request, asking the server to display images generated from that style, the time spent between the request and the answer would be too long. Obviously, I couldn't use only one model as I have trained one **Generator** network for each style individually.

While the application is running, a user can navigate through the web site by using the navbar at the top of the page. This navbar allows the user to:

1. Go to the **Home** page. This page is what a user first sees when they enter the web site and it contains a short description of the application. I implemented this with Flask by simply writing a function that returns the **index** template of the app. The route assigned to this function is the standard route ('/').

2. Go to the **About** page. This page shows some information about me, including my email and photo, and about the purpose of this application. The implementation in Flask is similar to the one above, but with a different route ('/about').

3. Choose one of the available art styles. This is where the fun begins. This option is presented in the form of a dropdown menu. When the user clicks on a style, they will be redirected to the page related to that style. Each of the 10 styles has it's own page (html template) and each page displays different information and images. This redirection is done with a single function for all styles. Its assigned route has a string parameter for the desired art style and based on the value of the parameter, the function returns one of the 10 style templates. Each of those 10 pages contains two buttons, one for generating art and one for seeing a fixed collage of images generated for that style (handpicked by me).

   If the user clicks on the **first** button (*Generate Paintings*), they will be redirected to a new page that displays a collage of paintings generated with the **Generator** for the chosen style. This redirection is implemented through a function that returns a new template and has a route with a string parameter for the style chosen as well as another fixed parameter ('generate') to differentiate it from the previous route. The *generate* template that is being returned displays an image that is generated dynamically by the server.

   The process used to generate and display the image:

   - The *generate* template receives the chosen style as a parameter when it is rendered. It uses this style as a parameter for the Jinja2 function *url_for* in order to call the plotting function for that style and receive the source of the image.[49] There are 10 plotting functions, one for each style, so we need the style parameter in order to differentiate between them.
   - The plotting function calls a helper function that does not have a route assigned.
   - The helper function generates the input vectors for the **Generator** network, feeds them to the network and receives the generated images. These images are then gathered together in a figure containing a grid using **Matplotlib**. This figure is converted into a byte stream, using the function *BytesIO* from the package *io*. The figure is then closed, because Matplotlib keeps figures in memory until they are closed explicitly, and the byte stream is returned.
   - The plotting function receives the byte stream and returns it as a .png image with the Flask helper function *send_file*.
   - The *generate* template receives the source of the generated image and displays the image.

The *generate* template also contains a button that refreshes the page so that a new grid of images is generated. This button is basically a copy of the first button.

If the user clicks on the **second** button (*See Some Great Generated Paintings*), they will be redirected to a static page that displays a collage of some of the best looking images I have generated for that art style. This is implemented through a function that receives a string parameter (the style) and renders a template. There is only one template for all 10 styles, but the source of the image displayed is dynamically chosen based on the style.

8.3 Aspect

I used bootstrap[1] in order to give the app a nice look. This way, I didn't have to write css code. I only had to assign the desired classes to the html items in order to get the website aspect I wanted.

8.3.1 What is common to each page

Each page of the application contains the same navbar at the top of it. The navbar displays four options:

1. **GenerateArt**. It is the name of the application. If the user clicks on it, they will be redirected to the **Home** page.
2. **Home**. Clicking on it redirects the user to the **Home** page.
3. **About**. Clicking on this option redirects the user to the **About** page.
4. **Art Styles**. This is a dropdown menu that has 10 available options, one for each art style covered by the application. If the user click on an art style, they will be redirected to that art style's page. Each style has its own page.

8.3.2 Home Page

This is the first page a user sees when they enter the web site. It displays the title of the application, **Generate Art**, as well as a description of the application and how to generate paintings. The description emphasizes on the fact that the paintings generated are the result of mathematical operations performed on vectors of randomly generated numbers.

[1] `https://getbootstrap.com/`. Last accessed 8th June 2021.

There are also two images displayed on the page. One is the *Mona Lisa* and the other depicts connections between artificial neurons. These pictures have the purpose of giving the viewer a sense of connection between the subjective appeal of art and the complexity and rigorousness of mathematics.

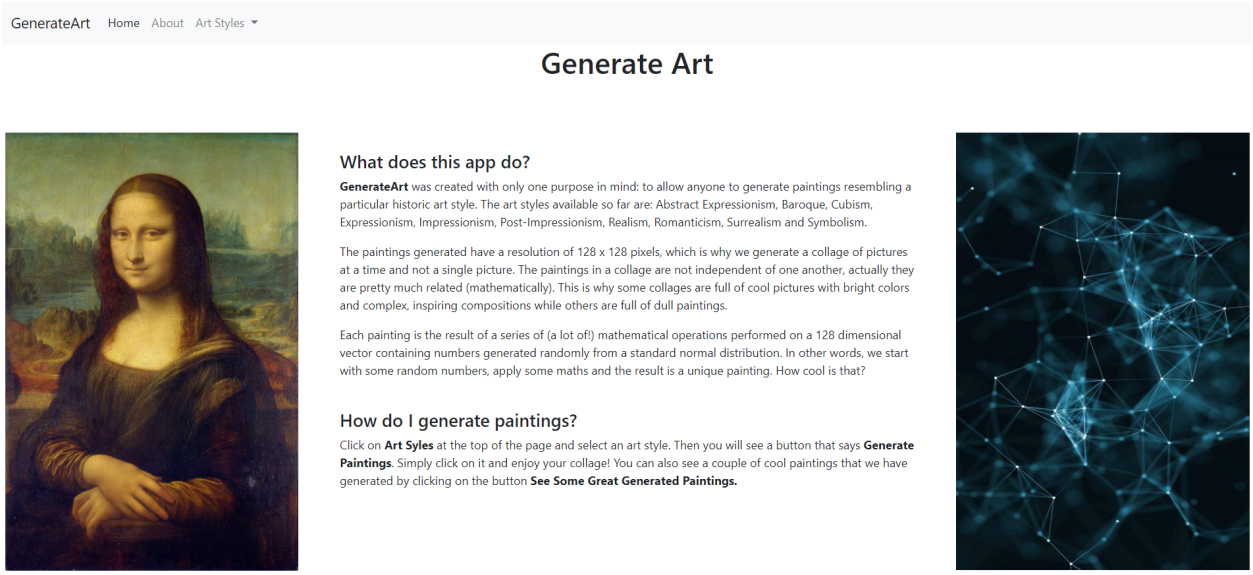

Figure 8.1: The Home Page.

8.3.3 About Page

This page contains information about me and about the purpose of the application. My name, my email, a photo of mine, the name of the university where I study are there. Two of my main passions are also mentioned, as they are relevant for the application: *Artificial Intelligence* and *Visual Art*. Also, the technology used in order to generate the images, namely *GANs*, as well as the resolution used, 128 x 128, are also mentioned. Finally, there is an image that makes the viewer think of the connection and balance between the section of the brain related to art, to creativity, and the one related to rationality, to mathematical exactness.

