Generative Art: A new approach with Generative Adversarial Networks

Said Moredi

School of Mathematics, Computer Science and Engineering

Department of Computer Science

City, University of London

Submitted in satisfaction of the requirements for the
Degree of Bachelor of Science
in Computer Science

Supervisor Dr. Ernesto Jimenez-Ruiz

May 2021

Abstract

This paper tackles the problem of generative art in the field of machine learning. Generative art has produced some of the most interesting pieces of geometrical and contemporary art however its repetitiveness becomes its downfall. This paper tackles these issues of generative art and future problems by suggesting an alternative method. A new technique using machine learning models with generative adversarial networks is proposed to tackle the problem. A deeper dive into the architecture of said models is analysed and evaluated for suitability of the problem at hand, producing art through machine learning models, specifically generative adversarial networks. Further analysis is laid out to in terms of performance and efficiency of the programs. As this program will be run on consumer level hardware, architectures where the processing time exceeds a few hours will be disregarded as this requires specialist hardware to be run efficiently. This paper will introduce a larger toolbelt for artists within the field of generative art and give insiders an easy to step towards the world of generative art with machine learning models.

List of Figures

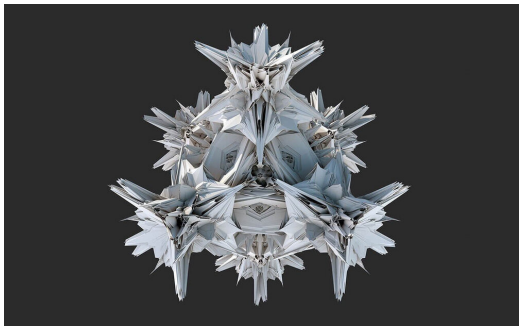
Table of Contents

Chapter 1

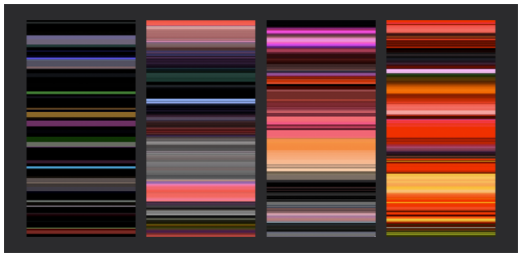
Introduction

1.1 Problem Description

What is the problem with standard generative art? Generative art flourishes in the world of geometry, abstraction, and mechanised production. Themes of futurism, constructivism and abstract expressionism are all by-products of years of generative art (Galanter, 2003). It has become a new medium to explore and express a new form of art . However, this approach of producing generative art consists of many variations of algorithms with no sense of machine learning within the final outcome of the image (Galanter, 2016). The techniques used within generative art focus more on repetitions of certain patterns within a 2D or 3D field. This can be impressive and does provide an endless form of appealing art to look at. Unfortunately, the linear approach to the production of the art takes away a level of creativity which is instilled in pieces created in real life. Eliminating the robotic approach within generative art and replacing it with a model which learns from the art, sprouts the same spark of human touch within the images. This introduces a new world of generative art that is yet to be explored. From this development, pieces of art that might be imagined by an individual can be generated seamlessly. A mix of Basquiat and Sandro Botticelli may be imagined, but with technologies like GAN Art can be brought to fruition. The use of specific datasets such as portraits or clothing can sprout new creations that resemble a human feel to the work created therefore allowing them to be used in a commercial aspect too (Mitchell, 2006).



(a) Work of (Hansmeyer, 2021)



(b) Work of (McCormac, 2021)

Figure 1.1: Showcase of Generative Art

How this GAN Art provide something new? GAN Art can provide a new approach towards art. As it becomes a more widely used medium the entry level to using a program like this can allow multiple custom datasets to be considered, each dataset can give an amalgamation of styles within a piece of art. Taking the styles of classical renaissance era paintings and murals mixed with new age abstract art can breed new creations that have yet to be imagined. From an artist's viewpoint, they can take the creations from GAN art as a form of inspiration or publish it as something they have created. Answering the questions of periods of art working in symphony is only one new use that GAN art will provide. Within the field of video games or other visual mediums such as film or photography, GAN art can be used to produce visual pieces which accompany the mediums (Fadaeddini, Majidi and Eshghi, 2018). It breaks apart the methodological approach that generative art carries out, the use of machine learning in art is not something new but it is definitely an area yet to be explored greater. Shifting generative arts procedural aspects and replacing them with forms of backpropagation is the aim of this project. The scope of this work is not to tunnel the use cases of GAN Art to producing and generating art but a program which can be tweaked to fit within other constraints, for example the creation commercial objects.

1.2 Project Objectives

GAN Art's main objective is to produce new pieces of art which was directly influenced by a selected dataset. Here, a large dataset of art images will be passed through the main program, through training the program will begin to output new images. For example, a dataset of portraits from the renaissance era should produce an image of a portrait with a blend of styles that were extracted from the original images. To obtain such datasets, I will be using Kaggle as it has vast range of images, which I can extract and process into my very own datasets (Kaggle, 2021). The pre-processing functions will have constraints put in to avoid anomalies in the main program.

1.2.1 Primary Objectives

The primary objective of this project is to produce new images of art once a large batch of images has been fed through the main program. The final image should reflect the influence of the dataset within the image. A blend of styles will be explored with a large dataset at the start. Once the model is fine-tuned to art, specific styles of art will be further explored, for example, expressionism, cubism and surrealism. Each style will need fine tuning of its own; once the base model is tuned its much easier to adjust for specific styles. This will essentially be done through various forms of dataset pre-processing. To achieve a machine learning program which can abide by these constraints, I will be developing my own Generative Adversarial Network (GAN). GAN's will be further explained in the paper. The program will have its own generator and discriminator function with various explorations into other deep learning models such as convolutional neural networks (CNN) and reinforcement learning (Albawi, Mohammed and Al-Zawi, 2017). The scope of this paper can be infinite in a sense as the world of machine learning and generative deep learning is not completely new but still novel in its implementation.

1.2.2 Sub-objectives

Most of the sub-objectives in this paper focus primarily on the usability of the software. However, this should not be confused with overall primary goal of generating deep learning art from a dataset. The sub-objectives are listed below:

- The application shall create unique pieces of art on each iteration. The uniqueness of the art can be analyzed by comparing pixels of the image on every new iteration. Using functions such as mean squared error (MSE) and structural similarity index to remedy faults of solely using MSE (Wang and Lu, 2018).
- The application shall be able to differentiate between different styles of art within the dataset. The datasets I will be using will most likely be labeled, in the case of an unlabeled dataset the program should be able to tell distinctive art styles apart.
- The application shall be able to differentiate the art period of each image. From the original dataset this can be carried out easily by assigning time periods to art movements. Once this function is carried out on the output of the GAN, it becomes increasingly more difficult and may not be a priority due to time constraints.
- The application shall control the level of influence a particular artist has had over a certain output. The first developments of this application will standardize the overall influence of all images in the dataset over the final image. The bias of this can be controlled via increasing the pool of a certain artists images in a dataset. Once the amount of an artist's images increases far more than the standardized images of other artists, a natural influence will begin to show within the program. Controlling the bias in this way is not intuitive, however it is by far the fastest way to increase specific influence over the final outputs of the program.

1.3 Project Beneficiaries

The beneficiaries of this project will be artists, designers, and any creative visual medium. For artists this can become a tool-belt to hone new skills which can be translated into many fields. GAN Art and other generative deep learning models will be a norm in advertisements, art installations and prints. The diverse nature of the program will be an advantage to be exploited into various industries, the question is more of a sense of usability. Designers can generate new patterns from existing ones and expand into a new creative field which is yet to be explored. In the commercial field the use of GAN Art can make way for fine-tuned models for specific advisors. In the sense a car manufacturer can use GAN Art to produce new sketch models of car body parts. This can open a new world of possibilities for commercial and personal use. GAN Art can also be expanded into academia, where researchers can take the same concepts and improve other types of GAN such as CycleGAN and BigGAN (Chu, Zhmoginov and Sandler, 2017). The use cases of GAN Art are truly infinite and is only held back by our imagination. On one hand it will be used to produce aesthetically pleasing art but can also improve on current technologies. Current technologies where efficiency is important, GAN can analyse any data which can be transformed into a tensor. This diverse input type means a range of data types can be converted and used. Standard GAN is able to predict missing data within a dataset, in the case of data missing in a series of art pieces, GAN Art can be used to produce an accurate enough result of the next image. Development of animations can be vastly increased by using GAN Art where the next frame of the sequence can be predicted, hence leading to less strenuous repetitive drawings for animators and larger throughput (Liang et al., 2017).

1.4 Work Performed

Every part of the GAN will be produced from scratch. I will be using Python as the programming language in conjunction with other popular machine learning libraries. TensorFlow, PyTorch, and Pillow are some of the main libraries that will be used

in the project. I am not completely familiar with all the libraries but with their documentation online I am confident in putting together an efficient GAN.

1.4.1 Library Rundown

TensorFlow TensorFlow is a software application which is popular for implementations of machine learning algorithms. Originally made by Google, it was then released in 2015 as open source (Abadi, 2016). TensorFlow takes inputs as multi-dimensional arrays, which are also called tensors. The flow essentially means a series of operations the tensor is put through to produce an output. TensorFlow is highly versatile in its use cases and diverse in its compatibility. This is one of the main reasons why I have decided to use TensorFlow. It gives me the opportunity in the future to start training of the model on one PC and still have the option to continue training on a much more powerful PC. Another advantage of TensorFlow is its ability to run on CPU and GPU; throughout this project I will be using a Nvidia GTX 1070 Ti with 8GB of VRAM. This will allow me to create much more complex models which will work efficiently on my hardware. However, if a user does not have my hardware, it's no issue at all as TensorFlow will simply switch to the CPU and continue processing. I will not be covering Keras in its own section as it is shipped with TensorFlow now.

PyTorch PyTorch is friendly for beginners and has good documentation for implementing machine learning models (Paszke et al., 2019). Tensors are used in PyTorch too as input. The performance in PyTorch is very impressive due to its compatibility with CUDA GPU's (Sanders and Kandrot, 2010). It has extensive neural network building blocks which can be fine-tuned to fit the preferred model. One of the shining advantages of PyTorch is its ability to easily implement backpropagation in models. This saves time in the development process of the application. The friendly intuitive design with a range of research and help provided online makes PyTorch a go to for machine learning.

Pillow Pillow is a library which allows us to work with and manipulate images. Most of the Pillow use cases will be carried out at the beginning of development. When working with images and transferring them into tensors, many constraints need to be put in place to minimize the chance of anomalies appearing in the model. Pillow will allow me to instill these constraints into the images before they are processed into the machine learning models. Constraints such as resolution, colour space and cropping of the image will be done using the Pillow library. There are many other libraries which I will be experimenting with to produce GAN Art. All those that make a significant change to the overall flow of the project will be noted throughout the methodology and results section of this paper (Pérez-Garcıa, Sparks and Ourselin, 2020).

1.5 Assumptions Made

In terms of technology, I will be developing on PyCharm, Google Colab + Kaggle and Jupyter notebooks. Although both Colab and Jupyter Notebooks do essentially the same thing, parts of the initial development will experiment on both platforms for performance and usability analysis. Assumptions about these technologies remaining free throughout my development have been assumed. I have no prior knowledge of GAN's and took on this project due to my interest in machine learning and art in all forms. Most of the research has been carried out by further reading into the subject of generative art through deep learning modules. Although this may be a novel implementation of the project it is something I pursued as I truly enjoy creating and tuning art. In terms of scheduling, the bulk of this project has been carried out in the research phase as I believe a deeper understanding of concepts can help me better execute an intuitive program. For hardware, assumptions were made that my current hardware can comfortably complete the training phase of the program and produce competent results that are satisfactory to the requirements of the project. Arrangements have been explored if my current hardware fails in any form.

1.6 Limitations

The development hardware of this project is limited, as there is hardware that can speed up computation, meaning a longer testing phase could have been carried out to further fine tune the model. As GAN's are relatively new with its first implementation in 2014, it still has many techniques to improve such as the usability of the program (Goodfellow et al., 2014). Although a few implementations have focused on GAN Art there is no specific books. The reading material on the subject is limited. TensorFlow and PyTorch libraries are relatively new to me and were not covered within my AI module. Most of the libraries and languages are self-taught, although I know the basics of the frameworks, to create a complex model will take a much greater time. Monetary limitations are also a big constraint, as mentioned previously new and bleeding edge hardware will provide a greater cushion to experiment with various methods. For example, running the training of the model for a longer period, leading to far better final results. The limitation of time is apparent in this project as training the model and then adapting it to new challenges means no other development can continue until the training is over.

1.7 Output Summary

At the end of development and testing, the project will produce a deep learning model which will generate art. It will be a series of Python programs that will lead to the final output of computer-generated art developed by a machine learning model. The components of the GAN can be separated from the original code and allow other developers to fine tune the components to their specific use cases. For instance, the discriminator module of the application can be extracted and be made stricter on the classification of input leading to longer training times but significantly better results. All the libraries and their specific versions will be noted to enhance compatibility with various systems whilst minimizing bugs. Documentation of the code and the setup of the environment to run the specific code will be outlined, allowing new users to input their own datasets and experiment with GAN Art. Adding these additional

outputs can only bolster new users and draw more attention to computer generated art with deep learning models. Analysis and graphs of the performance overhead of specific functions within the model will be carefully curated into charts to better understand areas of improvement within the program. Most of the analysis will be done on TensorBoard, an analysis and graphing tool for TensorFlow applications (TensorBoard, 2021).

Chapter 2

Literature Review

2.1 What are Generative Adversarial Networks?

Generative Adversarial Networks (GAN) is an unsupervised but not limited to, machine learning model (Goodfellow et al., 2014). It encompasses two machine learning models against each other in a min and max game, these components being the generator and discriminator. The generator from a series of random noise values tries to create a fake image which the discriminator might pass as a real image. The discriminators job is to classify real images from the training set as real or fake images produced from the generator. Through the loss functions of the generator and discriminator, GAN carries out back propagation to updates the weights on the generator and discriminator. Overtime the generator output will pass a threshold where the discriminator cannot distinguish the output as fake. This technique is not limited to images, any data which can be turned into tensor instances will work – highlighting the flexibility of GAN. Both the generator and components can use other machine learning models such a deep convolutional neural networks, perceptron's, and recurrent neural networks. GAN was created by Ian J. Goodfellow with a team of researchers (Goodfellow et al., 2014). From its inception the concept opened a new world of machine learning which is still continuously being explored. Currently there stands over dozens of implementations for GAN as it has quickly become a technique which is replacing inefficient learning models in the world of machine learning. Other uses such as data mining can be easily setup via having a large enough dataset of the original data.

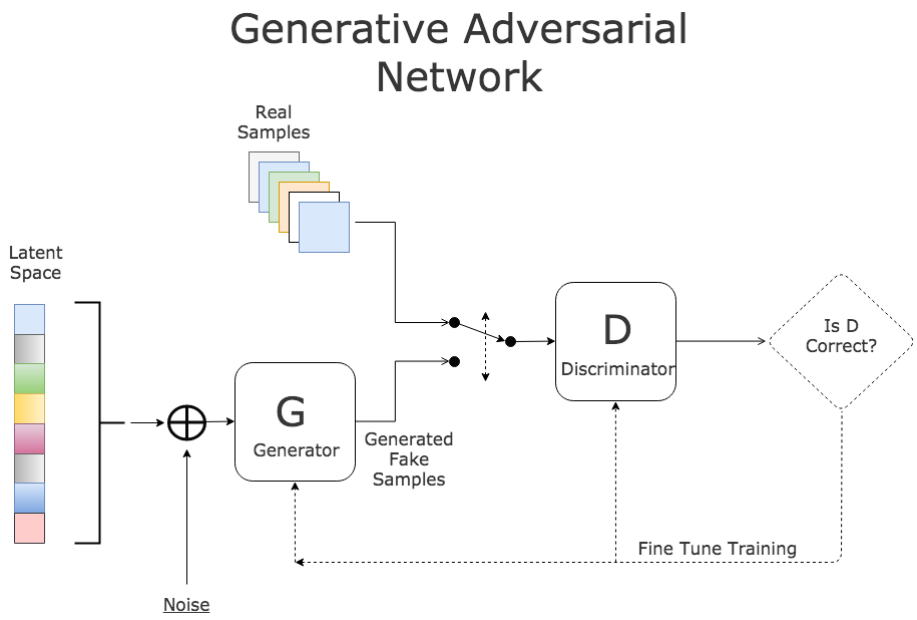


Figure 2.1: Brief explanation of GAN (Gharakhanian, 2021)

2.2 Deeper dive into GAN

At a surface level the generating component in the model generates data whilst the discriminative network evaluates them as real or counterfeit input. The generator will map to a latent space dependant on the data distribution of interest, the discriminator distinguishes each output of the generator to the actual true data it has learnt. The discriminator is put through training where it is fed original data and increases its accuracy by correctly evaluating if the data is from the actual dataset. Once the accuracy has built up it can easily classify the original data. From here on the generator will start to produce output, which is fed into the discriminator, the discriminator here will decide if the generator data is of the original dataset. To improve the generator and the discriminator, independent backpropagation on both components is carried out(Hecht-Nielsen, 1992). Backpropagation for the discriminator improves the model by better identifying phony data to real data, whilst the generator improves its model by learning what output do not work and allows it to produce better samples to fool the discriminator.

2.3 Specific GAN architecture: ArtGAN

ArtGAN is a project created by students in conjunction with researchers from the university of Japan and Malaysia, respectively (Tan et al., 2017). It proposes a new approach for GAN for conditional image synthesis. Loss functions in this implementation are carried out on a gradient and are backpropagated from the categorical discriminator to the generator. This in turn produces a higher quality image with a decreased training time. Although this type of backpropagation may be seen as cheating the generator to quickly produce acceptable output. However, the advantages far outweigh the pitfalls. With the use of an autoencoder within the categorical discriminator work hand in hand to improve the overall quality of the image. Taking the concepts of how humans learn to draw, from repetition of a singular object, ArtGAN's feedbacks the label given to the generated image by using the loss function of the discriminator. From here on the Discriminator is updated by a minimizing function. This makes the discriminator and generator far more efficient compared to a normal GAN whilst adding another function to the generator for pixel-wise reconstruction – this inherently is what improves the image quality of the images.

Improvements made on the architecture Improvements for these specific implementation, are using other deep learning models within the discriminator and generator then analysing the performance gain to the model whilst also keeping in mind qualitative differences. The function for pixel-reconstruction can be replaced with another that runs the generator through a convolutional neural network and removes pooling layers, this technique was used by Zhixin Zhang and other researchers (Zhang et al., 2020). Jointly attaching other GAN can be a technique to further enhance the quality of the results, however this comes at a cost of a long training period and greater risk of anomalies arising. A key finding in the original paper is when the same loss function to improve the generator was added to the discriminator the overall quality of the discriminator and results were significantly worse. This is the main reason why the discriminator has its own minimizing function. Many GAN take the approach of unsupervised learning, however including supervised elements to

the GAN can work in its favour to attain better performance from the discriminator and the generator.

2.4 Critical Context

ArtGAN is a complex model which individually curates' functions for the generator and discriminator. Although not the most complex in the world of GAN it is a steppingstone that this project will heavily take from. The loss functions on the generator and minimising functions on the generator are components that I want to replicate with my implementation. Labelling the generator with the loss functions of the discriminator will allow me to analyse how well the discriminator is performing and improve the generator outputs. The basic elements of GAN will be taken from the Goodfellow paper and allow me to build the foundation of my GAN Art. When experimenting with other machine learning models within my GAN I will be analysing the performance of each model when significant changes are made. Due to hardware limitations the changes may be minor in contrast to ArtGAN changes. I will be following a basic implementation of both of the described GAN as my knowledge of this is fairly new and it was not a part of my studies at university. However, wherever improvements can be made will be highlighted in the results section.

Chapter 3

Methodology

As this project will conduct multiple sprints throughout its development stage and be issued new changes at the end of a sprint's lifecycle – the agile methodology will be by far the most appropriate (Beck et al., 2001). From the initial proposed workplan in the project definition document, 3 sprints for the program will be created. Each iteration of the sprint will be increased in difficulty compared to the previous sprint. This does not mean every sprint will be kept in the final outcome as each version will be tested for performance, efficiency, and overall score. After the first implementation of the program a series of tests will be compiled which are present in the PDD created prior. Once the implementation has passed all the tests a report of new improvements will be carried out and implemented to continue development into the next phase. At the end of the second implementation the review of the changes made will be assessed compared to the first model. If the current implementation is satisfactorily greater than the previous implementation, then the sprint may move onto its final implementation. This essentially stops the sprouting of unnecessary versions that do not make significant changes to the application itself.

