Animated Background Character Generation Based on Generative Adversarial Network

A SEMINAR REPORT

Submitted by

P MAHESH - RA1811032020041
ARNAV TYAGI - RA1811032020028

Under the guidance of

Mr . Arun V

(Assistant Professor, Department of Computer Science and Engineering)

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY RAMAPURAM CAMPUS, CHENNAI -600089

MAY - 2021

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Deemed to be University u/S 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this seminar report **"Animated Background Character Generation Based on Generative Adversarial Network"** is the bonafide work of "*PMAHESH,*ARNAV TYAGI**"** who carried out the project work under my supervision.

SIGNATURE
SUPERVISOR
DEPARTMENT

<<Academic Designation>>
<<Department>>
<<Full address of the Dept & Institution >>

SIGNATURE
HEAD OF THE

<<Department>>
<<Full address of the Dept & Institution >>

Submitted for the Viva Voce Examination held on …….. at SRM Institute of Science and Technology , Ramapuram Campus, Chennai -600089

<<Signature>>
INTERNAL EXAMINER I

<<Signature>>
INTERNAL EXAMINER I

ABSTRACT

Using a computer to generate images with realistic images is a new direction in current computer vision research. This paper designs an image generation model based on the Generative Adversarial Network (GAN). This paper creates a model – a discriminator network and a generator network by eliminating the fully connected layer in the traditional network and applying batch normalization and deconvolution operations. This paper also uses a hyper-parameter to measure the diversity and quality of the generated image. The experimental results of the model on the CelebA dataset show that the model has excellent performance in face image generation. In this report we take this approach to an entirely new type of style, I.e hand drawn animation. This is inherently challenging due to the fact that CelebA dataset contains data of similar style, real world palette. But, In terms of anime, the art style differs and palette is volatile at best.

List of figures

Fig number	Description
Fig-1 :	a Classic case of mode collapse
Fig-2 :	dataset used
Fig 3 :	Discriminator
Fig 4 :	Generator
Fig 5:	Discriminator Architecture theory
Fig 6:	Generator Architecture theory
Fig-7 :	GAN
Fig -8	Discriminator Model code
Fig -9	Discriminator Architecture visual
Fig -10	Log_loss
Fig -11	Generator model code
Fig -12	Generator Architecture Visual
Fig -13	C-Model_ code
Fig -14	Step count
Fig -15	Combined Dataset
Fig -16	Discriminator training
Fig -17	Generator training
Fig -18	Logging of losses
Fig -19	Logging of weights
Fig -20	Sampling result images
Fig -21	Project flow diagram
Fig -22	Data flow diagram
Fig -23	Binary cross entropy
Fig -24	Nash Equilibrium
Fig -25	Loss graphs
Fig -26	Version 1 results
Fig -27	version 2 results
Fig -28	version 3 results
Fig -29	version 4 results_1
Fig -30	version 4 results_2
Fig -31	version 5 results

List of Abbreviations

- **GPU - Graphics processing Unit**
- **GANs - Generative Adversarial Networks**
- **SRGAN - super resolution GAN**
- **LOGAN - Latent optimization GAN**
- **DCGAN - Deep Convolutional Generatiev Adversarial Networks**
- **SRS - system requirement Specification**
- **ANN - Artificial Neural Network**
- **ENN - Encoder neural Network**
- **NN - Neural Network**

APPENDIX 3

TABLE OF CONTENTS

ABSTRACT i
LIST OF FIGURES ii
LIST OF TABLES: iii
LIST OF ABBREVIATIONS iv
CHAPTER 1: INTRODUCTION 1
1.1 Background 1
1.2 Problem Statement 1
1.3 Project Concept……2
CHAPTER 2:LITERATURE REVIEW.....3
CHAPTER 3:SYSTEM SPECIFICATIONS……...10
3.1 Introduction…….…….…10
3.2 System Specifications……...….10
3.3 Hardware Specifications…….…….…….…….……10
3.4 Budget…….…….…….……11
CHAPTER 4:DATASET…….…….....13
CHAPTER 5: METHODS AND ALGORITHMS USED…….…….……15
5.1 Generative Adversarial Networks…….…….……15
5.1.a Generator Network…….16
5.1.b Discriminator Network……...17
5.2 GANs…….…….…….……18
CHAPTER 6: EXPERIMENTS…….....…19
6.1 Data Processing……19
6.2 C-model Architecture…….21
CHAPTER 7: ARCHTECTURE DIAGRAMS…….…….23
CHAPTER 8: EVALUATION METRICS…….26
8.1 Log_loss……..27
8.2 Results……28
CHAPTER 9:MODULE EXPLAINATION AND CODE……32
CHAPTER 10: CONCLUSION…….36