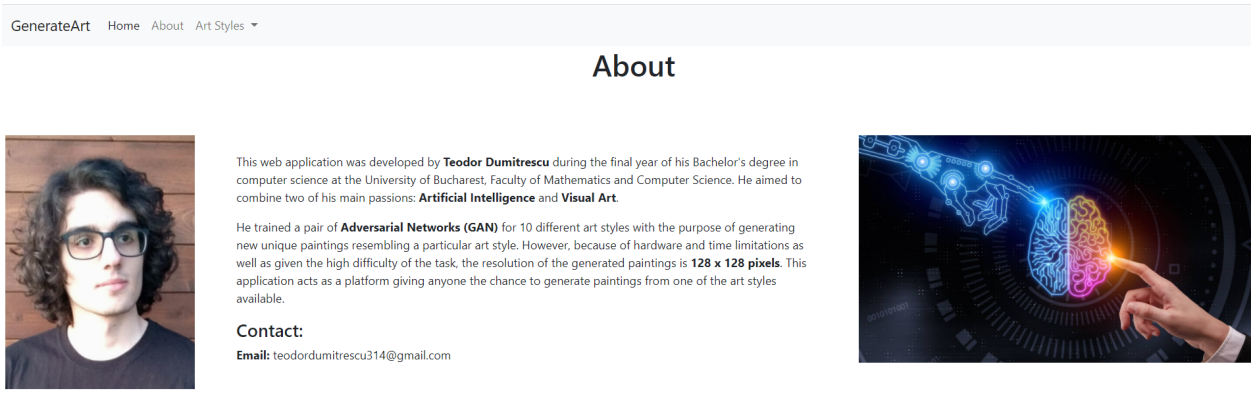

Figure 8.2: The About Page.

8.3.4 Style Page

There is one Style page for each of the 10 styles. The user gets to any of them through the dropdown menu **Art Styles**.

A Style page displays a short description of that art style. The description contains the time frame associated with the style, as well as some key characteristics that differentiate it from the other art styles. This description is framed by two paintings relevant for that style, one to the left and one to the right. Four other paintings are displayed in a line, at the bottom of the page. This alignment is obtain with the help of bootstrap. The body of the page is divided in 2 rows and each row is divided into 12 columns. The result is a responsive grid-like layout. The items displayed in the grid adjust their size based on the size of the window which is helpful especially for smaller screens, like those of mobiles. The first row of the grid is divided into: 3 columns for the first image, 6 columns for the description and 3 columns for the second image. In the second row, there are 4 images and each image takes 3 columns.

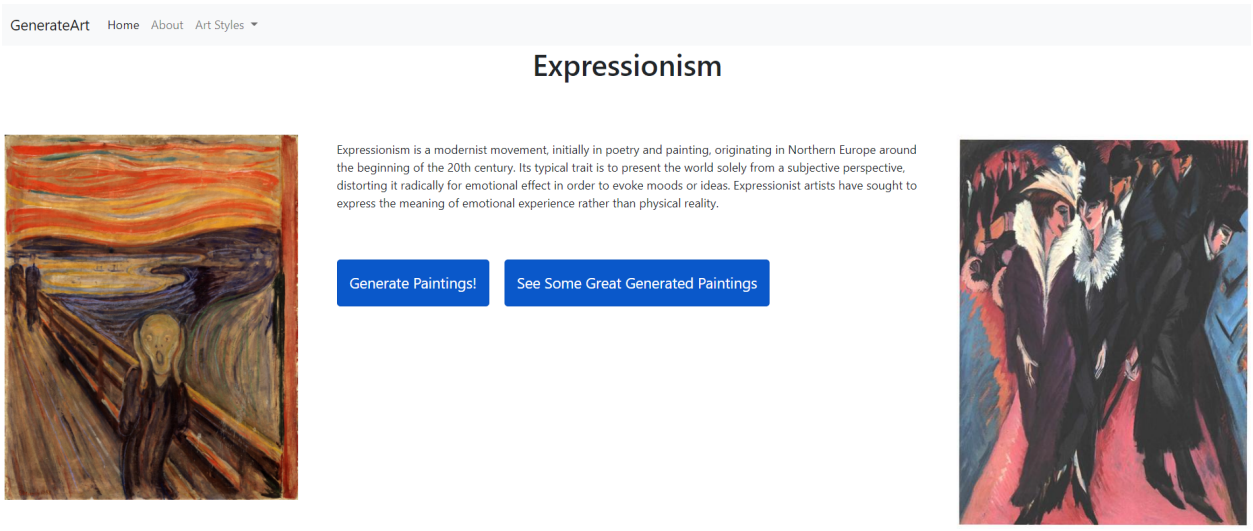

Figure 8.3: The Style page for expressionism. The user has to scroll down in order to see the other four paintings displayed.

There are also two buttons. The first one says **Generate Paintings** and it redirects the user to a new page. This page contains the name of the art style as a title and displays an image containing a 4 x 4 grid of generated paintings. The same **Generate Paintings** button appears under this image and if the user clicks on it, the page will be refreshed and the image grid will contain 16 new unique paintings.

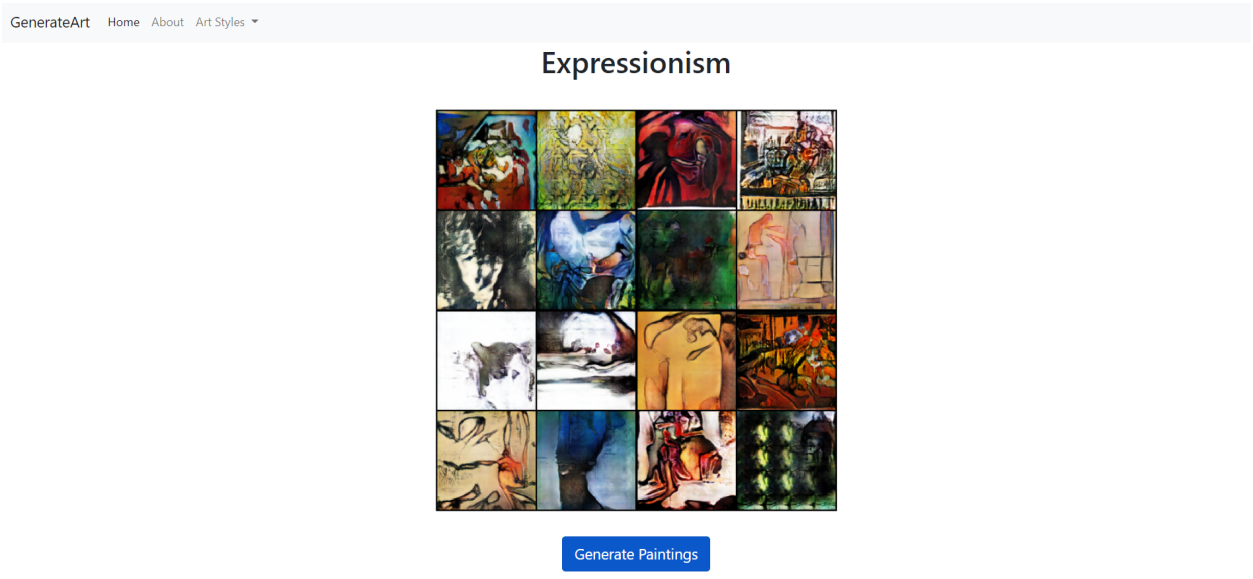

Figure 8.4: The page displayed after clicking on the **Generate Paintings** button.