3.1 Software Engineering Method

This project is carried out with the agile software methodology. The users of this project have been outlined in the beneficiary's section: essentially academics, artists, and commercial use. The scope of problems covers mostly machine learning problems and a complex overarching problem of generative art with the use of deep learning models. In terms of vision statement the program should allow users to input their

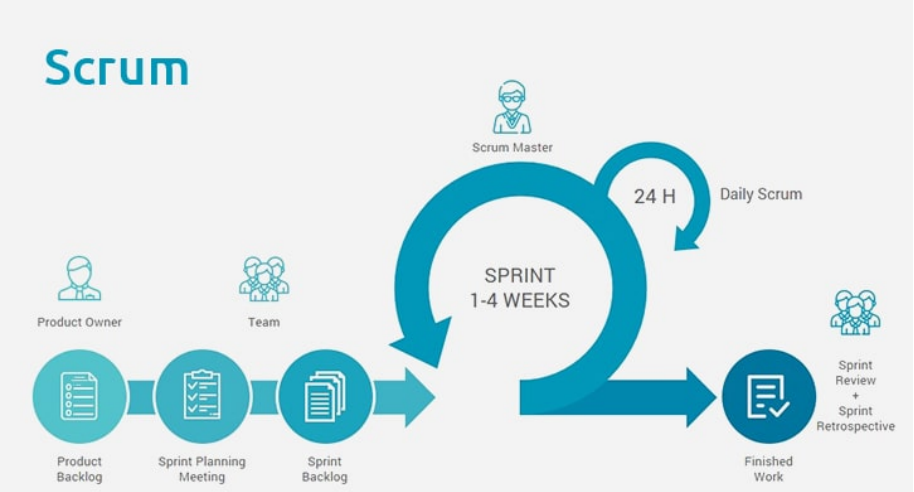


Figure 3.1: Brief explanation of Agile with Scrum framework (Pisuwala, 2021)

own datasets of images and produce art. The bulk of this project will be carried out by using the agile framework, scrum (Schwaber and Beedle, 2002). Due to this delivery focused method the 3 sprints have been predefined in the PDD, however revision of the sprints will be carried out before development and throughout development. If the development produces an outcome that satisfies the goals of the project means revisions to further improve the sprints will only support that goal. Short time periods have been defined for each individual sprint's lifecycle; this allows room for experiment within the development phase due to adaptable nature of the scum framework. Updates on the project will be carried out in a detailed and strategical method keeping the report as a form of consultant. Due to the review process put in at the end of the lifecycle of the sprint, this will allow the demos and results to be outlined. This ultimately contributes to the final revision of the sprint which will clearly state the benefits and shortcoming of each sprint. This project requires a scrum-based approach as it is mostly a research driven project, therefore allowing it to adapt to changing requirements.

3.2 Management Tools

In terms of version control systems, the project be will primarily be using GitHub, OneDrive and Kaggle to keep track of sprints and changes to the implementation. This also provides the ability of reverting back changes which may have caused bugs

within the main program. Extensive bug reports can be written with assignments to certain changes allowing for greater completeness tracking. An offline zip file will be kept on an external hard drive for added data availability. The Grantt chart provided in the PDD will further help with scheduling certain tasks for the project. Specifics of reporting and development can be found within the workplan. The chart is sufficient to keep track of the project progress, however for added reinforcement of management an extra Excel spreadsheet will be provided to outline the progression of the report itself. Each component of the development phase will be outline in an individual section within the method chapter. This further adds a greater security to completeness tracking. Individual tests of the components allow the project to evaluate the progress of the implementation on a micro level. Once all tests are satisfied, development naturally moves onto the next components of the implementation.

3.3 Work Plan Creation and Outline

The work plan was created by taking a template from popular software development use cases. Keeping the agile methodology in mind the construction of the work plan revolves around continuous revamping prior changes made to the code. As the bulk of the work plan contains research and preparing to write the code, it allows for a longer period to gain a deeper understanding of the subject matter. Sessions with the consultant allowed me to further breakdown the task at hand and pinpoint pitfalls within the original plan. Due to the time constraints instilled in the Grantt chart, originally proposed in the PDD, it allowed the project to stay on time for various deadlines therefore forcing the project to remain on time (Maylor, 2001). Prior to development and part of the research proposal is the planned architecture of the Art GAN, here many of the detailed steps and planning are set out for the project which is recorded within this report. With the detailed architecture of the GAN, another key part of the work plan is the requirements research. The most basic version of the program is expressed with additional features characterised into different sprints of the program. Whilst revisions of each individual sprints are done after completion. This is expressed within the Grantt chart as “Testing and Review”

and "Review changes from Previous Version". Taking a meticulous approach to changes made within the program diminishes repetitiveness of ineffective features which are counter intuitive towards the main goal of the project. An extra analysis phase towards the end of each sprint further reinforces this. On the completion of the three sprints, further analysis done on each iteration will allow for a clearer result to be put forward. The program which is the most efficient and most in accordance with the main goal of this project will be used as the final complete version.

3.3.1 Changes Made

The strategy of this project is heavily constructed to the constraints initialised in the agile software development methodology. The short deadlines for turnover of sprints and immediate analysis allows the application to be put through multiple test phases ensuring the highest quality of sprints presented at the end of the project. The following workplan will expand on the one created in the PDD. A thorough review of the work to be performed for the implementation will be highlighted. Each component yet to be developed will be put into a sprint subjection and further broken down into completion, tests, and review. Extra features will be extracted and placed after the initial basic requirements of GAN Art are met. In this instance working versions of the code can be presented without interference of new features. The final sprint produced will include only the best features of each sprint, therefore creating the best in class of features and usability for the user. Once the work plan has been executed the project will move onto the implantation phase and handled mostly through VCS with essential reporting added to this document. Every sprint will be allocated a phase of implementation followed by essential testing which will be updated on the original work plan proposed in the document.

3.4 Requirements Documentation and Completeness Tracking

As expressed in the main goal the key scope of this paper is to produce a program which produces generated art from machine learning models, particularly GAN. This

was the basis of my requirement documentation. All other secondary goals should be intertwined with this main concept, essentially leading to software that is streamlined and efficient. Each individual sprint of the program will at least satisfy this to be considered as a complete program. To keep track of all these changes the development phase was carried out whilst writing the report. This way initial changes to the sprint were noted and amended if not sufficient to the report. As GitHub was used for tracking changes too it allowed previous versions of the sprint to be considered within the final evaluation. A table of the requirements of the project were previously laid out in the PDD, these are further expanded on in this section. Each objective and sub-objective was analysed in relation to the final goal of producing generative art with machine learning. Many of the sub-objectives were removed and placed into an additional feature's category. This slight change considers the additional features to be a luxury and only implemented if they fit within the time frame of the project. The table of requirements will be used as a means of completeness tracking. To satisfy the requirements, the program must pass various tests attached to each objective. These are done in the means of processing the data, analysing outcomes, and passing strenuous checks. This way the program can be abstracted down to its basic components, once these components are working as intended – development can continue. This method mitigates bugs which can arise in the future, if bugs do appear they can be easily tracked and tackled due to the abstract nature of the program.

3.5 Architecture Design

Previously discussed in the literature review section, the ArtGAN architecture may seem the most pleasing for this implementation for creating generative machine learning art (Tan et al., 2017). As a means of experimentation and development of something original, this paper will only take small concepts that are used within the ArtGAN implementation. Deep convolutional generative adversarial networks will be the first novel implementation of the program (Gao et al., 2018). This neural architecture crucially removes fully connected hidden layers which in turn allows deeper

architectures to be put in place. Here much more information with a larger number of constraints can be seen within the final image of the program, fundamentally this makes the image quality better. A difference from the ArtGAN implementation is that it replaces pooling layers with strided convolutions for its discriminator and fractional-strided convolutions within its generator. This architecture will be further analysed within a testing phase explained the Results section. Experimentation with conditioned-attention normalised GAN (CAN-GAN) will be carried out to create a hybrid of many architectures as a final implementation (Shi et al., 2020). CAN-GAN was originally proposed for face age synthesis, due to its fascinating architecture which changes various facial features to show aging by attention mapping, a similar concept to StyleGAN mapping, the same fine tuning can be used within portraits. The larger control over various aspects of the image can break portraits down into components that in turn can be manipulated by the GAN. The final architecture will be proposed through a means of experimentation and fine tuning. Due to the architecture of GAN being vast in this current time, testing on a multitude of GAN will be the best way to produce an exciting final result. The general GAN architecture of a discriminator and generator with other finalising features will be kept the same as the original. This ensures the integrity of the objectives of this paper.

3.6 Implementation

Most of the development of this project will be done on PyCharm and Kaggle. A problem arises from using a specific IDE especially with various packages that regularly update such as TensorFlow and PyTorch. This stops other users using the code as the code will need to be regularly changed when said updates are put into place. After the end of development, a Kaggle notebook file will be submitted to ensure other users can run the code without having all the packages installed on their local hardware. This increases usability of the code and further allows newcomers to be easily introduced to generative art made via machine learning. With the agile methodology, implementation of the program becomes very easy. As this is a research

and experimental project many components are going to go through many versions to better satisfy the outcome. Meaning throughout development the project will always be moving forward if it sticks to its time constraints. Using the guidelines of the agile methodology, this paper will give sufficient time for experimentation throughout the development phase. When sufficient changes are to be enacted due to experimentation, they will be record within the project and within the VCS - GitHub. To make sure the project stays on track, the Grantt chart with the specific requirements will be used throughout the implementation phase.

3.7 Development

To make sure development is continuously on track the requirements set out prior will be used as a guide. Experimentation will be key within development to first test out architectures that I am familiar with but also in terms of finding new ways to efficiently carry out the task at hand. Generative art with machine learning can be developed in many ways, however with the use of new and cutting-edge GAN technology it can be tuned perfectly to fit the use case. This tuning will come as an experimentation phase in the development of the program. The experiments will fine tune the final images and add or remove limiting factors leading to new results. Not all experiments will be mentioned within this paper if they do not contribute significantly to the outcome. Libraries which are used in this paper will be further broken down and analysed to highlight their advantages and weaknesses. This allows new developers reading this paper to make new decisions with the results at hand. In turn, performance of the program can be vastly improved. This can be the case if new implementations that speed up training in the machine learning model become apparent and added swiftly for the betterment of the program.

3.8 Maintenance

To maintain the project and its code, everything will be uploaded onto GitHub and its other features will be utilised throughout the development phase (Github, 2016). As mentioned previously, machine learning packages depreciate very quickly. This

leads to code becoming unusable in a couple of months and needs to be changed to match the changes on the new package updates. To get around this, Kaggle can maintain the original versions of the package and the code will work on any machine. Within Kaggle the option to use a free GPU will be even more beneficial to increase compute time. Different sprints that are produced will be finalised as working models, therefore each sprint can be run independently and tested for its performance and overall usability. Bug reports will be created with a form and validated with specific changes put in place to resolve them. This way all bug reports can be accessed and cached to capture recurrences which in turn can be flagged. In the unlikely event of GitHub not working, VCS will be moved to an offline drive on two separate PC's. This is to protect all avenues of availability of the code and the project itself as whole, allowing for maintenance to be carried out wherever necessary.

3.9 Testing and Evaluation

Testing will be a key part of the development of this project. As testing determines if a certain sprint satisfies the requirements to continue onto the next development phase. The tests for each component will be written in a separate program. This approach stops bugs from arising with the original code to be tested. As previously stated, every component must satisfy the test clause to continue development. Bug's present will be noted and dealt with. Testing of the whole sprint will be done when it is complete, and no other requirements are yet to be satisfied. This allows for the completeness of the program to be maintained. Each test result will be logged so comparisons of each version of the components can be compared with the best one progressing forward to the final implementation. At the end of each sprint an evaluation phase will commence. The requirements of the program will be highlighted with the specific implementation which satisfies the said clause. At the end of each evaluation phase new additional features that were added prior will be analysed to see if they improve the overall program. This strips away unnecessary features that may be present in the program but do not move it towards its main objective or cause it to lose performance for minimal gain. This level of evaluation will in-turn

produce a program which completely satisfies its requirements and is efficient in nature. Therefore, when new developers pick up this code it will be refined enough for them to work on this and add changes without needing to strip down the code.

3.10 Experimentation and Data Analysis

Throughout the development of the program there will be a very high emphasis on experimentation. As the nature of GAN is unpredictable to an extent on datasets with high variance. To get around this problem various architects of GAN will be tested, and their results recorded. The outputs will be broken down into a multitude of categories, for example, image quality, resolution, and variance of noise. The results which show the best of all three categories with performance in mind too will be selected as the preferred architects and put forward. The documented results will be further analysed to capture pitfalls and evaluated into new future methods. As experimentation is a big part of this process it will take the bulk of the project time. Analysis will be carried out after each experimentation phase to evaluate if the changes made are useful for further development of the project. Matplotlib will be the analysis library of choice as it has an intuitive easy to use design and can breakdown many of the other components of the program for analysis (Hunter, 2007). With the use of Matplotlib a large number of graphs can be produced which can compare multiple variables are once for example the discriminator loss compared to the generator loss.

Chapter 4

Results

Three sprints were developed for this project. The final sprint produced is the most mature implementation of the project goal. It takes all the previous developments made and produces a sprint which addresses issues with older versions whilst maintaining better results. The changes will be highlighted in their respective versions. The chosen architectures with their strengths and weaknesses will be further explored. Reasons for such changes will be mentioned with code snippets showing the area changes have been made. Deep convolution generative adversarial networks (DCGAN) will be the starting novel implementation of producing generating art through machine learning models. The original paper will be used as a skeleton of the implementation with various experimental changes making the bulk of the program. Reasons for such changes will be mentioned and explained.

4.1 Chosen Dataset

The dataset chosen for the development phase is the "Best Artworks of All Time" submitted by user Icaro (icaro, 2019). This dataset consists of art produced by 50 of the most influential artists of all time. It comes with a comma separated values (CSV) file which has a range of information of each artist; name, years active, genre and nationality are some of the categories the artists are separated into. Within the dataset images.zip contains a collection of full-size images which are divided in folders and sequentially numbered for every artist within the dataset. Originally this data was scrapped from Art Challenge (ArtChallenge, 2021). Another zip file of resized.zip contains images which have been resized to save more space for loading

the data and processing it. This will be the main zip file used out the development phase. The project makes use of this dataset due to its high variance is colliding genres of art. It contains art from the 14th century to the 20th century, these large periods cover a range of art movements which can lead to very interesting final outcomes for this project.

4.1.1 Exploring the Dataset

The Torchvision library from PyTorch will be used for loading and resizing the dataset, it was chosen due it's speed in these processes (Marcel and Rodriguez, 2010). To keep the program as efficient as possible when working on the dataset the images are saved as a binary file (Eddelbuettela and Wub, n.d.). If a binary file already exists, the program will pick up the binary file and skip this part of the process. The WIDTH and HEIGHT variables refer to the resolution of the images, they will be used to resize the images accordingly. A 64x64 resolution may be low in resolution but it is a good baseline to further tune the model before upscaling into higher resolutions, saving time and performance. The function prepare_data produces the binary file which is later carried onto the other functions and processed accordingly. The else statement occurs if there is a binary file found in the path and continues to load it into the program; as seen in figure 4.1. To make sure the images have been converted correctly into a binary file, the function visualize_random_art which takes the dataset parameter, prints out a grid of random art chosen from the dataset; as seen in figure 4.2. Before the figure is created the dataset is shuffled and the for loop after selects random images and shows them as output.

```
WIDTH = 64
HEIGHT = 64
IMG_SIZE = (WIDTH,HEIGHT)
EPOCHS = 250
```

```
def prepare_data(path_of_file , path_of_data):
    #Look for saved file to save time loading and processing images
        between runs
```

```
print("Looking for saved binary file...")

if not os.path.isfile(path_of_file):
    print("\n File not found, creating new file...\n")
    dataset = []
    transform_ds = transforms.Compose([transforms.Resize(IMG_SIZE),])
        #define transformation

    image_folder = torchvision.datasets.ImageFolder(root=path_of_data,
                                    transform=transform_ds)

    print('Number of artworks found: ',len(image_folder))



    print("Converting images, this will take a few minutes")
    for i in range (len(image_folder)):
        image_array = numpy.array(image_folder[i][0])
        dataset.append(image_array)
        if (i%500 == 0):
            print("Pictures processed: ", i)

    print("Saving dataset binary file...")
    dataset = np.array(dataset, dtype=np.float32)
    dataset = (dataset - 127.5) / 127.5 #Normalize to [-1 , 1]
    numpy.save(path_of_file, dataset) #Save processed images as npy
        file

else:
    print("Data found, loading..")
    dataset = np.load(path_of_file)

print("Dataset length: ", len(dataset))
```

```
    return dataset

dataset = prepare_data(saved_binary_file , dataset_path)

#Using a TensorFlow Dataset to manage the images for easy shuffling,
    dividing etc
BATCH_SIZE = 128

training_dataset =
    tf.data.Dataset.from_tensor_slices(dataset).shuffle(60000).
batch(BATCH_SIZE)
```

```
Looking for saved binary file...

 File not found, creating  new file...

Number of artworks found:  8683
Converting images, this will take a few minutes
Pictures processed:  0
Pictures processed:  500
Pictures processed:  1000
Pictures processed:  1500
Pictures processed:  2000
Pictures processed:  2500
Pictures processed:  3000
Pictures processed:  3500
Pictures processed:  4000
Pictures processed:  4500
Pictures processed:  5000
Pictures processed:  5500
Pictures processed:  6000
Pictures processed:  6500
Pictures processed:  7000
Pictures processed:  7500
Pictures processed:  8000
Pictures processed:  8500
Saving dataset binary file...
Dataset length:  8683
```

Figure 4.1: Output of the function, prepare_data

```
def visualize\_random\_art(dataset):
    """
    function to plot some random images from the data set

    """
    np.random.shuffle(dataset) #Shuffle the images

    fig = plt.figure(figsize=(12,12))
    for i in range(1,37):
```

```
        fig.add_subplot(6,6,i)
        plt.imshow(dataset[i])
        plt.axis('off')

visualize_random_art(dataset)
```

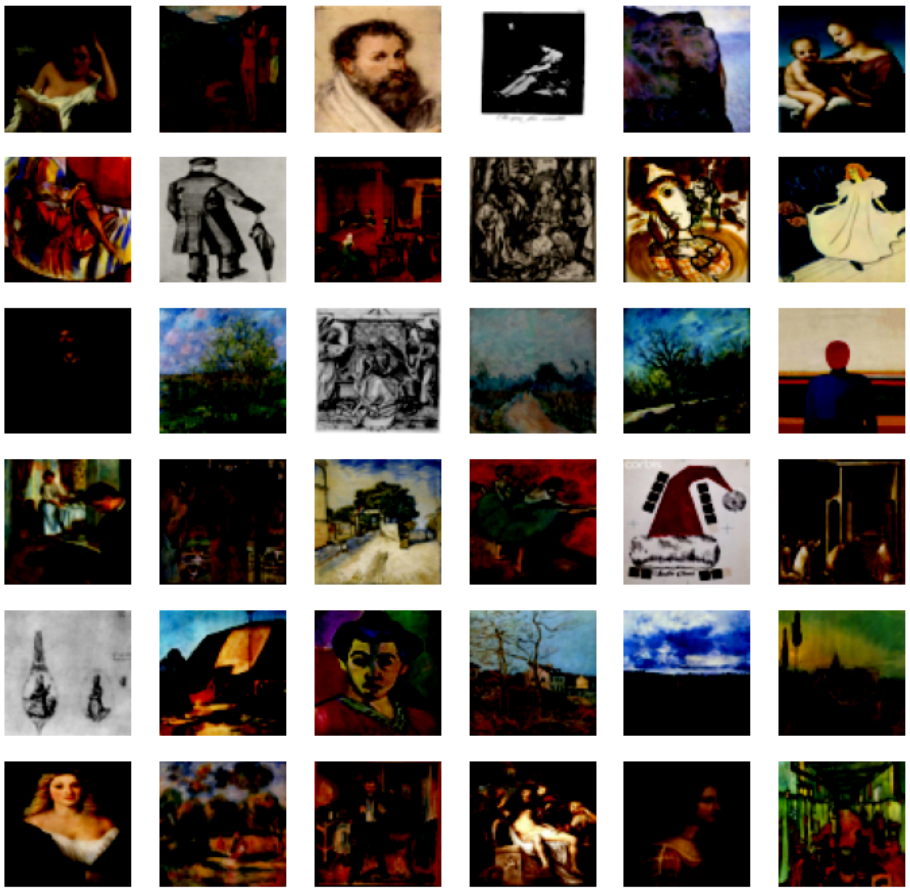
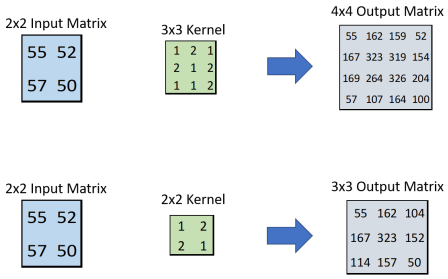


Figure 4.2: Output for the function visualise_random_art

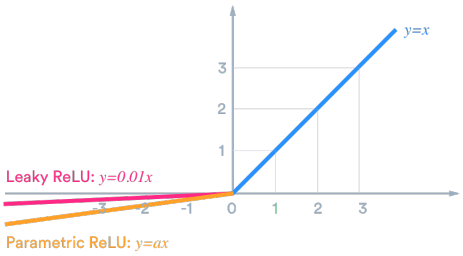
4.2 Version 1

4.2.1 Architecture Choices

The generator takes in a random seed to produce a generated image from it. The use of Conv2DTranspose deviates from the original DCGAN paper proposal; see figure 4.3a for a brief explanation. Reason for this is the increased efficiency and results of transposed convolutions. The input provided by the seed is transposed over the



(a) A brief explanation of Conv2DTranspose



(b) LeakyReLu activation function

Figure 4.3: Conv2DTranspose and LeakyReLu infographics

kernel to produce an output. These parameters are trainable and can produce better results in the long run. Upsampling and Conv2D is used for the generator in the original DCGAN paper; the problem with Upsampliung is that it does not have any trainable parameters, so it repeats its input to produce an output, leading to a large amount of generator information being lost. The discriminator in this architecture uses Conv2D, which is the same as the DCGAN paper. Batchnorm is used for normalization of the generator and discriminator for pre-processing of the data. The activation function of this particular architecture is the rectified linear unit (ReLU) activation function which is used for all layers of the generator except its final output layer which uses TanH. ReLU is popular for most deep learning models so it is a great place to start and further develop into new models. To improve the strength of the discriminator, in this case to classify which images are real or counterfeit, the Leaky ReLU activation function is used; see figure 4.3b for an infograph. The Adam optimizer is used for both the generator and the discriminator. As this is a relatively noisy problem due to the number of pixels and working on images, the Adam optimization algorithm will be able to handle the sparse gradients within the generator and discriminator.