CHAPTER 1: INTRODUCTION

1.1 Background

With the increasing use of deep learning in the field of computer vision, automatic image generation based on depth models has received more and more attention. By using the powerful learning ability of the depth model, the inherent distribution law in the data can be efficiently mined to generate images with similar distribution.High-quality generated images can also be used to expand the amount of image data in the dataset, which can alleviate the need for a large number of training samples during deep learning model training, so that practical applications such as face recognition have higher accuracy.

1.2 problem statement

We all love anime characters and are tempted to create our custom ones. However, it takes tremendous efforts to master the skill of drawing, after which we are first capable of designing our own characters. To bridge this gap, the automatic generation of anime characters offers an opportunity to bring a custom character into existence without professional skill. Besides the benefits for a non-specialist, a professional creator may take advantages of the automatic generation for inspiration on animation and game character design; a Doujin RPG developer may employ copyright-free facial images to reduce designing costs in game production.

1.3 Project concept

The main objective of developing this project is

- To develop a Architecture that is able to handle a Volatile pattern of images and able to successfully generate new images based on learned latent distribution.
- To determine the faults and mode collapse conditions which may lead to breakdown of GAN

we propose a model that produces anime faces at high quality with promising rate of success. Our contribution can be described as three-fold: A clean dataset, which we collected from various sources, a suitable GAN model, based on DRAGAN, and our approach to train a GAN from images without tags, which can be leveraged as a general approach to training supervised or conditional model without tag data.

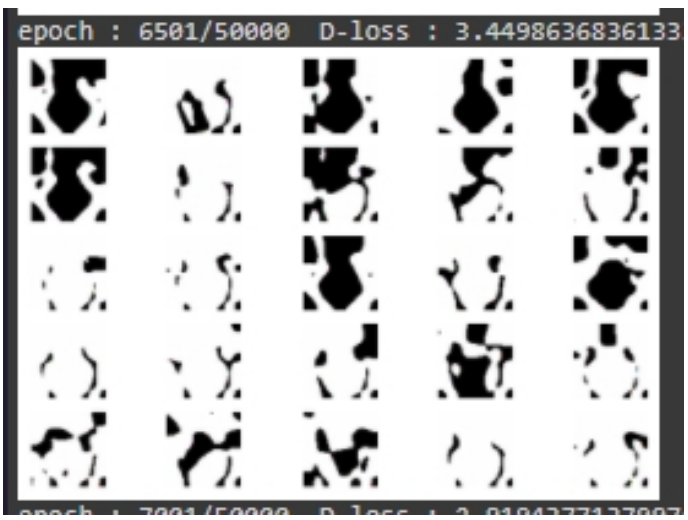


Fig-1
A classic case of mode collapse.

CHAPTER TWO: LITERATURE REVIEW

2.1 Introduction

Existing literature provides several attempts for generation facial images of anime characters. Among them are Mattya and Rezoolab who first explored the generation of anime character faces right after the appearance of DCGAN. Later, Hiroshiba proposed the conditional generation model for anime character faces. Also, codes are made available online focusing on anime faces generation such as IllustrationGAN and AnimeGAN. However, since results from these works are blurred and distorted on an untrivial frequency, it still remains a challenge to generate industry-standard facial images for anime characters.In this report, we propose a model that produces anime faces at high quality with promising rate of success.