The second button says **See Some Great Generated Paintings** and it redirects the user to another page. This page displays an image with a grid with 5 rows and 4 columns,

containing 20 paintings generated for the selected style. I have selected 20 generated paintings for each of the 10 art styles. A set of 20 paintings is the result of searching through perhaps 1,000 generated pictures so they are not representative of what a single batch of generated images should look like. However, they are representative of what any user can generate with multiple tries.

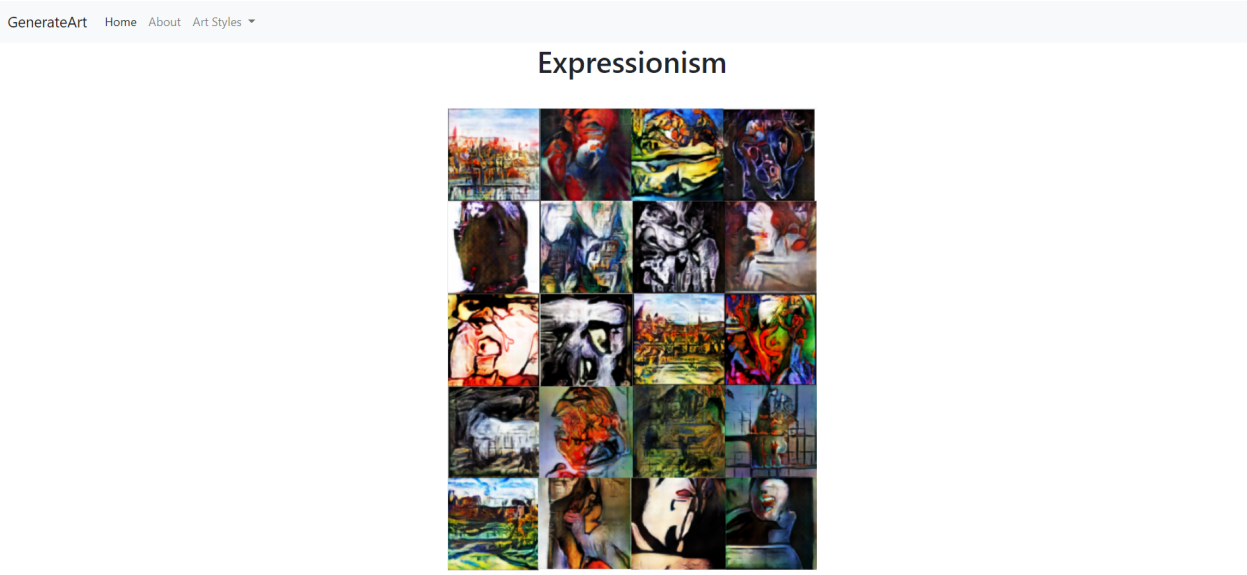

Figure 8.5: The page displayed after clicking on the **See Some Great Generated Paintings** button.

Chapter 9

Conclusion

I believe that I have accomplished my goal of generating good looking paintings that resemble a particular art style. Also, a secondary goal of mine was allowing ordinary people to generate art in an easy manner, without having to write code or spend time painting themselves. I believe I have accomplished this goal as well.

During my experiments, I have tried several notable GAN architectures, including: DCGAN, WGAN-GP, ProGAN and Conditional GAN and also incorporated various techniques specific to the GANs research field to these architectures. I have encountered many barriers, mostly hardware and time limitations, but also low amount of training examples and some poor choices along the road. I have fully experienced the difficulties of training GANs. The small amount of research available in the direction of generating art with GANs didn't help me that much either.

However, in spite of these limitations, I have completed my task in a satisfactory way. Perhaps if I had access to the computer that allowed me to train continuously and with a decent speed right from the beginning of my experiments, I would have been able to find an architecture that produces good looking images even for 256 x 256 resolution, while still generating paintings that resemble a particular style.

As for future work, I see three directions worth following:

1. Adjusting the current architecture in order to increase the quality and the diversity of the generated paintings even more. For some art styles, there is a bit of Mode Collapse happening and if I could get rid of it, the images generated would be even more diverse.

2. Going from 128 x 128 to 256 x 256 resolution. I have attempted to do this but was unsuccessful. With better hardware and more time on my hands I believe this direction could lead to some interesting results.

3. Making the web application more complex. Some features that come to my mind that would bring considerable value to the application are:

    - Implementing a log in system and allowing the users to save the images generated that they enjoy the most in some kind of inventory of their own. Perhaps saving small images and not whole grids of them and allowing the user to build collages only with paintings they like.
    - Allowing the user to replace an image at a time from a grid of generated images. This way, the user could ”reroll” small paintings until the grid is full of images they enjoy.
    - Adding even more available styles for users to generate paintings resembling those styles.

Bibliography

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *NIPS*, 2014.

[2] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4401–4410.

[3] A. Elgammal, B. Liu, M. Elhoseiny, and M. Mazzone, "Can: Creative adversarial networks generating "art" by learning about styles and deviating from style norms," in *8th International Conference on Computational Creativity, ICCC 2017*. Georgia Institute of Technology, 2017.

[4] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[5] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," 2017.

[6] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[7] N. Kodali, J. Abernethy, J. Hays, and Z. Kira, "On convergence and stability of gans," *arXiv preprint arXiv:1705.07215*, 2017.

[8] F. Farnia and A. Ozdaglar, "Do gans always have nash equilibria?" in *International Conference on Machine Learning*. PMLR, 2020, pp. 3029–3039.

[9] Z. Zhang, M. Li, and J. Yu, "On the convergence and mode collapse of gan," in *SIGGRAPH Asia 2018 Technical Briefs*, 2018, pp. 1–4.

[10] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," in *Proceedings*

of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 6629–6640.

[11] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *NIPS*, 2016.

[12] H. Thanh-Tung and T. Tran, "Catastrophic forgetting and mode collapse in gans," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–10.

[13] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks," *stat*, vol. 1050, p. 17, 2017.

[14] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks: the official journal of the International Neural Network Society*, vol. 61, pp. 85–117, 2015.

[15] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*. PMLR, 2017, pp. 214–223.

[16] M. Lucic, K. Kurach, M. Michalski, O. Bousquet, and S. Gelly, "Are gans created equal? a large-scale study," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 698–707.