4.2.2 Generator Design

The function make_generator_model which takes seed size and channels as parameters. This architecture is true to the original DCGAN paper. This sequential model is used as it is generally good for most of neural network projects and works well. ReLU is used as the activation function for the first dense layer, with the

seed_size being the input_dim. Batch Normalization is used to normalize all the input. The normalization process stabilizes the learning of the model, the input units is set to have zero mean and unit variance. This is key for the model to not fail when it begins training. The final layer has no Batch Normalization added it to it as it causes sample oscillation and model instability, which was tested and confirmed in an experimentation phase. Essentially leaving Batch Normalization within the final layer results in outputs which vary from non-trained outputs (images which are pure noise values), to semi-trained outputs. LeakyReLU is used from here on out as the soul activation layer except for the final layer. This activation function deals well with gradients which is the perfect use case for image generation. Setting the alpha in LeakyReLU, which is the slope of leak, was set to 0.2 – as per the original paper. This deviates from the original GAN paper (Goodfellow et al., 2014) which uses maxout activation. Nevertheless, LeakyReLU preserves as much information as possible throughout training. The dropout rate was set to 0.4 for the middle layers, this makes sure the network learns a redundant representation of all the input. This prevents overfitting within the network and makes the network more robust. As each node takes the consensus over an ensemble of networks which dramatically improves performance. This generator model complies well enough with the original DCGAN performance and is fine-tuned later on in the code to work best for the chosen dataset.

```
def make_generator_model(seed_size, channels):
    """
    This function builds the generator for DCGAN
    Parameters:
        seed_size:according to authors of DCGAN , Generator takes this
            random seed and generates an image
        channels : output channels that image will have
    """
    model = tf.keras.Sequential()

    model.add(Dense(4*4*512,activation="relu",input_dim=seed_size)) #64x64
        units
```

```
model.add(Reshape((4,4,512)))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha = 0.2))


model.add(Conv2DTranspose(512, (5, 5), strides=(2, 2), padding='same',
    use_bias=False))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha =0.3))
model.add(Dropout(0.4))

model.add(Conv2DTranspose(256, (5, 5), strides=(2, 2), padding='same',
    use_bias=False))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha = 0.2))
model.add(Dropout(0.4))

model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
    use_bias=False))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha =0.3))
model.add(Dropout(0.4))

model.add(Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same',
    use_bias=False ))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha = 0.2))

model.add(Conv2DTranspose(channels, (5, 5), strides=(1, 1),
    padding='same', use_bias=True, activation='tanh'))
```

```
    return model
```

4.2.3 Discriminator Design

A standard sequential model inspired by the DCGAN paper is used as the discriminator model. Creating a discriminator with a high degree of accuracy can give an upper hand in producing final images which are very accurate to the original. This is essentially a convolutional neural network (CNN) based image classifier, trying to tell the real or fake images apart. The Conv2D layers allows the network to learn its own spatial down sampling, again different from maxpooling with strided convolutions. The program doubles the filters of the Conv2D layers, whilst giving the input shape on the first layer. The input_shape function takes the height, width and colour channel of the input images, in this case 64x64, channel 3 which represents the RGB colour space. Batch Normalization is within the middle layers of the discriminator model, but they are avoided in the input layers as it leads to oscillation and model instability. A momentum of 0.9 is set to Batch Normalization which in turn increases the performance of the model. The LeakyReLU activation function with a slope of 0.2 is added to all layers except the final layer, where a simple sigmoid function is used. LeakyReLU functions promote a healthy gradient flow of activating inputs, this essentially improves the learning process of both the generator and discriminator. Before feeding the model into a sigmoid function all the layers are flattened. From here the network decides if a particular image is real, which it will evaluate to a scaler probability ratio close to 1, or counterfeit which is a scaler probability ratio close to 0.

```
def make_discriminator_model(image_shape):
    """
    This function builds a discriminator which is a CNN based image
        classifier
    and will output probability values of what it thinks is fake or real
        with values close to
    0 being fake and 1 being real.
```

```
    Parameters :
        image_shape : input_image shape (h x w x c)
    """
    model = tf.keras.Sequential()
    model.add(Conv2D(64, kernel_size=5, strides=2, input_shape=(64, 64,
        3), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(128, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(256, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(512, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Flatten())
    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    return model
```

4.2.4 Generator Loss

The generator loss (cross entropy) is a measurement of how well it performed at fooling the discriminator. If the input presented to the discriminator by the generator is classified as real, the generator done a good enough job to bypass the detection of the discriminator. The model is penalized proportionally to how much the predicted

probability distribution varies from the expected probability distribution of the fake image due to the existence of cross entropy. It serves as a foundation for the error, which is then backpropagated via the generator and discriminator, allowing the next batch to perform better.

```
def generator_loss(fake_output):
    """
    The generators loss is a measurement of how good it performed at
        fooling the discriminator.
    If the discriminator classifies the fake images as 1, the generator
        did a good job.

    Parameters :
     fake_output : fake image from generator
    """
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

4.2.5 Discriminator Loss

The discriminators loss function (cross entropy) is based on its ability to distinguish real images from the original dataset and fake ones provided by the generator. It compares the real images to an array of ones, with one being real, it then compares its prediction of counterfeit images on an array of zeroes, with anything close to zero being fake. The goal of this function is to classify all real images as one and all fake images as zero. The loss is signified is signified by adding the two losses of both classification processes together. In this case if the discriminator were to predict a probability of 0.1 when the actual value is 1, it would be considered a very bad result and a high loss value. For a perfect model to exists the log loss will be 0, which is extremely hard to do in machine learning models. The discriminator_loss function takes the parameters of the real output defined by our discriminator model and the fake output provided by our generator model. The first line of code real_loss compares the real image to an array of ones and gains a probability of the output being real (probability close to one). The second line fake_loss compares the fake

output produced by the generator to an array of zeroes and gains a probability of the output being fake (probability close to zero). The penultimate line adds both of the probability predictions together and finally they are returned as the total_loss, which is the output of the discriminator_loss function.

```
def discriminator_loss(real_output, fake_output):
    """
    The discriminators loss is based on its ability to distinguish real
        images from fakes.
    It compares its predictions on real images to an array of ones
        (remember 1 being real)
    and its predictions on fake images to an array of zeros (0 being fake).
    The goal is to classify all real images as 1 and all fakes as 0.
    The total loss is then these two losses added together.

    Parameters :
        real_output : real image from the dataset
        fake_output : fake image from the dataset
    """
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

4.2.6 Optimization and Checkpoints

The Adam optimizer is used for the generator and the discriminator. Both models' optimizers are separated as training occurs separately for each one. As a result, changes to one model come at the detriment of the other. However, this does improve overall image results in the grand scheme of things. This also means that if one of the models' parameters is modified, the structure of the optimization problem being solved changes. The Adam optimizer seeks and equilibrium between the generator and the discriminator. In the original DCGAN paper the learning rate is set to

0.0002, this was too low in this case, so changing it to 0.0001 with the leaving momentum set to 0.5 help stabilize the training. These hyper tuned parameters worked well and gave impressive results. A simple checkpoint function was written to increase efficiency of the program and deal with unexpected stoppages. These checkpoints are called later if training is resumed. The method generate_images generates a plot of 21 images. It needs 21 images as input, images are stacked in a single tensor as generated_images2, once it loops to 21 it shows all the images.

```
#The two models optimizers are separated because we train them separately.
# I found a slightly lower generator LR to be beneficial.
#Beta value of 0.5 generates more stable models as per the findings in
    the paper
#"Unsupervised representation learning with deep convolutional generative
    adversarial networks"

generator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)
```

```
!mkdir ./training_checkpoints

#checkpoint for saving a model

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

4.2.7 Training

The number of epochs to train for has been set to 150. The train_step function takes images as its parameters and returns the generator and discriminator loss.

The training begins with the generator receiving a random seed, which is then used to produce an image. The discriminator then sorts the images into two categories: fake and real. Each model's loss is measured separately, and the gradients are modified. The modification of the gradients with back propagation from the generator and discriminator loss improves the model significantly on each epoch. Since we're updating the weights of two models that use different losses, they're both reliant on each other, so this needs to be coded explicitly. Remember that the purpose of discriminator training is to increase the likelihood of correctly classifying a given input as real or false. This can be accomplished by ascending the stochastic gradient of the discriminator. A batch of real samples will be generated from the training set and transferred to the discriminator. Following that, the loss would be estimated, allowing the gradients to be calculated backwards. Another batch of fake samples from the current generator are taken and passed through the discriminator, the loss is the calculated. The gradients from both are accumulated through a backward pass. After this the discriminators optimizer is called. Throughout the training process, the generator's main goal is to produce better fakes. In this instance the program uses GradientTape to compute the generators loss and computes the generators gradient in a backward pass. At the end of this process the generators parameters are updated with an optimizer step.

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    """
    The training begins by providing a random seed to the generator, which
        is then used to generate
    an image. The discriminator then classifies images from both the fake
        and real dataset.
    The loss is calculated separately for each model and the gradients are
        updated.

    Parameters :
```

```
    images : images to be trained on
    """
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)
      #generate_images(generated_images)
    gradients_of_generator = gen_tape.gradient(gen_loss,
        generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
        discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
    generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
        discriminator.trainable_variables))

    return gen_loss , disc_loss

def train(dataset, epochs):
  for epoch in range(epochs):
    gen_loss_list = []
    disc_loss_list = []
    start = time.time()

    for image_batch in dataset:
        t =  train_step(image_batch)
```

```
        gen_loss_list.append(t[0])
        disc_loss_list.append(t[1])
    # Produce images for the GIF as you go
    # display.clear_output(wait=True)
    g_loss = sum(gen_loss_list) / len(gen_loss_list) #calculate losses
    d_loss = sum(disc_loss_list) / len(disc_loss_list)
    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time for epoch {} is {} sec'.format(epoch + 1,
        time.time()-start))

    print(f'Epoch {epoch+1}, gen loss = {g_loss}, disc loss = {d_loss}')
# Generate after the final epoch
display.clear_output(wait=True)
```

```
Epoch 139, gen loss = 4.021149158477783, disc loss = 0.48265889286994934
Time for epoch 140 is 13.968857288360596 sec
Epoch 140, gen loss = 3.536060094833374, disc loss = 0.2987475097179413
Time for epoch 141 is 13.972722053527832 sec
Epoch 141, gen loss = 3.795200109481811 5, disc loss = 0.16455313563346863
Time for epoch 142 is 13.97269058227539 sec
Epoch 142, gen loss = 3.8699722290039062, disc loss = 0.15631920099258423
Time for epoch 143 is 13.957346439361572 sec
Epoch 143, gen loss = 4.024540901184082, disc loss = 0.14344005286693573
Time for epoch 144 is 13.985370874404907 sec
Epoch 144, gen loss = 4.179721832275391, disc loss = 0.4071517586708069
Time for epoch 145 is 13.929293870925903 sec
Epoch 145, gen loss = 3.5912346839904785, disc loss = 0.26517924666404724
Time for epoch 146 is 13.95818567276001 sec
Epoch 146, gen loss = 3.802884817123413, disc loss = 0.1712428629398346
Time for epoch 147 is 14.011319398880005 sec
Epoch 147, gen loss = 3.9605305194854736, disc loss = 0.12468817830085754
Time for epoch 148 is 13.957943201065063 sec
Epoch 148, gen loss = 4.002618789672852, disc loss = 0.13388872146606445
Time for epoch 149 is 13.993006944656372 sec
Epoch 149, gen loss = 4.406344413757324, disc loss = 0.3692184090614319
Time for epoch 150 is 14.43604588508606 sec
Epoch 150, gen loss = 3.997933864593506, disc loss = 0.22565357387065887
```

Figure 4.4: Training ran for 150 Epochs

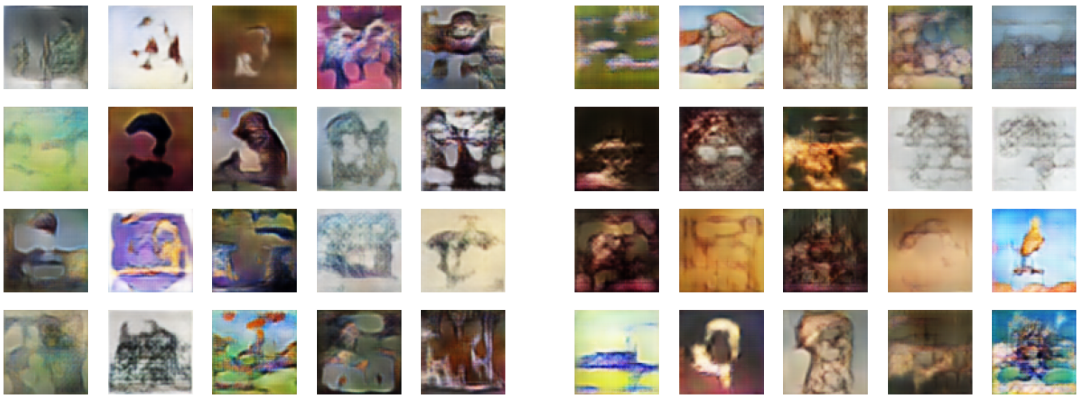


Figure 4.5: Results for 64x64 DCGAN: Version 1

4.2.8 Outcomes

The final images produced by the network are definitely satisfactory. As it is the first implementation of generative art through machine learning models, the techniques used were mostly from a hand full of papers. All of the images are abstract in nature with various indistinguishable shapes are mapped out onto the images. The colour palette takes from many different genres of paintings showing the model has learnt from all of its given input. The dataset can be blamed to an extent as it exhibits art from various movements within many different time periods. At the end it produces visually pleasing images with little amounts of noise. As noise is not so much of an issue in the images, a larger focus can be put on other objects showing in the image. The strength of the DCGAN architecture is visible in this instance as it can withstand a dataset with a range of inputs and still produce satisfactory results. The hyper tuning of the parameters allows the program to consistently learn and update its weight accordingly which works well for the final outcome. The use of the Adam optimizer limits the vanishing gradient effect on the generator and the discriminator. The robustness of the program to produce an .h5 file which can be applied to another dataset will allow it to be used on multiple data sets without the need for retraining. Overall, the training time for the model was short, running it on Kaggle further helped make the case for this. The main problem arising from the outcomes is the relatively small resolution. This problem will be expanded on and given a clear explanation on the next version.

4.3 Version 2

4.3.1 Changes Made

One of the issues with the first sprint was its low resolution. No discernible objects can be made out in the final image. Although it can be considered to achieve its objective function it lacks in many ways as the final implementation of the program. A clear goal to overcome the low resolution is to write the same architecture for 128x128 resolution. After this resolution has been overcome, the same program can be written for 256x256 resolution. The new changes may increase training time but may wield better results. The results this program expects to produce overall GAN stability and better image quality. Consideration for High-resolution Deep Convolutional Generative Adversarial Networks (HDCGAN) were explored however the architecture adds additional features which considerably slow down overall performance but do not wield any better results. This led to the thought that the architecture may not be suitable for high quality generated images. As there are more variables to consider and a much larger generator and discriminator model, it causes the program many times to run out of GPU memory. If it were to solely run on a CPU the DCGAN can take huge amounts of time for a single epoch. Currently a single epoch is completed in roughly 13 seconds on Kaggle with a 16GB GPU. The program tries to produce the best results for the least amount of training and processing time. This is important as it stops inefficient architectures being developed which may produce better results as a lost to performance.

```
WIDTH = 128
HEIGHT = 128
IMG_SIZE = (WIDTH,HEIGHT)
EPOCHS = 150
```

```
noise = tf.random.normal([1,SEED_SIZE])

generated_image = generator(noise, training=False)
```

```
plt.imshow(generated_image[0, :, :, 0])

image_shape = (HEIGHT, HEIGHT, IMAGE_CHANNELS)

discriminator = make_discriminator_model(image_shape)
print(discriminator(generated_image))
```

4.3.2 Implementation of Changes

To scale up the GAN to a higher resolution the WIDTH and HEIGHT variables were changes accordingly to 128x128. The training process was stable for 50 epochs, as seen in the figure 4.6. The discriminator in both models worked very well in both cases staying roughly between 0.3 to 0.4 loss, meaning the model was classifying images correctly to a high degree. At the beginning of the training phase in the first few epochs the generator has not learnt enough to produce good enough results for the discriminator; the generator loss stays high sub 10 epochs. Then gradually the generator loss decreases to 40 epochs, around 30-40 epochs there is the smallest amount of generator loss. Once its past 40 the generator loss starts to shift higher. The results produced for 50 epochs showed slight improvements to the original image. This is in the case of producing new distinguishable objects in the photo repeatedly with minimum amounts of noise. DCGAN architecture is meant to produce better results the longer the model is trained for. In this case the epochs for the program were changed to the original 150 and retrained on the dataset. The discriminator loss stayed consistent as the previous models. However, the generator loss in the model after 50 epochs diverges and fluctuates higher as the number of epochs increases. This shows a level of instability to the architecture which this project attempts to avoid. A lot of reasons can be the backbone of this problem such as bad initialization of the generator model. ArtGAN expressed previously in the literature review uses a form of mapping from the original images to produce better results instantly. This can be a workaround for the problem at hand but other factors such as local minima, saddle point loss, or simply the instability of the DCGAN architecture on large resolution

images, can contribute much to the problem. The generator loss may have decreased if trained for even more epochs however this would have not showed an intuitive and robust design. A new solution arises from this unique problem, Wassertein GAN with Gradient Penalty.

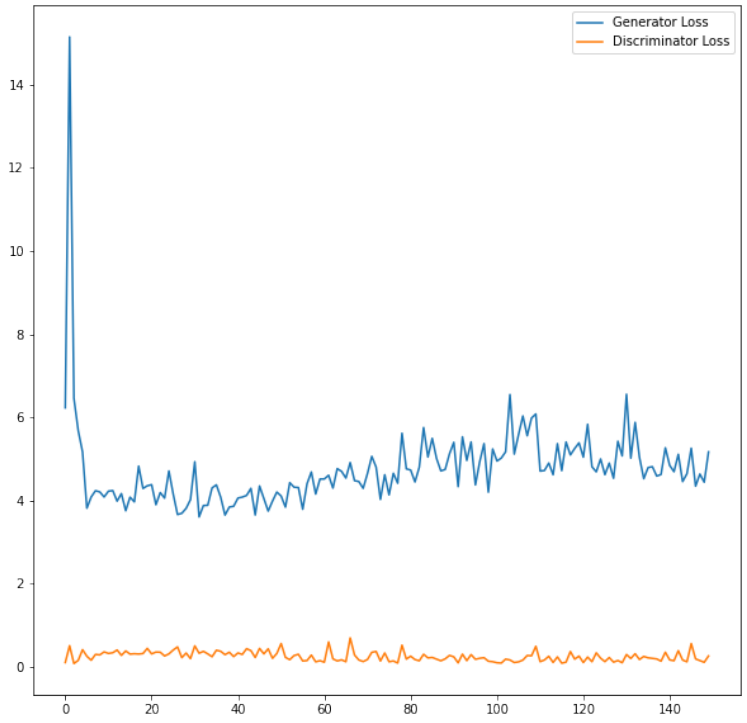


Figure 4.6: Generator and Discriminator Loss for 150 epochs

```
#The two models optimizers are separated because we train them separately.
# I found a slightly lower generator LR to be beneficial.
#Beta value of 0.5 generates more stable models as per the findings in
    the paper
#"Unsupervised representation learning with deep convolutional generative
    adversarial networks"

generator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)
```

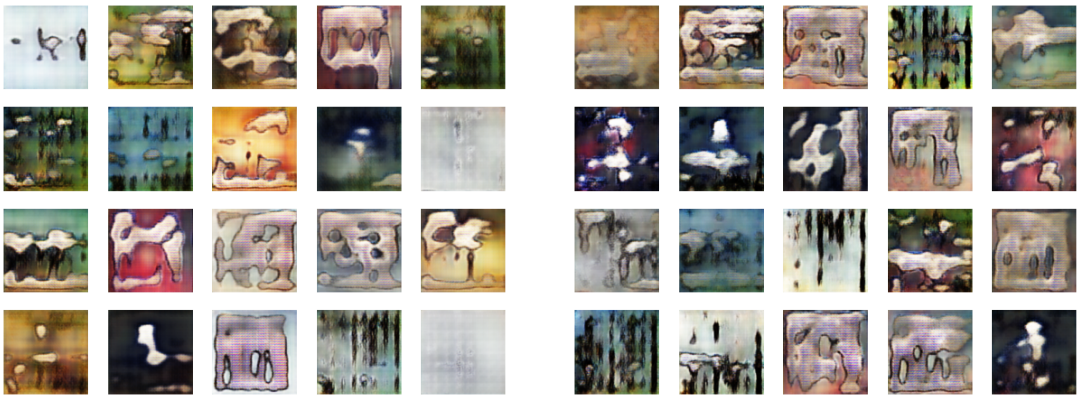


Figure 4.7: Results for 128x128 DCGAN: Version 2

```
def generate_images(generated_images2):
  # Notice ‘training‘ is set to False.
  # This is so all layers run in inference mode (batchnorm).
    generated_images2 =0.5 * generated_images2 + 0.5

    fig = plt.figure(figsize=(10,10))
    for i in range(1,21):
        fig.add_subplot(5,5,i)
        plt.imshow(generated_images2[i])
        plt.axis('off')

    plt.show()
```

4.4 Version 3

4.4.1 Architecture Choices

To develop a training procedure which is stable and does not increase in generator loss throughout the program, the architecture of the GAN will change to the Wasserstein GAN with Gradient Penalty (WGAN-GP) (Gulrajani et al., 2017). This GAN has better training stability and leaps over the hyper tuning issues of a DCGAN. The first model may have been fixed with strenuous hyper tuning of each parameter however it would have been counter intuitive as it would have been constrained to

the specific dataset. The loss in most GAN models provides a sense of ambiguity to what is specifically going wrong in the program and can not be used to better the program; only used to capture the progress of improvements. With WGAN-GP the loss can be used as a termination criteria where training will stop if the loss converges with the actual discriminator, essentially stopping training at the correct time with the highest degree of accuracy. In the case of this paper this is extremely helpful to pinpoint the lowest generator loss, subsequently producing better images.