Here we studied and did review on many research papers.By following those papers , we understood the complete architecture and functioning of Generative Adversarial neural networks. The review of the papers are arranged the following way :

1. Title of the paper
2. Author
3. Year (Research paper released)
4. Concept

S.no	title	year	concept
1.	REALISTIC FACE IMAGE GENERATION BASED ON GENERATIVE ADVERSARIAL NETWORK	2019	Using a computer to generate images with realistic images is a new direction in current computer vision research. This paper designs an image generation model based on the Generative Adversarial Network (GAN). This paper creates a model – a discriminator network and a generator network by eliminating the fully connected layer in the traditional network and applying batch normalization and deconvolution operations. This paper also uses a hyper-parameter to measure the diversity and quality of the generated image.
2.	StarGAN Based Facial Expression Transfer for Anime Characters	2020	Human facial expression transfer has been well explored using Generative Adversarial Networks. Also, in case of anime style images, several successful attempts have been made to generate high-quality anime face images using GAN approach. However, the task of anime facial expression transfer is not well studied yet due to the lack of a clean labeled anime dataset. We address this issue from both data and model perspectives, by providing a clean labeled anime dataset and leveraging the use of the StarGAN image-to-image translation framework.

3.	Generating Anime from Real Human Image with Adversarial Training	2019	With growing interest in animation movie, the demand for building an automated system to convert the real video into animation is higher than ever. The frame-by-frame editing of animation generation process is very costly and time-consuming. To help to generate animation quickly and easily in an automated process we proposed a generative model that converts real-world images into corresponding-animation image without losing important details of the source image. We used a variation of the generative adversarial network as underlying architecture with custom loss function to preserve the content of source image which converting to animation image.
4.	Towards the Automatic Anime Characters Creation with Generative Adversarial Networks	2017	Automatic generation of facial images has been well studied after the Generative Adversarial Network (GAN) came out. There exists some attempts applying the GAN model to the problem of generating facial images of anime characters, but none of the existing work gives a promising result. In this work, we explore the training of GAN models specialized on an anime facial image dataset. We address the issue from both the data and the model aspect, by collecting a more clean, well-suited dataset and leverage proper, empirical application of DRAGAN. With quantitative analysis and case studies we demonstrate that our efforts lead to a stable and high-quality model.

5.	Full-body High-resolution Anime Generation with Progressive Structure-conditional Generative Adversarial Networks	2018	Progressive Structure-conditional Generative Adversarial Networks (PSGAN), a new framework that can generate fullbody and high-resolution character images based on structural information. Recent progress in generative adversarial networks with progressive training has made it possible to generate high-resolution images. However, existing approaches have limitations in achieving both high image quality and structural consistency at the same time.

CHAPTER 3: SYSTEM SPECIFICATIONS

3.1 Introduction:

A System Requirements Specification (SRS) (also known as a Software Requirements Specification) is a set of documentation that describes the features and behavior of a system or software application. It includes a variety of elements (see below) that attempts to define the intended functionality required by the customer to satisfy their different users.
Now, this system has intended in such a way that it takes fewer resources to work out properly. The development of this project needs the following hardware and software requirements.

3.2 Software Requirements:

- Language / Packages : Python
- Version : Python 3.8.5
- Operating systems : Windows 10
- Communication protocol : HTTP Protocol

3.3 Hardware Requirements:

- Processor Clock speed : AMD Ryzen 5 with 2.0GHz
- RAM : 12 GB
- Hard disk capacity : 30 GB
- GPU : 4GB Nividia GeForce GTX 1050

3.4 Budget

The budget of completion for developing the heart disease prediction system will require various software and hardware devices. The application is averagely expensive to build but if happens to be as successful as the developer sees it to be it will bring forth enough profit to cover the costs undergone.

The table below explains the planned budget in Indian Rupee to develop the system:	**SOFTWARE / COMPANY**	**PRICE**
HARDWARE		
Computer	Operating System	Windows - 25000
Hard disk	Cloud storage	8,500
GPU	Nividia Gtx1050	10,000
Internet Connection	Atria Convergence Technologies	3,000
Total	46,500	

CHAPTER 4: DATASETS

1.1 Data collection and preprocessing

There are number of preexisting datasets which have the required data, but the nature of this project was to attempt the learn variations or “volatile palette” of anime characters. So, we scraped around 86,000 images from internet over the course of 5 days.