[17] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," in *International Conference on Learning Representations*, 2018.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[19] E. W. Weisstein, "Convolution," *https://mathworld. wolfram. com/*, 2003.

[20] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[21] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, 2016. [Online]. Available: http://distill.pub/2016/deconv-checkerboard

[22] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *stat*, vol. 1050, p. 11, 2018.

[23] S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry, "How does batch normalization help optimization?" in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 2488–2498.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[25] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, "Spectral normalization for generative adversarial networks," in *International Conference on Learning Representations*, 2018.

[26] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *NIPS*, 2017.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[28] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[29] B. M. Randles, I. V. Pasquetto, M. S. Golshan, and C. L. Borgman, "Using the jupyter notebook as a tool for open science: An empirical study," in *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 2017, pp. 1–2.

[30] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, *Jupyter Notebooks-a publishing format for reproducible computational workflows.*, 2016, vol. 2016.

[31] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

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

[33] M. Grinberg, *Flask web development: developing web applications with python*. O'Reilly Media, Inc., 2018.

[34] Y. LeCun, "1.1 deep learning hardware: Past, present, and future," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 12–19.

[35] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[36] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[37] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 317–324.

[38] E. Bisong, "Google colaboratory," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 59–64.

[39] T. Carneiro, R. V. M. Da Nóbrega, T. Nepomuceno, G.-B. Bian, V. H. C. De Albuquerque, and P. P. Reboucas Filho, "Performance analysis of google colaboratory as a tool for accelerating deep learning applications," *IEEE Access*, vol. 6, pp. 61 677–61 685, 2018.

[40] V. V. Salian, "Generating art using gans," 2020, last accessed 6 June 2021. [Online]. Available: https://blog.jovian.ai/generating-art-with-gans-352ceef3d51f

[41] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2020, pp. 8107–8116.

[42] T. Salimans and D. P. Kingma, "Weight normalization: a simple reparameterization to accelerate training of deep neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016, pp. 901–909.

[43] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," in *International conference on machine learning*. PMLR, 2017, pp. 2642–2651.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[45] T. Tieleman and G. Hinton, "Neural networks for machine learning," *Coursera (Lecture 65-RMSprop)*, 2012.

[46] T. Van Laarhoven, "L2 regularization versus batch and weight normalization," *arXiv preprint arXiv:1706.05350*, 2017.

[47] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014.

[48] Y. Qin, N. Mitra, and P. Wonka, "How does lipschitz regularization influence gan training?" in *European Conference on Computer Vision*. Springer, 2020, pp. 310–326.

[49] A. Ronacher, "Jinja2 documentation," 2008.

Appendix A

Random selection of images generated for each style

The images generated with the final **GAN** architecture displayed in this work are handpicked from a large pool of generated images. This means that I had generated perhaps 60 images before choosing one of them to be displayed as one of the best looking images generated for a particular art style. In this appendix I want to give the reader a sense of how an ordinary batch of random generated paintings would look like, for each art style. I chose to display 4 x 4 grids of paintings mostly because this is the format used in the application.