4.4.2 Inherent Changes in GAN Architecture

There are two probability distributions in GAN: one for the generator and another for the discriminator. The generator's probability distribution attempts to match the discriminator's probability distribution as closely as possible. When this happens, the generator would be able to be classified as a real image by the discriminator. On a graph this is shifting the distribution of the generator as close to the distribution of the discriminator, as seen in figure 4.8. This arises a new problem of how distance is defined in a probability distribution, specifically the distance between the probability distributions of the generator and the discriminator. In WGAN-GP the Wasserstein distance is used which vastly improves the convergence of the two distributions.

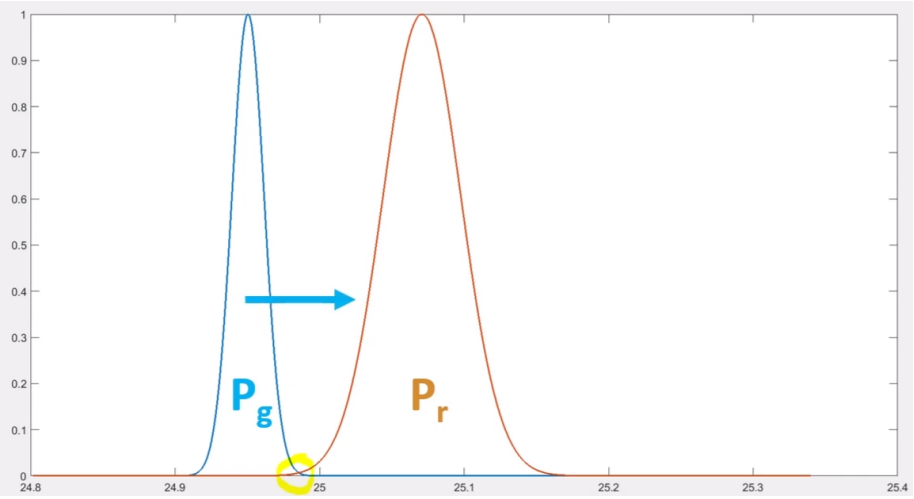


Figure 4.8: Pg represents the generators probability distribution, whilst Pr represents the probability distribution of the real images - Wasserstein defines a way to minimise the distance of the two distributions

4.4.3 Explanation of Architetcture

One of the main reasons to switch to the WGAN-GP as it solves the stability issues that occurred with DCGAN. WGAN-GP uses a special kind of loss function known as the Wasserstein-Loss, this is one of the main contributors which prevents mode collapse in the network. Mode collapse is when the network outputs only a specific number of classes rather than the whole network. Another reason to switch to the WGAN-GP is its ability to be run for as long as possible. As DCGAN progressively gets worse as the number of epochs increases, WGAN-GP benefits and gets better for more iterations. This means the best possible outcomes can only be put forward in this implementation. A significant change in the architecture from DCGAN is the introduction of a "CRITIC" instead of a discriminator. As a lot of useful information can be lost from classify images through ones and zeroes, the "CRITIC" in WGAN-GP scores images with real numbers. Here a larger amount of information is passed onto the generator to significantly improved itself on each iteration. The critic in this architecture is trained more than the generator which sets a higher boundary for images to be classified as real. In a standard WGAN-GP the critic is trained five times more than the generator. This useful loss metric is related to the convergence of the generators and improves sample quality, which refers to the images sent to the critic. It also improves stability of the GAN whilst preventing mode collapse. Gradient penalty improves the Adam optimizer that was initially used in the DCGAN. The momentum from the Adam optimizer is completely removed. To change the training of the critic the WGAN-GP takes an interpolation of a real image and a generated image, this is done by taking a random number between epsilon and one. The norm of the gradient is taken which satisfies the Lipchitz constraint put on the critic.

```
class Critic(nn.Module):
    '''
    Critic Class
    Values:
        im_chan: the number of channels in the images, fitted for the
            dataset used, a scalar 3 for rgb
```

```
        hidden_dim: the inner dimension, a scalar
    '''
    def __init__(self, im_chan=3, hidden_dim=64):
        super(Critic, self).__init__()
        self.crit = nn.Sequential(
            self.make_crit_block(im_chan, hidden_dim),
            self.make_crit_block(hidden_dim, hidden_dim * 2),
            self.make_crit_block(hidden_dim * 2, im_chan,
                final_layer=True),
        )

    def make_crit_block(self, input_channels, output_channels,
        kernel_size=4, stride=2, final_layer=False):
        '''
        Function to return a sequence of operations corresponding to a
            critic block of WGAN;
        a convolution, a batchnorm (except in the final layer), and an
            activation (except in the final layer).
        Parameters:
            input_channels: how many channels the input feature
                representation has
            output_channels: how many channels the output feature
                representation should have
            kernel_size: the size of each convolutional filter, equivalent
                to (kernel_size, kernel_size)
            stride: the stride of the convolution
            final_layer: a boolean, true if it is the final layer and
                false otherwise
                      (affects activation and batchnorm)
        '''
        if not final_layer:
            return nn.Sequential(
```

```
                nn.Conv2d(input_channels, output_channels, kernel_size,
                    stride),
                nn.LeakyReLU(0.2, inplace=True),
            )
        else:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size,
                    stride),
            )

    def forward(self, image):
        '''
        Function for completing a forward pass of the critic: Given an
            image tensor,
        returns a 1-dimension tensor representing fake/real.
        Parameters:
            image: a flattened image tensor with dimension (im_chan)
        '''
        crit_pred = self.crit(image)
        return crit_pred.view(len(crit_pred), -1)
```

4.4.4 Gradient Penalty

The gradient penalty is computed in two functions. Two scenarios can arise for computing the gradient: computing the gradient with respect to the images and computing the gradient penalty given the gradient. The gradient is first computed by taking the interpolation of two images, one from the generator one from the discriminator. This is done by weighing the fake and real images via a random vector between epsilon and 1, then these two numbers are added together. Once the intermediate is calculated the critic produces a new output for the image. Then the program computes the gradient of the critics score on the mixed images (output) with respect to the pixels of the mixed images input. To compute the gradient pen-

alty given the gradient the magnitude of each image's gradient is calculated, this is also called the norm. After this the program calculates the penalty by squaring the distance between each magnitude and the ideal norm of one by taking the mean of all squared distances. This means that instability is minimized as the mean of these gradients will make partial changes to the generator. This also stops the exploding gradient problem where the discriminator might want to change the generator by a very large amount, again, reducing instability in the network and preventing mode collapse. The function get_gradient takes the critic model, a batch of real images, a batch of fake images and epsilon, which is a vector of the uniformly random proportions of the real and fake images per mixed image. This function returns the gradient of the critics scores with respect to the given mixed image. The function gradient_penalty takes a gradient from the previous function and returns the gradient penalty. The function calculates the magnitude of each image gradient and penalises the mean quadratic distance of each magnitude to one given a batch of image gradients.

```
def get_gradient(crit, real, fake, epsilon):
    '''
    Return the gradient of the critic's scores with respect to mixes of
        real and fake images.
    Parameters:
        crit: the critic model
        real: a batch of real images
        fake: a batch of fake images
        epsilon: a vector of the uniformly random proportions of real/fake
            per mixed image
    Returns:
        gradient: the gradient of the critic's scores, with respect to the
            mixed image
    '''
    # Mix the images together
    mixed_images = real * epsilon + fake * (1 - epsilon)
```

```
    # Calculate the critic's scores on the mixed images
    mixed_scores = crit(mixed_images)

    # Take the gradient of the scores with respect to the images
    gradient = torch.autograd.grad(
        #we need to take the gradient of outputs with respect to inputs.

        inputs=mixed_images,
        outputs=mixed_scores,

        # These other parameters have to do with the pytorch autograd
            engine works
        grad_outputs=torch.ones_like(mixed_scores),
        create_graph=True,
        retain_graph=True,
    )[0]
    return gradient
```

```
def gradient_penalty(gradient):
    '''
    Return the gradient penalty, given a gradient.
    Given a batch of image gradients, you calculate the magnitude of each
        image's gradient
    and penalize the mean quadratic distance of each magnitude to 1.
    Parameters:
        gradient: the gradient of the critic's scores, with respect to the
            mixed image
    Returns:
        penalty: the gradient penalty
    '''
    # Flatten the gradients so that each row captures one image
    gradient = gradient.view(len(gradient), -1)
```

```
    # Calculate the magnitude of every row
    gradient_norm = gradient.norm(2, dim=1)

    # Penalize the mean squared distance of the gradient norms from 1

    penalty = (1/len(gradient)) *torch.sum(( gradient_norm - 1 )**2)

    return penalty
```

4.4.5 Generator Loss

The generator loss is calculated by maximizing the critics prediction on the generator's fake images. given the score for all fake images the mean is calculated. The function get_gen_loss takes the critics scores of the fake images as a parameter and returns the loss of the generator. The model is penalised proportionally to how much the predicted probability distribution varies from the expected probability distribution for a given image using the loss function. The higher the predicted output of produced images, the lower the critic's loss when analysing generated images.

```
def get_gen_loss(crit_fake_pred):
    '''
    Return the loss of a generator given the critic's scores of the
        generator's fake images.
    Parameters:
        crit_fake_pred: the critic's scores of the fake images
    Returns:
        gen_loss: a scalar loss value for the current batch of the
            generator
    '''

    gen_loss = -torch.mean(crit_fake_pred)
```

```
    return gen_loss
```

4.4.6 Discriminator Loss

In terms of the critic, the loss is calculated by maximizing the distance between the critic's predictions on the real images and the prediction on the fake images, this is done whilst also adding a gradient penalty. The gradient penalty is weighed accord to lambda. Given the scores for all the images in the batch, the program uses the mean of them. For a hundred epochs the critics loss gradually increases whilst the generator loss decreases consistently. This is a large improvement over the first iteration of this program. It does not struggle at 50 epochs as the first DCGAN, it continually improves itself. This essentially means the quality of the images produced are going to increase as long as the training continues. A WGAN-GP can be running for as long as it needs to be. This flexibility allowed me to run the model for over 300 epochs which produced outstanding results. Whilst still maintaining a relatively small critic loss and decreasing its generator loss on each iteration.

```
def get_crit_loss(crit_fake_pred, crit_real_pred, gp, c_lambda):
    '''
    Return the loss of a critic given the critic's scores for fake and
        real images,
    the gradient penalty, and gradient penalty weight.
    Parameters:
        crit_fake_pred: the critic's scores of the fake images
        crit_real_pred: the critic's scores of the real images
        gp: the unweighted gradient penalty
        c_lambda: the current weight of the gradient penalty
    Returns:
        crit_loss: a scalar for the critic's loss, accounting for the
            relevant factors
    '''
```

```
    crit_loss = torch.mean(crit_fake_pred) -torch.mean(crit_real_pred) +
        c_lambda*gp

    return crit_loss
```

4.4.7 Training

The training of this model is very slow compared to the DCGAN; this is understandable as WGAN-GP is a much heavier process in compute power. The gradient penalty requires to compute the gradient of a gradient – this means potentially a few minutes per epoch depending on the hardware. To get the best results WGAN-GP should be run for as long as possible on a powerful GPU. Another change from the original DCGAN is that the critic is updated multiple times every time the generator is updated. This essentially helps the generator from overpowering the critic. WGAN-GP does not directly improve the overall performance of a GAN – but it does save time from tuning hyper parameters meticulously. WGAN-GP also increases the stability of the model and avoids mode collapse which saves more time in the long run. WGAN-GP will be able to train in a much more stable way than the vanilla DCGAN although it will generally run a bit slower in terms of compute time. The training constraints of WGAN-GP original paper have been put in place for the best results (FIGURE). The training process is visualized throughout to keep track of the changes. The training process is written in one large for loop starting with updating the critic. The function of the average critic loss in the batch is calculated and used in the visualization process. After this the gradients and optimizer are updated, the first half of the for loop returns the average critic loss. The generator and weights are updated accordingly. The generator loss is also calculated and used further down for visualization and analysis purposes. This training process is much heavier, but it introduces new avenues to explore deeper into GAN for generative art. It also wields the best results out of the three models presented in this paper. Each image is far more advanced than the average output delivered by the vanilla DCGAN.

```
cur_step = 0
```

```
generator_losses = []
critic_losses = []
for epoch in range(n_epochs):
    # Dataloader returns the batches
    for real, _ in tqdm(dataloader):
        cur_batch_size = len(real)
        real = real.to(device)

        mean_iteration_critic_loss = 0
        for _ in range(crit_repeats):
            #Update critic
            crit_opt.zero_grad()
            fake_noise = get_noise(cur_batch_size, 64, device=device)
            fake = gen(fake_noise)
            crit_fake_pred = crit(fake.detach())
            crit_real_pred = crit(real)

            epsilon = torch.rand(real.shape ,device=device,
                requires_grad=True)
            gradient = get_gradient(crit, real, fake.detach(), epsilon)
            gp = gradient_penalty(gradient)
            crit_loss = get_crit_loss(crit_fake_pred, crit_real_pred, gp,
                c_lambda)

            # Keep track of the average critic loss in this batch
            mean_iteration_critic_loss += crit_loss.item() / crit_repeats
            # Update gradients
            crit_loss.backward(retain_graph=True)
            # Update optimizer
            crit_opt.step()
        critic_losses += [mean_iteration_critic_loss]

        #Update generator
```

```
gen_opt.zero_grad()
fake_noise_2 = get_noise(cur_batch_size, 64, device=device)
fake_2 = gen(fake_noise_2)
crit_fake_pred = crit(fake_2)

gen_loss = get_gen_loss(crit_fake_pred)
gen_loss.backward()

# Update the weights
gen_opt.step()

# Keep track of the average generator loss
generator_losses += [gen_loss.item()]

#Visualization code
if cur_step % display_step == 0 and cur_step > 0:
    gen_mean = sum(generator_losses[-display_step:]) / display_step
    crit_mean = sum(critic_losses[-display_step:]) / display_step
    print(f"Step {cur_step}: Generator loss: {gen_mean}, critic
        loss: {crit_mean}")
    show_tensor_images(fake)
    show_tensor_images(real)
    step_bins = 20
    num_examples = (len(generator_losses) // step_bins) * step_bins
    plt.plot(
        range(num_examples // step_bins),
        torch.Tensor(generator_losses[:num_examples]).view(-1,
            step_bins).mean(1),
        label="Generator Loss"
    )
    plt.plot(
        range(num_examples // step_bins),
```

```
            torch.Tensor(critic_losses[:num_examples]).view(-1,
                step_bins).mean(1),
            label="Critic Loss"
        )
        plt.savefig(f"figure_loss_epoch_step_{cur_step}_.png")
        plt.legend()
        plt.show()

    cur_step += 1
```

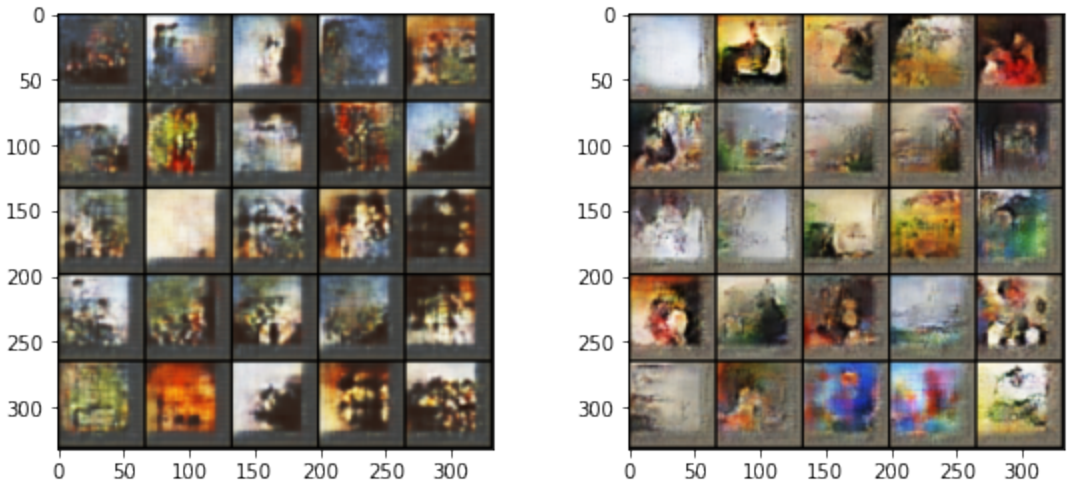


Figure 4.9: Results for 64x63 WGAN-GP: Version 3

Chapter 5

Summary

5.1 Interpretations

The results of this paper show that generative art can be produced through machine learning models, specifically generative adversarial networks. The range of results shows there is still a lot of work to do in terms of creating light architectures for generative art with the use of machine learning models. From the literature this paper extracts key findings in machine learning models and implements them intuitively into programs which produce art. The evolution of the first vanilla DCGAN to the WGAN-GP highlights the changing nature of machine learning models to combat specific problems, mode collapse in this case.

5.2 Implications

This paper helps in adding material to the newly formed world of generative adversarial networks. It also aids in understanding generative adversarial networks and bringing new interest to the subject. In terms of generative art, it solves the problem of mechanised production within it and introduces a new machine learning approach. The machine learning approach encompasses a flair of human touch by feeding the network a large dataset of artworks from a wide time period. Breaking down the linear approach to art can be beneficial for existing generative art artists. GAN are on the boundary of breaking out into the commercial world. Its achilleas is hyper tuning which can render some datasets being completely disregarded. The

WGAN-GP in this paper solves this problem and allows for a range of datasets to be used.

5.3 Limitations

The results of the GAN are subjective, however that is the case with all art. Noise capturers could have been built however they cannot be translated into meaningful graphs. This paper suggests when generative art produced through machine learning models is successful it should be compared to real life art. This way members of the audience can be asked to decide "what piece is created through a machine learning model?". However, this test is past the scope of this paper as it solely focuses on the generation of the art. Although important advancements have been made in GAN, they are still relatively new technology, so implications of the results of this paper are still yet to be further analysed. If new technologies and models become apparent the results of this paper may implicate new findings.

5.4 Recommendations

From here new architecture can be better adapted into producing art. Although specific implementations exist such as StyleGAN, StyleGAN2 and ArtGAN; they require very strong hardware to get some form of acceptable results. This paper used hardware that was readily available and provides alternatives of running the code in dedicated environments online. New methods of computer vision entangled with GAN implementations maybe able to combat the problem of the heavy compute power required to run complex GAN. As making this technology readily available will allow it to thrive and progress forward.

5.5 Conclusion

Machine learning models were created and used to produce generative art. This paper produced three programs which satisfy the main objective. The first version follows a novel vanilla Deep Convolutional Generative Adversarial Network architecture and implements it to produce interesting art with minimal noise. The second version of this program creates a high-resolution version of these images and begins to identify pitfalls in the architecture such as generator loss increasing after a certain epoch. The advantages and disadvantages of the architecture are highlighted, for example the WGAN-GP may produce a better final output but takes a large performance hit. This is show in simple experts from the program running with a single epoch taking roughly 3-4 minutes to execute. Whilst the first DCGAN took roughly 30 minutes to execute the WGAN-GP for the same number of epochs takes around 2 and a half hours. This paper takes the approach of finding new important architectures for generative art as it is something that is rapidly advancing. As generative artists breakout into the 3D plane, the world of machine learning models for generative art lacks behind. The contributions of this paper such as designing and implementing exciting new architectures for the task can bridge that gap.

References

Abadi, Martın (2016). 'TensorFlow: learning functions at scale'. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pp. 1–1 (cit. on p. 6).

Albawi, Saad, Tareq Abed Mohammed and Saad Al-Zawi (2017). 'Understanding of a convolutional neural network'. In: *2017 International Conference on Engineering and Technology (ICET)*. Ieee, pp. 1–6 (cit. on p. 3).

ArtChallenge (2021). *http://artchallenge.ru/?lang=en* (cit. on p. 23).

Beck, Kent et al. (2001). 'Manifesto for agile software development'. In: (cit. on p. 14).

Chu, Casey, Andrey Zhmoginov and Mark Sandler (2017). 'Cyclegan, a master of steganography'. In: *arXiv preprint arXiv:1712.02950* (cit. on p. 5).

Eddelbuettela, Dirk and Wush Wub (n.d.). 'RcppCNPy: Reading and writing NumPy binary files'. In: () (cit. on p. 24).

Fadaeddini, Amin, Babak Majidi and Mohammad Eshghi (2018). 'A case study of generative adversarial networks for procedural synthesis of original textures in video games'. In: *2018 2nd National and 1st International Digital Games Research Conference: Trends, Technologies, and Applications (DGRC)*. IEEE, pp. 118–122 (cit. on p. 2).

Galanter, Philip (2003). 'What is generative art? Complexity theory as a context for art theory'. In: *In GA2003–6th Generative Art Conference*. Citeseer (cit. on p. 1).

– (2016). 'Generative art theory'. In: *A Companion to Digital Art*, pp. 146–180 (cit. on p. 1).

Gao, Fei et al. (2018). 'A deep convolutional generative adversarial networks (DCGANs)-based semi-supervised method for object recognition in synthetic aperture radar (SAR) images'. In: *Remote Sensing* 10.6, p. 846 (cit. on p. 18).