Fig -2

This dataset consists of 86,000 images which will be used for training in an unsupervised manner to identify latent distribution. The main challenge with such a vast dataset is there is too much variance in patterns and feature maps of images which in turn will affect model if architecture is not sufficient enough to learn from this pool. To counter this, we used an iterative approach where we progressively increase the complexity of architecture until we approximate optimal architecture and parameters to cater to this data.
Each image is resized to 24*24*3 and normalized to maintain consistency and also to make sure that it doesnt tax too much on GPU.

CHAPTER 5: METHODS AND ALGORITHMS USED

The main Objective of this project is to demonstrate the power of GANs by attempting to learn “volatile patterns”. as stated earlier, we are going to start at basic GANs and work our way towards DCGANs

5.1 Generative Adversarial Networks

Generative adversarial networks are a unique breed of Unsupervised Deep learning algorithms. They are made by combination of a ANN and ENN. They emulate a zero sum game where any contributions made by one are always negated by the latter, this form of learning is attributed to Game theory, where both NN compete to reduce their loss, only to be obstructed by another network. But on larger context, they both improve over time.

A simple GANs consists of 2 parts/NN:

- Discriminator

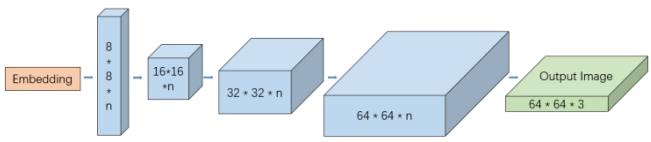


Fig -3

- Generator

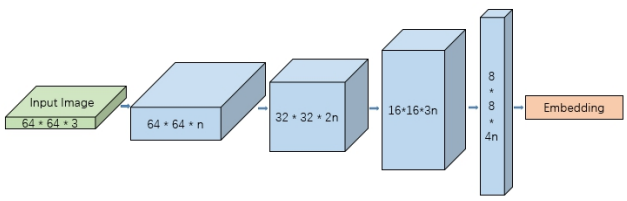


Fig-4

generator attempts to generate a real sample to spoof the discriminator, and the discriminator attempts to distinguish between the real sample and the generated sample from the generator.this is basis of a GAN.

5.1.a Generator Model

The generator model takes a fixed-length random vector as input and generates a sample in the domain.

The vector is drawn from randomly from a Gaussian distribution, and the vector is used to seed the generative process. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution.This vector space is referred to as a latent space, or a vector space comprised of latent variables. Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.

A latent variable is a random variable that we cannot observe directly.

We often refer to latent variables, or a latent space, as a projection or compression of a data distribution. That is, a latent space provides a compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples.

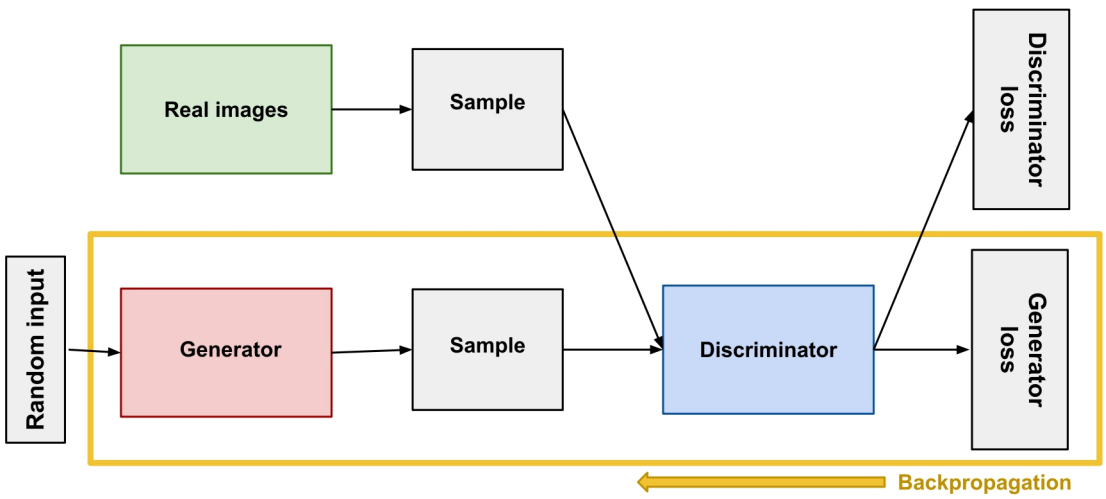


Fig-5

5.1.b Discriminator Model

The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated).