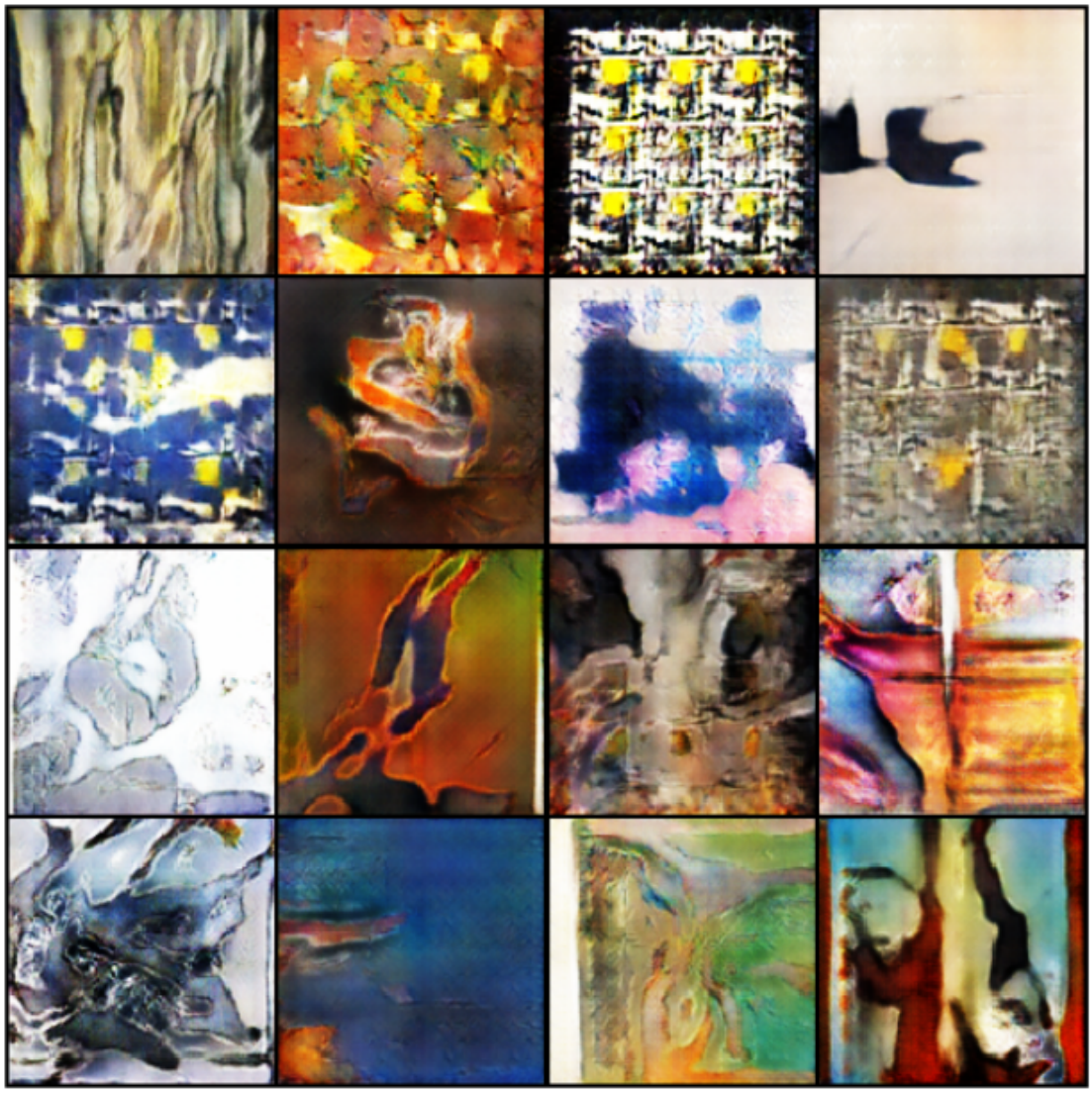

Figure A.1: Abstract Expressionism

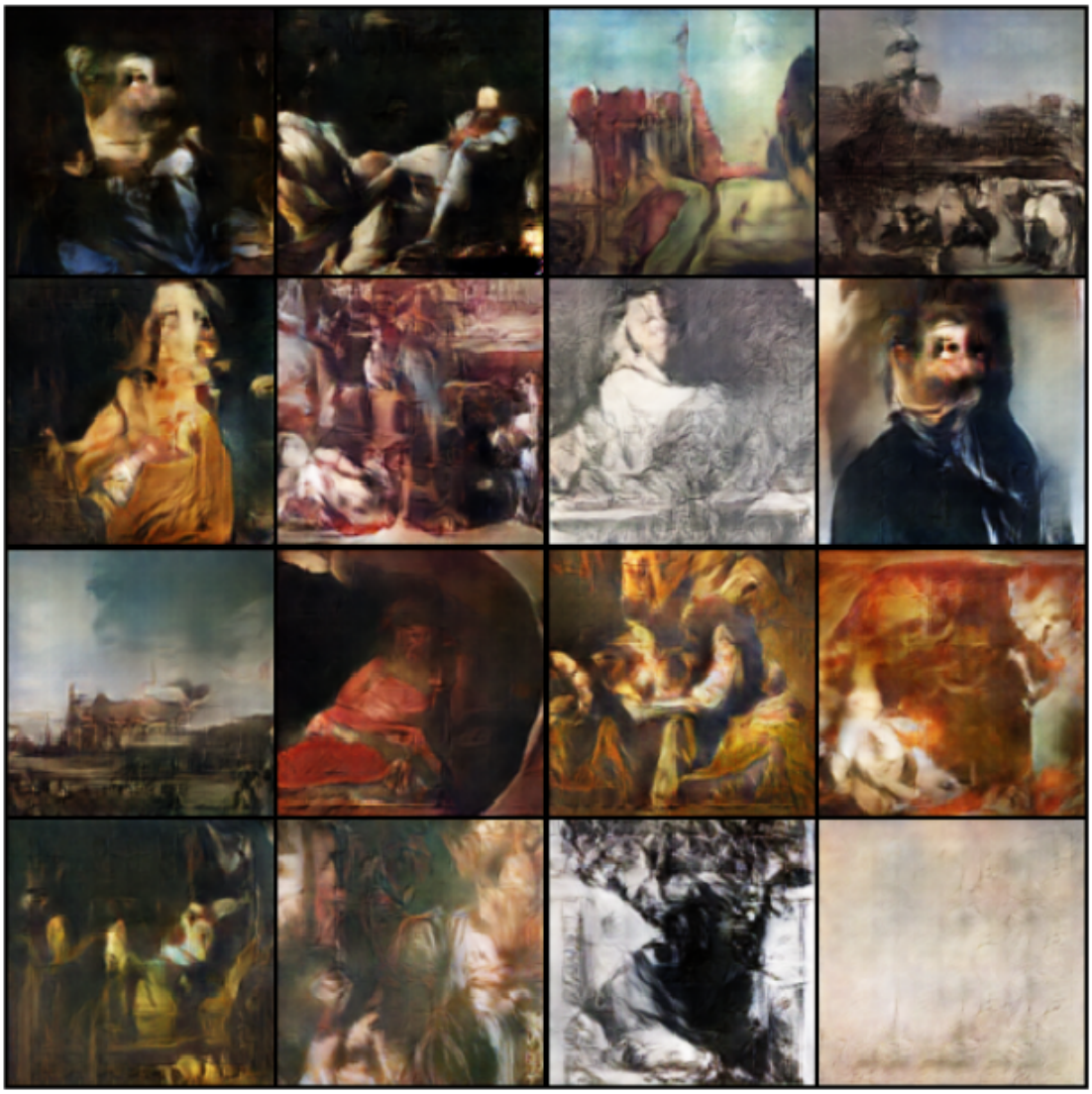

Figure A.2: Baroque