Gharakhanian, Al (2021). *https://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html* (cit. on p. 11).

Github, Inc (2016). 'GitHub'. In: *URl: https://github. com/(visited on 04/23/2014)* (cit. on p. 20).

Goodfellow, Ian J et al. (2014). 'Generative adversarial networks'. In: *arXiv preprint arXiv:1406.2661* (cit. on pp. 8, 10, 29).

Gulrajani, Ishaan et al. (2017). 'Improved training of wasserstein gans'. In: *arXiv preprint arXiv:1704.00028* (cit. on p. 43).

Hansmeyer, Michael (2021). *https://aiartists.org/generative-art-design* (cit. on p. 2).

Hecht-Nielsen, Robert (1992). 'Theory of the backpropagation neural network'. In: *Neural networks for perception*. Elsevier, pp. 65–93 (cit. on p. 11).

Hunter, John D (2007). 'Matplotlib: A 2D graphics environment'. In: *IEEE Annals of the History of Computing* 9.03, pp. 90–95 (cit. on p. 22).

icaro (2019). *https://www.kaggle.com/ikarus777/best-artworks-of-all-time* (cit. on p. 23).

Kaggle (2021). *https://www.kaggle.com/* (cit. on p. 3).

Liang, Xiaodan et al. (2017). 'Dual motion gan for future-flow embedded video prediction'. In: *proceedings of the IEEE international conference on computer vision*, pp. 1744–1752 (cit. on p. 5).

Marcel, Sébastien and Yann Rodriguez (2010). 'Torchvision the machine-vision package of torch'. In: *Proceedings of the 18th ACM international conference on Multimedia*, pp. 1485–1488 (cit. on p. 24).

Maylor, Harvey (2001). 'Beyond the Gantt chart:: Project management moving on'. In: *European management journal* 19.1, pp. 92–100 (cit. on p. 16).

McCormac, Jon (2021). *https://aiartists.org/generative-art-design* (cit. on p. 2).

Mitchell, Tom Michael (2006). *The discipline of machine learning*. Vol. 9. Carnegie Mellon University, School of Computer Science, Machine Learning ... (cit. on p. 1).

Paszke, Adam et al. (2019). 'Pytorch: An imperative style, high-performance deep learning library'. In: *arXiv preprint arXiv:1912.01703* (cit. on p. 6).

Pérez-Garcıa, Fernando, Rachel Sparks and Sebastien Ourselin (2020). 'TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning'. In: *arXiv preprint arXiv:2003.04696* (cit. on p. 7).

Pisuwala, Ubaid (2021). *https://www.peerbits.com/blog/agile-software-development.html* (cit. on p. 15).

Sanders, Jason and Edward Kandrot (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional (cit. on p. 6).

Schwaber, Ken and Mike Beedle (2002). *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River (cit. on p. 15).

Shi, Chenglong et al. (2020). 'CAN-GAN: Conditioned-attention normalized GAN for face age synthesis'. In: *Pattern Recognition Letters* 138, pp. 520–526 (cit. on p. 19).

Tan, Wei Ren et al. (2017). 'ArtGAN: Artwork synthesis with conditional categorical GANs'. In: *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE, pp. 3760–3764 (cit. on pp. 12, 18).

TensorBoard (2021). *https://www.tensorflow.org/tensorboard/* (cit. on p. 9).

Wang, Weijie and Yanmin Lu (2018). 'Analysis of the mean absolute error (MAE) and the root mean square error (RMSE) in assessing rounding model'. In: *IOP conference series: materials science and engineering*. Vol. 324. 1. IOP Publishing, p. 012049 (cit. on p. 4).

Zhang, Zhixin et al. (2020). 'High-quality face image generation based on generative adversarial networks'. In: *Journal of Visual Communication and Image Representation* 71, p. 102719 (cit. on p. 12).

Appendix A
Appendix

A.1 Project Definition Document

Project Definition Document

BSc (Hons) Computer Science

Generative Art: A new approach with Generative Adversarial Networks

No proprietary arrangements required.

GAN Art

Supervisor:
Dr Ernesto Jimenez-Ruiz

Proposed By:
Said Moredi
+44(0)7783030344
said.moredi@city.ac.uk

Said Moredi

+44(0)7783030344

Word Count : 1437

Contents

1 Proposal

1.1 Brief introduction to Generative Adversarial Networks

Generative Adversarial Networks (GAN) is an unsupervised machine learning model. It encompasses two machine learning models against each other in a min and max game, which are the generator and discriminator. The generator from a series of random noise values tries to create a fake image which the the discriminator might pass as a real image. The discriminators job is to classify real images from the training set as real or fake images produced from the generator. Through the loss functions of the generator and discriminator, GAN carries out back propagation to updates the weights on the generator and discriminator. Overtime the generator output will pass a threshold where the discriminator cannot distinguish the output as fake.

1.2 Problems to be solved

This project takes a deep dive into generative art with the use of Generative Adversarial Networks (GAN). As this topic is relatively new with first implementations of GAN's from 2014 [Goodfellow et al., 2014] which outlines a new world of neural networks yet to be explored. I was inspired by the work of Mario Klingemann (Figure 1 shows his work)

Figure 1: Work of Mario Klingemann

Figure 2: User Submission on ArtBreeder

where he uses multiple convolutional neural networks to produce new portraits from a deep nested database of artworks. Another source of inspiration is the websites ArtBreeder [ArtBreeder, 2021b] where users are given the choice of choosing images that are strung through a GAN and produce new outcomes. My implementation will focus on a GAN which takes an extensive amount of images and produces a single piece of art.

2 Project Objectives

2.1 Main Objective

The main objective of GAN Art will be to exhibit a piece of art from a relatively vast range of artworks that it was fed. The dataset of paintings and various forms of art will be attained from Kaggle. [Kaggle, 2021] From the dataset I will extrapolate significant data that has created the unique output. For example, from an output of a single piece of art, the application should be able to tell what necessary the images were used in the production of the result.

2.2 Sub-Objectives

1. The application shall create unique pieces of art on each iteration.

   Test: Run through a script which compares the pixels of the previous output.

2. The application shall be able to differentiate between the different styles of art within the dataset.

   Test: Check setup where the original label of the image is compared to a the final output.

3. The application shall be able to differentiate the art period of each image.

   Test: A print function which highlights the corresponding art year that produced the final image.

4. The application shall produce percentages of art styles used that lead to its output.

   Test: Early stopping within the application can ensure the figures produced are correct.

5. The application should read in images from the provided dataset and use it to construct its output

   Test: Tracking of the used images in the console can prove this.

6. The application shall convert data from images into machine readable instructions to manipulate and produce new results

   Test: Pre-tests of converting raw data from images into numbers can show this action

7. The application shall format the given dataset of images into a uniform format and size

   Test: Printing the size of the images can confirm this with a boolean function

8. The application shall control the level of influence a particular artist has had over a certain output

   Test: Standardising the number of artworks for all artists can affect the final output of the image, whilst allowing more influence from other artists can be displayed via increasing their artworks over the GAN.

9. The GAN will use two different methods (neural networks) for the discriminator and generator.

   Test: Running seperate tests on the discriminator and generator before adding them into the main program.

3 Research Questions

1. Will the GAN outputs retain information of the original input?
2. Does the GAN Art application run efficiently compared to its input size?
3. Has the GAN produced output which is too similar to the input?
4. Can we produce plots of the loss for the model?

4 Beneficiaries

I believe GAN Art will produce a greater tool-belt for artists and designers alike to look at GAN as a new form medium to gain inspiration from. The complex nature of the aspect can be fine tuned to produce art which is tailored to the user. Many commercial uses arise from this technology as the abstract application can produce unique outputs that can be used in advertisements, prints or art installations. Developers that are looking for an insight into GAN will find this project useful. Researchers and any individual interested with the use of neural networks in art will also find this useful.

5 Work Plan

I will be using the Agile methodology for software development through out this project. An outline of my personal deadlines and the time frame given will be added at the end. Through out the project I will document every test I have carried out for my sub-objectives that I have laid out previously. The procedure and results of each test will be documented in my report. With the grant chart I can breakdown my sub-objectives further allowing me to make incremental development throughout the project. I will be using Github for VCS, this way I have every version of the program and the progress can be monitored. Through the use of Git I can evaluate the speed of my development and adjust to the deadlines I have set out for myself.

5.1 Version 1

1. **Process Database:** The database will be loaded in and run through a few tests. First printing out the relevant information about the artists involved for example the style of art they are notable for. Then the images will be processed to the correct size, allowing each image to have the same resolution - making it easier further down the line for the GAN.

2. **Implementation:** To implement my design I will be using the Spyder IDE. I will also be using TesnorFlow2 and Keras libraries. These will be the main libraries that I will use for my GAN, however many of the other libraries I will outline in the documentation.

3. **Testing and Review:** Here I will breakdown each component for my application and put it through a series of tests. Each test will be documented by what the test was and the results from it. This way I can be sure to continue development without any faults in the foundation of the program. The review section will allow me to add further comments in to the documentation; making it easier for readers to understand the code.

5.2 Version 2

1. **Review Changes from Previous Version:** I will be using Github for VCS, this way I can see the necessary changes in the review section. The code will be analysed again and the changes can be reverted or kept, adhering to the review process. I believe having a second review phase will minimise the amount of faults that will arise for future versions.

2. **Implementation of Review:** The changes outlined in the second review will be finalised by adding the specific changes to the main application. A simple table with the changes and the status will be outlined allowing me to keep track of changes.

3. **Testing and Bug Report:** All the changes made will be sent through a series of new tests in-conjunction with the first version of the program. The bug report will note down any new abnormalities in the program - a chart of this will be produced.

4. **Comparison and Analysis of Previous Versions:** The changes in this version will be analysed for efficiency and speed of computation then compared to the initial version. This section will allow make it easier for me to refactor code in the next version leading to a fast and efficient program.

5.3 Version 3

1. **Outline Additional Features:** Once a working version of the program is stable and running, I will start to add additional features. This can only be done once the application is running efficiently with no bugs. Additional features could be: user interface, adding an additional GAN, producing many images from a single run and labelling the final image with percentages of influence.

2. **Implement Additional Features:** The new features will be added incrementally allowing me to maintain the original code as much as possible - this leads to less errors when adding extra components to the program

3. **Final Code Test and Refactor:** The final strenuous tests will allow me to cover all possibilities of failure of my project. The refactoring of code will allow it to be concise and efficient. The documentation for the code will be produced and reviewed here, adding final changes if they require it.

Figure 3 shows this in a Grant Chart

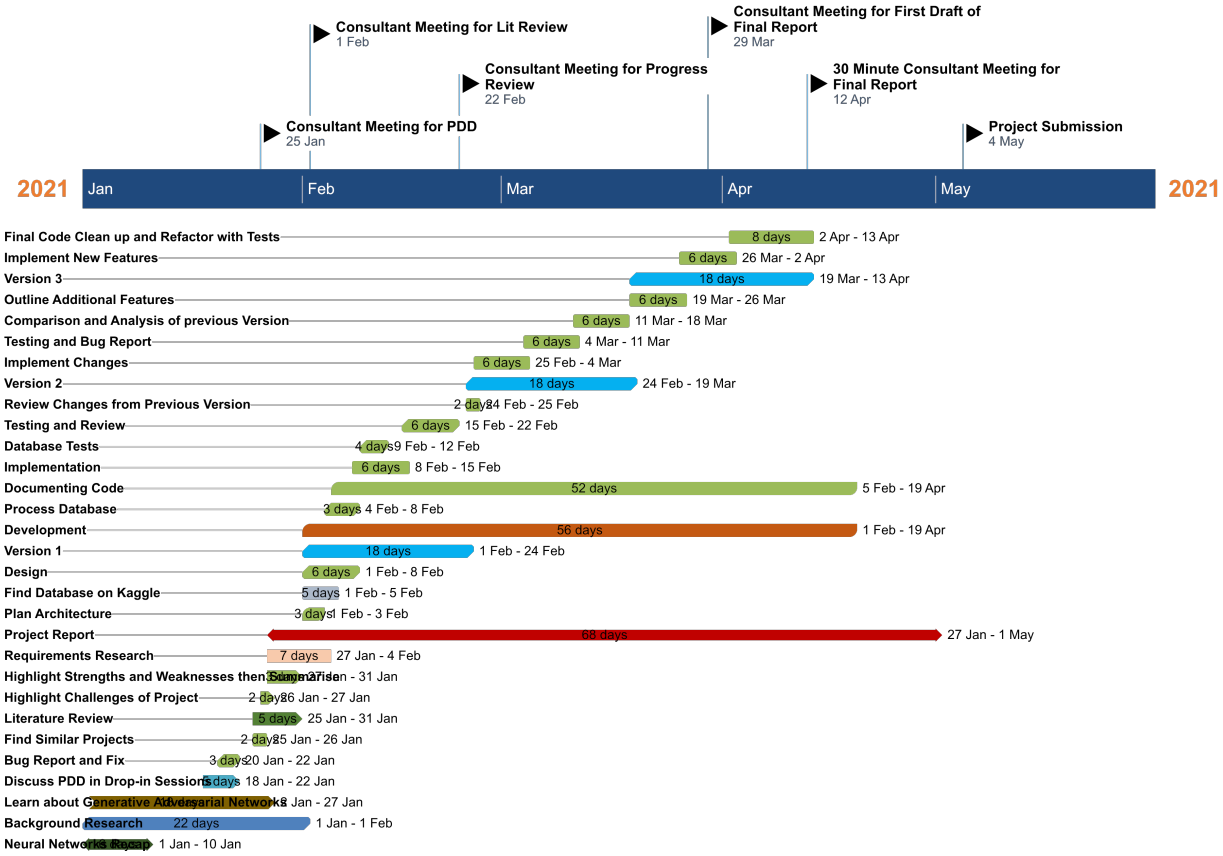


Figure 3: Work plan: Read from the bottom up

6 Risks & Ethics Checklist

The table below highlights the risks and mitigations of the project. 1

ID	RISK	IMPACT 1 - Low 5 - High	Likelihood 1 - Low 5 - High	Severity = Impact x Likelihood	MITIGATION
01	The images of the database are corrupted.	2	1	2	There are plenty of free and open source online databases that I can use.
02	The Spyder IDE stops working.	2	1	2	The project can be moved to Jupyter Notebooks.
03	Hardware Failure (eg. GPU breaks)	1	2	2	Cloud computing sources and free versions such as Jupyter Notebooks.
05	Program does not produce unique art	5	1	5	Review of the code and bug reports.
06	Program cannot differentiate between different styles of art	3	2	6	Processing the data with multiple tests.
07	Program cannot differentiate between art periods of the image	3	2	6	Process data before development.
08	Program does not show percentages of influence	1	4	4	Keeping track of the changes in the Neural Network can stop this.
09	Program does not read in data from the data-set	1	1	1	Manipulating the data can stop this from occurring.

Table 1: Severity 1- 8 = Low, 9 - 16 = Medium, 17 - 25 = High

References

[ArtBreeder, 2021a] ArtBreeder (2021a). Artbreeder website. https://www.artbreeder.com/i?k=967b574a5615674adc9ef0fd.

[ArtBreeder, 2021b] ArtBreeder (2021b). Artbreeder website. https://www.artbreeder.com/.

[Goodfellow et al., 2014] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks. *arXiv preprint arXiv:1406.2661*.

[Kaggle, 2021] Kaggle (2021). Kaggle website. https://www.kaggle.com/.

[Klingemann, 2021] Klingemann, M. (2021). Medium website. https://miro.medium.com/max/1200/0*UI6CmdkEWNLHEAsg.png.

Research Ethics Review Form: BSc, MSc and MA Projects

Computer Science Research Ethics Committee (CSREC)

http://www.city.ac.uk/department-computer-science/research-ethics

Undergraduate and postgraduate students undertaking their final project in the Department of Computer Science are required to consider the ethics of their project work and to ensure that it complies with research ethics guidelines. In some cases, a project will need approval from an ethics committee before it can proceed. Usually, but not always, this will be because the student is involving other people ("participants") in the project.

In order to ensure that appropriate consideration is given to ethical issues, all students must complete this form and attach it to their project proposal document. There are two parts:

PART A: Ethics Checklist. All students must complete this part.
The checklist identifies whether the project requires ethical approval and, if so, where to apply for approval.

PART B: Ethics Proportionate Review Form. Students who have answered "no" to all questions in A1, A2 and A3 and "yes" to question 4 in A4 in the ethics checklist must complete this part. The project supervisor has delegated authority to provide approval in such cases that are considered to involve MINIMAL risk. The approval may be ***provisional*** – *identifying the planned research as* likely to involve MINIMAL RISK. In such cases you must additionally seek ***full approval*** from the supervisor as the project progresses and details are established. ***Full approval*** must be acquired in writing, before beginning the planned research.

A.1 If you answer YES to any of the questions in this block, you must apply to an appropriate external ethics committee for approval and log this approval as an External Application through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
1.1	Does your research require approval from the National Research Ethics Service (NRES)? *e.g. because you are recruiting current NHS patients or staff?* *If you are unsure try - https://www.hra.nhs.uk/approvals-amendments/what-approvals-do-i-need/*	**NO**
1.2	Will you recruit participants who fall under the auspices of the Mental Capacity Act? *Such research needs to be approved by an external ethics committee such as NRES or the Social Care Research Ethics Committee - http://www.scie.org.uk/research/ethics-committee/*	**NO**
1.3	Will you recruit any participants who are currently under the auspices of the Criminal Justice System, for example, but not limited to, people on remand, prisoners and those on probation? *Such research needs to be authorised by the ethics approval system of the National Offender Management Service.*	**NO**
A.2 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee, you must apply for approval from the Senate Research Ethics Committee (SREC) through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
2.1	Does your research involve participants who are unable to give informed consent? *For example, but not limited to, people who may have a degree of learning disability or mental health problem, that means they are unable to make an informed decision on their own behalf.*	**NO**
2.2	Is there a risk that your research might lead to disclosures from participants concerning their involvement in illegal activities?	**NO**
2.3	Is there a risk that obscene and or illegal material may need to be accessed for your research study (including online content and other material)?	**NO**

2.4	Does your project involve participants disclosing information about special category or sensitive subjects? *For example, but not limited to: racial or ethnic origin; political opinions; religious beliefs; trade union membership; physical or mental health; sexual life; criminal offences and proceedings*	**NO**
2.5	Does your research involve you travelling to another country outside of the UK, where the Foreign & Commonwealth Office has issued a travel warning that affects the area in which you will study? *Please check the latest guidance from the FCO - http://www.fco.gov.uk/en/*	**NO**
2.6	Does your research involve invasive or intrusive procedures? *These may include, but are not limited to, electrical stimulation, heat, cold or bruising.*	**NO**
2.7	Does your research involve animals?	**NO**
2.8	Does your research involve the administration of drugs, placebos or other substances to study participants?	**NO**
A.3 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee or the SREC, you must apply for approval from the Computer Science Research Ethics Committee (CSREC) through Research Ethics Online - https://ethics.city.ac.uk/ **Depending on the level of risk associated with your application, it may be referred to the Senate Research Ethics Committee.**		*Delete as appropriate*
3.1	Does your research involve participants who are under the age of 18?	**NO**
3.2	Does your research involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)? *This includes adults with cognitive and / or learning disabilities, adults with physical disabilities and older people.*	**NO**
3.3	Are participants recruited because they are staff or students of City, University of London? *For example, students studying on a particular course or module.* *If yes, then approval is also required from the Head of Department or Programme Director.*	**NO**
3.4	Does your research involve intentional deception of participants?	**NO**
3.5	Does your research involve participants taking part without their informed consent?	**NO**
3.5	Is the risk posed to participants greater than that in normal working life?	**NO**
3.7	Is the risk posed to you, the researcher(s), greater than that in normal working life?	**NO**
A.4 If you answer YES to the following question and your answers to all other questions in sections A1, A2 and A3 are NO, then your project is deemed to be of MINIMAL RISK. **If this is the case, then you can apply for approval through your supervisor under PROPORTIONATE REVIEW. You do so by completing PART B of this form.** **If you have answered NO to all questions on this form, then your project does not require ethical approval. You should submit and retain this form as evidence of this.**		*Delete as appropriate*
4	Does your project involve human participants or their identifiable personal data? *For example, as interviewees, respondents to a survey or participants in testing.*	**NO**

PART B: Ethics Proportionate Review Form (THIS PART DOES NOT APPLY TO ME)

If you answered YES to question 4 and NO to all other questions in sections A1, A2 and A3 in PART A of this form, then you may use PART B of this form to submit an application for a proportionate ethics review of your project. Your project supervisor has delegated authority to review and approve this application under proportionate review. You must receive final approval from your supervisor in writing before beginning the planned research.

However, if you cannot provide all the required attachments (see B.3) with your project proposal (e.g. because you have not yet written the consent forms, interview schedules etc), the approval from your supervisor will be ***provisional***. You **must** submit the missing items to your supervisor for approval prior to commencing these parts of your project. Once again, you must receive written confirmation from your supervisor that any provisional approval has been superseded by with ***full approval*** of the planned activity as detailed in the full documents. **Failure to follow this procedure and demonstrate that final approval has been achieved may result in you failing the project module.**

Your supervisor may ask you to submit a full ethics application through Research Ethics Online, for instance if they are unable to approve your application, if the level of risks associated with your project change, or if you need an approval letter from the CSREC for an external organisation.

