Bachelor of Science in Computer Science and Engineering

GENERATING COMPLEX HENNA PATTERNS USING GENERATIVE ADVERSARIAL NETWORKS (GANs)

Submitted by

Sayeda Samia Nasrin
Student ID: 16102083

Supervised by

Risul Islam Rasel
Assistant Professor of CSE

Department of Computer Science and Engineering
Chittagong Independent University
Chattogram,
Bangladesh

February, 2021

CANDIDATE'S DECLARATION

This is to certify that the work presented in this thesis, titled, "Generating Complex Henna Patterns using Generative Adversarial Networks (GANs)", is the outcome of the investigation and research carried out by me under the supervision of Mr. Risul Islam Rasel.

It is also declared that neither this project nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

.....

Name : Sayeda Samia Nasrin

Student ID : 16102083

Date : 25th February, 2021

CERTIFICATION

The thesis "Generating Complex Henna Patterns using Generative Adversarial Networks (GANs)", Submitted by ID No: 16102083, Session: Spring 2020, has been accepted as satisfactory in partial fulfilment of the requirement for the degree of Computer Science & Engineering, on 25th February 2021.

BOARD OF EXAMINERS

1. ----------
   Risul Islam Rasel
   Assistant Professor of CSE

2. ----------
   Atiqur Rahman
   Assistant Professor of CSE

3. ----------
   Md. Sajjatul Islam
   Assistant Professor and Head of CSE

4. ----------
   Dr. Aseef Iqbal
   Associate Professor and Dean of SSE

ACKNOWLEDGEMENT

All praise to Allah, the Almighty, for blessing me with the successful completion of this thesis. However, my heartfelt gratitude and appreciation also must be conveyed to the following people for their support and trust in me and making this dream into a reality.

I would like to express my sincerest gratitude to my supervisor Mr. Risul Islam Rasel, Assistant Professor, Department of Computer Science and Engineering (CSE), Chittagong Independent University, Bangladesh for his guidance, care and feedback.

I also would like to thank my father Mr. M Fajal Ahmed Haron, my mother Mrs Safika Yesmin Rashna, my younger brother Mr. Saiham Al Sabit, my elder sister in law Foujia Hossain and my husband Mr. Saad Bin Sajjad for their tremendous support during my thesis.

Chittagong Sayeda Samia Nasrin

February 2021

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ABSTRACT

Among all the living organisms on the Earth, mankind has flourished not owing to its strength, size or speed, but rather due to its intellect. And as a species which attempts to manifest and magnify its capabilities in its creations, it is inevitable that some of the creations will be imbued with intelligence. This quest of imparting intelligence into our creations have led to the inception and progress of Artificial Intelligence (AI). With the advent of powerful processors and techniques in a computer, arenas in AI have emerged which was only existent in fantasy in the past. One such arena is the generation of things like text or images using AI, and this was made possible with the research and development of deep neural networks, more specifically the brand-new class of neural networks known as Generative Adversarial Networks (GAN). While certain aspects of human activity such as game playing have been effectively been imitated and even improved in computers using AI, the concept of generation of much more creative endeavours such as producing meaningful text or abstract or finite artwork by computers were a far cry until recent times. Utilizing the unique nature of GANs, this is now possible. While different sorts of artwork and arts have been generated using GANs, traditional art or design patterns which have historically been tied to a particular segment of the demographics or geography have been largely left unexplored so far. In this regard, this work aims at the generation of Henna design patterns, which is a widely popular work of art involving complex creative designs in the Indian sub-continent and parts of Asia. The HennaGan introduced in this thesis shows that Deep Convolutional Neural Network (DCGANs) can be used to generate henna design images with variations effectively. It also creates the base for research into creative Henna art generation using DCGANs and provides insights into how the network parameters can be tuned to obtain a good result.

Chapter 1: Introduction

Since the inception of computers and computer systems, a major field of research has been AI. Although concepts such as neural networks were conceived by great computer-scientists decades before the explosive iterations of computation capacities owing to Moore's Law, it is only in recent times that these concepts have seen a resurgence in research and applications.

The utilisation of Graphics Processor Unit (GPU) in the domain of neural networks and machine learning has led to the immense growth in the adoption of existing AI techniques and the invention of new ones. Such a technique is the use of a deep neural network in the synthesis of artificial images that resemble the original images on which the network was trained on. The resemblance is to such a degree that the dedicated components in the network for detection of the artificially generated images can no longer differentiate between the original and the generated images. This class of neural networks is known as Generative Adversarial Networks (GAN).

1.1. Problem Statement

A lot of work in the Deep learning sector has been on prediction or classification or similar tasks. GAN is a relatively new Deep learning technique and the use cases of this technology are still being guessed and worked upon. There has been some work on generating images of human faces or animals and fake videos. There has also been some considerable work in the artistic sphere, especially in style transfer and art imitation. However, the amount of work done in generating patterned design or abstract design has been relatively few. Henna or Mehndi design is a huge sector of creative design arts in the world, especially in the Indian subcontinent.

Henna is a form of body art or temporary tattoo mostly applied on hands and feet. It is extracted from a plant called the Henna tree. Its scientific name is '*Lawsonia inermis'*. It has been in practice since time immemorial, especially in South-Asia and the Arab regions. In many South Asian countries like Bangladesh, India, Pakistan, during traditional wedding festivals Puja or Eid celebrations it is traditionally applied on the

hands. The decorations are usually complex patterns and may remain visible for several days after applied once.

Henna art is a complex form of art that requires very good skills to produce good work. Hence, it requires extreme concentration and attention to detail. And the number of designs that can be drawn is also infinite. Recent progress in AI has emboldened researchers and developers to try out works of art and literature using computers. However, the area of art remains a fertile area of investigation for AI. In this context, an investigation into techniques available for generating henna art without human effort is an interesting avenue of research and application.

The driving factor would be realizing the vision of an AI fully capable of generating Henna design patterns and arts on par with human henna designers, which is a highly sought out skill in society and the fashion industry. Maybe it would even generate new concepts of henna design which could provide new insights for the creative flourish of the human henna artists. This work thus aims at the understanding and generating of Henna design patterns using GAN.

1.2. Motivations

GANs have been used to generate text, sounds, or images when trained on a related dataset. Works of art have also been produced by specialised GANs. But the world of arts is huge, and as such, there remain multiple avenues of art categories yet to investigate in the context of their generation by computers. This work aims at bridging the gap between the cutting-edge neural network technique of GAN and the centuries old traditional art industry of Henna.

1.3. Goal of the Thesis

The objectives of this project are:

1. To prepare a manually curated dataset of henna designs on a single hand.
2. To train a deep neural network on the dataset.
3. To generate new Henna design patterns using GAN (Generative Adversarial Network).

4. To perform an evaluation.

1.4. Scope

- Creating a manual data set that consists of henna art images of relatively good quality which will be fed into the network. Covering all sorts of henna art is not feasible so instead, the focus will be on henna art on the hands.
- Generation of single-hand henna designs. As the data set is used for training consists only of single-hand henna images, generating similar images is within the scope.
- Qualitative analysis by human retrospection.

1.5. Structure of the Report

The rest of the document is organized as follows: In Chapter 2, Literature Review of previous and existing works are discussed. In Chapter 3, the Methodology is described. Chapter 4 is about the Result and Performance analysis. In Chapter 5, the Summary and the Future works which can be done are discussed.

Chapter 2: Literature Review

This section tries to give an overview of GANs and its derivatives, and how this development can usher in a new era of creativity by utilizing these techniques for generating art and literature.

2.1 GANs

The evolution of ANNs and more specifically CNNs have led to the inception of a brand new category of machine learning technique called GAN. It came into being following the research of a young scientist named Ian Goodfellow and his team in 2014 [1]. Since its introduction, it has revolutionized the ANN world with its unique ideas and applications in unsupervised, supervised, and reinforcement learning. GANs have two competing neural networks called the 'Generator' and the 'Discriminator' which compete against each other in a zero-sum game. The generative network generates an entity and the discriminator network evaluates it, so basically the two networks are adversaries of one another. This goes on till the generator generates such quality of candidates that are indistinguishable from the ones the discriminator was trained with. Only backpropagation techniques, forward propagation techniques, and dropout algorithms are used in GANs. Figure 1 shows the basic architecture of GANs.

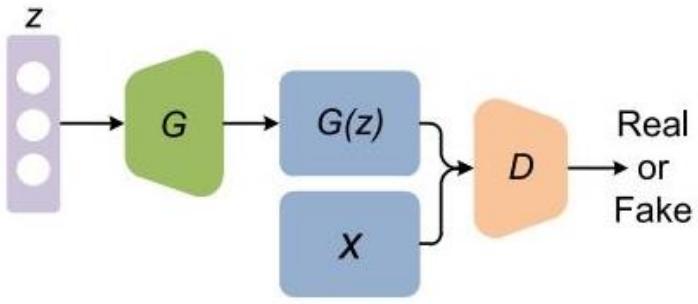

Figure 1: Basic Architecture of GAN [2]

2.1.1. Types of GANs

After the introduction of the original GAN, research and development into this particular ANN have progressed by leaps and bounds. Hence, numerous variations and implementations of the same exist today. A survey by Zhaoqing Pan et al. reveals the recent progress in this regard [2]. GANs have been broadly classified based on

Architecture optimization and objective function optimization. Further categorization based on convolutional, condition-based or auto-encoder based architecture is also possible.

Table 1: Classification of GANs [2]

Architecture Optimization Based GANs	Convolution based GANs	DCGAN
	Condition based GANs	CGAN; InfoGAN; ACGAN
	Autoencoder based GANs	AAE; BiGAN; ALI; AGE; VAE-GAN
Objective Function Optimization Based GANs	Unrolled GAN; f-GAN; Mode-Regularized GAN; AGE; VAE-GAN	

Table 1 presents the classificational overview of GANs. Figure 2 presents the architecture of select GAN types, namely CGAN, ACGAN and infoGAN. Figure 3 shows the generator and discriminator in action for training with real images of cats while trying to generate fake images of cats. The generator starts off with simple depictions which look nothing like a cat initially. But later on, after multiple feedbacks from the discriminator the generator improves upon the initial artwork and presents fake images which resemble the original images very closely, thus the discriminator cannot 'discriminate' among fake and real images anymore. This is shown in Figure 4.

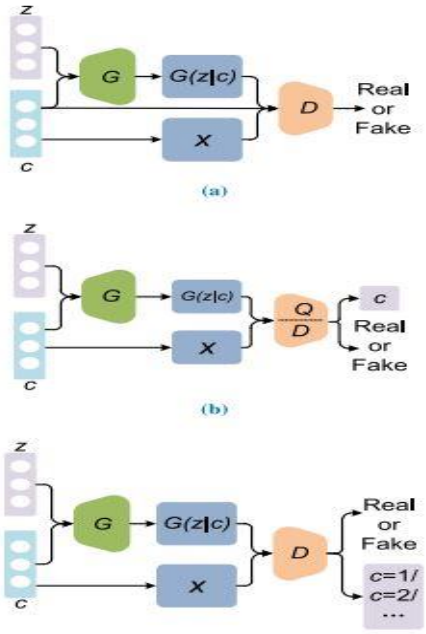

Figure 2: Architecture of (a) CGAN (b) infoGAN (c) ACGAN

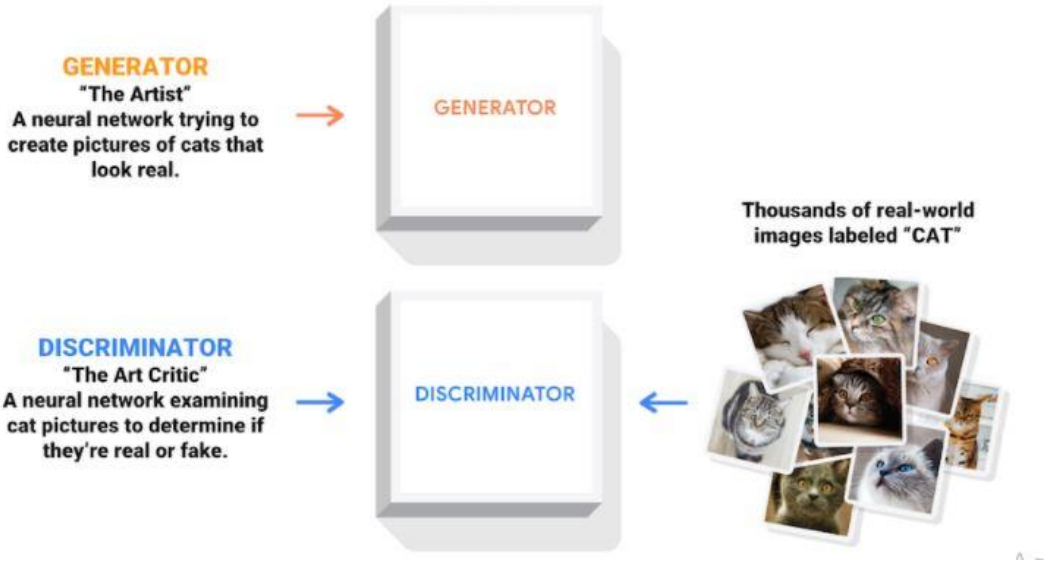

Figure 3: Generator and discriminator example in GAN [3]

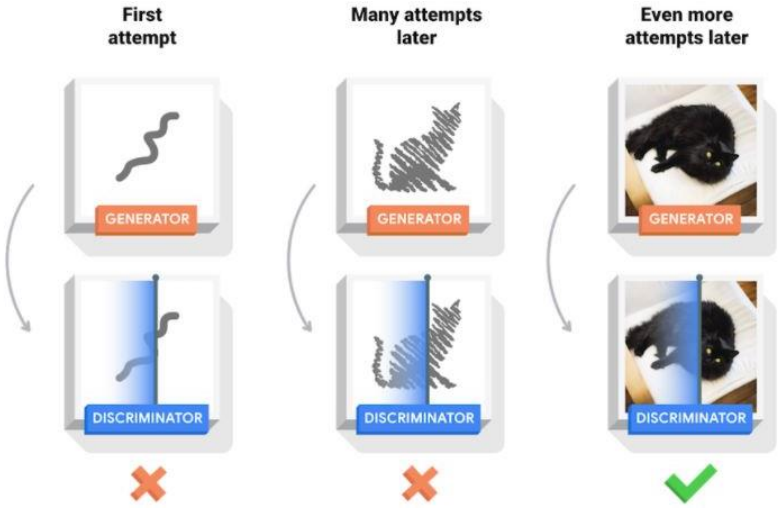

Figure 4: GAN in action [3]

2.2 Deep Convolutional Generative Neural Network (DCGAN)

Among the broad range of options available, CNN based DCGANs for unsupervised learning are found to be very capable in the synthesis of artificial text, images etc. DCGAN was the product of research into unsupervised learning with CNN [3]. The success of DCGANs lies in the simplicity of its implementation. Figure 5 demonstrates the architecture of the generator in DCGAN. All the max-pooling is replaced with convolutional strides. For up-sampling, transposed convolution is used. Fully

connected layers are eliminated, and batch normalization is used except for the output layer of the generator and the input layer of the discriminator. ReLU is used in the generator except for the output which uses tanh while LeakyReLU is used in the discriminator. Adam optimizer is used with tuned hyper-parameters.

The difference between GAN and DCGAN is explained in a study by Ian Goodfellow, one of the authors of the original GAN paper [5]:

• Use batch normalization layers in most layers of both the discriminator and the generator, with the two mini-batches for the discriminator normalized separately. The last layer of the generator and the first layer of the discriminator are not batch normalized so that the model can learn the correct mean and scale of the data distribution.