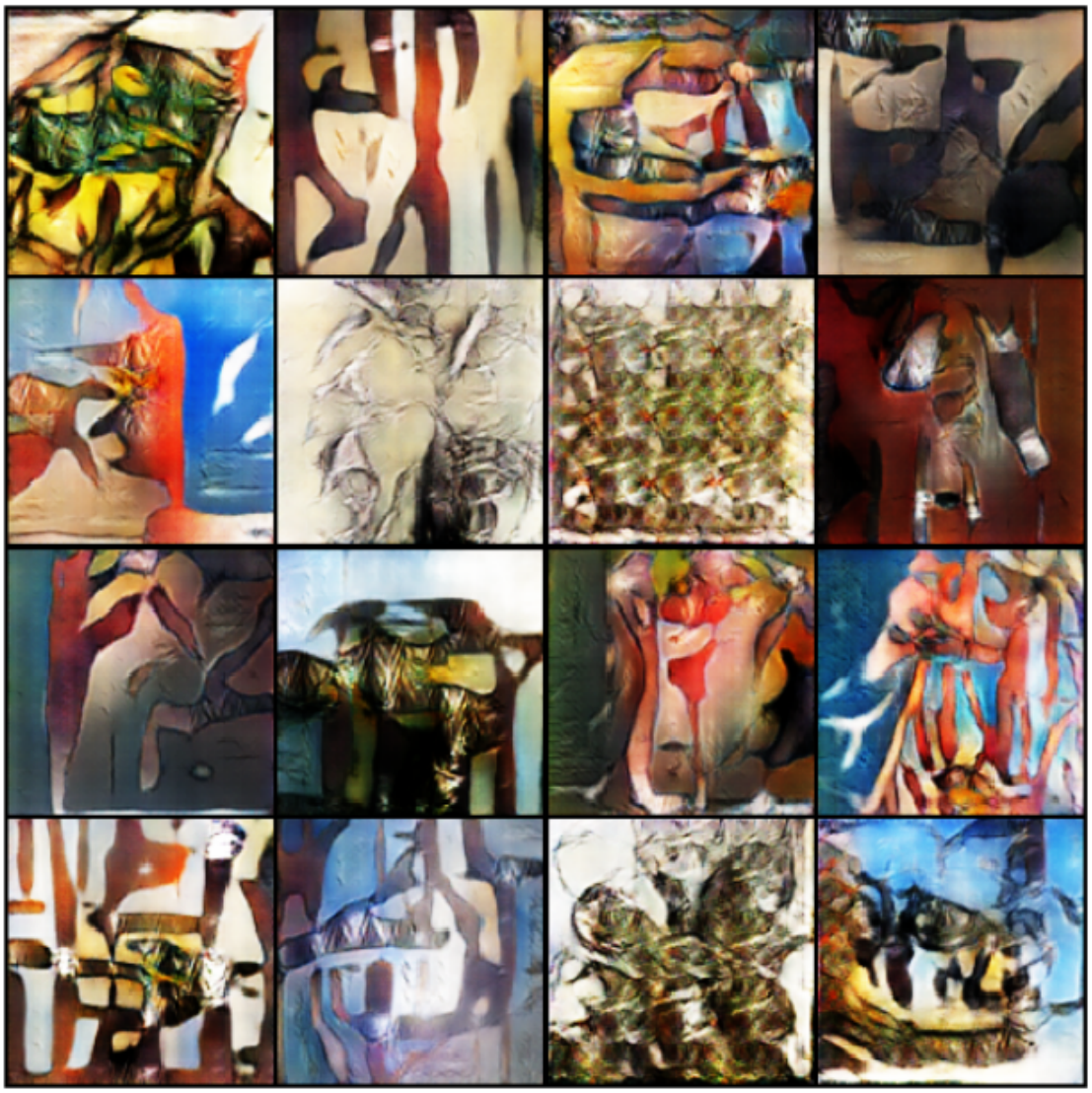

Figure A.3: Cubism

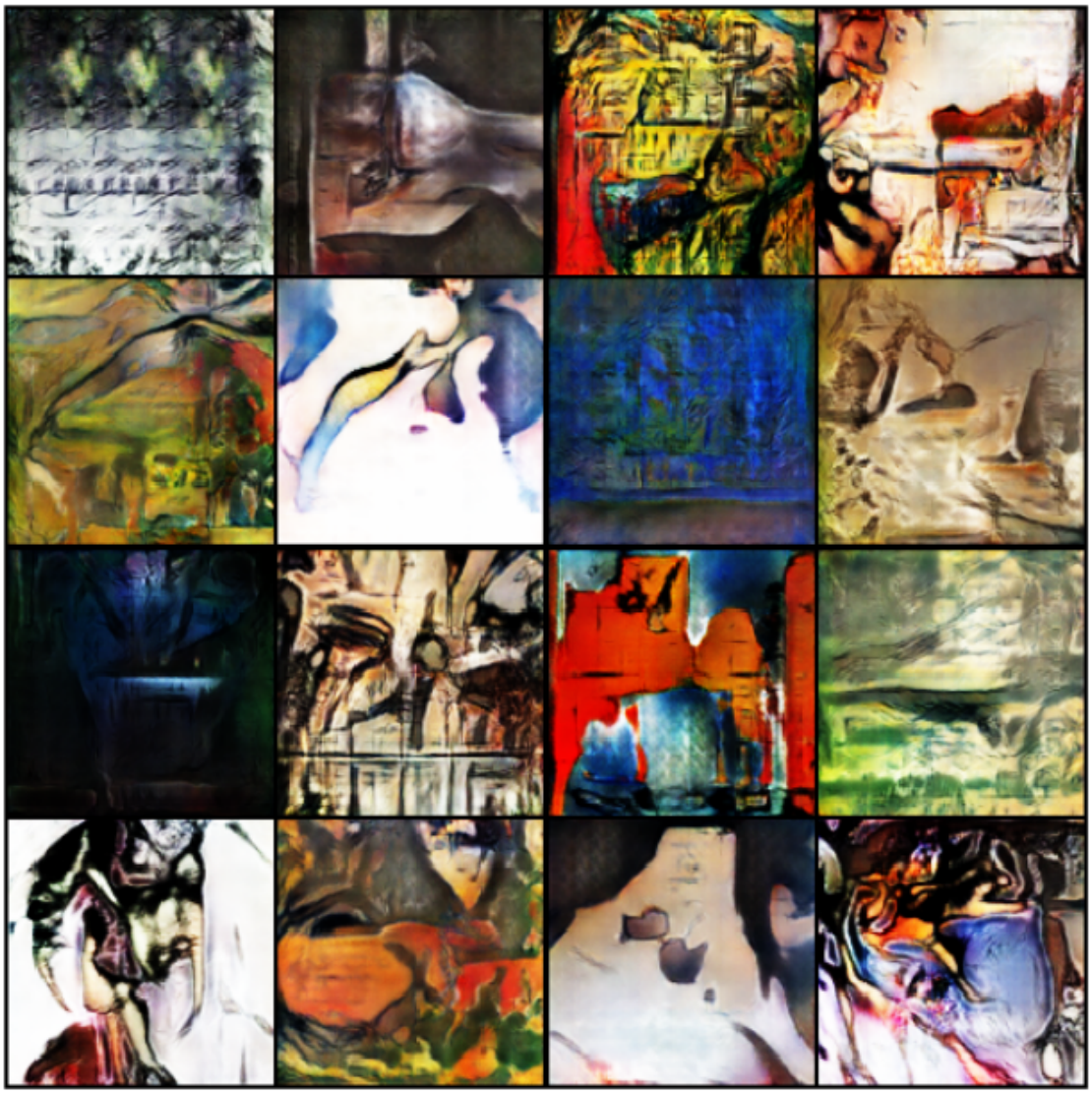

Figure A.4: Expressionism

Figure A.5: Impressionism