The real example comes from the training dataset. The generated examples are output by the generator model.

The discriminator is a normal (and well understood) classification model.

After the training process, the discriminator model is discarded as we are interested in the generator.

Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data.

> *We propose that one way to build good image representations is by training Generative Adversarial Networks (GANs), and later reusing parts of the generator and discriminator networks as feature extractors for supervised tasks*

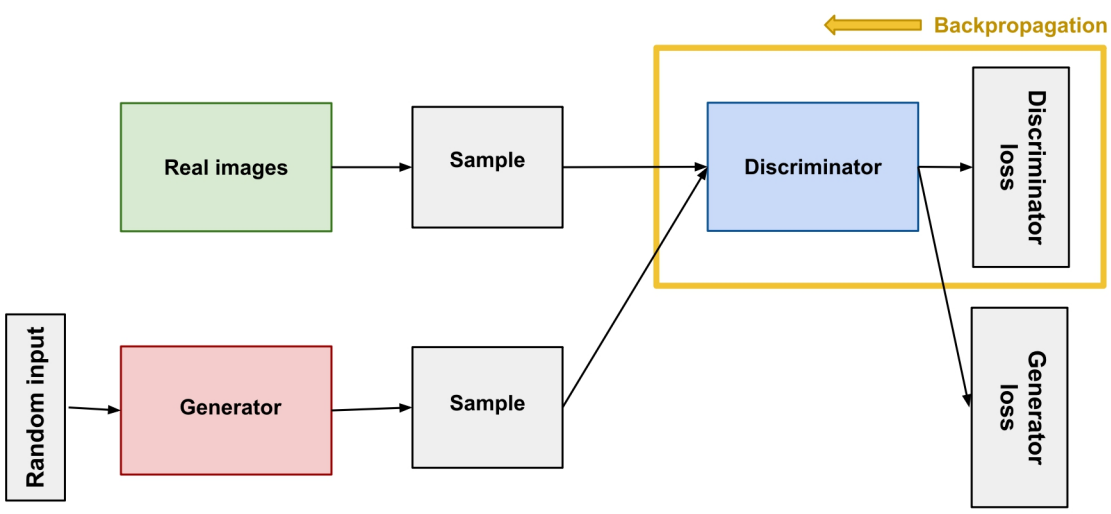


Fig - 6

5.2 GAN

Because a GAN contains two separately trained networks, its training algorithm must address two complications:

1. GANs must juggle two different kinds of training

   (generator and discriminator).

2. GAN convergence is hard to identify.

The generator and the discriminator have different training processes.GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs.
2. The generator trains for one or more epochs.

We keep the generator constant during the discriminator training phase. As discriminator training tries to figure out how to distinguish real data from fake, it has to learn how to recognize the generator's flaws. That's a different problem for a thoroughly trained generator than it is for an untrained generator that produces random output.

Similarly, we keep the discriminator constant during the generator training phase. Otherwise the generator would be trying to hit a moving target and might never converge.