• The overall network structure is mostly borrowed from the all-convolutional Net. This architecture contains neither pooling nor “unpooling” layers. When the generator needs to increase the spatial dimension of the representation it uses transposed convolution with a stride greater than 1.

• The use of Adam optimizer rather than SGD with momentum.

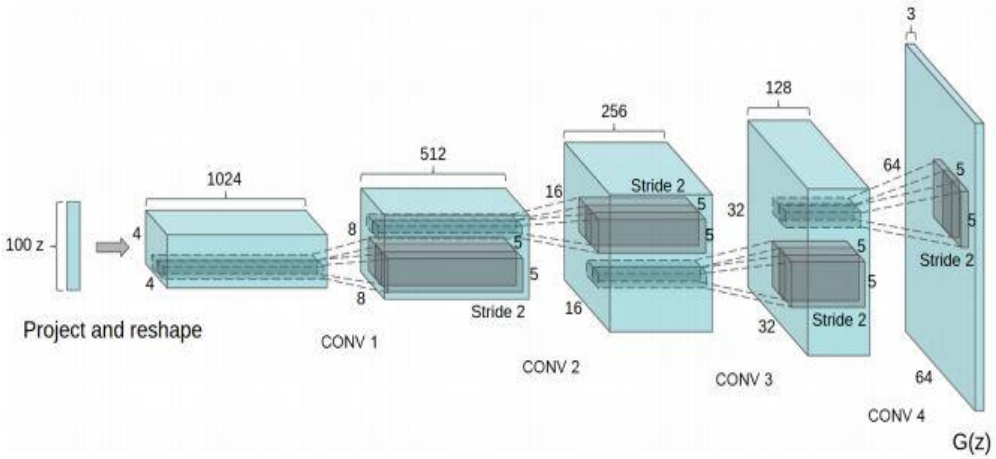

Figure 5: DCGAN Generator architecture [4]

2.3 Previous Work

After the introduction of GANs, there has been an immense interest in its research. A product of those endeavours has been DCGAN introduced by Alec Radford et al [3]. A lot of research into improving, adapting and modifying DCGAN is present across the machine learning research field as well. The original paper on DCGAN presented examples of generated images after networks were trained on the LSUN bedroom dataset, Imagenet-1K and a custom face dataset containing 3 million images of 10 thousand people collected from the internet. The DCGAN was further evaluated on the CIFAR-10 dataset even though it was not trained on it, just to show the domain robustness of the network. They also demonstrated vector arithmetic on the face samples. Figure 6 demonstrates some of the results of this operation from the paper.

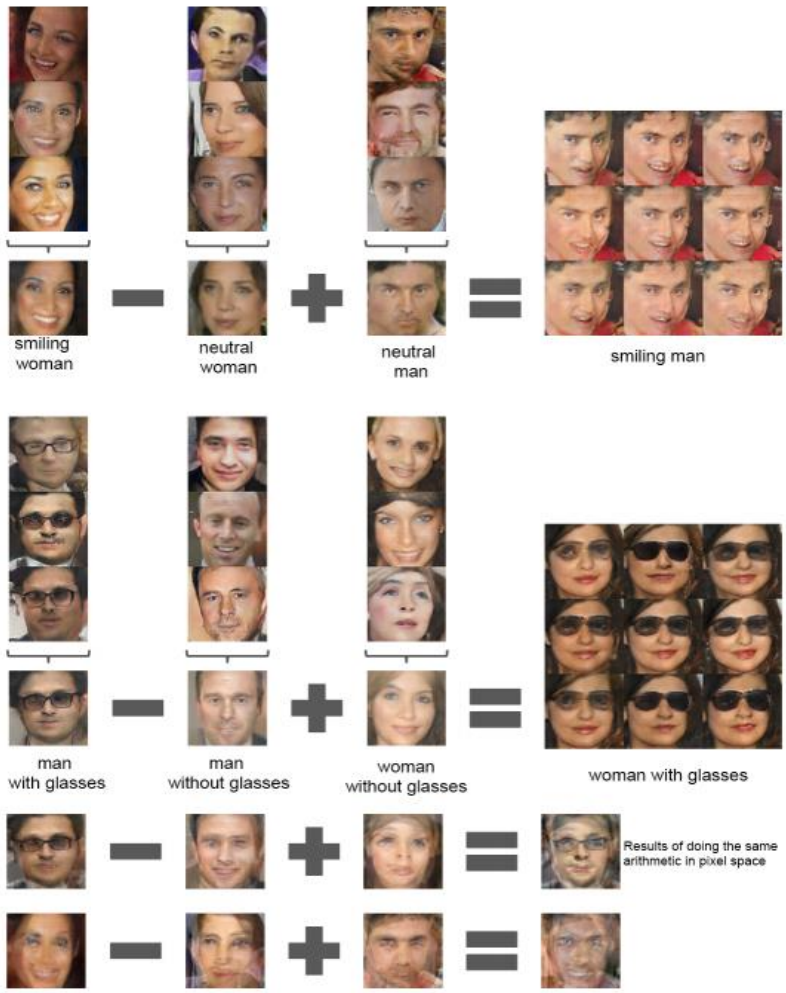

Figure 6: Vector Arithmetic on face dataset using DCGAN [3]

The creators of GAN and DCGAN teamed up in 2016 to produce a work which suggested techniques for improving the training of GANs [6]. The attempt was at providing several techniques for greater stabilization during training and proposed a metric for evaluation. They tried using semi-supervised learning instead of completely unsupervised learning and performed experiments on the MNIST, CIFAR-10, SVHN and ImageNet. Figure 7 demonstrates the visual difference in generated images using normal DCGAN and improved DCGAN. It clearly shows that using the improved techniques, the DCGAN can identify features in the animals such as noses, eyes etc. better.

Figure 7: Samples generated from the ImageNet dataset with normal DCGAN. (Left) Samples generated by an improved DCGAN (Right) [6]

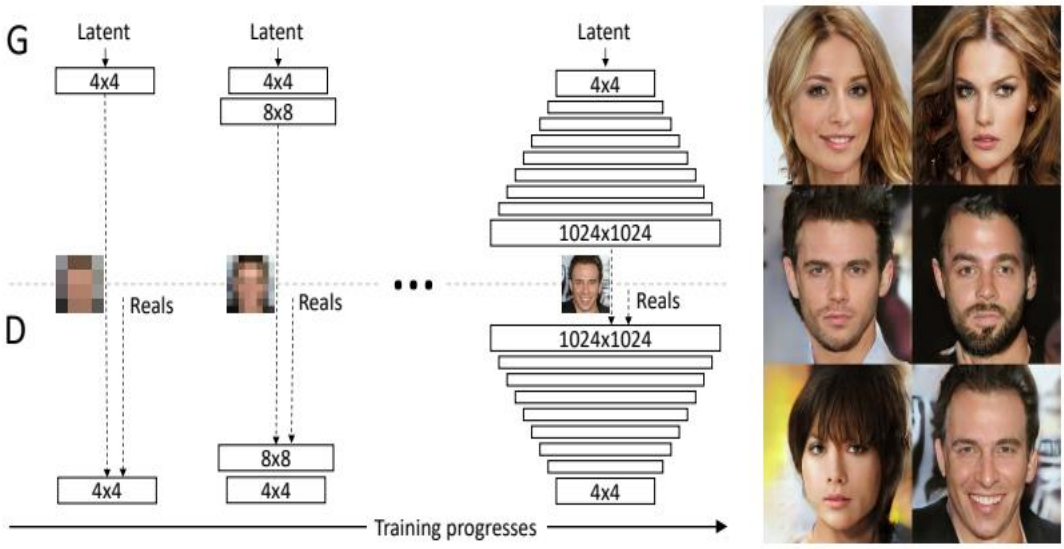

Figure 8: Progressively growth of GANS starting from 4x4 pixel images and generating 1024x1024 pixel images [8]

Figure 9: High-quality images generated using BigGAN [9]

Conditional GANs are another improvement on mainstream GANs which integrate the class labels of the data to improve the training [7]. This work was experimented on unimodal and multimodal networks.

Progressively Growing of GANs is another work which uses a multi-scale architecture to solve the problem where the GAN instability is increased due to target image resolution size [8]. This work allows the generator and discriminator to grow progressively starting from a low resolution to a high resolution. Figure 8 demonstrates how the GANs gradually builds up from a low 4x4 pixel resolution to 1024x1024 pixel resolution.

Another work which introduced a model called BigGAN which was trained on the ImageNet dataset of 128x128 pixel images uses a lot of components like self-attention, spectral normalization etc. [9]. Some of the results are shown in Figure 9. This model is a large-scale GAN and boasts of an impressive score on the Inception Score (IS) scale which was introduced in the earlier mentioned work for the improvement of GANs [6]. As because this model is a large-scale implementation, it is hard to implement it on a single machine.

GANs using the style transfer technique resulted in an architecture which was unsupervised, automatically learned and had separation of attributes in the resulting images. This model is termed as the StyleGAN [10]. StyleGAN was modified and used to create synthetic images of various items. Figure 10 shows a glimpse of the un-curated set of images that can be generated using StyleGAN when trained on a dataset containing human faces.

Figure 10: Uncurated set of Images generated by StyleGAN on the FFHQ dataset [10]

Figure 11: Abstract art images generated by GANGogh [11]

StyleGAN was modified and used to create synthetic images of various items. The model was named GANGogh [11]. Figure 11 shows some abstract art images generated using this model.

There have been many attempts to generate creative works using GAN. These works range from simple images to complicated artwork, from simple text to poetry, from simple clips to swap faces in a video. It is a highly prospective field which is in its early days of exploration and promises a lot of exciting use cases as well as the potential to rival human creators one day.

Figure 12: Portrait images by art-GAN [12]

Figure 13: African mask images generated using DCGAN [13]

DCGAN is one of the candidates in this sector. It has been used for generating artwork to solve the fault data generation problem [12]. Using DCGAN for generating artwork is an ongoing effort. Various works have been trying to utilise DCGANs for generating various sorts of artwork. Such a repository of DCGAN generated art is art-DCGAN which can generate various types of artwork based on categories [12]. Here the dataset was image-scrapped from wiki-art. Figure 12 shows some portrait images generated by art-GAN. There can also be other categories like landscape, abstract landscape etc.

Another work was focused on trying to generate a much more abstract form of artwork which is African masks. This model was trained on a dataset having images of historic African masks work by various tribes [13]. It had generated images with multiple resolutions. Figure 13 shows some 128x128 pixels images generated by this DCGAN model.

Figure 14: Some images from the original anime face dataset [14]

Figure 15: Generated anime faces [14]

Anime is a very popular hand-drawn computer animation which is originated from Japan. DCGAN was also utilised to try to generate faces styled like anime characters [14]. Figure 14 shows a snapshot of original anime faces from the training data set while Figure 15 shows some generated anime faces.

India is a very cultural place with the traditional and vibrant artistic ecosystem. One of the traditional artwork is 'Rangoli'. DCGAN has also been utilised to generate such Rangoli images [15]. Figure 16 shows a sample from the original dataset for Rangoli images. Figure 17 shows a portion of the output of DCGAN generated Rangoli images after 500 epochs of operation.

Figure 16: Some original images from the Rangoli dataset [15]

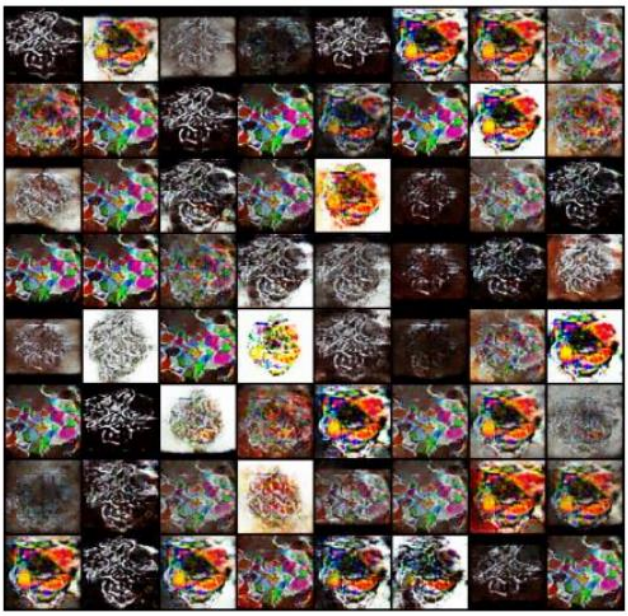

Figure 17: Fake Rangoli images generated using DCGAN. Snapshot after 500 epochs [15]

Besides GAN, a work based on Recurrent Neural Network (RNN) was done to generate MNIST images, CIFAR-10 images [16]. The architecture that was used is Deep Recurrent Attentive Writer (DRAW). This paper introduced DRAW neural network architecture for image generation. Figure 18 and 1 represents generated some outputs generated by this technique.

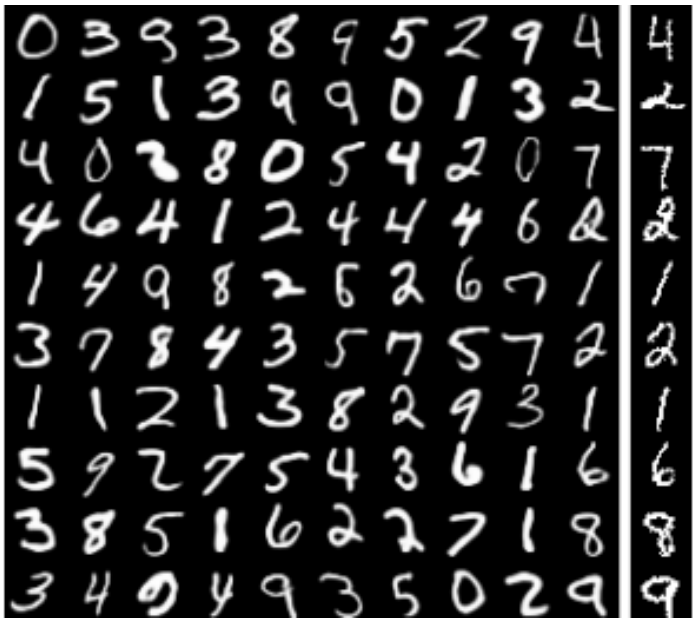

Figure 18: Generated MNIST images using RNN [16]

Figure 19: Generated CIFAR images using RNN [16]

A table of the summary of the papers used in the literature review has been given below:

Table 2: Summarized Literature Review

No	Name of the Paper	Dataset	Technique	Outcome
1.	Unsupervised representation learning with deep convolutional generative adversarial networks. [3]	LSUN bedroom dataset Imagenet-1K and a custom face dataset	DCGAN	Introduced DCGAN
2.	Improved Techniques for Training GANs [6]	MNISIST CIFAR-10 SVHN ImageNet	Semi-supervised learning instead Of completely unsupervised learning	Improved techniques for DCGAN
3.	Conditional Generative Adversarial Nets [7]	MNIST digit MIR Flickr	GAN with additional parameter y.	Introduced ConditionalGAN
4.	Progressive Growing of GANs for Improved Quality, Stability, and Variation. [8]	CIFAR10	Generator & Discriminator starts from low resolution and grows both progressively	Improved GANs method for Improved Quality, Stability and Variation
5.	Large Scale GAN Training for High Fidelity Natural Image Synthesis. [9]	ImageNet	GAN	Generated Image with high image resolution.
6.	A style-based generator architecture for generative adversarial networks. [10]	Flickr-Faces-HQ	GANs with style transfer technique	Introduced styleGAN
7.	RangoliGAN: Generating Realistic Rangoli Images using GAN [15]	Custom made dataset of 1079 images	DCGAN	Generating Rangoli Images
8.	Draw: A recurrent neural network for image generation [16]	MNIST CIFAR10	RNN	Introduces RNN based draw neural network for image generation

Chapter 3: Methodology

This section describes the methodology used in the implementation of HennaGAN.