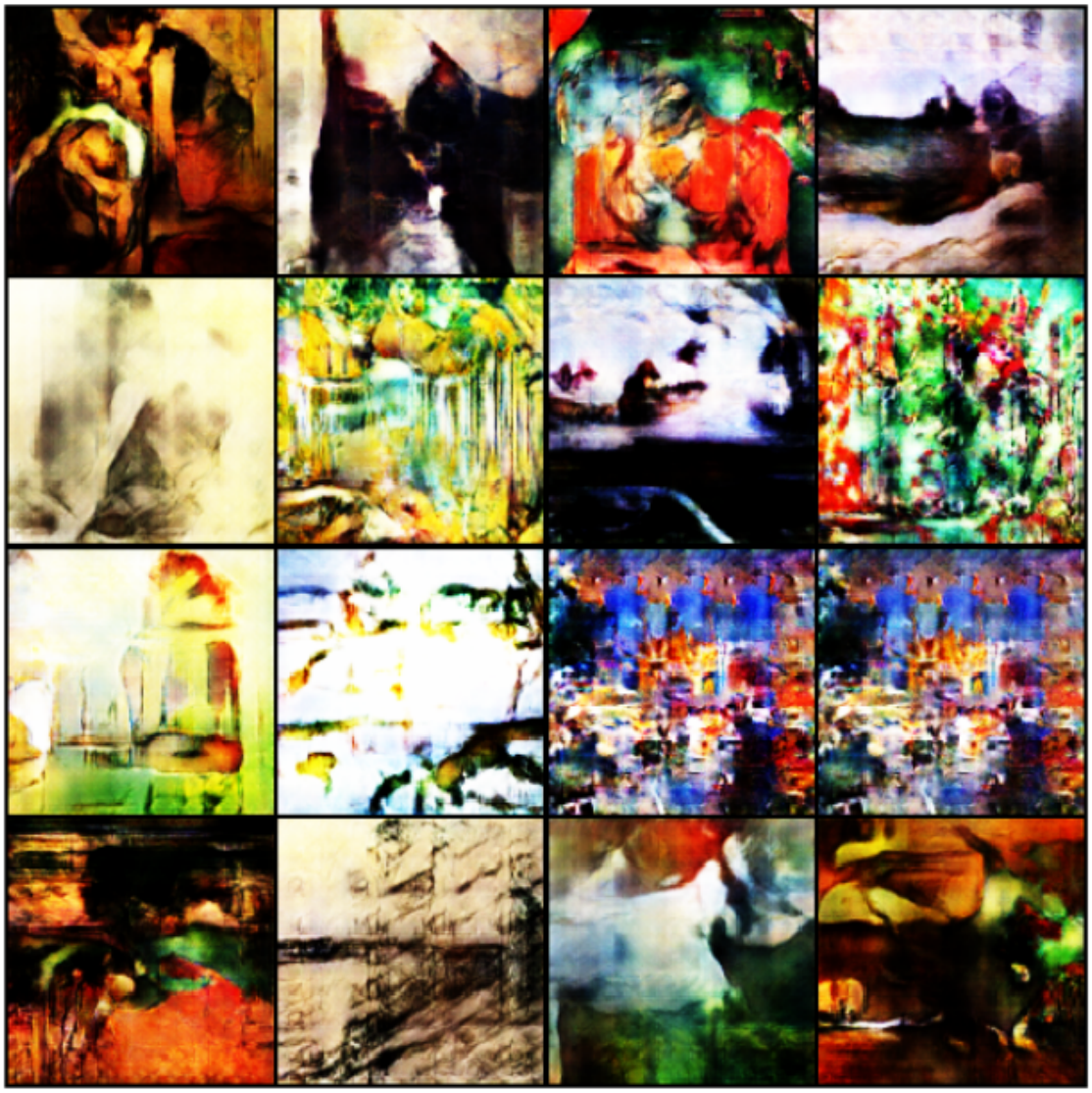

Figure A.6: Post-Impressionism

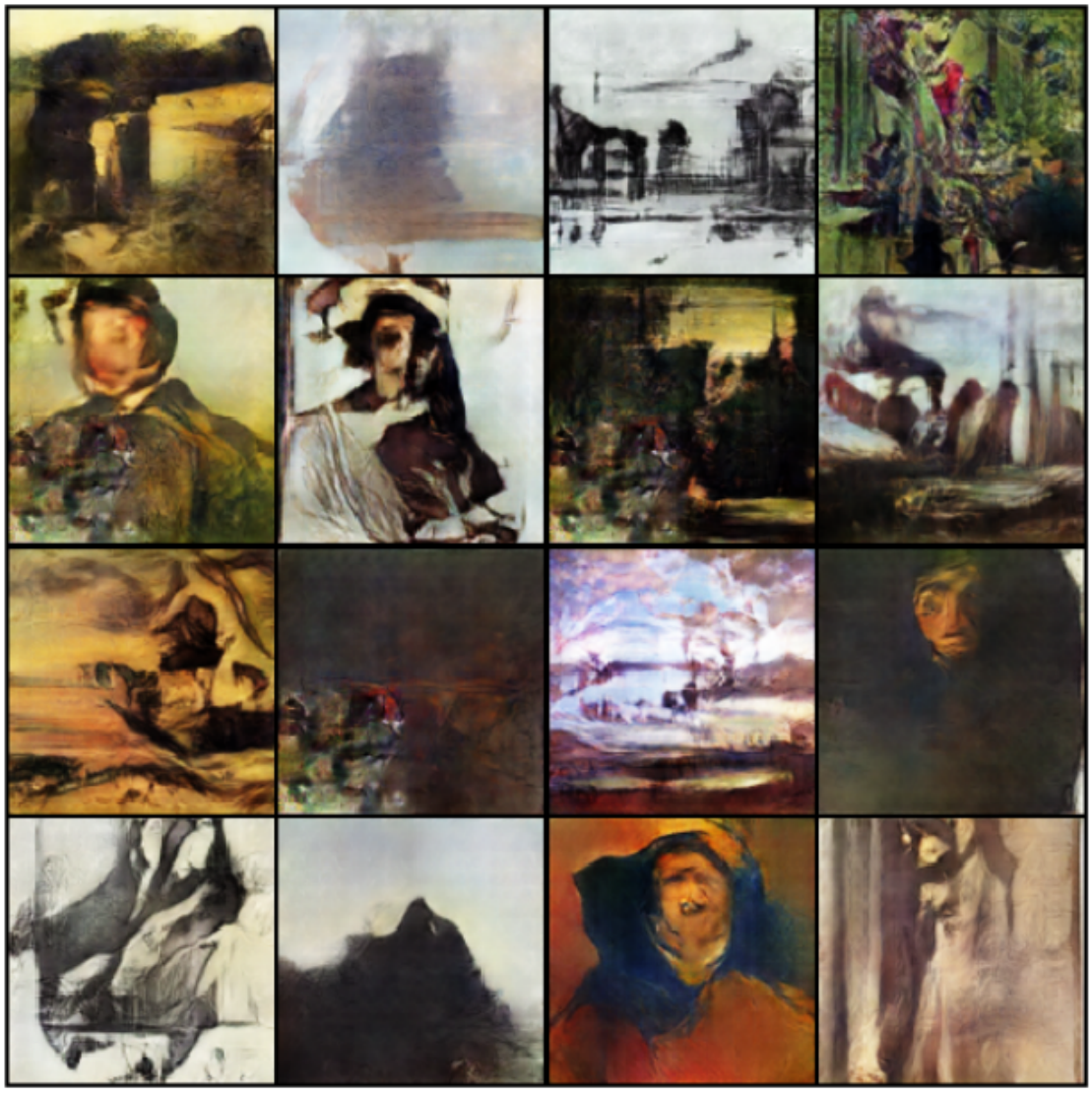

Figure A.7: Realism