B.1 The following questions must be answered fully. All grey instructions must be removed.		*Delete as appropriate*
1.1.	Will you ensure that participants taking part in your project are fully informed about the purpose of the research?	**YES / NO**
1.2	Will you ensure that participants taking part in your project are fully informed about the procedures affecting them or affecting any information collected about them, including information about how the data will be used, to whom it will be disclosed, and how long it will be kept?	**YES / NO**
1.3	When people agree to participate in your project, will it be made clear to them that they may withdraw (i.e. not participate) at any time without any penalty?	**YES / NO**
1.4	Will consent be obtained from the participants in your project? Consent from participants will be necessary if you plan to involve them in your project or if you plan to use identifiable personal data from existing records. "Identifiable personal data" means data relating to a living person who might be identifiable if the record includes their name, username, student id, DNA, fingerprint, address, etc. *If YES, you must attach drafts of the participant information sheet(s) and consent form(s) that you will use in section B.3 or, in the case of an existing dataset, provide details of how consent has been obtained.* *You must also retain the completed forms for subsequent inspection. Failure to provide the completed consent request forms will result in withdrawal of any earlier ethical approval of your project.*	**YES / NO**
1.5	Have you made arrangements to ensure that material and/or private information obtained from or about the participating individuals will remain confidential?	**YES / NO**

B.2 If the answer to the following question (B2) is YES, you must provide details		*Delete as appropriate*
2	Will the research be conducted in the participant's home or other non-University location? *If **YES**, you must provide details of how your safety will be ensured.*	**YES / NO**

B.3 Attachments **ALL of the following documents MUST be provided to supervisors if applicable.** **All must be considered prior to final approval by supervisors.** **A written record of final approval must be provided and retained.**	***YES***	***NO***	***Not Applicable***
Details on how safety will be assured in any non-University location, including risk assessment if required (see B2)			**X**
Details of arrangements to ensure that material and/or private information obtained from or about the participating individuals will remain confidential (see B1.5) *Any personal data must be acquired, stored and made accessible in ways that are GDPR compliant.*			**X**
Full protocol for any workshops or interviews**			**X**
Participant information sheet(s)**			**X**
Consent form(s)**			**X**
Questionnaire(s)** *sharing a Qualtrics survey with your supervisor is recommended.*			**X**
Topic guide(s) for interviews and focus groups**			**X**
Permission from external organisations or Head of Department** *e.g. for recruitment of participants*			**X**

approval *can still be given, under the condition that you must submit the final versions of all items to your supervisor for approval at a later date.* ***All*** *such items* ***must*** *be seen and approved by your supervisor before the activity for which they are needed begins. Written evidence of* ***final approval*** *of your planned activity must be acquired from your supervisor before you commence.*

Changes

If your plans change and any aspects of your research that are documented in the approval process change as a consequence, then any approval acquired is invalid. If issues addressed in Part A (the checklist) are affected, then you must complete the approval process again and establish the kind of approval that is required. If issues addressed in Part B are affected, then you must forward updated documentation to your supervisor and have received written confirmation of approval of the revised activity before proceeding.

Templates for Consent and Information

You must use the templates provided by the University as the basis for your participant information sheets and consent forms. You **must** adapt them according to the needs of your project before you submit them for consideration.

Participant Information Sheets, Consent Forms and Protocols must be consistent. Please ensure that this is the case prior to seeking approval. Failure to do so will slow down the approval process.

We strongly recommend using Qualtrics to produce digital information sheets and consent forms.

Further Information

http://www.city.ac.uk/department-computer-science/research-ethics
https://www.city.ac.uk/research/ethics/how-to-apply/participant-recruitment
https://www.city.ac.uk/research/ethics

A.2 Source Code

A.2.1 Version 1

```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display

import os

# Keras functions
import tensorflow as tf
from tqdm import tqdm
from keras.optimizers import Adam
from keras.models import Sequential, Model, load_model
from keras.layers import Dense, Conv2D, Conv2DTranspose, Reshape, Flatten
from keras.layers import Dropout, LeakyReLU, BatchNormalization
from keras.layers import Activation, ZeroPadding2D, UpSampling2D
from keras.layers import Input, Reshape
from matplotlib import pyplot
from IPython.display import clear_output

# Numpy functions
import numpy
import numpy as np
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy.random import randn
from numpy.random import randint
from numpy import zeros
from numpy import ones
from numpy import asarray

#Torchvision for fast and easy loading and resizing
import torchvision
import torchvision.transforms as transforms

from PIL import Image
```

```
WIDTH = 128
HEIGHT = 128
IMG_SIZE = (WIDTH,HEIGHT)
EPOCHS = 150

#for efficient data loading and manipulation, the images are save as
    binary file
#to reproduce the file the code is below
#but if the saved file is already present the function will load it and
    not
#create it from scratch

dataset_path = "../input/best-artworks-of-all-time/resized"
saved_binary_file = "./training_data.npy"


def prepare_data(path_of_file , path_of_data):
    #Look for saved file to save time loading and processing images
        between runs
    print("Looking for saved binary file...")

    if not os.path.isfile(path_of_file):
        print("\n File not found, creating new file...\n")
        dataset = []
        transform_ds = transforms.Compose([transforms.Resize(IMG_SIZE),])
            #define transformation

        image_folder = torchvision.datasets.ImageFolder(root=path_of_data,
                                    transform=transform_ds)

        print('Number of artworks found: ',len(image_folder))


        print("Converting images, this will take a few minutes")
        for i in range (len(image_folder)):
            image_array = numpy.array(image_folder[i][0])
            dataset.append(image_array)
            if (i%500 == 0):
                print("Pictures processed: ", i)

        print("Saving dataset binary file...")
        dataset = np.array(dataset, dtype=np.float32)
        dataset = (dataset - 127.5) / 127.5 #Normalize to [-1 , 1]
        numpy.save(path_of_file, dataset) #Save processed images as npy
            file

    else:
        print("Data found, loading..")
        dataset = np.load(path_of_file)
```

```
    print("Dataset length: ", len(dataset))

    return dataset

dataset = prepare_data(saved_binary_file , dataset_path)

#Using a TensorFlow Dataset to manage the images for easy shuffling,
    dividing etc
BATCH_SIZE = 128

training_dataset =
    tf.data.Dataset.from_tensor_slices(dataset).shuffle(60000).batch(BATCH_SIZE)

import matplotlib.pyplot as plt
import random

def visualize_random_art(dataset):
    """
    function to plot some random images from the data set

    """
    np.random.shuffle(dataset) #Shuffle the images

    fig = plt.figure(figsize=(12,12))
    for i in range(1,37):
        fig.add_subplot(6,6,i)
        plt.imshow(dataset[i])
        plt.axis('off')

visualize_random_art(dataset)

def make_generator_model(seed_size, channels):
    """
    This function builds the generator for DCGAN
    Parameters:
        seed_size:according to authors of DCGAN , Generator takes this
            random seed and generates an image
        channels : output channels that image will have
    """
    model = tf.keras.Sequential()

    model.add(Dense(4*4*512,activation="relu",input_dim=seed_size)) #64x64
        units
    model.add(Reshape((4,4,512)))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))

    model.add(Conv2DTranspose(256, (5, 5), strides=(2, 2), padding='same',
        use_bias=False))
    model.add(BatchNormalization())
```

```
    model.add(LeakyReLU(alpha = 0.2))
    model.add(Dropout(0.4))

    model.add(Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same',
        use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha =0.3))
    model.add(Dropout(0.4))

    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
        use_bias=False ))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))

    model.add(Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same',
        use_bias=False ))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))

    model.add(Conv2DTranspose(16, (5, 5), strides=(2, 2), padding='same',
        use_bias=False ))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))
    model.add(Conv2DTranspose(channels, (5, 5), strides=(1, 1),
        padding='same', use_bias=True, activation='tanh'))
 #   assert model.output_shape == (None, 64, 64, 3) , f"output shape
     mismatch {model.output_shape}"

    return model

SEED_SIZE = 100
IMAGE_CHANNELS = 3

generator = make_generator_model(SEED_SIZE ,IMAGE_CHANNELS)

def make_discriminator_model(image_shape):
    """
    This function builds a discriminator which is a CNN based image
        classifier
    and will output probability values of what it thinks is fake or real
        with values close to
    0 being fake and 1 being real.

    Parameters :
        image_shape : input_image shape (h x w x c)
    """
    model = tf.keras.Sequential()
    model.add(Conv2D(64, kernel_size=5, strides=2, input_shape=(128, 128,
        3), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
```

```
    model.add(Conv2D(128, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(256, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(512, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Flatten())
    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    return model

SEED_SIZE = 100
IMAGE_CHANNELS = 3

generator = make_generator_model(SEED_SIZE ,IMAGE_CHANNELS)
discriminator = make_discriminator_model([128,128,3])

noise = tf.random.normal([1,SEED_SIZE])

generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0])

image_shape = (HEIGHT, HEIGHT, IMAGE_CHANNELS)

discriminator = make_discriminator_model(image_shape)
print(discriminator(generated_image))

cross_entropy = tf.keras.losses.BinaryCrossentropy()

def discriminator_loss(real_output, fake_output):
    """
    The discriminators loss is based on its ability to distinguish real
        images from fakes.
    It compares its predictions on real images to an array of ones
        (remember 1 being real)
    and its predictions on fake images to an array of zeros (0 being fake).
    The goal is to classify all real images as 1 and all fakes as 0.
    The total loss is then these two losses added together.

    Parameters :
        real_output : real image from the dataset
```

```
        fake_output : fake image from the dataset
    """
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    """
    The generators loss is a measurement of how good it performed at
        fooling the discriminator.
    If the discriminator classifies the fake images as 1, the generator
        did a good job.

    Parameters :
     fake_output : fake image from generator
    """
    return cross_entropy(tf.ones_like(fake_output), fake_output)

#The two models optimizers are separated because we train them separately.
# I found a slightly lower generator LR to be beneficial.
#Beta value of 0.5 generates more stable models as per the findings in
    the paper
#"Unsupervised representation learning with deep convolutional generative
    adversarial networks"

generator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)

!mkdir ./training_checkpoints

#checkpoint for saving a model

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

def generate_images(generated_images2):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
    generated_images2 =0.5 * generated_images2 + 0.5

    fig = plt.figure(figsize=(10,10))
    for i in range(1,21):
        fig.add_subplot(5,5,i)
        plt.imshow(generated_images2[i])
        plt.axis('off')
```

```
    plt.show()

EPOCHS = 150
noise_dim = 100
num_examples_to_generate = 16

# You will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    """
    The training begins by providing a random seed to the generator, which
        is then used to generate
    an image. The discriminator then classifies images from both the fake
        and real dataset.
    The loss is calculated separately for each model and the gradients are
        updated.

    Parameters :
    images : images to be trained on
    """
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
        generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
        discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
        generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
        discriminator.trainable_variables))

    return gen_loss , disc_loss ,generated_images

import matplotlib.pyplot as plt
```

```
generator_losses = []
discriminator_losses = []

def train(dataset, epochs):
  for epoch in range(epochs):
    gen_loss_list = []
    disc_loss_list = []
    start = time.time()

    for image_batch in tqdm(dataset):

        t =  train_step(image_batch)
        gen_loss_list.append(t[0])
        disc_loss_list.append(t[1])

    # Produce images for the GIF as you go
    # display.clear_output(wait=True)
    g_loss = sum(gen_loss_list) / len(gen_loss_list) #calculate losses
    d_loss = sum(disc_loss_list) / len(disc_loss_list)
    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)
      generate_images(t[2])
    print ('Time for epoch {} is {} sec'.format(epoch + 1,
       time.time()-start))

    print(f'Epoch {epoch+1}, gen loss = {g_loss}, disc loss = {d_loss}')
    generator_losses.append(g_loss)
    discriminator_losses.append(d_loss)


  # Generate after the final epoch
  display.clear_output(wait=True)

train(training_dataset, EPOCHS) #run only for training

#code for visualizing losses

fig = plt.figure(figsize=(10,10))
plt.plot(
               range(EPOCHS) ,
               generator_losses,
               label="Generator Loss"
           )
plt.plot(
               range(EPOCHS) ,
               discriminator_losses,
               label="Discriminator Loss"
           )
```

```
plt.legend()
plt.savefig("loss_figure.png")
plt.show()
```

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

```
generated_images2 = generator(seed2, training = False)
generated_images2 =0.5 * generated_images2 + 0.5
fig = plt.figure(figsize=(10,10))

plt.imshow(generated_images2[0])
plt.axis('off')
plt.savefig("image_generated.png")
plt.show()
```

```
generator.save("./trained_generator.h5")
print("Saved model to disk")

checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
import keras
seed2 = tf.random.normal([1, 100])
generator = keras.models.load_model("./trained_generator.h5")

seed = tf.random.normal([22 , noise_dim])

generated_images2 = generator(seed, training = False)
generate_images(generated_images2)
```

A.2.2 Version 2

```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
```

```
import os

# Keras functions
import tensorflow as tf
from tqdm import tqdm
from keras.optimizers import Adam
from keras.models import Sequential, Model, load_model
from keras.layers import Dense, Conv2D, Conv2DTranspose, Reshape, Flatten
from keras.layers import Dropout, LeakyReLU, BatchNormalization
from keras.layers import Activation, ZeroPadding2D, UpSampling2D
from keras.layers import Input, Reshape
from matplotlib import pyplot
from IPython.display import clear_output

# Numpy functions
import numpy
import numpy as np
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy.random import randn
from numpy.random import randint
from numpy import zeros
from numpy import ones
from numpy import asarray

#Torchvision for fast and easy loading and resizing
import torchvision
import torchvision.transforms as transforms

from PIL import Image

WIDTH = 128
HEIGHT = 128
IMG_SIZE = (WIDTH,HEIGHT)
EPOCHS = 150

#for efficient data loading and manipulation, the images are save as
    binary file
#to reproduce the file the code is below
#but if the saved file is already present the function will load it and
    not
#create it from scratch

dataset_path = "../input/best-artworks-of-all-time/resized"
saved_binary_file = "./training_data.npy"

def prepare_data(path_of_file , path_of_data):
```

```
    #Look for saved file to save time loading and processing images
        between runs
    print("Looking for saved binary file...")

    if not os.path.isfile(path_of_file):
        print("\n File not found, creating new file...\n")
        dataset = []
        transform_ds = transforms.Compose([transforms.Resize(IMG_SIZE),])
            #define transformation

        image_folder = torchvision.datasets.ImageFolder(root=path_of_data,
                                    transform=transform_ds)

        print('Number of artworks found: ',len(image_folder))


        print("Converting images, this will take a few minutes")
        for i in range (len(image_folder)):
            image_array = numpy.array(image_folder[i][0])
            dataset.append(image_array)
            if (i%500 == 0):
                print("Pictures processed: ", i)

        print("Saving dataset binary file...")
        dataset = np.array(dataset, dtype=np.float32)
        dataset = (dataset - 127.5) / 127.5 #Normalize to [-1 , 1]
        numpy.save(path_of_file, dataset) #Save processed images as npy
            file

    else:
        print("Data found, loading..")
        dataset = np.load(path_of_file)

    print("Dataset length: ", len(dataset))

    return dataset

dataset = prepare_data(saved_binary_file , dataset_path)

#Using a TensorFlow Dataset to manage the images for easy shuffling,
    dividing etc
BATCH_SIZE = 128

training_dataset =
    tf.data.Dataset.from_tensor_slices(dataset).shuffle(60000).batch(BATCH_SIZE)


import matplotlib.pyplot as plt
import random
```

```
def visualize_random_art(dataset):
    """
    function to plot some random images from the data set

    """
    np.random.shuffle(dataset) #Shuffle the images

    fig = plt.figure(figsize=(12,12))
    for i in range(1,37):
        fig.add_subplot(6,6,i)
        plt.imshow(dataset[i])
        plt.axis('off')

visualize_random_art(dataset)

def make_generator_model(seed_size, channels):
    """
    This function builds the generator for DCGAN
    Parameters:
        seed_size:according to authors of DCGAN , Generator takes this
            random seed and generates an image
        channels : output channels that image will have
    """
    model = tf.keras.Sequential()

    model.add(Dense(4*4*512,activation="relu",input_dim=seed_size)) #64x64
        units
    model.add(Reshape((4,4,512)))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))

    model.add(Conv2DTranspose(256, (5, 5), strides=(2, 2), padding='same',
        use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))
    model.add(Dropout(0.4))

    model.add(Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same',
        use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha =0.3))
    model.add(Dropout(0.4))

    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
        use_bias=False ))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))

    model.add(Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same',
        use_bias=False ))
```

```
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))

    model.add(Conv2DTranspose(16, (5, 5), strides=(2, 2), padding='same',
        use_bias=False ))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha = 0.2))
    model.add(Conv2DTranspose(channels, (5, 5), strides=(1, 1),
        padding='same', use_bias=True, activation='tanh'))
 #   assert model.output_shape == (None, 64, 64, 3) , f"output shape
     mismatch {model.output_shape}"

    return model

SEED_SIZE = 100
IMAGE_CHANNELS = 3

generator = make_generator_model(SEED_SIZE ,IMAGE_CHANNELS)

def make_discriminator_model(image_shape):
    """
    This function builds a discriminator which is a CNN based image
        classifier
    and will output probability values of what it thinks is fake or real
        with values close to
    0 being fake and 1 being real.

    Parameters :
        image_shape : input_image shape (h x w x c)
    """
    model = tf.keras.Sequential()
    model.add(Conv2D(64, kernel_size=5, strides=2, input_shape=(128, 128,
        3), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(128, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(256, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(512, kernel_size=5, strides=2, padding='same'))
    model.add(BatchNormalization(momentum=0.9))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Flatten())
    model.add(Dense(1))
    model.add(Activation('sigmoid'))
```

```
    return model

SEED_SIZE = 100
IMAGE_CHANNELS = 3

generator = make_generator_model(SEED_SIZE ,IMAGE_CHANNELS)
discriminator = make_discriminator_model([128,128,3])

noise = tf.random.normal([1,SEED_SIZE])

generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0])

image_shape = (HEIGHT, HEIGHT, IMAGE_CHANNELS)

discriminator = make_discriminator_model(image_shape)
print(discriminator(generated_image))


cross_entropy = tf.keras.losses.BinaryCrossentropy()

def discriminator_loss(real_output, fake_output):
    """
    The discriminators loss is based on its ability to distinguish real
        images from fakes.
    It compares its predictions on real images to an array of ones
        (remember 1 being real)
    and its predictions on fake images to an array of zeros (0 being fake).
    The goal is to classify all real images as 1 and all fakes as 0.
    The total loss is then these two losses added together.

    Parameters :
        real_output : real image from the dataset
        fake_output : fake image from the dataset
    """
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    """
    The generators loss is a measurement of how good it performed at
        fooling the discriminator.
    If the discriminator classifies the fake images as 1, the generator
        did a good job.

    Parameters :
```

```
    fake_output : fake image from generator
    """
    return cross_entropy(tf.ones_like(fake_output), fake_output)


#The two models optimizers are separated because we train them separately.
# I found a slightly lower generator LR to be beneficial.
#Beta value of 0.5 generates more stable models as per the findings in
    the paper
#"Unsupervised representation learning with deep convolutional generative
    adversarial networks"

generator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4,0.5)

!mkdir ./training_checkpoints

#checkpoint for saving a model

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)


def generate_images(generated_images2):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
    generated_images2 =0.5 * generated_images2 + 0.5

    fig = plt.figure(figsize=(10,10))
    for i in range(1,21):
        fig.add_subplot(5,5,i)
        plt.imshow(generated_images2[i])
        plt.axis('off')

    plt.show()


EPOCHS = 150
noise_dim = 100
num_examples_to_generate = 16

# reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    """
    The training begins by providing a random seed to the generator, which
        is then used to generate
    an image. The discriminator then classifies images from both the fake
        and real dataset.
    The loss is calculated separately for each model and the gradients are
        updated.

    Parameters :
    images : images to be trained on
    """
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
        generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
        discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
        generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
        discriminator.trainable_variables))

    return gen_loss , disc_loss ,generated_images


import matplotlib.pyplot as plt

generator_losses = []
discriminator_losses = []

def train(dataset, epochs):
  for epoch in range(epochs):
    gen_loss_list = []
    disc_loss_list = []
    start = time.time()
```

```
    for image_batch in tqdm(dataset):

        t =  train_step(image_batch)
        gen_loss_list.append(t[0])
        disc_loss_list.append(t[1])

    # Produce images for the GIF as you go
    # display.clear_output(wait=True)
    g_loss = sum(gen_loss_list) / len(gen_loss_list) #calculate losses
    d_loss = sum(disc_loss_list) / len(disc_loss_list)
    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)
      generate_images(t[2])
    print ('Time for epoch {} is {} sec'.format(epoch + 1,
       time.time()-start))

    print(f'Epoch {epoch+1}, gen loss = {g_loss}, disc loss = {d_loss}')
    generator_losses.append(g_loss)
    discriminator_losses.append(d_loss)


  # Generate after the final epoch
  display.clear_output(wait=True)

train(training_dataset, EPOCHS) #run only for training

#code for visualizing losses

fig = plt.figure(figsize=(10,10))
plt.plot(
              range(EPOCHS) ,
              generator_losses,
              label="Generator Loss"
          )
plt.plot(
              range(EPOCHS) ,
              discriminator_losses,
              label="Discriminator Loss"
          )
plt.legend()
plt.savefig("loss_figure.png")
plt.show()


checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))


generated_images2 = generator(seed, training = False)
generated_images2 =0.5 * generated_images2 + 0.5
```

```
fig = plt.figure(figsize=(10,10))

plt.imshow(generated_images2[0])
plt.axis('off')
plt.savefig("image_generated.png")
plt.show()


generator.save("./trained_generator.h5")
print("Saved model to disk")

checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
import keras
seed2 = tf.random.normal([1, 100])
generator = keras.models.load_model("./trained_generator.h5")

seed = tf.random.normal([22 , noise_dim])

generated_images2 = generator(seed, training = False)
generate_images(generated_images2)
```