3.1 Dataset

For any deep learning task, however trivial it is, there must be a dataset which contains items on which the neural network is supposed to be trained on. The case for using DCGAN for generating henna designs is no different, which means there is a requirement of a dataset of henna image designs the likes of which we want to reproduce with newer designs.

3.1.1. Dataset Collection

Henna is traditionally applied on the hands, most of the women in South Asia and the Arab countries, mostly on the eve of various occasions like marriage ceremonies. Henna is also applied to other parts of the body, but for this work, the focus is on henna designs on a single hand. Therefore, the dataset must contain a decent amount of images with henna designs on the hand. For this purpose, initially, 10000 images were Figure 20 represents a portion of the original images collected.

Figure 20: A portion of the original images without the background removal

The duplicates were discarded using libraries of Python and using a technique known as image hashing. Image hashing employs a hashing function on an image and this generates a unique image fingerprint. The fingerprints are generated for each image in the dataset and the images with the same fingerprints are discarded, which eliminates all duplicates.

3.1.2. Dataset Curation

Although the 1915 images collected had no objects, text or other irrelevant designs on the hands themselves, they still had a lot of background noise. This had to be removed or reduced. The first attempt was using a Python script with the algorithm 'GrabCut', which is a technique from OpenCV to remove the background from images [17]. However, the results were not satisfactory as GrabCut employs a rectangular shape to define the area of interest and removes the background which is within the boundaries of that rectangle, but the individual elements of interest had to be marked with a separate colour. This was a very time-consuming process and was unsuitable to employ for a fairly large dataset of more than 1000 images.

Figure 21: A portion of training images with background removed

The second attempt was employing Adobe Photoshop to remove the background. This fared relatively well and even better than the GrabCut algorithm. Around 1000 images from the selected 1915 were curated using this technique. However, using Adobe Photoshop had a steep learning curve and, on average, more than 10 minutes were required to curate a single image and remove the background. Hence, this method was deemed too slow.

The third attempt at curating the dataset was done using the website 'remove.bg' [18]. This website provides a very intuitive drag and drop interface for uploading images. It then employs AI to remove the background from the image. However, as with the case of all AI-based software, the results were not perfect and some of them had to be further curated to remove little fragments of the background which remained. The remaining 915 images were curated with this online tool. Figure 21 shows a portion of the curated dataset.

3.2 Architecture

Figure 22 depicts the high-level view of the architecture of HennaGAN. There are mainly two neural networks involved in the process, the Generator network and the Discriminator network.

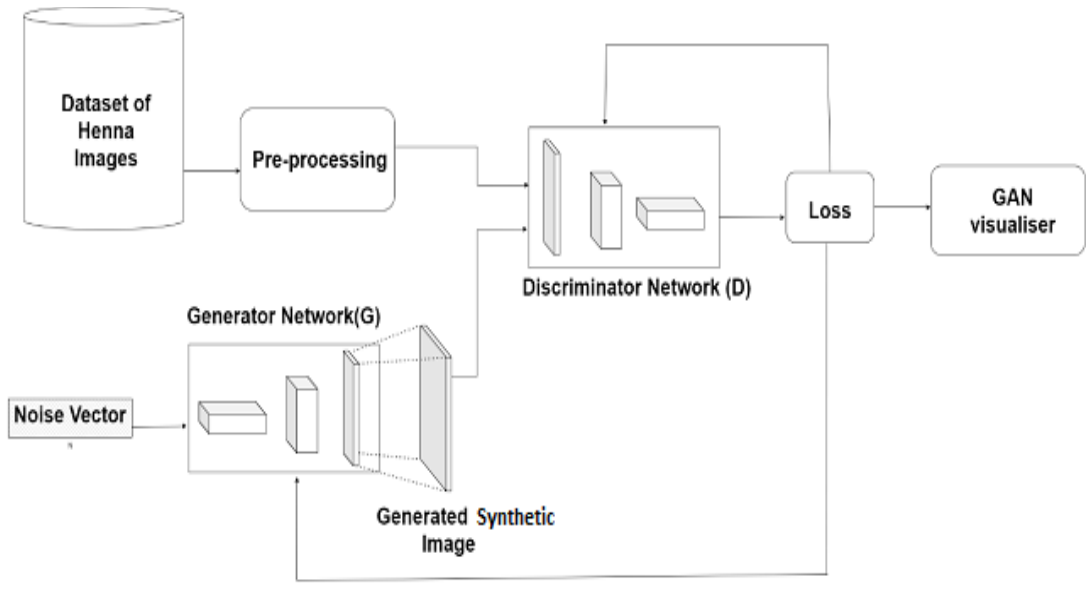

Figure 22: Overall Architecture of HennaGAN

There is the dataset which stores the original images of hands containing henna design patterns on them. They are pre-processed before being used in the training of the networks. There are many techniques used in the architecture such as loss functions, activation functions, custom weight initialization etc. These individual components and methods are further described in the upcoming sections. Finally, a visualizer shows a demonstration of the evolution of the generated images from the beginning until the very end when the zero-sum game between the generator and the discriminator stops.

In the data set of henna images, there are 1915 manually curated images. This data set goes through a preprocessing step in which the images are resized, centre-cropped, transformed into tensors and normalized. The discriminator is to be trained on this data set.

A vector with the dimension of 100 will be passed to the Generator. The vector is known as the noise vector. Then it uses convolutional network transpose 2d and upscales the image to 1024 channels and then 4 by 4 pixels. After that 8 by 8 pixels with 512 channels then it keeps doing until it obtains 64 by 64 and 3 RGB channels. Hence it generates a random image of 64-pixel size. After the generation of the image, it will be evaluated by the discriminator.

The task of the discriminator is to try to find out whether the fake images are close enough to the original images or not. For this, a probability distribution is used, that is the distance in a metric which shows how far or near the generated images are from the original ones. While evaluating the generated image using the probability distribution discriminator uses a loss function. Both the discriminator and the generator can then learn from this loss function.

From the original GAN paper, the GAN loss function is defined as the following:

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{x\sim p_z(z)}[\log(1 - D(G(z)))] \ldots \text{(i)}$$

If ‘x’ represents the image, ‘D(x)’ is the discriminator network that outputs a scalar probability that indicates whether ‘x’ originated from the training data or the generator.

'D(x)' is high when 'x' comes from the training data and low when it comes from the generator.

For the generator 'G(z)', 'z' is a latent space vector sampled from a standard normal distribution. The aim of 'G' is to approximate the distribution of the training data 'P_{data}' to generate the fake samples from that approximated distribution 'P_g'.
Therefore, '*D(G(z))*' is a scalar which is the probability that the generator outputs the real image. The discriminator and generator play a zero-sum game where the generator tries to minimize the probability that the discriminator will predict the output as a fake '*(log (1- D(G(z)))',* while the discriminator tries to maximize the probability that it detects the outputs correctly *'(logD(x))*'.

Theoretically, the end to this min-max game is when the discriminator guesses correctly and '$P_g = P_{data}$'. However, this is an active area of research and the convergence of GANs are difficult to achieve. So after evaluating the generated image discriminator pass a loss function.

3.3 Generator

In simple words, a generator can be thought of an artist who tries to reproduce artwork as close as possible to the original artworks of an artist. It converts the latent space vector to the data space. It generates images of the same size as the training images, along with the RGB colour channels. Generators were used for 32x32, 64x64 and 128x128 pixel images. The following code shows the generator class for 64x64 image size. The 'nc' is the number of colour channels, which is 3 in this case for RGB images, 'ngf' is the size of feature maps which are propagated through the generator and 'nz' is the latent space vector.

Generator learns from the feedback provided by the discriminator and tries to improve the generation of images. This generator has 4 segments sequentially each of which consists of a transposed convolutional layer ('nn.ConvTranspose2d'), a batch normalization layer ('nn.BatchNorm2d') and a rectified linear unit ('nn.ReLU').

```
1. # Generator Code
2. class Generator(nn.Module):
3.     def __init__(self, ngpu):
4.         super(Generator, self).__init__()
5.         self.ngpu = ngpu
6.         self.main = nn.Sequential(
7.             # input is Z, going into a convolution
8.             nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),

9.             nn.BatchNorm2d(ngf * 8),
10.            nn.ReLU(True),
11.            # state size. (ngf*8) x 4 x 4
12.            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False
   ),
13.            nn.BatchNorm2d(ngf * 4),
14.            nn.ReLU(True),
15.            # state size. (ngf*4) x 8 x 8
16.            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=Fals
   e),
17.            nn.BatchNorm2d(ngf * 2),
18.            nn.ReLU(True),
19.            # state size. (ngf*2) x 16 x 16
20.            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),

21.            nn.BatchNorm2d(ngf),
22.            nn.ReLU(True),
23.            # state size. (ngf) x 32 x 32
24.            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
25.            nn.Tanh()
26.            # state size. (nc) x 64 x 64
27.        )
28.
29.    def forward(self, input):
30.        return self.main(input)
31.
32.   #Create the generator
33.    netG = Generator(ngpu).to(device)
34.    if (device.type == 'cuda') and (ngpu > 1):
35.      netG = nn.DataParallel(netG, list(range(ngpu)))
36.
37.    netG.apply(weights_init)
38.    Print(netG)
```

Figure 23: Generator Code

Batch normalization is used to make the neural network faster and more stable. The ReLu function is the most widely used function in neural networks. It enables the input to propagate as the output without any change in its value if it is positive, otherwise outputs zero. Which means, no negative values can pass through a ReLu activation function. It has an additional transposed layer at the end just before the closing activation function, a hyperbolic tangent function (‘nn.tanh’). Using this activation function aids better in capturing image features like the treatment of dark and light colours equally.

Then the generator is instantiated and the weights are initialized. Finally, the model can be printed to observe how the generator is structured.

3.4 Discriminator

Discriminator can be thought of a CNN based binary classifier, which classifies the generated images by the generator as either real or fake. It gives a scalar probability as output. The Discriminator were also used for 32x32, 64x64 and 128x128 pixel images. The given code portrays the discriminator class which takes input image of 64x64 pixels with 3 channels to distinguish between real and fake. The parameter 'ndf' sets the depth of feature maps propagated through the discriminator.

```
1. class Discriminator(nn.Module):
2.     def __init__(self, ngpu):
3.         super(Discriminator, self).__init__()
4.         self.ngpu = ngpu
5.         self.main = nn.Sequential(
6.             # input is (nc) x 64 x 64
7.             nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
8.             nn.LeakyReLU(0.2, inplace=True),
9.             # state size. (ndf) x 32 x 32
10.            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
11.            nn.BatchNorm2d(ndf * 2),
12.            nn.LeakyReLU(0.2, inplace=True),
13.            # state size. (ndf*2) x 16 x 16
14.            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
15.            nn.BatchNorm2d(ndf * 4),
16.            nn.LeakyReLU(0.2, inplace=True),
17.            # state size. (ndf*4) x 8 x 8
18.            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
19.            nn.BatchNorm2d(ndf * 8),
20.            nn.LeakyReLU(0.2, inplace=True),
21.            # state size. (ndf*8) x 4 x 4
22.            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
23.            nn.Sigmoid()
24.            )
25.
26.    def forward(self, input):
27.     return self.main(input)
28.
29.    #Create the discriminator
30.    netD = Discriminator(ngpu).to(device)
31.    if (device.type == 'cuda') and (ngpu > 1):
32.      netD = nn.DataParallel(netD, list(range(ngpu)))
33.    netD.apply(weights_init)
34.
35.    print(netD)
```

Figure 24: Discriminator Code

The discriminator is then processed by 3 sequential segments similar to the generator which are convolutional layer (‘nn.ConvTranspose2d’), a rectified linear unit (‘nn.LeakyReLU’) and a batch normalization layer (‘nn.BatchNorm2d’). Leaky ReLu is used to help the gradients flow easier. The output of leaky ReLu is positive for the positive inputs and for the negative input the output is a controlled negative value, controlled by a parameter called alpha. Finally ends with 4^{th} segment that is sigmoid function which generates output as a probability.

Then the discriminator is finally created and initialized with weights initialization function. Also, like the generator model, it can be printed to observe how the discriminator is structured.

3.5 BCELoss

The discriminator and the generator must learn through loss functions and optimizers. The Binary Cross-Entropy (BCEloss) loss function is used. It has the log components of both the discriminator and the generator. It is defined as:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = -[y_n . \log x_n + (1 - y_n) . \log(1 - x_n)] \dots \text{(ii)}$$

Gradients are numeric calculations which allow to adjust parameters of a network, or in other words, it is used to update weights in a neural network. And loss is used to calculate the gradients. Loss functions are used to calculate this loss. BCE loss is used for binary classification tasks. The discriminator is also a binary classifier. The real label is defined as 1 while the fake label is set to 0. These labels are useful for the calculation of the losses of both the discriminator and the generator. Adam optimizer is used for both the networks with modifiable learning rates and beta1 hyperparameters. Using separate learning rates for the discriminator and the generator is known as Two Time-Scale Update Rule (TTUR). The training loop will be explained in the implementation chapter, where the learning rate, input noise, optimizer and loss functions are used.

3.6 Hyper Parameters

Hyperparameters mean such parameters in terms of machine learning which control the learning process. In this work, the hyperparameters are epochs, batch size, learning rate for training, image size, length of the latent vector, use of gpu or cpu indicator and the depth of feature maps for the discriminator and generator. GANs are very sensitive to hyperparameter changes.

Table 3: Vital hyperparameters used for HennaGAN

Epoch	100, 300, 500, 1000	
Image Size	32x32, 64x64, 128x128	
Batch Size	32, 64, 128, 256, 512	
Learning Rate	Without TTUR	0.0002, 0.0005
	With TTUR	d : 0.0002 & g : 0.002, d : 0.0002 & g : 0.0005

3.7 Workflow

The workflow followed will be as follows:

1. The curated data set is loaded
2. The networks are built and compiled
3. The discriminator network is trained
4. The generator network is trained
5. The two networks are engaged in a zero-sum game till they converge

Then the zero-sum game is started. Ian Goodfellow demonstrates the game which is depicted in Figure 25.

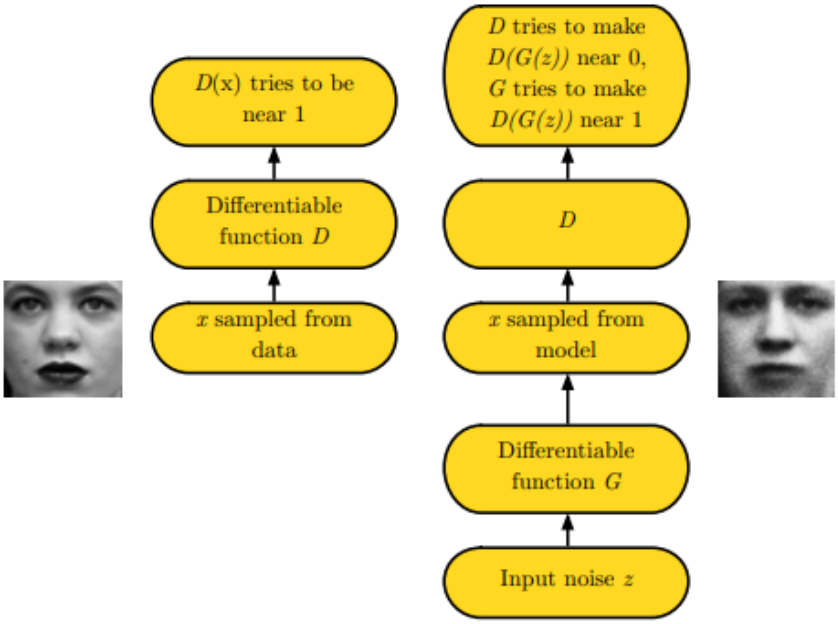

Figure 25: Zero-sum game in GANs [11]

The training follows the Algorithm 1 listed in the paper for improvements on the original GAN architecture with some modifications to tune it for better results. Figure 26 shows the pseudocode for the proposed Algorithm 1. This algorithm tries to converge using the GAN minimax equation which is explained earlier in section 4.5.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

Figure 26: Pseudocode for Algorithm 1 in the paper [2]

Chapter 4: Result & Performance Analysis

Following the workflow steps and implementation details, several experimental runs were conducted. Each run had a different configuration. The hyperparameters were tuned to find the best result. The dataset and the environment were the same for all the runs. Below, the runs are described.