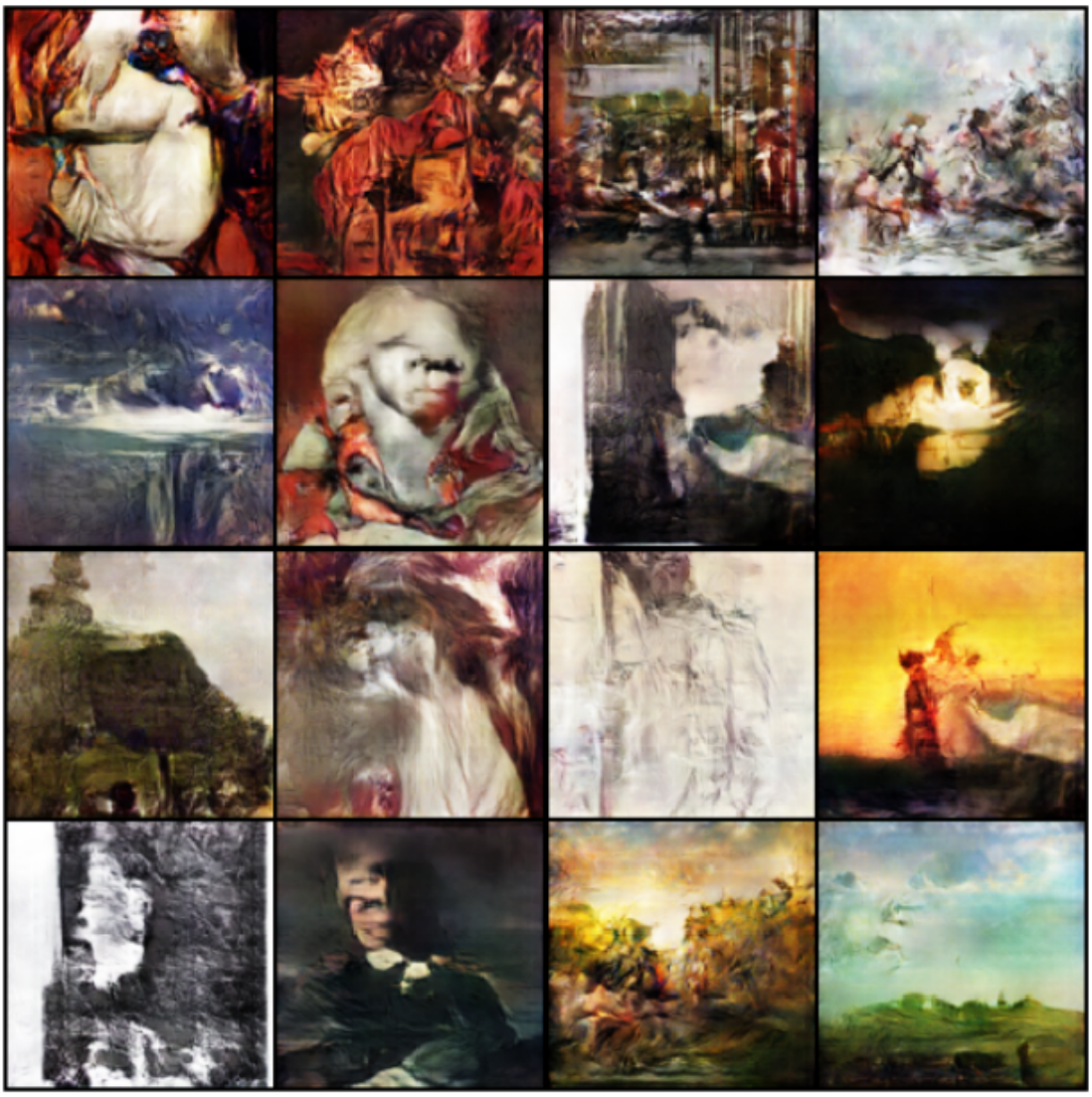

Figure A.8: Romanticism

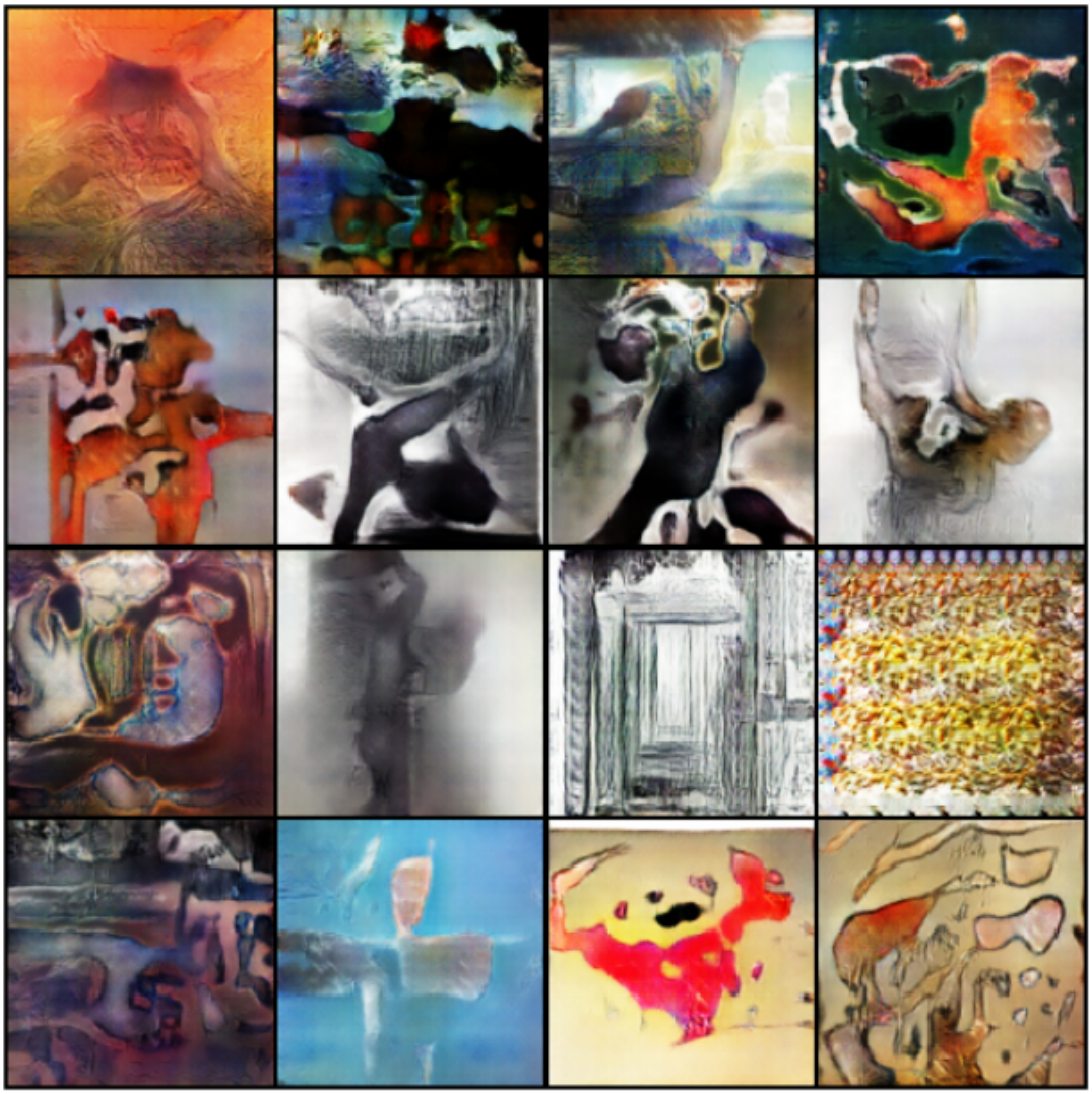

Figure A.9: Surrealism

Figure A.10: Symbolism