A.2.3 Version 3

```
import torch
import torchvision
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader ,Dataset
import matplotlib.pyplot as plt
torch.manual_seed(0)

def show_tensor_images(image_tensor, num_images=25, size=(1, 28, 28)):
    '''
    Function for visualizing images: Given a tensor of images, number of
        images, and
    size per image, plots and prints the images in an uniform grid.
    '''
    image_tensor = (image_tensor + 1) / 2
    image_unflat = image_tensor.detach().cpu()
    image_grid = make_grid(image_unflat[:num_images], nrow=5)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.savefig(f"images_figure.png")
    plt.show()

def make_grad_hook():
    '''
    Function to keep track of gradients for visualization purposes,
```

```
    which fills the grads list when using model.apply(grad_hook).
    '''
    grads = []
    def grad_hook(m):
        if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
            grads.append(m.weight.grad)
    return grads, grad_hook

import random

seed = 42
random.seed(seed)
torch.manual_seed(seed)

path_of_data = "../input/best-artworks-of-all-time/resized"


import gc

gc.collect()

torch.cuda.empty_cache() #clears gpu memory


class DataSet(Dataset):
    """
    A typical dataset class that is used to create dataloader for the model
    """
    def __init__(self , path_of_data) :
        super(DataSet, self).__init__()


        transform_ds = transforms.Compose([transforms.Resize((64
            ,64)),transforms.ToTensor() ,\
                                           transforms.Normalize((0.5, 0.5,
                                              0.5), (0.5, 0.5, 0.5))])
        self.images = torchvision.datasets.ImageFolder(root=path_of_data,
                                        transform=transform_ds)
    def __len__(self):
        return len(self.images)
    def __getitem__(self , index):
        return self.images[index]
dataset = DataSet(path_of_data)




dataloader = DataLoader(
    dataset ,
    batch_size=64, pin_memory=True, shuffle=True
```

```
)

class Generator(nn.Module):
    '''
    Generator Class
    Values:
        z_dim: the dimension of the noise vector, a scalar
        im_chan: the number of channels in the images, fitted for the
            dataset used, a scalar 3 for rgb
        hidden_dim: the inner dimension, a scalar
    '''
    def __init__(self, z_dim=100, im_chan=3, hidden_dim=64):
        super(Generator, self).__init__()
        self.z_dim = z_dim
        # Build the neural network
        self.gen = nn.Sequential(
            self.make_gen_block(z_dim, hidden_dim * 4 , kernel_size = 5
                ,stride = 2),
            self.make_gen_block(hidden_dim * 4, hidden_dim * 4,
                kernel_size=5,stride =1),
            self.make_gen_block(hidden_dim * 4, hidden_dim*2 ,
                kernel_size=4,stride = 1),
            self.make_gen_block(hidden_dim * 2, hidden_dim ,
                kernel_size=4,stride = 1),
            self.make_gen_block(hidden_dim, 32 , kernel_size=4,stride = 2),

            self.make_gen_block(32, im_chan , kernel_size=2,
                final_layer=True,stride = 2),
        )

    def make_gen_block(self, input_channels, output_channels,
        kernel_size=3, stride=2, final_layer=False):
        '''
        Function to return a sequence of operations corresponding to a
            generator block of WGAN;
        a transposed convolution, a batchnorm (except in the final layer),
            and an activation.
        Parameters:
            input_channels: how many channels the input feature
                representation has
            output_channels: how many channels the output feature
                representation should have
            kernel_size: the size of each convolutional filter, equivalent
                to (kernel_size, kernel_size)
            stride: the stride of the convolution
            final_layer: a boolean, true if it is the final layer and
                false otherwise
                      (affects activation and batchnorm)
        '''
        if not final_layer:
```

```
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels,
                    kernel_size, stride ),
                nn.BatchNorm2d(output_channels),
                nn.ReLU(inplace=True),
                )
            else:
                return nn.Sequential(
                    nn.ConvTranspose2d(input_channels, output_channels,
                        kernel_size, stride),
                    nn.Tanh(),
                )

        def forward(self, noise):
            '''
            Function for completing a forward pass of the generator: Given a
                noise tensor,
            returns generated images.
            Parameters:
                noise: a noise tensor with dimensions (n_samples, z_dim)
            '''

            return self.gen(noise)

    def get_noise(n_samples, z_dim=100, device='cpu'):
        '''
        Function for creating noise vectors: Given the dimensions (n_samples,
            z_dim)
        creates a tensor of that shape filled with random numbers from the
            normal distribution.
        Parameters:
          n_samples: the number of samples to generate, a scalar
          z_dim: the dimension of the noise vector, a scalar
          device: the device type
        '''
        return torch.randn(n_samples, z_dim,1,1 ,device=device)


    class Critic(nn.Module):
        '''
        Critic Class
        Values:
            im_chan: the number of channels in the images, fitted for the
                dataset used, a scalar 3 for rgb
            hidden_dim: the inner dimension, a scalar
        '''
        def __init__(self, im_chan=3, hidden_dim=64):
            super(Critic, self).__init__()
            self.crit = nn.Sequential(
                self.make_crit_block(im_chan, hidden_dim*4),
```

```
            self.make_crit_block(hidden_dim*4, hidden_dim*8 ),
            self.make_crit_block(hidden_dim*8 , hidden_dim*16 ,
                kernel_size=4 ,stride=3),
            self.make_crit_block(hidden_dim*16, 1 ,kernel_size=5,stride=3),

        )


    def make_crit_block(self, input_channels, output_channels,
        kernel_size=4, stride=2, final_layer=False ,bias=True):
        '''
        Function to return a sequence of operations corresponding to a
            critic block of WGAN;
        a convolution, a batchnorm (except in the final layer), and an
            activation (except in the final layer).
        Parameters:
            input_channels: how many channels the input feature
                representation has
            output_channels: how many channels the output feature
                representation should have
            kernel_size: the size of each convolutional filter, equivalent
                to (kernel_size, kernel_size)
            stride: the stride of the convolution
            final_layer: a boolean, true if it is the final layer and
                false otherwise
                      (affects activation and batchnorm)
        '''
        if not final_layer:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size,
                    stride ,padding = 1 ,bias=bias),
                nn.LeakyReLU(0.2, inplace=True),
            )
        else:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size,
                    stride=1 ,padding=0 ,bias = bias ),
            )

    def forward(self, image):
        '''
        Function for completing a forward pass of the critic: Given an
            image tensor,
        returns a 1-dimension tensor representing fake/real.
        Parameters:
            image: a flattened image tensor with dimension (im_chan)
        '''
        crit_pred = self.crit(image)
        return crit_pred.view(len(crit_pred), -1)
```

```
fake_noise = get_noise(1 ,100)
generator = Generator()
i = generator(fake_noise)
print(i.shape)
critic = Critic()
c = critic(i)
print(c)

n_epochs = 38 #the number of times you iterate through the entire dataset
    when training
z_dim = 100 #the dimension of the noise vector
display_step = 50#how often to display/visualize the images
batch_size = 64 #the number of images per forward/backward pass
lr = 1e-4# 0.0002 #the learning rate
beta_1 = 0.5#the momentum terms
beta_2 = 0.999#the momentum terms
c_lambda = 10#weight of the gradient penalty
crit_repeats = 7 #number of times to update the critic per generator
    update
device = 'cuda' #the device type

#initializing generartor and critic objects
#also their weights
gen = Generator(z_dim).to(device)
gen_opt = torch.optim.Adam(gen.parameters(), lr=lr, betas=(beta_1,
    beta_2))
crit = Critic().to(device)
crit_opt = torch.optim.Adam(crit.parameters(), lr=lr, betas=(beta_1,
    beta_2))

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



def get_gradient(crit, real, fake, epsilon):
    '''
    Return the gradient of the critic's scores with respect to mixes of
        real and fake images.
    Parameters:
        crit: the critic model
        real: a batch of real images
        fake: a batch of fake images
```

```
        epsilon: a vector of the uniformly random proportions of real/fake
            per mixed image
    Returns:
        gradient: the gradient of the critic's scores, with respect to the
            mixed image
    '''
    # Mix the images together
    mixed_images = real * epsilon + fake * (1 - epsilon)

    # Calculate the critic's scores on the mixed images
    mixed_scores = crit(mixed_images)

    # Take the gradient of the scores with respect to the images
    gradient = torch.autograd.grad(
        #we need to take the gradient of outputs with respect to inputs.

        inputs=mixed_images,
        outputs=mixed_scores,

        # These other parameters have to do with the pytorch autograd
            engine works
        grad_outputs=torch.ones_like(mixed_scores),
        create_graph=True,
        retain_graph=True,
    )[0]
    return gradient



def gradient_penalty(gradient):
    '''
    Return the gradient penalty, given a gradient.
    Given a batch of image gradients, you calculate the magnitude of each
        image's gradient
    and penalize the mean quadratic distance of each magnitude to 1.
    Parameters:
        gradient: the gradient of the critic's scores, with respect to the
            mixed image
    Returns:
        penalty: the gradient penalty
    '''
    # Flatten the gradients so that each row captures one image
    gradient = gradient.view(len(gradient), -1)

    # Calculate the magnitude of every row
    gradient_norm = gradient.norm(2, dim=1)

    # Penalize the mean squared distance of the gradient norms from 1

    penalty = (1/len(gradient)) *torch.sum(( gradient_norm - 1 )**2)
```

```
    return penalty


def get_gen_loss(crit_fake_pred):
    '''
    Return the loss of a generator given the critic's scores of the
        generator's fake images.
    Parameters:
        crit_fake_pred: the critic's scores of the fake images
    Returns:
        gen_loss: a scalar loss value for the current batch of the
            generator
    '''

    gen_loss = torch.mean(crit_fake_pred)

    return gen_loss



def get_crit_loss(crit_fake_pred, crit_real_pred, gp, c_lambda):
    '''
    Return the loss of a critic given the critic's scores for fake and
        real images,
    the gradient penalty, and gradient penalty weight.
    Parameters:
        crit_fake_pred: the critic's scores of the fake images
        crit_real_pred: the critic's scores of the real images
        gp: the unweighted gradient penalty
        c_lambda: the current weight of the gradient penalty
    Returns:
        crit_loss: a scalar for the critic's loss, accounting for the
            relevant factors
    '''

    crit_loss = torch.mean(crit_real_pred) -torch.mean(crit_fake_pred) +
        c_lambda*gp

    return crit_loss


import matplotlib.pyplot as plt

cur_step = 0
generator_losses = []
critic_losses = []
for epoch in range(n_epochs):
    # Dataloader returns the batches
```

```
for real, _ in tqdm(dataloader):
    cur_batch_size = len(real)
    real = real.to(device)

    mean_iteration_critic_loss = 0
    for _ in range(crit_repeats):
        #Update critic
        crit_opt.zero_grad()
        fake_noise = get_noise(cur_batch_size, 100, device=device)
        fake = gen(fake_noise)
        crit_fake_pred = crit(fake.detach())
        crit_real_pred = crit(real)

        epsilon = torch.rand(real.shape ,device=device,
            requires_grad=True)
        gradient = get_gradient(crit, real, fake.detach(), epsilon)
        gp = gradient_penalty(gradient)
        crit_loss = get_crit_loss(crit_fake_pred, crit_real_pred, gp,
            c_lambda)

        # Keep track of the average critic loss in this batch
        mean_iteration_critic_loss += crit_loss.item() / crit_repeats
        # Update gradients
        crit_loss.backward(retain_graph=True)
        # Update optimizer
        crit_opt.step()
    critic_losses += [mean_iteration_critic_loss]

    #Update generator
    gen_opt.zero_grad()
    fake_noise_2 = get_noise(cur_batch_size, 100, device=device)
    fake_2 = gen(fake_noise_2)
    crit_fake_pred = crit(fake_2)

    gen_loss = get_gen_loss(crit_fake_pred)
    gen_loss.backward()

    # Update the weights
    gen_opt.step()

    # Keep track of the average generator loss
    generator_losses += [gen_loss.item()]

    #Visualization code
    if cur_step % display_step == 0 and cur_step > 0:
        gen_mean = sum(generator_losses[-display_step:]) / display_step
        crit_mean = sum(critic_losses[-display_step:]) / display_step
        print(f"Epoch {epoch} Step {cur_step}: Generator loss:
            {gen_mean}, critic loss: {crit_mean}")
        show_tensor_images(fake)
```

```
            show_tensor_images(real)
            step_bins = 20
            num_examples = (len(generator_losses) // step_bins) * step_bins
            plt.plot(
                range(num_examples // step_bins),
                torch.Tensor(generator_losses[:num_examples]).view(-1,
                    step_bins).mean(1),
                label="Generator Loss"
            )
            plt.plot(
                range(num_examples // step_bins),
                torch.Tensor(critic_losses[:num_examples]).view(-1,
                    step_bins).mean(1),
                label="Critic Loss"
            )
            plt.savefig(f"figure_loss_epoch_step_{cur_step}_.png")
            plt.legend()
            plt.show()

        cur_step += 1


i =  generator(get_noise(64,100))

show_tensor_images(i)
```

A.2.4 Requirements

Library Requirements

```
imageio==2.9.0
ipython==7.22.0
keras==2.4.3
matplotlib==3.4.1
numpy==1.20.2
Pillow==8.2.0
tensorflow==2.4.1
torch==1.8.1
torchvision==0.9.1
tqdm==4.60.0
```

Dataset used `https://www.kaggle.com/ikarus777/best-artworks-of-all-time`

A.2.5 Instructions to execute code

The easiest way to run this code is to follow the links for each version. This will direct you to a Kaggle notebook where you can press the Run all function at the top. This will require you to have a Kaggle account or create one. To improve the

performance of the model, enable the GPU accelerator on the right-hand tab. Follow the following links for each version of the code.

Version 1:

`https://www.kaggle.com/moredeyethejedeye/dcgan-v1`

Version 2:

`https://www.kaggle.com/moredeyethejedeye/dcgan-v1-2`

Version 3:

`https://www.kaggle.com/moredeyethejedeye/wgan-gp-ver-3`

GitHub Repo

`https://github.com/samsdead/Gan_Art`

A.3 Outputs

A.3.1 Version 1

Click the following for the outputs. Figure A.1

A.3.2 Version 2

Click the following for the outputs. Figure A.2

A.3.3 Version 3

Click the following for the outputs. Figure A.3

A.4 Template Research Paper Submission

Generative Art: A new approach with Generative Adversarial Networks*

Said Moredi[1][0000−1111−2222−3333]

[1] City, University of London, London, Northampton Square
[2] info@city-university-of-london.co.uk
https://www.city.ac.uk/about/schools/mathematics-computer-science-engineering/computer-science

Abstract. THIS TEMPLATE PAPER IS NOT COMPLETE. IT REQUIRES EDITING AND PEER REVIEWING. This paper tackles the problem of generative art in the field of machine learning. Generative art has produced some of the most interesting pieces of geometrical and contemporary art – however, its repetitiveness becomes its downfall. This paper tackles these issues of generative art and future problems by suggesting an alternative method. A new technique using machine learning models with generative adversarial networks is proposed to tackle the problem. A deeper dive into the architecture of said models is analysed and evaluated for suitability of the problem at hand, producing art through machine learning models, specifically generative adversarial networks. Further analysis is laid out to in terms of performance and efficiency of the programs. As this program will be run on consumer level hardware, architectures where the processing time exceeds a few hours will be disregarded as this requires specialist hardware to be run efficiently. This paper will introduce a larger toolbelt for artists within the field of generative art and give insiders an easy to step towards the world of generative art with machine learning models.

Keywords: Generative Art · Generative Adversarial Networks (GAN) · Deep Convolutional GAN (DCGAN) · Wasserstein GAN with Gradient Penalty (WGAN - GP) · Machine Learning

1 Problem Description

1.1 What is the problem with standard generative art?

Generative art flourishes in the world of geometry, abstraction, and mechanised production. Themes of futurism, constructivism and abstract expressionism are all by-products of years of generative art [**?**]. It has become a new medium to explore and express a new form of art . However, this approach of producing generative art consists of many variations of algorithms with no sense of machine learning within the final outcome of the image [6]. The techniques used within

* Supported by City, University of London

generative art focus more on repetitions of certain patterns within a 2D or 3D field. This can be impressive and does provide an endless form of appealing art to look at. Unfortunately, the linear approach to the production of the art takes away a level of creativity which is instilled in pieces created in real life. Eliminating the robotic approach within generative art and replacing it with a model which learns from the art, sprouts the same spark of human touch within the images. This introduces a new world of generative art that is yet to be explored. From this development, pieces of art that might be imagined by an individual can be generated seamlessly. A mix of Basquiat and Sandro Botticelli may be imagined, but with technologies like GAN Art can be brought to fruition. The use of specific datasets such as portraits or clothing can sprout new creations that resemble a human feel to the work created therefore allowing them to be used in a commercial aspect too [**?**].

Subsequent paragraphs, however, are indented.

How this GAN Art provide something new? GAN Art can provide a new approach towards art. As it becomes a more widely used medium the entry level to using a program like this can allow multiple custom datasets to be considered, each dataset can give an amalgamation of styles within a piece of art. Taking the styles of classical renaissance era paintings and murals mixed with new age abstract art can breed new creations that have yet to be imagined. From an artist's viewpoint, they can take the creations from GAN art as a form of inspiration or publish it as something they have created. Answering the questions of periods of art working in symphony is only one new use that GAN art will provide. Within the field of video games or other visual mediums such as film or photography, GAN art can be used to produce visual pieces which accompany the mediums fadaeddini2018case. It breaks apart the methodological approach that generative art carries out, the use of machine learning in art is not something new but it is definitely an area yet to be explored greater. Shifting generative arts procedural aspects and replacing them with forms of backpropagation is the aim of this project. The scope of this work is not to tunnel the use cases of GAN Art to producing and generating art but a program which can be tweaked to fit within other constraints, for example the creation commercial objects.

1.2 This papers Objectives

GAN Art's main objective is to produce new pieces of art which was directly influenced by a selected dataset. Here, a large dataset of art images will be passed through the main program, through training the program will begin to output new images. For example, a dataset of portraits from the renaissance era should produce an image of a portrait with a blend of styles that were extracted from the original images. To obtain such datasets, I will be using Kaggle as it has vast range of images, which I can extract and process into my very own datasets kaggleWeb. The pre-processing functions will have constraints put in to avoid anomalies in the main program.

Work Performed Every part of the GAN will be produced from scratch. I will be using Python as the programming language in conjunction with other popular machine learning libraries. TensorFlow, PyTorch, and Pillow are some of the main libraries that will be used in the project. I am not completely familiar with all the libraries but with their documentation online I am confident in putting together an efficient GAN.

1.3 Assumptions Made

In terms of technology, I will be developing on PyCharm, Google Colab + Kaggle and Jupyter notebooks. Although both Colab and Jupyter Notebooks do essentially the same thing, parts of the initial development will experiment on both platforms for performance and usability analysis. Assumptions about these technologies remaining free throughout my development have been assumed. I have no prior knowledge of GAN's and took on this project due to my interest in machine learning and art in all forms. Most of the research has been carried out by further reading into the subject of generative art through deep learning modules. Although this may be a naïve implementation of the project it is something I pursued as I truly enjoy creating and tuning art. In terms of scheduling, the bulk of this project has been carried out in the research phase as I believe a deeper understanding of concepts can help me better execute an intuitive program. For hardware, assumptions were made that my current hardware can comfortably complete the training phase of the program and produce competent results that are satisfactory to the requirements of the project. Arrangements have been explored if my current hardware fails in any form.

1.4 Recap of GAN

Generative Adversarial Networks (GAN) is an unsupervised but not limited to, machine learning model. It encompasses two machine learning models against each other in a min and max game, these components being the generator and discriminator. The generator from a series of random noise values tries to create a fake image which the discriminator might pass as a real image. The discriminators job is to classify real images from the training set as real or fake images produced from the generator. Through the loss functions of the generator and discriminator, GAN carries out back propagation to updates the weights on the generator and discriminator. Overtime the generator output will pass a threshold where the discriminator cannot distinguish the output as fake. This technique is not limited to images, any data which can be turned into tensor instances will work – highlighting the flexibility of GAN. Both the generator and components can use other machine learning models such as deep convolutional neural networks, perceptron's, and recurrent neural networks.

2 First Implementation

The generator takes in a random seed to produce a generated image from it. The use of Conv2DTranspose deviates from the original DCGAN paper proposal. The reason for this is the increased efficiency and results of transposed convolutions. The input provided by the seed is transposed over the kernel to produce an output. These parameters are trainable and can produce better results in the long run. Upsampling and Conv2D is used for the generator in the original DCGAN paper; the problem with Upsampliung is that it does not have any trainable parameters, so it repeats its input to produce an output, leading to a large amount of generator information being lost. The discriminator in this architecture uses Conv2D, which is the same as the DCGAN paper. Batchnorm is used for normalization of the generator and discriminator for pre-processing of the data. The activation function of this particular architecture is the rectified linear unit (ReLU) activation function which is used for all layers of the generator except its final output layer which uses TanH. ReLU is popular for most deep learning models so it is a great place to start and further develop into new models. To improve the strength of the discriminator, in this case to classify which images are real or counterfeit, the Leaky ReLU activation function is used. The Adam optimizer is used for both the generator and the discriminator. As this is a relatively noisey problem due to the number of pixels and working on images, the Adam optimization algorithm will be able to handle the sparse gradients within the generator and discriminator.

2.1 Loss

Generator Loss The generator loss (cross entropy) is a measurement of how well it performed at fooling the discriminator. If the input presented to the discriminator by the generator is classified as real, the generator done a good enough job to bypass the detection of the discriminator. The model is penalized proportionally to how much the predicted probability distribution varies from the expected probability distribution of the fake image due to the existence of cross entropy. It serves as a foundation for the error, which is then backpropagated via the generator and discriminator, allowing the next batch to perform better.