4.1 Experimental Runs

The total number of experimental runs with the variable hyper-parameters are listed below in Table 4. In the ‘Learning Rate’ column‘d’ stands for the discriminator and ‘g’ for the generator.

Some of the runs listed in the table are described below. Each of the descriptions has the graph which shows the plotting of discriminator and generator losses versus the number of iterations. The issue with GANs is that they are overly sensitive to hyper-parameter changes. Sometimes, the losses are not very intuitive. As the two networks compete against each other, improvement in one often means the loss is higher for the other. Intuitively, when the losses converge, the images generated are expected to be better in quality. However, this is not always the case. The quality of images thus must be determined from the observation of both the loss functions and actual observation.

Table 4: Experimental Runs

No	Epoch	Image Size	Batch Size	Learning Rate
1	100	32	64	0.0002
2	100	64	64	0.0002
3	100	64	64	0.0005
4	100	64	64	d-0.0002,g-0.002
5	100	64	64	d-0.0002,g-0.0005
6	100	128	64	0.0002
7	300	32	64	0.0002
8	300	64	32	0.0002
9	300	64	64	0.0002
10	300	64	128	0.0002
11	300	64	256	0.0002
12	300	64	512	0.0002
13	300	128	64	0.0002
14	300	64	64	0.0005
15	300	64	64	d-0.0002,g-0.002
16	300	64	64	D-0.0002,g-0.0005
17	500	64	32	0.0002
18	500	64	64	0.0002
19	500	64	128	0.0002
20	500	64	256	0.0002
21	500	64	512	0.0002
22	1000	64	64	0.0002

4.2 Runs with the different image sizes

The following attempts are made with variable image size while keeping the other hyper-parameters constant except the epochs. The hyper-parameter learning rate is 0.0002 and the beta1 for Adam optimizer 0.5 as suggested by the DCGAN paper.

4.2.1. Image size 32x32 pixel, 100 epochs

Qualitative analysis: The first attempt was to try with images having the size of 32x32 pixels. The training was run for 100 epochs. From the generated images as seen in Figure 27, the hands are neither intelligible nor are the designs on them. There are also images with are just random colours without any definite shape.

Quantitative analysis: The graph for the discriminator and the generator does not coincide or intersect at any point in time during the training. There are brief overlaps after roughly 1200 iterations but as no real convergence between the two networks whatsoever is observed in Figure 28, the generated images have almost no resemblance to the training set of images.

Figure 27: Comparison of real versus generated images for 100 epochs, image size 32 pixels and learning rate 0.0002

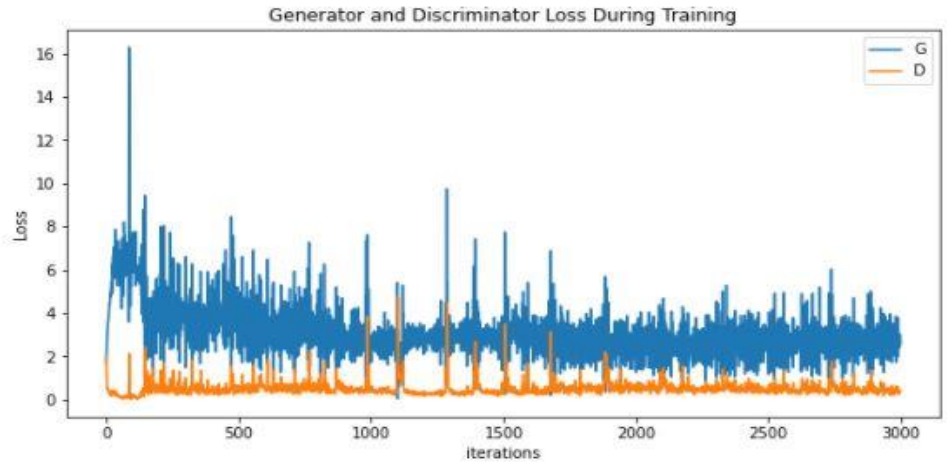

Figure 28: Graph for loss comparison for 100 epochs, image size 32 and learning rate 0.0002

4.2.2. Image size 32x32 pixel, 300 epochs

Figure 29: Comparison of real versus generated images for 300 epochs, image size 32 pixels and learning rate 0.0002

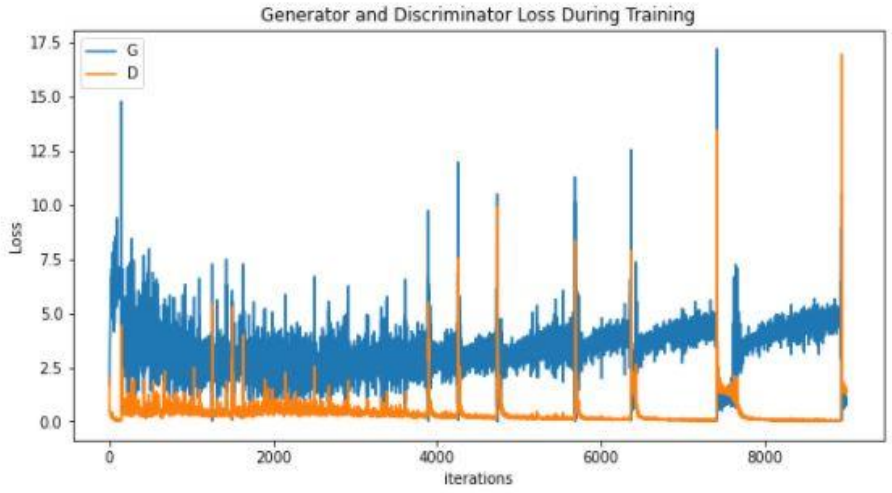

Figure 30: Graph for loss comparison for 300 epochs, image size 32 and learning rate 0.0002

Qualitative analysis: The same model was trained for 300 epochs. The results as shown in Figure 29 are slightly better than the previous figure. But still, some images have disfigured hands and no designs are visible clearly. So overall, this configuration cannot be considered ideal.

Quantitative analysis: The graph shows the networks converge at around 7800 iterations and there are also sudden coincident points before this intersection. The

convergence is rather abrupt. The generator loss suddenly drops below 1 and coincides with the suddenly rising discriminator function. After the convergence the loss is restored to values which are similar to the previously observed values. This behaviour can be termed as irregular.

4.2.3. Image size 64x64 pixel, 100 epochs

Qualitative analysis: Figure 31 portrays result after 100 epochs for 64x64 pixels which came out marginally better than the 32x32 pixels for the same epochs. This resolution is widely used for DCGAN implementations.

Figure 31: Comparison of real versus generated images for 100 epochs, image size 64 pixels, batch size 64 and learning

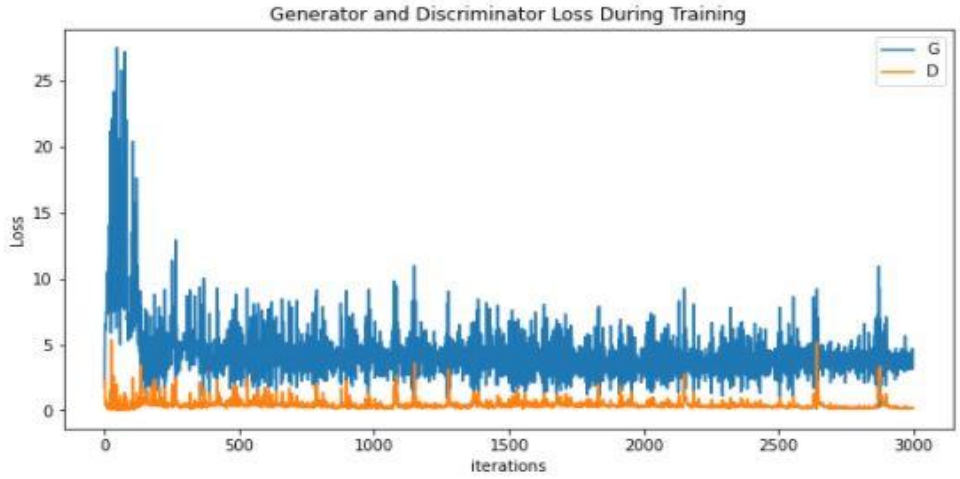

Figure 32: Graph for loss comparison for 100 epochs, image size 64 and learning rate 0.0002

Quantitative analysis: Figure 32 shows the generator loss versus the discriminator loss graph for this configuration. The bounce at the beginning suggests the model tries to improve it. Although the networks do not converge, from the qualitative analysis it can be seen that the generated images were better that the run with the same configuration but with resolution 32x32 pixels.

4.2.4. Image size 64x64 pixel, 300 epochs

Qualitative analysis: As can be observed in Figure 33, the quality of the generated images worsened with 300 epochs than with 100 epochs.

Figure 33: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 64 and learning rate 0.0002

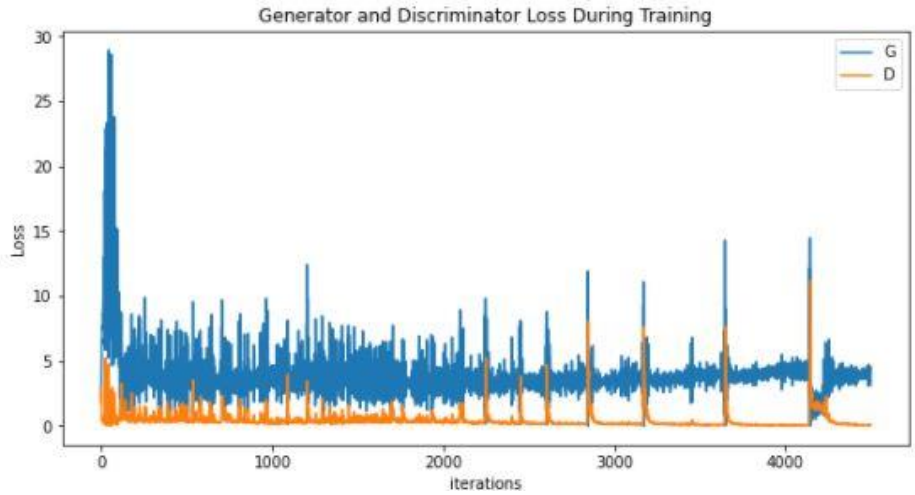

Figure 34: Graph for loss comparison for 300 epochs, image size 64 and learning rate 0.0002

Quantitative analysis: Figure 34 shows the graph for this configuration. It is a lot similar to the graph of the network with the same configuration at 32x32 pixels. The generator and discriminator losses coincide heavily after 4000 iterations, but then immediately branch out in opposite directions.

4.2.5. Image size 128x128 pixel, 300 epochs

Qualitative analysis: Figure 35 displays output for training on the resolution of 128x128 pixels. Here a significant number of images have finger shape. Some of them have shades of henna design as well. Although the designs are not very well defined. But it is a drastic improvement over the model with the same configuration but with resolution 64x64 pixels.

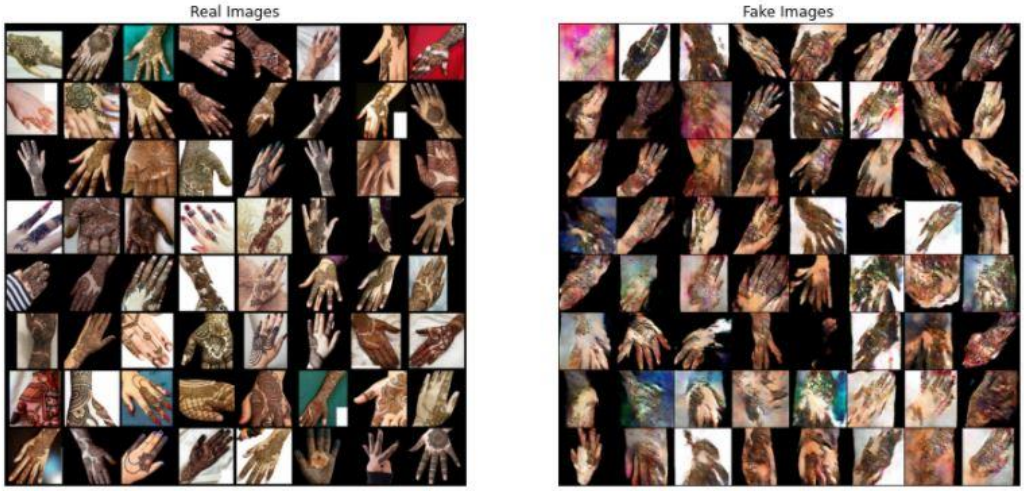

Figure 35: Comparison of real versus generated images for 300 epochs, image size 128 pixels, batch size 64 and learning rate 0.0002

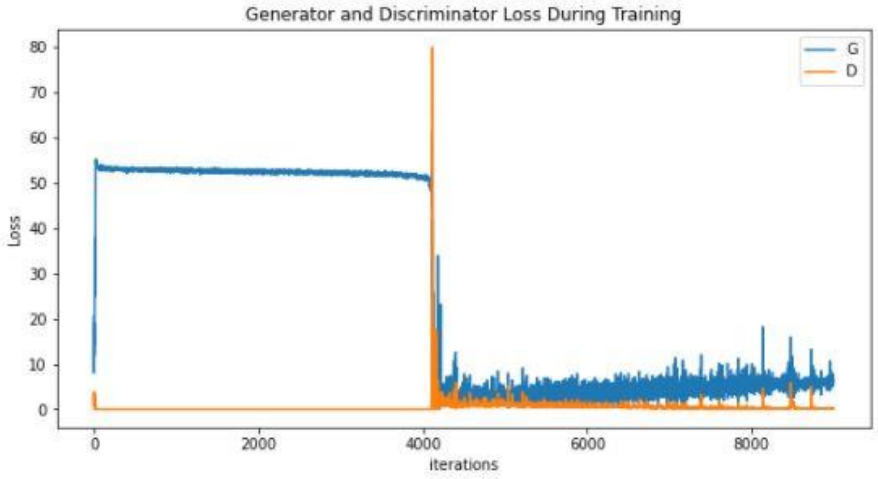

Figure 36: Graph for loss comparison for 300 epochs, image size 128 and learning rate 0.0002

Quantitative analysis: If we look at the graph of this output at figure 36, discriminator and generator didn't meet for 4000 iterations but after that, we have an overwhelming change. The discriminator is continuously at 0 for 4000 iterations while the generator is fixed till 4000 iterations. The networks perform best at immediately after 4000 iterations and then the discriminator stays close to zero, indicating convergence failure.

4.3 Runs with the different batch sizes

For the following runs under this sub-section, the batch size is different, images are 64x64 pixels and the training was run for variable epochs, while the learning rate is constant at 0.0002.

4.3.1. Batch size 32, 300 epochs

Qualitative analysis: The results obtained for 300 epochs are shown in Figure 37 and the graph of the losses in Figure 36. While some of the generated images seem to have generated hands properly, the henna designs themselves are quite ambiguous. Furthermore, some of the hands have more fingers or are completely distorted.

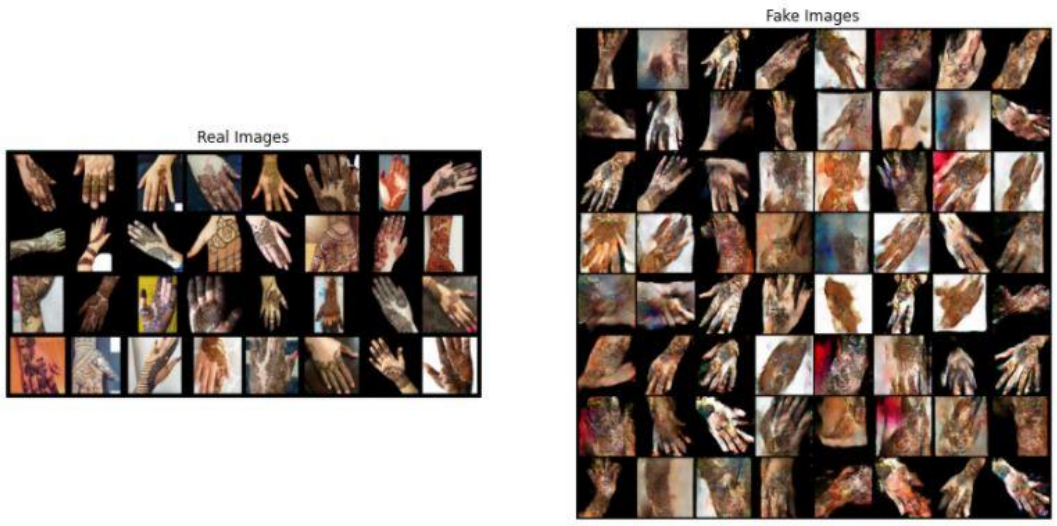

Figure 37: Comparison of real versus generated images for 300 epochs, batch size 32 and learning rate 0.0002

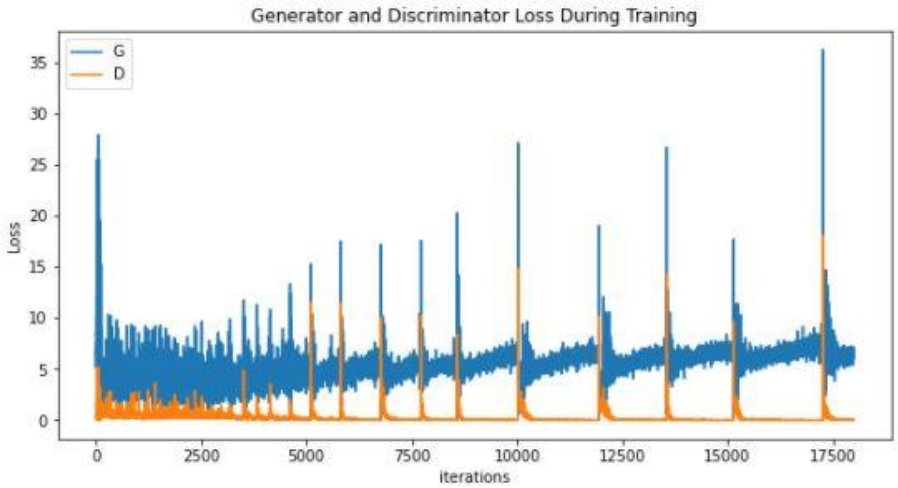

Figure 38: Graph for loss comparison for 300 epochs, batch size 32 and learning rate 0.0002

Quantitative analysis: The graph from Figure 38 shows the generator loss gradually decreasing but with regular spikes in values. At the same values, the discriminator loss also seems to spike in tandem. For the majority of the time, the discriminator loss is zero, which indicates convergence failure, possibly due to mode collapse.

4.3.2. Batch size 64, 100 epochs

Qualitative analysis: Figure 39 shows some significant shapes with design which were generated after running the network for 100 epochs with a batch size of 64. That is, each time, 64 samples were used at a time. Some of the images have no meaningful shape and design, while some have design which is wrongly in the background, while others have a lot of noise.

Figure 39: Comparison of real versus generated images for 100 epochs, batch size 64 and learning rate 0.0002

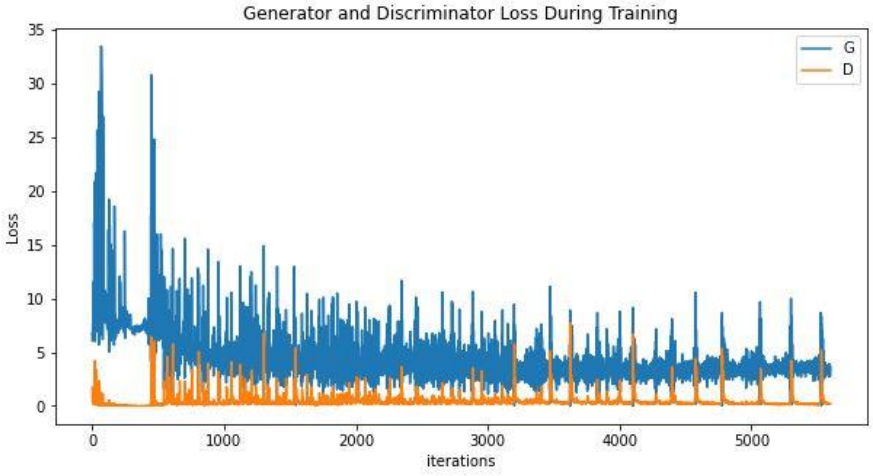

Figure 40: Graph for loss comparison for 100 epochs, batch size 64 and learning rate 0.0002

Quantitative analysis: In the graph in Figure 40 at first the generator and the discriminator starts at a distance from each other. There are two massive spikes in the generator loss function. Eventually the distance which was existent between the generator and the discriminator is minimized and also later intersects at some points. But the discriminator loss more or less stayed towards zero, resulting in convergence failure.

4.3.3. Batch size 64, 300 epochs

Figure 41: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 64 and learning rate 0.0002

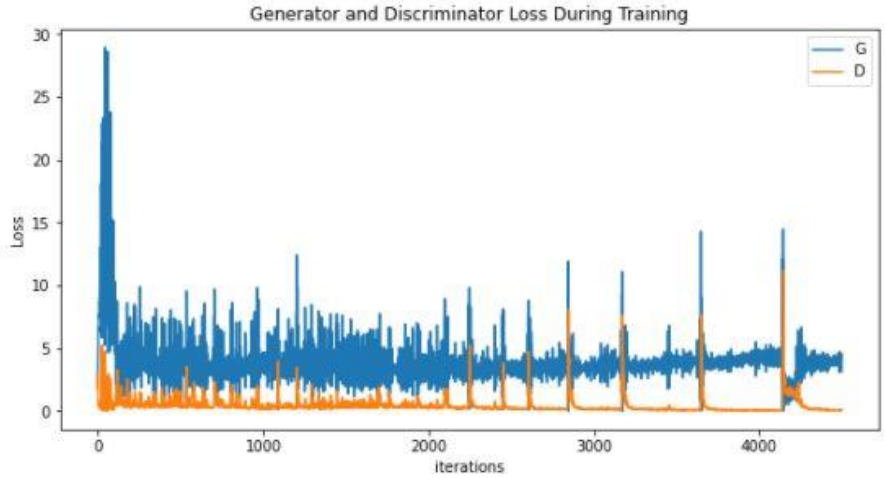

Figure 42: Graph for loss comparison for 300 epochs, batch size 64 and learning rate 0.0002

Qualitative analysis: The result after 300 epochs for the same batch shows that the number of defined shapes increased and undefined shapes decreased. Although there are also some images with a lot of noise in the background.

Quantitative analysis: As with previous runs with 300 epochs, the graph depicted in Figure 42 is also similar. The loss functions start out separately from one another, however, the networks coincide suddenly after 4000 iterations. But then the discriminator loss falls to zero which indicates convergence failure. The generator loss again increases and maintains a steady course after the convergence.

4.3.4. Batch size 64, 500 epochs

Qualitative analysis: The outcome after 500 epochs for batch 64 shows in figure 43. There are a few proper shaped hands but the rest are not in proper shape. The designs are rather not intelligible at all.

Quantitative analysis: Figure 44 depicts the generator and discriminator loss for this configuration. The discriminator loss more or less remains at zero with regular spikes and coincidences with the generator loss at regular intervals after a coincidence at around 5000 iterations. This means running for 500 epochs is not very helpful in generating good images.

Figure 43: Comparison of real versus generated images for 500 epochs, image size 64 pixels, batch size 64, and learning rate 0.0002

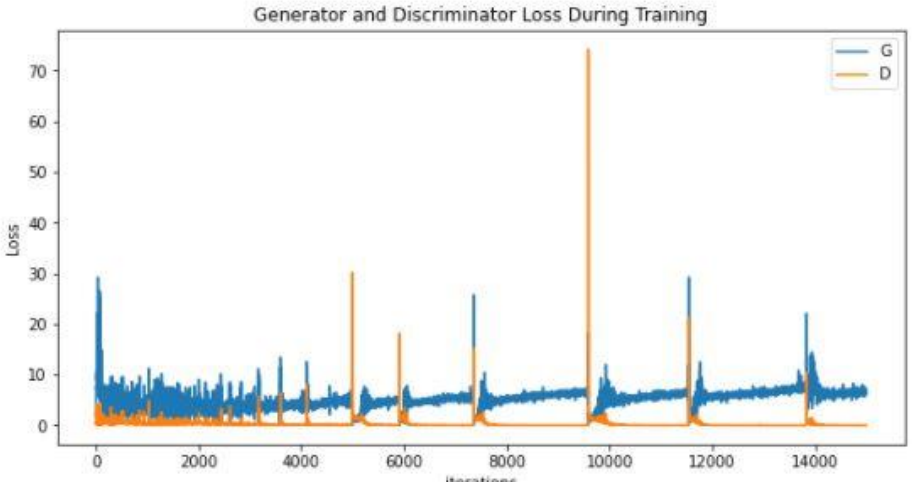

Figure 44: Graph for loss comparison for 500 epochs, batch size 64 and learning rate 0.0002

4.3.5. Batch size 64, 1000 epochs

Qualitative analysis: After 1000 epochs the result came out as over-fit. Some of the images seem to be like a hand but the rest are not even close to a handshape. The results are a result of overfitting and it is safe to say running this network for 1000 epochs or more will not generate any quality images.

Quantitative analysis: Figure 46 shows that the graph is almost identical to the previous graph with 500 epochs. As the model over fits, this graph warrants no further interesting discussion.

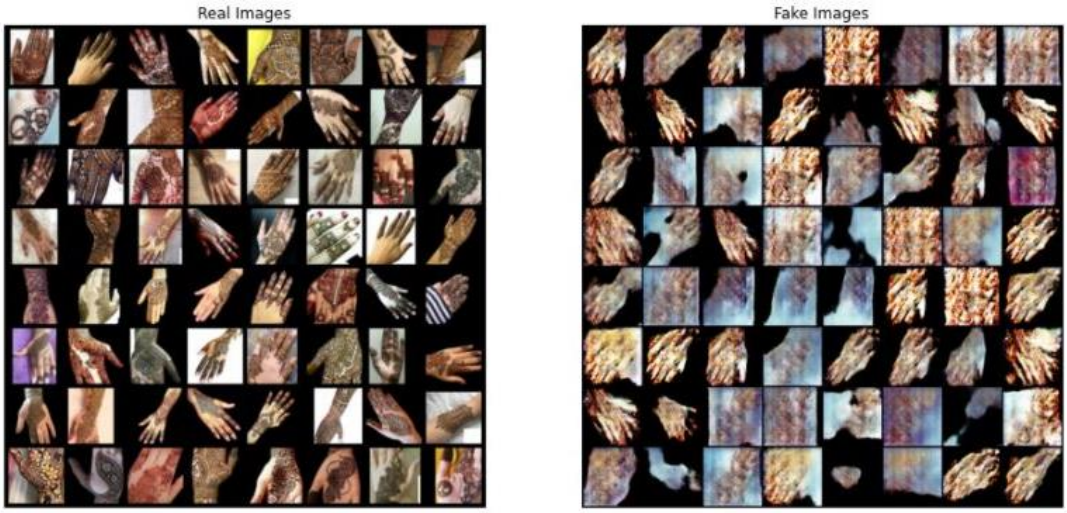

Figure 45: Comparison of real versus generated images for 1000 epochs, image size 64 pixels, batch size 64 and learning rate 0.0002

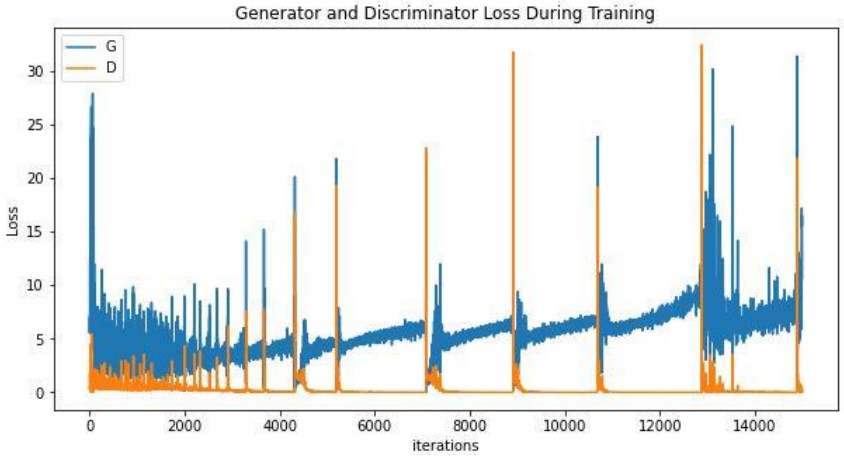

Figure 46: Graph for loss comparison for 1000 epochs, batch size 64 and learning rate 0.0002

4.3.6. Batch size 128, 300 epochs

Qualitative analysis: After 300 epochs for batch 128 the generated result shows some nicely generated hands with some of the designs visible as well. But there are also some distorted hands and some of the designs have spilled onto the background as well.

Quantitative analysis: Figure 48 shows a huge bounce of the generator loss at the beginning of the network. After that, the generator loss is pretty much stable apart from sudden spikes. The discriminator loss is more or less at zero with minor spikes and major spikes at the same iterations as the generator loss spikes. The networks do not converge.