Discriminator Loss The discriminators loss function (cross entropy) is based on its ability to distinguish real images from the original dataset and fake ones provided by the generator. It compares the real images to an array of ones, with one being real, it then compares its prediction of counterfeit images on an array of zeroes, with anything close to zero being fake. The goal of this function is to classify all real images as one and all fake images as zero. The loss is signified is signified by adding the two losses of both classification processes together. In this case if the discriminator were to predict a probability of 0.1 when the actual value is 1, it would be considered a very bad result and a high loss value. For a perfect model to exists the log loss will be 0, which is extremely hard to do in machine

learning models. The discriminator_loss function takes the parameters of the real output defined by our discriminator model and the fake output provided by our generator model. The first line of code real_loss compares the real image to an array of ones and gains a probability of the output being real (probability close to one). The second line fake_loss compares the fake output produced by the generator to an array of zeroes and gains a probability of the output being fake (probability close to zero). The penultimate line adds both of the probability predictions together and finally they are returned as the total_loss, which is the output of the discriminator_loss function.

2.2 Outcomes

The final images produced by the network are definitely satisfactory. As it is the first implementation of generative art through machine learning models, the techniques used were mostly from a hand full of papers. All of the images are abstract in nature with various indistinguishable shapes are mapped out onto the images. The colour palette takes from many different genres of paintings showing the model has learnt from all of its given input. The dataset can be blamed to an extent as it exhibits art from various movements within many different time periods. At the end it produces visually pleasing images with little amounts of noise. As noise is not so much of an issue in the images, a larger focus can be put on other objects showing in the image. The strength of the DCGAN architecture is visible in this instance as it can withstand a dataset with a range of inputs and still produce satisfactory results. The hyper tuning of the parameters allows the program to consistently learn and update its weight accordingly which works well for the final outcome. The use of the Adam optimizer limits the vanishing gradient effect on the generator and the discriminator. The robustness of the program to produce an .h5 file which can be applied to another dataset will allow it to be used on multiple data sets without the need for retraining. Overall, the training time for the model was short, running it on Kaggle further helped make the case for this. The main problem arising from the outcomes is the relatively small resolution. This problem will be expanded on and given a clear explanation on the next version.

3 Second Implementation

One of the issues with the first sprint was its low resolution. No discernible objects can be made out in the final image. Although it can be considered to achieve its objective function it lacks in many ways as the final implementation of the program. A clear goal to overcome the low resolution is to write the same architecture for 128x128 resolution. After this resolution has been overcome, the same program can be written for 256x256 resolution. The new changes may increase training time but may wield better results. The results this program expects to produce overall GAN stability and better image quality. Consideration

for High-resolution Deep Convolutional Generative Adversarial Networks (HD-CGAN) were explored however the architecture adds additional features which considerably slow down overall performance but do not wield any better results. This led to the thought that the architecture may not be suitable for high quality generated images. As there are more variables to consider and a much larger generator and discriminator model, it causes the program many times to run out of GPU memory. If it were to solely run on a CPU the DCGAN can take huge amounts of time for a single epoch. Currently a single epoch is completed in roughly 13 seconds on Kaggle with a 16GB GPU. The program tries to produce the best results for the least amount of training and processing time. This is important as it stops inefficient architectures being developed which may produce better results as a lost to performance.

3.1 Solutions

To scale up the GAN to a higher resolution the WIDTH and HEIGHT variables were changes accordingly to 128x128. The training process was stable for 50 epochs. The discriminator in both models worked very well in both cases staying roughly between 0.3 to 0.4 loss, meaning the model was classifying images correctly to a high degree. At the beginning of the training phase in the first few epochs the generator has not learnt enough to produce good enough results for the discriminator; the generator loss stays high sub 10 epochs. Then gradually the generator loss decreases to 40 epochs, around 30-40 epochs there is the smallest amount of generator loss. Once its past 40 the generator loss starts to shift higher. The results produced for 50 epochs showed slight improvements to the original image. This is in the case of producing new distinguishable objects in the photo repeatedly with minimum amounts of noise. DCGAN architecture is meant to produce better results the longer the model is trained for. In this case the epochs for the program were changed to the original 150 and retrained on the dataset. The discriminator loss stayed consistent as the previous models. However, the generator loss in the model after 50 epochs diverges and fluctuates higher as the number of epochs increases. This shows a level of instability to the architecture which this project attempts to avoid. A lot of reasons can be the backbone of this problem such as bad initialization of the generator model. Art-GAN expressed previously in the literature review uses a form of mapping from the original images to produce better results instantly. This can be a workaround for the problem at hand but other factors such as local minima, saddle point loss, or simply the instability of the DCGAN architecture on large resolution images, can contribute much to the problem. The generator loss may have decreased if trained for even more epochs however this would have not showed an intuitive and robust design. A new solution arises from this unique problem, Wassertein GAN with Gradient Penalty.

4 Final Implementation

To develop a training procedure which is stable and does not increase in generator loss throughout the program, the architecture of the GAN will change to the Wasserstein GAN with Gradient Penalty (WGANGP). This GAN has better training stability and leaps over the hyper tuning issues of a DCGAN. The first model may have been fixed with strenuous hyper tuning of each parameter however it would have been counter intuitive as it would have been constrained to the specific dataset. The loss in most GAN models provides a sense of ambiguity to what is specifically going wrong in the program and can not be used to better the program; only used to capture the progress of improvements. With WGAN the loss can be used as a termination criteria where training will stop if the loss converges with the actual discriminator, essentially stopping training at the correct time with the highest degree of accuracy. In the case of this paper this is extremely helpful to pinpoint the lowest generator loss, subsequently producing better images.

4.1 Reasoning for the Architecture Changes

One of the main reasons to switch to the WGAN-GP as it solves the stability issues that occurred with DCGAN. WGAN-GP uses a special kind of loss function known as the Wasserstein-Loss, this is one of the main contributors which prevents mode collapse in the network. Mode collapse is when the network outputs only a specific number of classes rather than the whole network. Another reason to switch to the WGAN-GP is its ability to be run for as long as possible. As DCGAN progressively gets worse as the number of epochs increases, WGAN-GP benefits and gets better for more iterations. This means the best possible outcomes can only be put forward in this implementation. A significant change in the architecture from DCGAN is the introduction of a "CRITIC" instead of a discriminator. As a lot of useful information can be lost from classify images through ones and zeroes, the "CRITIC" in WGAN-GP scores images with real numbers. Here a larger amount of information is passed onto the generator to significantly improved itself on each iteration. The critic in this architecture is trained more than the generator which sets a higher boundary for images to be classified as real. In a standard WGAN-GP the critic is trained fives times more than the generator. This useful loss metric is related to the convergence of the generators and improves sample quality, which refers to the images sent to the critic. It also improves stability of the GAN whilst preventing mode collapse. Gradient penalty improves the Adam optimizer that was initially used in the DCGAN. The momentum from the Adam optimizer is completely removed. To change the training of the critic the WGAN-GP takes an interpolation of a real image and a generated image, this is done by taking a random number between epsilon and one. The norm of the gradient is taken which satisfies the Lipchitz constraint put on the critic.

4.2 Gradient Penalty

The gradient penalty is computed in two functions. Two scenarios can arise for computing the gradient: computing the gradient with respect to the images and computing the gradient penalty given the gradient. The gradient is first computed by taking the interpolation of two images, one from the generator one from the discriminator. This is done by weighing the fake and real images via a random vector between epsilon and 1, then these two numbers are added together. Once the intermediate is calculated the critic produces a new output for the image. Then the program computes the gradient of the critics score on the mixed images (output) with respect to the pixels of the mixed images input. To compute the gradient penalty given the gradient the magnitude of each image's gradient is calculated, this is also called the norm. After this the program calculates the penalty by squaring the distance between each magnitude and the ideal norm of one by taking the mean of all squared distances. This means that instability is minimized as the mean of these gradients will make partial changes to the generator. This also stops the exploding gradient problem where the discriminator might want to change the generator by a very large amount, again, reducing instability in the network and preventing mode collapse. The function get_gradient takes the critic model, a batch of real images, a batch of fake images and epsilon, which is a vector of the uniformly random proportions of the real and fake images per mixed image. This function returns the gradient of the critics scores with respect to the given mixed image. The function gradient_penalty takes a gradient from the previous function and returns the gradient penalty. The function calculates the magnitude of each image gradient and penalises the mean quadratic distance of each magnitude to one given a batch of image gradients.

4.3 Loss

Generator Loss The generator loss is calculated by maximizing the critics prediction on the generator's fake images. given the score for all fake images the mean is calculated. The function get_gen_loss takes the critics scores of the fake images as a parameter and returns the loss of the generator. The model is penalised proportionally to how much the predicted probability distribution varies from the expected probability distribution for a given image using the loss function. The higher the predicted output of produced images, the lower the critic's loss when analysing generated images.

Discriminator Loss (Critic) In terms of the critic, the loss is calculated by maximizing the distance between the critic's predictions on the real images and the prediction on the fake images, this is done whilst also adding a gradient penalty. The gradient penalty is weighed accord to lambda. Given the scores for all the images in the batch, the program uses the mean of them. For a hundred epochs the critics loss gradually increases whilst the generator loss decreases consistently. This is a large improvement over the first iteration of this program.

It does not struggle at 50 epochs as the first DCGAN, it continually improves itself. This essentially means the quality of the images produced are going to increase as long as the training continues. A WGAN-GP can be running for as long as it needs to be. This flexibility allowed me to run the model for over 300 epochs which produced outstanding results. Whilst still maintaining a relatively small critic loss and decreasing its generator loss on each iteration.

4.4 Training

The training of this model is very slow compared to the DCGAN; this is understandable as WGAN-GP is a much heavier process in compute power. The gradient penalty requires to compute the gradient of a gradient – this means potentially a few minutes per epoch depending on the hardware. To get the best results WGAN-GP should be run for as long as possible on a powerful GPU. Another change from the original DCGAN is that the critic is updated multiple times every time the generator is updated. This essentially helps the generator from overpowering the critic. WGAN-GP does not directly improve the overall performance of a GAN – but it does save time from tuning hyper parameters meticulously. WGAN-GP also increases the stability of the model and avoids mode collapse which saves more time in the long run. WGAN-GP will be able to train in a much more stable way than the vanilla DCGAN although it will generally run a bit slower in terms of compute time. The training constraints of WGAN-GP original paper have been put in place for the best results (FIGURE). The training process is visualized throughout to keep track of the changes. The training process is written in one large for loop starting with updating the critic. The function of the average critic loss in the batch is calculated and used in the visualization process. After this the gradients and optimizer are updated, the first half of the for loop returns the average critic loss. The generator and weights are updated accordingly. The generator loss is also calculated and used further down for visualization and analysis purposes. This training process is much heavier, but it introduces new avenues to explore deeper into GAN for generative art. It also wields the best results out of the three models presented in this paper. Each image is far more advanced than the average output delivered by the vanilla DCGAN.

5 Findings

5.1 Interpretations

The results of this paper show that generative art can be produced through machine learning models, specifically generative adversarial networks. The range of results shows there is still a lot of work to do in terms of creating light architectures for generative art with the use of machine learning models. From the literature this paper extracts key findings in machine learning models and implements them intuitively into programs which produce art. The evolution of the first vanilla DCGAN to the WGAN-GP highlights the changing nature of machine learning models to combat specific problems, mode collapse in this case.

5.2 Implications

This paper helps in adding material to the newly formed world of generative adversarial networks. It also aids in understanding generative adversarial networks and bringing new interest to the subject. In terms of generative art, it solves the problem of mechanised production within it and introduces a new machine learning approach. The machine learning approach encompasses a flair of human touch by feeding the network a large dataset of artworks from a wide time period. Breaking down the linear approach to art can be beneficial for existing generative art artists. GAN are on the boundary of breaking out into the commercial world. Its achilleas is hyper tuning which can render some datasets being completely disregarded. The WGAN-GP in this paper solves this problem and allows for a range of datasets to be used.

5.3 Limitations

The results of the GAN are subjective, however that is the case with all art. Noise capturers could have been built however they can not be translated into meaningful graphs. This paper suggests when generative art produced through machine learning models is successful it should be compared to real life art. This way members of the audience can be asked to decide "what piece is created through a machine learning model?". However, this test is past the scope of this paper as it solely focuses on the generation of the art. Although important advancements have been made in GAN, they are still relatively new technology, so implications of the results of this paper are still yet to be further analysed. If new technologies and models become apparent the results of this paper may implicate new findings.

5.4 Recommendations

From here new architecture can be better adapted into producing art. Although specific implementations exist such as StyleGAN, StyleGAN2 and ArtGAN; they require very strong hardware to get some form of acceptable results. This paper used hardware that was readily available and provides alternatives of running the code in dedicated environments online. New methods of computer vision entangled with GAN implementations maybe able to combat the problem of the heavy compute power required to run complex GAN. As making this technology readily available will allow it to thrive and progress forward.

5.5 Conclusion

This paper aimed to show machine learning models to produce generative art. It managed to produce three programs which can satisfy the main objective. The first version follows a naïve vanilla Deep Convolutional Generative Adversarial Network architecture and implements it to produce interesting art with minimal noise. The second version of this program creates a high-resolution version of

these images and begins to identify pitfalls in the architecture such as generator loss increasing after a certain epoch. To combat this a new architecture is addressed and briefly explained with its implementation is shown after every section. The advantages and disadvantages of the architecture are highlighted, for example the WGAN-GP may produce a better final output but takes a large performance hit. This is show in simple experts from the program running with a single epoch taking roughly 3-4 minutes to execute. Whilst the first DCGAN took roughly 30 minutes to execute the WGAN-GP for the same number of epochs takes around 2 and a half hours. This paper takes the approach of finding new important architectures for generative art as it is something that is rapidly advancing. As generative artists breakout into the 3D plane, the world of machine learning models for generative art lacks behind. The contributions of this paper such as designing and implementing exciting new architectures for the task can bridge that gap.

References

1. Author, F.: Article title. Journal **2**(5), 99–110 (2016)
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). https://doi.org/10.10007/1234567890
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017
6. @articlegalanter2016generative, title=Generative art theory, author=Galanter, Philip, journal=A Companion to Digital Art, pages=146–180, year=2016, publisher=Wiley Online Library

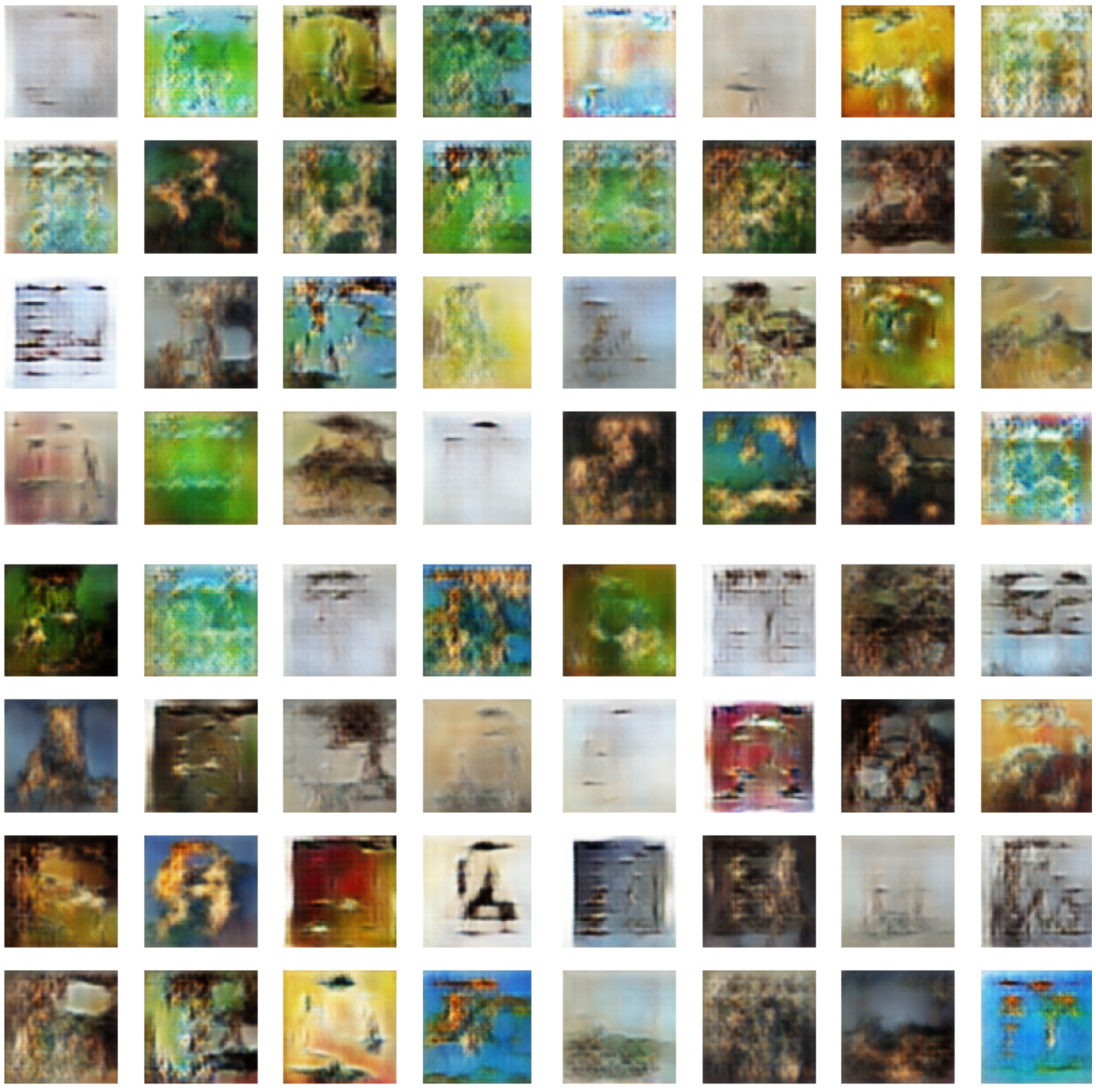


Figure A.1: Outputs for version 1 of the program

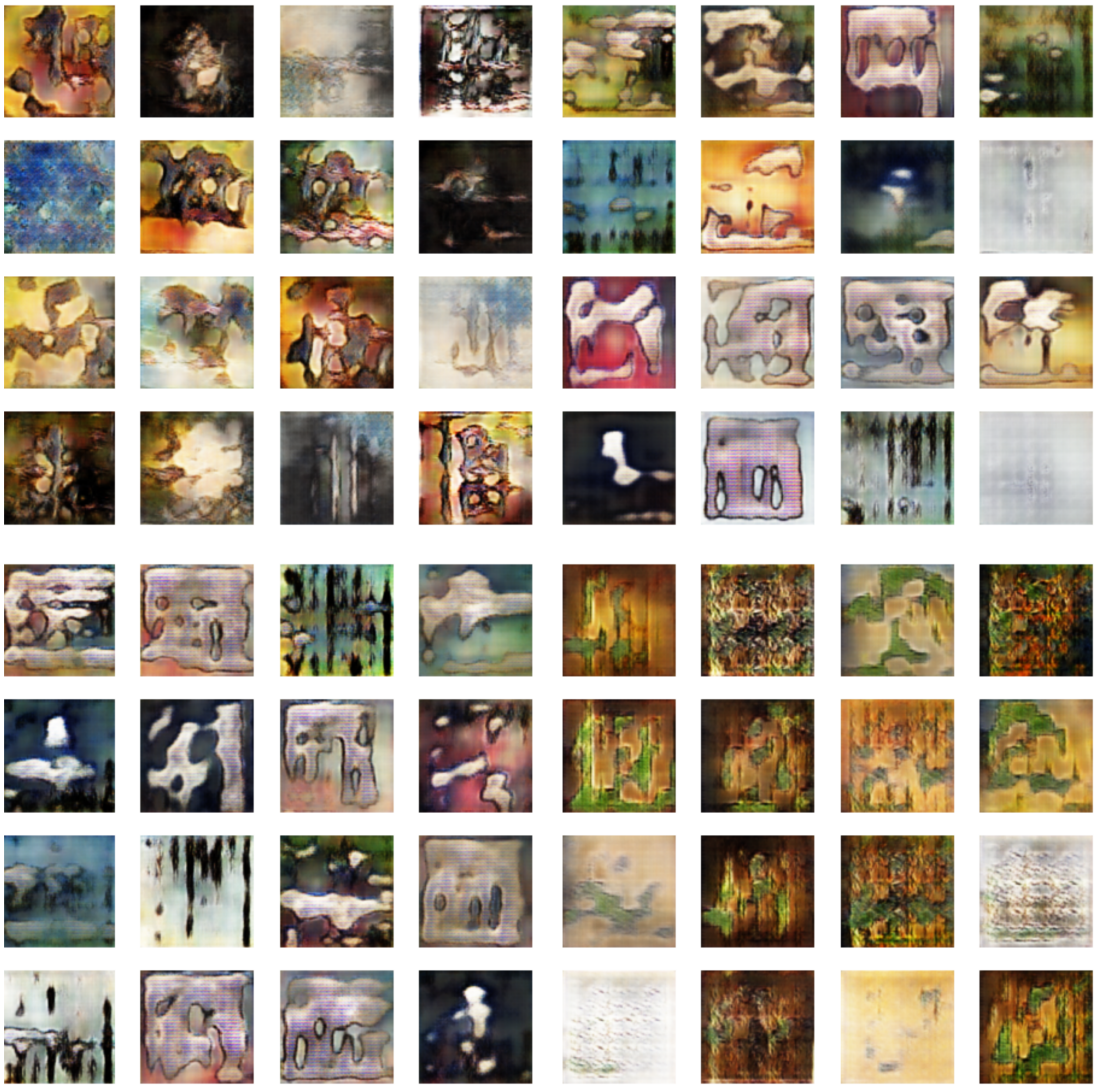


Figure A.2: Outputs for version 2 of the program

Figure A.3: Outputs for version 3 of the program