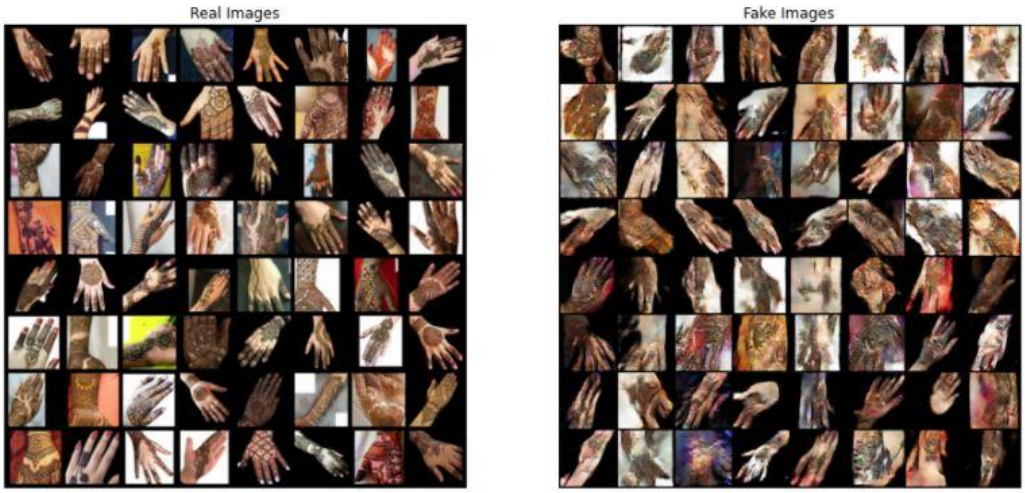

Figure 47: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 128 and learning rate 0.0002

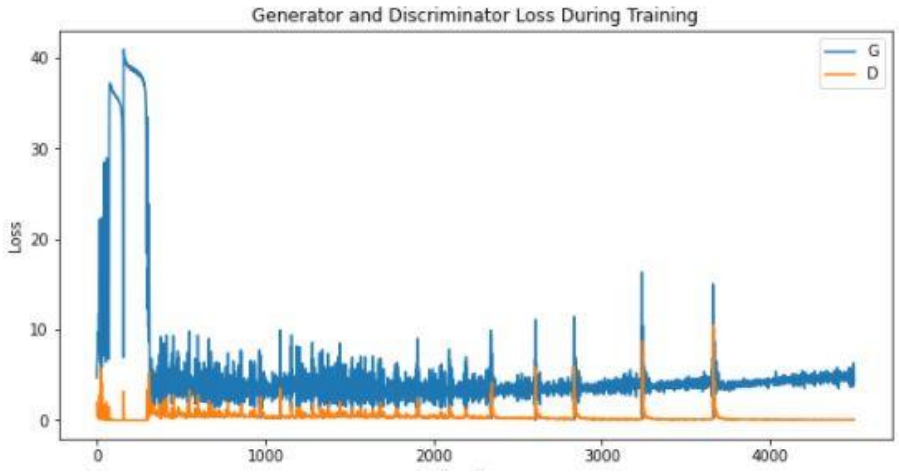

Figure 48: Graph for loss comparison for 300 epochs, batch size 128 and learning rate 0.0002

4.3.7. Batch size 256, 300 epochs

Qualitative analysis: For batch 256 there are some handshapes with designs, some have spilled onto the background, and some are just random noisy images. The quality overall is not good.

Quantitative analysis: Figure 50 depicts the graph for this configuration which is almost similar to the immediately previous graph and has some coincident spikes. There are no convergences whatsoever. The discriminator sticks at almost a constant of 0 with occasional spikes while the generator loss initially slides below 5 and remains almost constant throughout the entire time period.

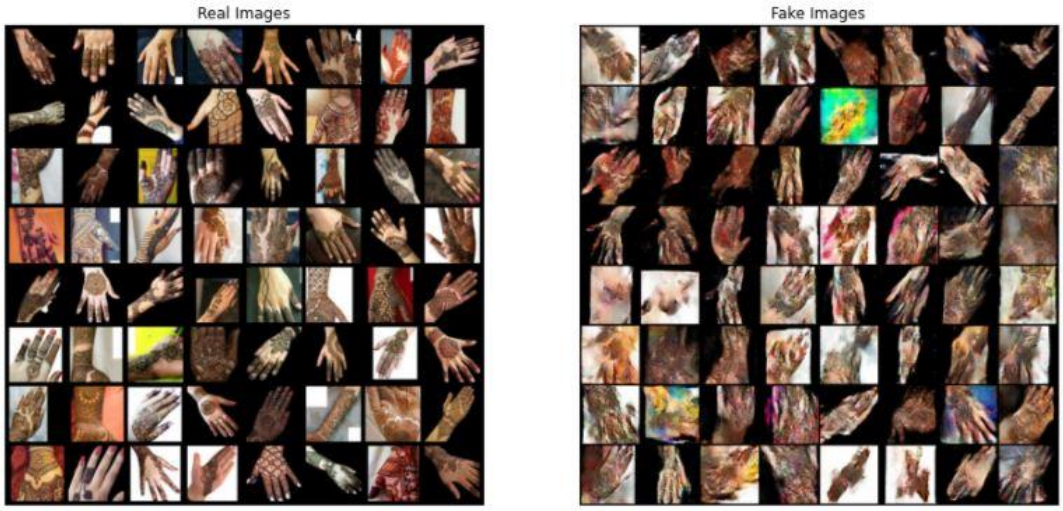

Figure 49: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 256 and learning rate 0.0002

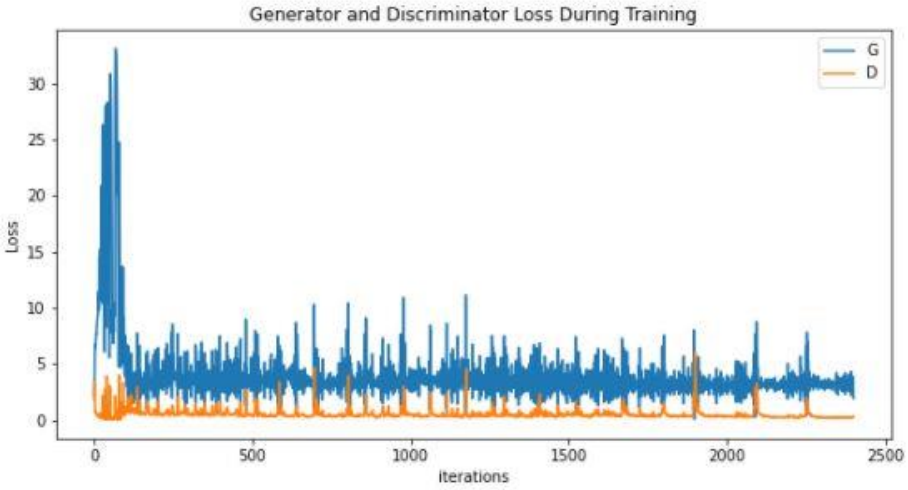

Figure 50: Graph for loss comparison for 300 epochs, batch size 256 and learning rate 0.0002

4.3.8. Batch size 512, 300 epochs

Qualitative analysis: After 300 epochs for batch 512 the outcomes shown cannot be called a handshape in figure 51. So the overall quality worsens with 300 epochs.

Quantitative analysis: The graph is similar to the previous two graphs. But as the quality of generated images is worse, the plot is not very interesting for a discussion.

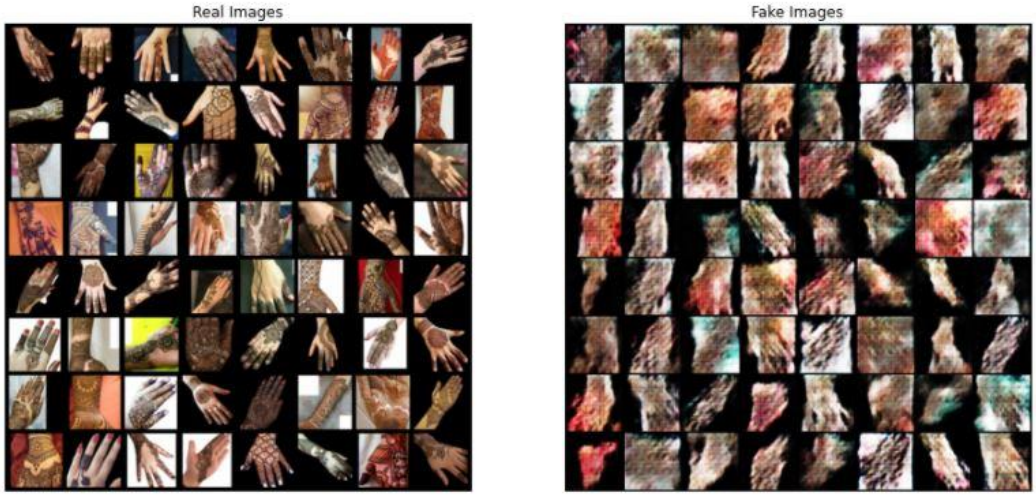

Figure 51: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 512, and learning rate 0.0002

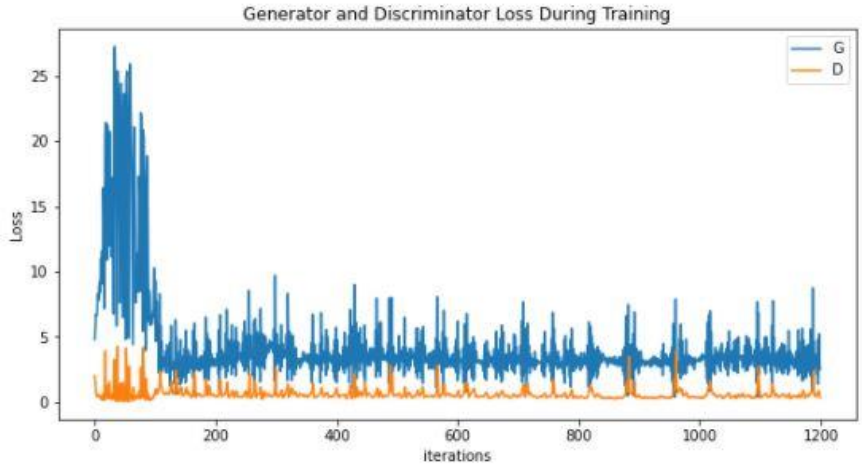

Figure 52: Graph for loss comparison for 300 epochs, batch size 512, and learning rate 0.0002

4.3.9. Batch size 1024, 300 epochs

Qualitative analysis: In the case of batch 1024 the result obtained after 300 epochs does not contain any single hand shape. Also, there are no designs rather all look like noise. The quality of images is probably the worst quality with this configuration.

Quantitative analysis: The network losses do never coincide or converge. This shows why the quality of the generated images is so bad in this case. The generator loss function initially sees a spike in its values until about a 100 iterations, then drops to around a constant value of 5. In contrast, the discriminator almost entirely stays around zero.

Figure 53: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 1024, and learning rate 0.0002

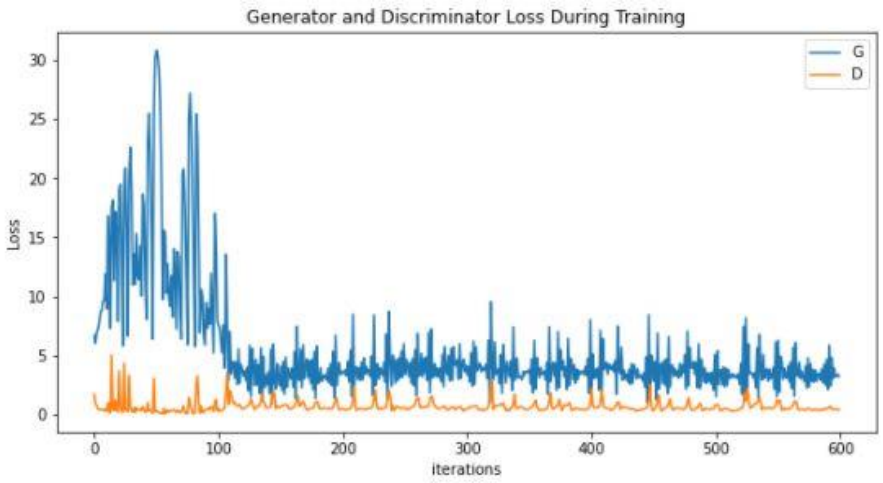

Figure 54: Graph for loss comparison for 300 epochs, batch size 1024, and learning rate 0.0002

4.4 Runs with the different learning rates

For the following runs, the batch size was kept at 64 since the better results were obtained in this setting. The learning rate was made variable along with the epochs, while the images are all 64x64 pixels.

4.4.1. Learning Rate 0.0005, 100 epochs

Qualitative analysis: The result achieved by learning rate 0.0005 after 100 epochs in Figure 55 has a slightly worse outcome than the result achieved by learning rate 0.0002 with the same configuration.

Quantitative analysis: The Generator and discriminator losses converge immediately at the beginning of the training and then diverge slowly. After the divergence, the generator loss stays constant at around 7 or 8 while occasionally springing past 10 while the discriminator stays at around 0. There are also spikes in the discriminator function which coincide with the spikes of the counterpart generator function at the same point in time.

Figure 55: Comparison of real versus generated images for 100 epochs, image size 64 pixels, batch size 64, and learning rate 0.0005

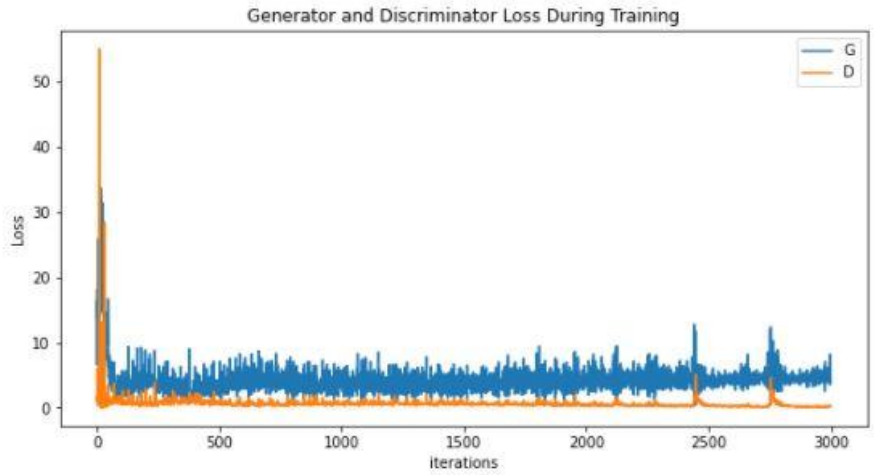

Figure 56: Graph for loss comparison for 100 epochs, image size 64, batch size 64, and learning rate 0.0005

4.4.2. Learning Rate 0.0005, 300 epochs

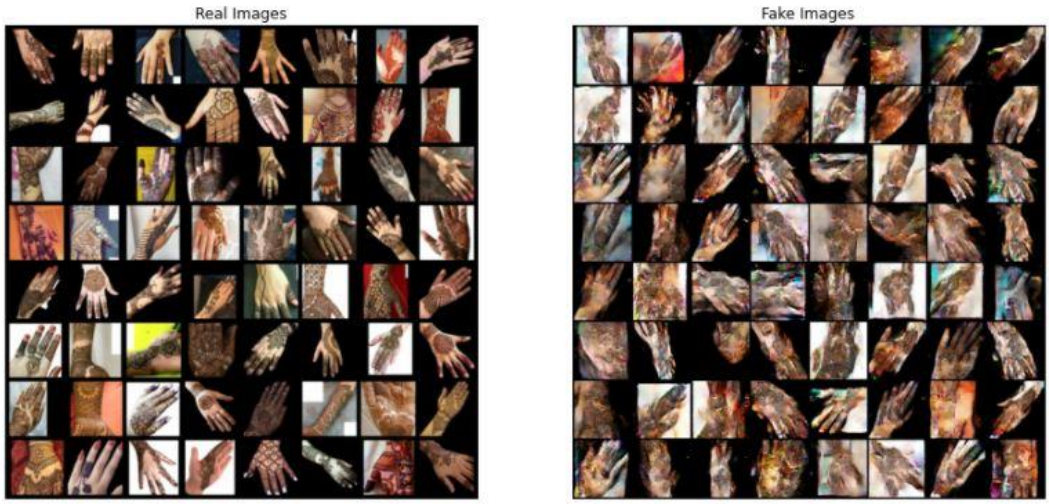

Figure 57: Comparison of real versus generated images for 300 epochs, image size 64 pixels, batch size 64, and learning rate 0.0005

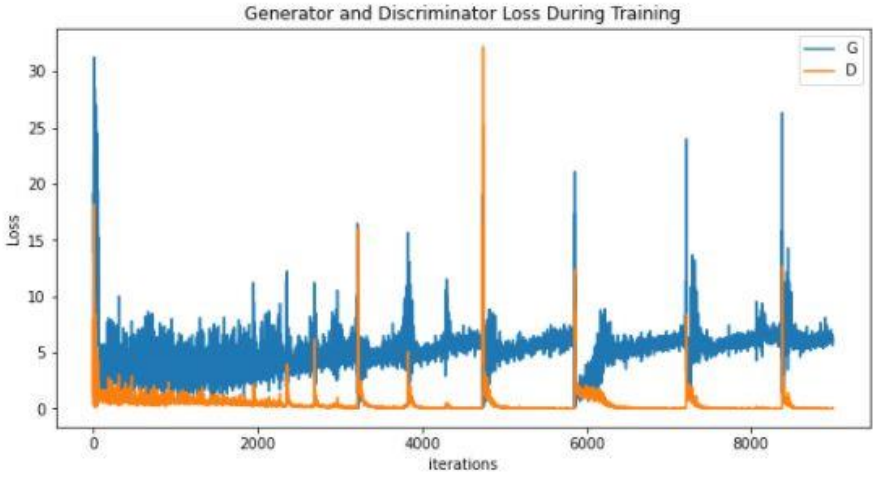

Figure 58: Graph for loss comparison for 300 epochs, image size 64, batch size 64 and learning rate 0.0005

Qualitative analysis: Though 100 epochs had good results for a learning rate of 0.0002, for 0.0005 it was worse. But After a run for 300 epochs with 0.0005, this result improves very slightly with proper handshape and in some cases a visible design. Although a lot of generated images are very noisy and of bad quality overall.

Quantitative analysis: The graph in Figure 58 shows the discriminator and generator losses starting together and then immediately losing the convergence. The two losses

coincide again at 3000 iterations and then diverge with the discriminator loss staying close to zero apart from coincident spikes with the generator loss.

4.4.3. Learning Rate 0.0002(discriminator) & 0.002(generator), 100 epochs

Qualitative analysis: The result achieved by learning rate 0.0002(discriminator) and 0.002(generator) after 100 epochs in Figure 59 shows a worse outcome than the result achieved by learning rate 0.0002 for both generator and discriminator with the same configuration in Figure 39.

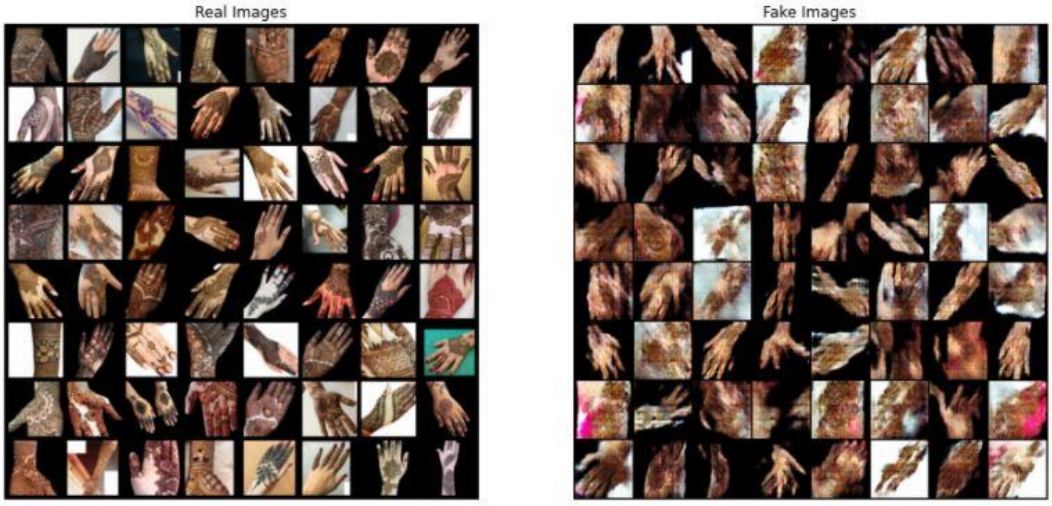

Figure 59: Comparison of real versus generated images for 100 epochs, image size 64 pixels, batch size 64, and learning rate 0.0002(D), 0.002(G)

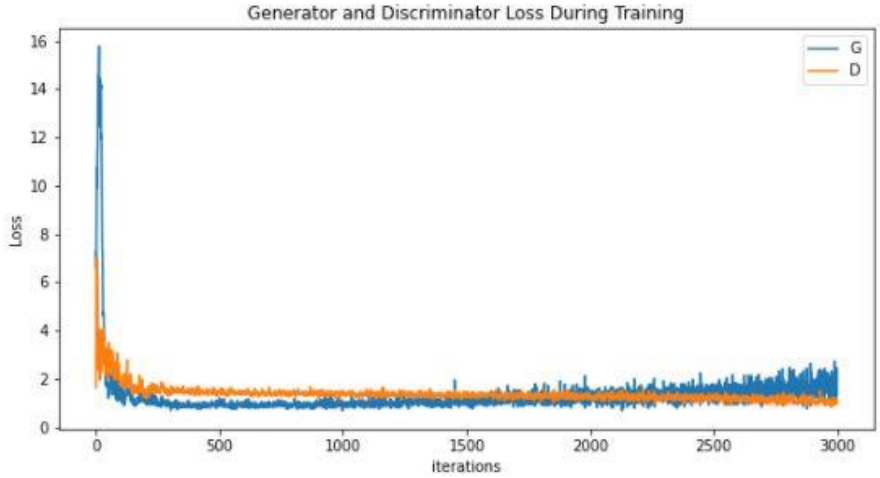

Figure 60: Graph for loss comparison for 100 epochs, image size 64, batch size 64, and learning rate 0.0002(D), 0.002(G)

Quantitative analysis: Though the outcome looks worse, the graphical loss shows better convergence than Figure 40. Figure 60 displays the graph in which the discriminator and generator losses converge at around 1500 iterations and then gradually start to diverge. Even though the loss functions, the quality of the generated images is not extraordinarily better than the ones where the loss functions never converged.

4.4.4. Learning Rate 0.0002(discriminator) & 0.0005(generator), 100 epochs

Figure 61: Comparison of real versus generated images for 100 epochs, image size 64 pixels, batch size 64 and learning rate 0.0002(D), 0.0005(G)

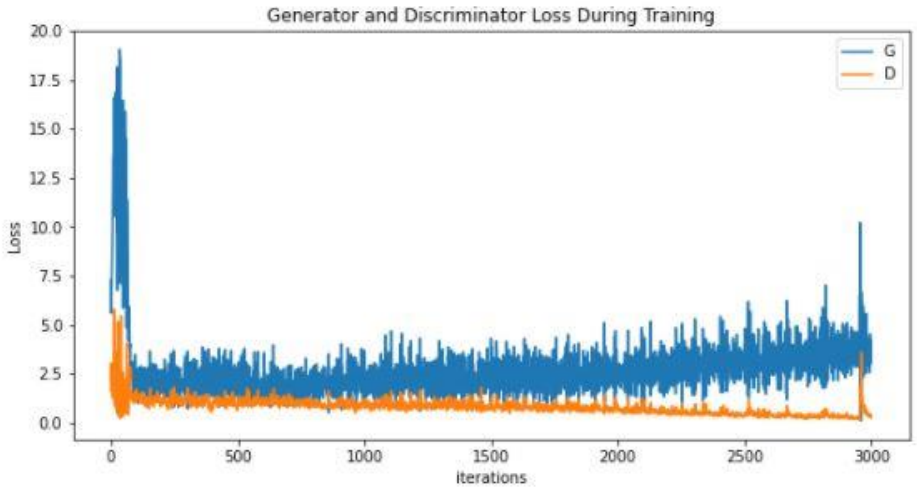

Figure 62: Graph for loss comparison for 100 epochs, image size 64, batch size 64, and learning rate 0.0002(D), 0.0005(G)

Qualitative analysis: The result achieved by learning rate 0.0002(discriminator) and 0.0005(generator) after 100 epochs in Figure 61 shows a weaker outcome than the result achieved by learning rate 0.0002(discriminator) and 0.002(generator) with the same configuration in Figure 59.

Quantitative analysis: The graph in Figure 62 shows the discriminator and generator losses starting together and then after some time losing the convergence. The two losses coincide again at 3000 iterations and then diverge with the discriminator loss staying close to zero apart from coincident spikes with the generator loss.

Chapter 5: Conclusion

5.1 Challenges

In the course of conducting this research and implementation of the concept, challenges were faced which ranged across a broad array of categories. The main pain points are briefly described in the following sections.

5.1.1. Limitations in Colab

The implementation was done in a Google Colab Notebook. Although it is a very robust cloud environment with GPU support, it has its fair share of limitations. The main hindrance to using Google Colab is the lack of fine-grained control. There are many configurations that can be modified in a physical setup which are not possible. And even the GPUs are shared. All the resources are hosted in the cloud which is a shared environment. As the resources are shared, there is a limit to their usage. And the GPU is always not available as well. If the notebook is idle for a while, the training switches to the slower CPU configuration. This switch occurs due to the high demands for the GPUs in the cloud. Due to this variable environment and configuration switching, it is hard to accurately determine the run time of the training.

5.1.2. Limitation in the dataset

The number of images in the training dataset plays a huge role in the performance of the neural network. But unfortunately, the number of available images suitable for being trained is limited. Around 10,000 images were collected from various sources. But many of those images had to be discarded due to being duplicates or being unsuitable due to background noise. A lot of the images also had other designs along with Henna designs which also had to be discarded.

All the selected images did not have the same rotation and orientation of the hand, which was a major challenge in generating uniform handshape and orientation.

5.1.3. Limitations in evaluation

Although there exists some metrics like Frechet Inception Distance (FID) or Inception score (IS), there is still no definite metric that is agreed upon to evaluate GANs. Furthermore, evaluating artwork is an abstract activity and thus the evaluation of the generated images is very difficult in absence of specific metrics and values.

5.2 Summary

GANs are a promising avenue to creative endeavours. This work provides a brief survey of the capabilities and types of GAN available for generating art. The attempt to find an AI model to generate a traditional artwork like Henna design is novel in this regard and it demonstrates that it is possible to use computers to generate such arts.

The configuration of GANs to generate desirable quality in the artwork is also an important theme that was worked on. The various hyper-parameters that are possible to be manipulated to fine-tune the network to generate better quality images were discussed. An intuition about these hyper-parameters can be gained from the quantitative and qualitative analysis done on the generated images.

From the observations we can say for the smaller dataset, batch 128 works better. Besides batch 32 and batch 64 also work well but batch size more than 128 doesn't work well with a smaller dataset.

For different learning rates, two approaches were tried. One was keeping the learning rate the same for both generator and discriminator where learning rate 0.0002 gave a better result in 100 epochs than a learning rate of 0.0005. Another approach was putting different learning rates for generators and discriminators which is called Two Time Scale Update Rule (TTUR) where a better result was obtained for learning rate 0.0002(d) & 0.002(g) than learning rate 0.0002(d) and 0.0005(g).

Also in the experiments, three dimensions of 32 x 32 pixels, 64 x 64 pixels, and 128 x 128 pixels were used where dimensions of 32 x 32 and 64 x 64 had better outcomes for this smaller dataset.

5.3 Future work

This thesis can be further improved in the future upon some selected ideas mentioned below:

- A larger dataset can be used for better training of the network.
- A better-curated dataset with hands having the same orientation and pose can be used to generate better images.
- More experimenting on the hyper-parameters may be done to improve the model.
- Artwork similar to Henna, like tattoos, etc. can be generated using the same technique.
- A finely tuned model for generating Henna designs can be used in a specially designed robot arm that can apply these designs on human hands.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative Adversarial Networks," in *International Conference on Neural Information Processing Systems* , 2014.

[2] Z. Pan, W. Yu, X. Yi, A. Khan, F. Yuan and Y. Zheng, "Recent Progress on Generative Adversarial Networks (GANs): A Survey," *IEEE Access,* vol. 7, pp. 36322-36333, 2019.

[3] Radford, Alec, L. Metz and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," Cornell University, 2015.

[4] Google, "Deep Convolutional Generative Adversarial Network," Google, [Online]. Available: https://www.tensorflow.org/tutorials/generative/dcgan?hl=en. [Accessed 7 4 2020].

[5] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," in *NIPS*, 2017.

[6] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford and X. Chen, "Improved Techniques for Training GANs," *arXiv*, 2016.

[7] M. Mirza and S. Osindero, *Conditional Generative Adversarial Nets,* arXiv, 2014.

[8] T. Karras, T. Aila, S. Laine and J. Lehtinen, *Progressive Growing of GANs for Improved Quality, Stability, and Variation,* arXiv, 2018.

[9] A. Brock, J. Donahue and K. Simonyan, *Large Scale GAN Training for High Fidelity Natural Image Synthesis,* arXiv, 2019.

[10] L. S. A. T. Karras T, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019.

[11] K. Jones, "GANGogh: Creating Art with GANs," 18 07 2017. [Online]. Available: https://towardsdatascience.com/gangogh-creating-art-with-gans-8d087d8f74a1. [Accessed 01 03 2020].

[12] W. Z. J. W. a. H. W. Y. Du, "DCGAN Based Data Generation for Process Monitoring," in *IEEE 8th Data Driven Control and Learning Systems Conference (DDCLS)*, Dali, China, 2019.

[13] R. Barrat, "art-DCGAN," 2017. [Online]. Available: https://github.com/robbiebarrat/art-DCGAN. [Accessed 05 04 2020].

[14] V. Dibia, “ART + AI — Generating African Masks using (Tensorflow and TPUs),” 28 12 2018. [Online]. Available: https://towardsdatascience.com/african-masks-gans-tpu-9a6b0cf3105c. [Accessed 1 5 2020].

[15] H. A. Wibowo, “Generate Anime Style Face Using DCGAN and Explore Its Latent Feature Representation,” 13 4 2019. [Online]. Available: https://towardsdatascience.com/generate-anime-style-face-using-dcgan-and-explore-its-latent-feature-representation-ae0e905f3974. [Accessed 17 4 2020].

[16] S.Sainath and S.Sridhar, “RangoliGAN: Generating Realistic Rangoli Images using GAN,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 07, no. 04, 2020.

[17] D. I. G. A. R. D. W. D. Gregor K, *Draw: A recurrent neural network for image generation.*, 2015.

[18] OpenCV, “Interactive Foreground Extraction using GrabCut Algorithm,” [Online]. Available: https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html. [Accessed 3 4 2020].

[19] Kaleido AI GmbH, “removebg,” [Online]. Available: https://www.remove.bg/. [Accessed 7 4 2020].

[20] M. Haenlein and A. Kaplan, “A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence,” *California Management Review*, vol. 61, pp. 5-14, 2019.

[21] S. K. Sarvepalli, “Deep Learning in Neural Networks: The science behind an Artificial Brain,” Liverpool Hope University, Liverpool, 2015.

[22] Y. Zhou, “Sentiment classification with deep neural networks,” Tampere University, Tampere, 2019.

[23] UC Business Analytics R Programming Guide, “Feedforward Deep Learning Models,” [Online]. Available: http://uc-r.github.io/feedforward_DNN. [Accessed 11 3 2020].

[24] F. Rosenblatt, “The perceptron–a perceiving and recognizing automaton,” Cornell Aeronautical Laboratory, 1957.

[25] DeepAI, “Sigmoidal Nonlinearity,” [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/sigmoidal-nonlinearity. [Accessed 27 3 2020].

[26] Cluzters.ai, “Introduction to Artificial Neural Networks,” [Online]. [Accessed 27 3 2020].

[27] P. T. Perez, “Deep Learning: Recurrent Neural Networks,” 18 10 2018. [Online]. Available: https://medium.com/deeplearningbrasilia/deep-learning-recurrent-neural-networks-f9482a24d010. [Accessed 28 3 2020].

[28] S. Yan, “Understanding LSTM and its diagrams,” 14 3 2016. [Online]. Available: https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714. [Accessed 28 3 2020].

[29] J. Brownlee, “A Gentle Introduction to Long Short-Term Memory Networks by the Experts,” 24 5 2017. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/. [Accessed 29 3 2020].

[30] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation,* vol. 9, no. 8, p. 1735–1780, 1997.

[31] E. R. Kandel, “An introduction to the work of David Hubel and Torsten Wiesel,” *The Journal of physiology,* 2009.

[32] D. K. Fukushima, *Neocognitron,* Kansai: Scholarpedia, 2007.

[33] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, “ Backpropagation Applied to Handwritten Zip Code Recognition,” AT&T Bell Laboratories, 1989.

[34] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” in *Proceedings of the IEEE*, 1998.

[35] D. A. Arora, “Neural Network: Activation Function,” [Online]. Available: https://anujaarora.com/2019/01/13/neural-network-activation-function/. [Accessed 4 3 2020].

[36] S. Sharma, “Activation Functions in Neural Networks,” [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed 4 3 2020].

[37] T. Karras, S. Laine and T. Aila, *A Style-Based Generator Architecture for Generative Adversarial Networks,* arXiv, 2019.

[38] Google, “An end-to-end open source machine learning platform,” [Online]. Available: https://www.tensorflow.org/?hl=en. [Accessed 20 3 2020].

[39] Facebook, “Torch,” [Online]. Available: http://torch.ch/. [Accessed 27 3 2020].

[40] “PyTorch,” [Online]. Available: https://web.archive.org/web/20180615190804/https://pytorch.org/about/. [Accessed 25 3 2020].

[41] “ABOUT US,” [Online]. Available: https://numpy.org/about/. [Accessed 26 4 2020].

[42] “Matplotlib,” [Online]. Available: https://matplotlib.org/. [Accessed 28 3 2020].

[43] M. Gupta, “NUMPY in Python for Machine Learning,” [Online]. Available: https://medium.com/technofunnel/numpy-in-python-for-machine-learning-4040109aeaee. [Accessed 28 3 2020].

[44] [Online]. Available: https://colab.research.google.com/.

[45] J. Brownlee, “Supervised and Unsupervised Machine Learning Algorithms,” 16 3 2016. [Online]. Available: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/. [Accessed 28 3 2020].

[46] K. M, M. T, I. W, G. HA and S. R, “A survey on unsupervised machine learning algorithms for automation, classification and maintenance.,” *International Journal of Computer Applications,* vol. 119, 2015.

[47] A. E. a. B. L. a. M. E. a. M. Mazzone, *CAN: Creative Adversarial Networks, Generating "Art" by Learning About Styles and Deviating from Style Norms,* 2017.

[48] D. J. S. K. Brock A, *Large scale gan training for high fidelity natural image synthesis,* 2018.

Appendix

Appendix A: Generator

```
# Generator Code
#image_size 32
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 4, kernel_size, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 4 x 4
            nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 8 x 8
            nn.ConvTranspose2d( ngf * 2, ngf, kernel_size, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 16 x 16
            nn.ConvTranspose2d( ngf, nc, kernel_size, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 32 x 32
        )

#image_size 64
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, kernel_size, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, kernel_size, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, kernel_size, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, kernel_size, 2, 1, bias=False),
```

```
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

# image_size 128
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(     nz, ngf * 16, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 16),
            nn.ReLU(True),
            # state size. (ngf*16) x 4 x 4
            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 8 x 8
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 16 x 16
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 32 x 32
            nn.ConvTranspose2d(ngf * 2,     ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 64 x 64
            nn.ConvTranspose2d(    ngf,      nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 128 x 128
        )

    def forward(self, input):
        return self.main(input)

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

      # Print the model
      print(netG)
```

Appendix B: Discriminator

```
#Discriminator_code
#image_size_32
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
```

```
8.              #  # input is (nc) x 32 x 32
9.              nn.Conv2d(nc, ndf, 4, stride=2, padding=1, bias=False),
10.             nn.LeakyReLU(0.2, inplace=True),
11.             # state size. (ndf) x 16 x 16
12.             nn.Conv2d(ndf, ndf * 2, 4, stride=2, padding=1, bias=False),
13.             nn.BatchNorm2d(ndf * 2),
14.             nn.LeakyReLU(0.2, inplace=True),
15.             # state size. (ndf*2) x 8 x 8
16.             nn.Conv2d(ndf * 2, ndf * 4, 4, stride=2, padding=1, bias=False)
    ,
17.             nn.BatchNorm2d(ndf * 4),
18.             nn.LeakyReLU(0.2, inplace=True),
19.             # state size. (ndf*4) x 4 x 4
20.             nn.Conv2d(ndf * 4, 1, 4, stride=1, padding=0, bias=False),
21.             nn.Sigmoid()
22.             # state size. 1
23.         )
24.
25. #image_size_64
26. class Discriminator(nn.Module):
27.     def __init__(self, ngpu):
28.         super(Discriminator, self).__init__()
29.         self.ngpu = ngpu
30.         self.main = nn.Sequential(
31.             # input is (nc) x 64 x 64
32.             nn.Conv2d(nc, ndf, kernel_size, 2, 1, bias=False),
33.             nn.LeakyReLU(0.2, inplace=True),
34.             # state size. (ndf) x 32 x 32
35.             nn.Conv2d(ndf, ndf * 2, kernel_size, 2, 1, bias=False),
36.             nn.BatchNorm2d(ndf * 2),
37.             nn.LeakyReLU(0.2, inplace=True),
38.             # state size. (ndf*2) x 16 x 16
39.             nn.Conv2d(ndf * 2, ndf * 4, kernel_size, 2, 1, bias=False),
40.             nn.BatchNorm2d(ndf * 4),
41.             nn.LeakyReLU(0.2, inplace=True),
42.             # state size. (ndf*4) x 8 x 8
43.             nn.Conv2d(ndf * 4, ndf * 8, kernel_size, 2, 1, bias=False),
44.             nn.BatchNorm2d(ndf * 8),
45.             nn.LeakyReLU(0.2, inplace=True),
46.             # state size. (ndf*8) x 4 x 4
47.             nn.Conv2d(ndf * 8, 1, kernel_size, 1, 0, bias=False),
48.
49.             # 1x1x1
50.             nn.Sigmoid()
51.         )
52.
53. #image_size_128
54. class Discriminator(nn.Module):
55.     def __init__(self, ngpu):
56.         super(Discriminator, self).__init__()
57.         self.ngpu = ngpu
58.         self.main = nn.Sequential(
59.             # input is (nc) x 128 x 128
60.             nn.Conv2d(nc, ndf, 4, stride=2, padding=1, bias=False),
61.             nn.LeakyReLU(0.2, inplace=True),
62.             # state size. (ndf) x 64 x 64
63.             nn.Conv2d(ndf, ndf * 2, 4, stride=2, padding=1, bias=False),
64.             nn.BatchNorm2d(ndf * 2),
65.             nn.LeakyReLU(0.2, inplace=True),
66.             # state size. (ndf*2) x 32 x 32
67.             nn.Conv2d(ndf * 2, ndf * 4, 4, stride=2, padding=1, bias=False)
    ,
68.             nn.BatchNorm2d(ndf * 4),
```

```
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 16 x 16
            nn.Conv2d(ndf * 4, ndf * 8, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 8 x 8
            nn.Conv2d(ndf * 8, ndf * 16, 4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*16) x 4 x 4
            nn.Conv2d(ndf * 16, 1, 4, stride=1, padding=0, bias=False),
            nn.Sigmoid()
            # state size. 1
        )
    def forward(self, input):
        return self.main(input)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
# to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

Appendix C: Training

```
%matplotlib inline
from __future__ import print_function
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.style
import matplotlib as mpl
mpl.style.use('classic')
from IPython.display import HTML

# Set random seem for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
```

```
torch.manual_seed(manualSeed)
# Load the Drive helper and mount
from google.colab import drive
#https://drive.google.com/drive/folders/1EUnErTN650NrOh7kXqB4V5Kxoq_1L_1r
# This will prompt for authorization.
drive.mount('/content/drive')
# Root directory for dataset
dataroot = ""
# Number of workers for dataloader
workers = 4

# Batch size during training
batch_size = 64
# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64 #64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100
# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002
# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

kernel_size=4

# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
```

```
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=True).cpu(),(1,2,0)))

# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)


      # Initialize BCELoss function
      criterion = nn.BCELoss()

      # Create batch of latent vectors that we will use to visualize
      #  the progression of the generator
      fixed_noise = torch.randn(64, nz, 1, 1, device=device)

      # Establish convention for real and fake labels during training
      real_label = 1
      fake_label = 0

      # Setup Adam optimizers for both G and D
      optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
      optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
      # Training Loop

      # Lists to keep track of progress
      img_list = []
      G_losses = []
      D_losses = []
      iters = 0

      print("Starting Training Loop...")
      # For each epoch
      for epoch in range(num_epochs):
          # For each batch in the dataloader
          for i, data in enumerate(dataloader, 0):

              ############################
              # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
              ###########################
              ## Train with all-real batch
              netD.zero_grad()
              # Format batch
              real_cpu = data[0].to(device)
              #real_cpu = (data.unsqueeze(dim=1).type(torch.FloatTensor)).to(device)
              b_size = real_cpu.size(0)
              label = torch.full((b_size,), real_label,dtype=torch.float, device=device, )
              # Forward pass real batch through D
              output = netD(real_cpu).view(-1)
              # Calculate loss on all-real batch
              errD_real = criterion(output, label)
              # Calculate gradients for D in backward pass
```

```
145.                 errD_real.backward()
146.                 D_x = output.mean().item()
147.
148.                 ## Train with all-fake batch
149.                 # Generate batch of latent vectors
150.                 noise = torch.randn(b_size, nz, 1, 1, device=device)
151.                 # Generate fake image batch with G
152.                 fake = netG(noise)
153.                 label.fill_(fake_label)
154.                 # Classify all fake batch with D
155.                 output = netD(fake.detach()).view(-1)
156.                 # Calculate D's loss on the all-fake batch
157.                 errD_fake = criterion(output, label)
158.                 # Calculate the gradients for this batch
159.                 errD_fake.backward()
160.                 D_G_z1 = output.mean().item()
161.                 # Add the gradients from the all-real and all-fake batches
162.                 errD = errD_real + errD_fake
163.                 # Update D
164.                 optimizerD.step()
165.
166.                 ############################
167.                 # (2) Update G network: maximize log(D(G(z))) using 'log D'
    trick
168.                 ###########################
169.                 netG.zero_grad()
170.                 label.fill_(real_label)  # fake labels are real for generato
    r cost
171.                 # Since we just updated D, perform another forward pass of a
    ll-fake batch through D
172.                 output = netD(fake).view(-1)
173.                 # Calculate G's loss based on this output
174.                 errG = criterion(output, label)
175.                 # Calculate gradients for G
176.                 errG.backward()
177.                 D_G_z2 = output.mean().item()
178.                 # Update G
179.                 optimizerG.step()
180.
181.                 # Output training stats
182.                 if i % 50 == 0:
183.                     print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x):
     %.4f\tD(G(z)): %.4f / %.4f'
184.                           % (epoch, num_epochs, i, len(dataloader),
185.                              errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
186.
187.                 # Save Losses for plotting later
188.                 G_losses.append(errG.item())
189.                 D_losses.append(errD.item())
190.
191.                 # Check how the generator is doing by saving G's output on f
    ixed_noise
192.                 if (iters % 500 == 0) or ((epoch == num_epochs-
    1) and (i == len(dataloader)-1)):
193.                     with torch.no_grad():
194.                         fake = netG(fixed_noise).detach().cpu()
195.                     img_list.append(vutils.make_grid(fake, padding=2, normal
    ize=True))
196.
197.                 iters += 1
198.
199.       plt.figure(figsize=(10,5))
```

```
        plt.title("Generator and Discriminator Loss During Training")
        plt.plot(G_losses,label="G",color='magenta')
        plt.plot(D_losses,label="D",color='blue')
        plt.xlabel("iterations")
        plt.ylabel("Loss")
        plt.legend()
        plt.show()

        #GIF show
        #%%capture
        fig = plt.figure(figsize=(8,8))
        plt.axis("off")
        ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in
     img_list]
        ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_dela
    y=1000, blit=True)

        HTML(ani.to_jshtml())

        # Grab a batch of real images from the dataloader
        real_batch = next(iter(dataloader))

        # Plot the real images
        plt.figure(figsize=(15,15))
        plt.subplot(1,2,1)
        plt.axis("off")
        plt.title("Real Images")
        plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:6
    4], padding=5, normalize=True).cpu(),(1,2,0)))

        # Plot the fake images from the last epoch
        plt.subplot(1,2,2)
        plt.axis("off")
        plt.title("Fake Images")
        plt.imshow(np.transpose(img_list[-1],(1,2,0)))
        plt.show()
```

Appendix D: Duplicate Remove

```
from collections import defaultdict
from PIL import Image
import argparse
import os
import hashlib
import imagehash


def getAverageHash(imgPath):
    try:
        return imagehash.average_hash(Image.open(imgPath))
    except:
        return None

def getDataset(image):
    """Get the images from the directory, partitioned by class."""
    exts = ["jpg", "png"]

    imageClasses = []  # Images, separated by class.
    for subdir, dirs, files in os.walk(image):
        images = []
        for fileName in files:
```

```
23.             # (imageClass, imageName) = (os.path.basename(subdir), fileName
    )
24.             imageName = fileName
25.             if any(imageName.lower().endswith("." + ext) for ext in exts):

26.                 images.append(os.path.join(subdir, fileName))
27.         imageClasses.append(images)
28.     return imageClasses
29. 
30. 
31. def runOnClass(args, imgs):
32.     """Find and remove duplicates within an image class."""
33.     d = defaultdict(list)
34.     for imgPath in imgs:
35.         imgHash = getAverageHash(imgPath)
36.         if imgHash:
37.             d[imgHash].append(imgPath)
38. 
39.     numFound = 0
40.     for imgHash, imgs in d.items():
41.         if len(imgs) > 1:
42.             print("{}: {}".format(imgHash, " ".join(imgs)))
43.             numFound += len(imgs) - 1  # Keep a single image.
44. 
45.             if args.delete:
46.                 largestImg = max(imgs, key=os.path.getsize)
47.                 print("Keeping {}.".format(largestImg))
48.                 imgs.remove(largestImg)
49.                 for img in imgs:
50.                     os.remove(img)
51. 
52.             if args.sha256:
53.                 print("")
54.                 for img in imgs:
55.                     print(hashlib.sha256(open(img, 'rb').read()).hexdigest(
    ))
56.                 print("")
57.     return numFound
58. 
59. if __name__ == '__main__':
60.     parser = argparse.ArgumentParser()
61.     parser.add_argument('inplaceDir', type=str,
62.                         help="Directory of images, divided into "
63.                         "subdirectories by class.")
64.     parser.add_argument('--delete', action='store_true',
65.                         help="Delete the smallest duplicate images instead
    "
66.                         "of just listing them.")
67.     parser.add_argument('--sha256', action='store_true',
68.                         help="Show sha256 sum for duplicate images")
69.     args = parser.parse_args()
70. 
71.     numFound = 0
72.     for imgClass in getDataset(args.inplaceDir):
73.         numFound += runOnClass(args, imgClass)
74.     print("\n\nFound {} total duplicate images.".format(numFound))
```