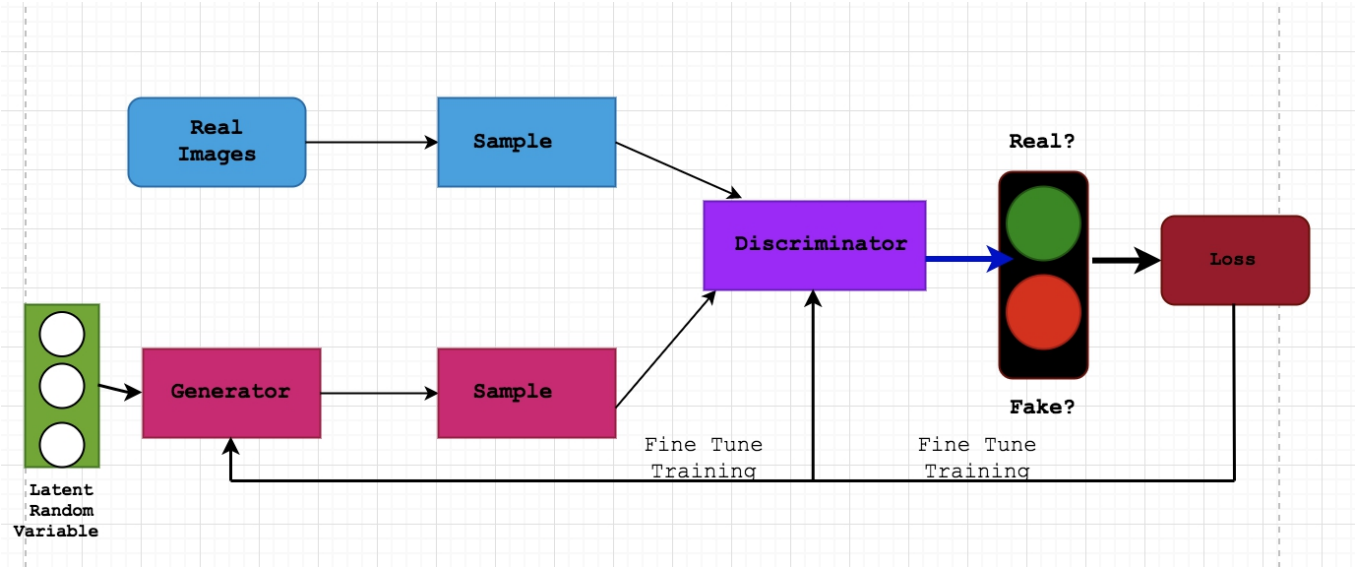


Fig - 7

CHAPTER 6: EXPERIMENTS

6.1 Data preprocessing

6.1.1 Image Preprocessing

With 86,000 images, the task is very difficult to manage using ImageDataGenerator as preproduction experiments suggested that ImageDataGenerator provided in Tensorflow library is too restrictive in terms of image sampling and it only works with single NN architectures as opposed to Multi NN architectures as we have implemented, in attempt to replicate process of ImageDataGenerator we have defined a function and used GLOB to process images into an Tensor at batch size of 64, this function reads 64 random images from folder and stores in a Tensor variable. We Resized all images to 64*64*3 size , normalized,converted them to Tensor and stored in an array of (86,000*64*64*3) array.

6.1.2: Noise Sampling

An integral part of GANs is to find the latent distribution of dataset, to find this, we always make use of Control Distribution, to pick Control Distribution, we go with popular choice of Normal distribution. Wherein, for every step in epoch we generate a random noise of size 64*64*1 and send it through generator to get output.

6.1.3 Train loop

Since Tensorflow doesn't have support for multi NN architectures yet, we have to get creative and define our own training loop which trains GAN in a certain way

1. Sample real images
2. Sample fake images by generator
3. Train discriminator on real and fake images
4. Freeze discriminator weights

5. Train Generator on fake images

6. Update losses and perform Gradient descent

7. Append losses and log them

8. Save weights at certain epochs for check-pointing

6.2 Final Architectures and Hyper-parameters

6.2.1a : discriminator architecture

Discriminator's Architecture is relatively easier as compared to Generator which is 200x hard to make.it consists of

1. 3 Convolution layers with 4*4 kernels, stride 2 parameters and Leaky ReLu activation function

2. flatten layer and dropout with p value 0.4

3. Dense layer/FC layer with 1 out_features and SIGMOID activation.

```
def discriminator_model(img_size):
  i = Input(shape = img_size)
  x = Conv2D(128, 3)(i)
  x = LeakyReLU()(x)
  # out_dim : 62 x 62 x128

  x = Conv2D(128, 4, strides = 2)(x)
  x = LeakyReLU()(x)
  # out_dim : 30 x 30 x 128

  x = Conv2D(128, 4, strides = 2)(x)
  x = LeakyReLU()(x)
  # out_dim : 14 x 14 x 128

  x = Conv2D(128, 4, strides = 2)(x)
  x = LeakyReLU()(x)
  x = Flatten()(x)
  # out_dim : 1 x 4608

  x = Dropout(0.4)(x)

  x = Dense(1, activation = "sigmoid")(x)

  model = Model(i,x,name = "discriminator")
  return model
```

Fig - 8

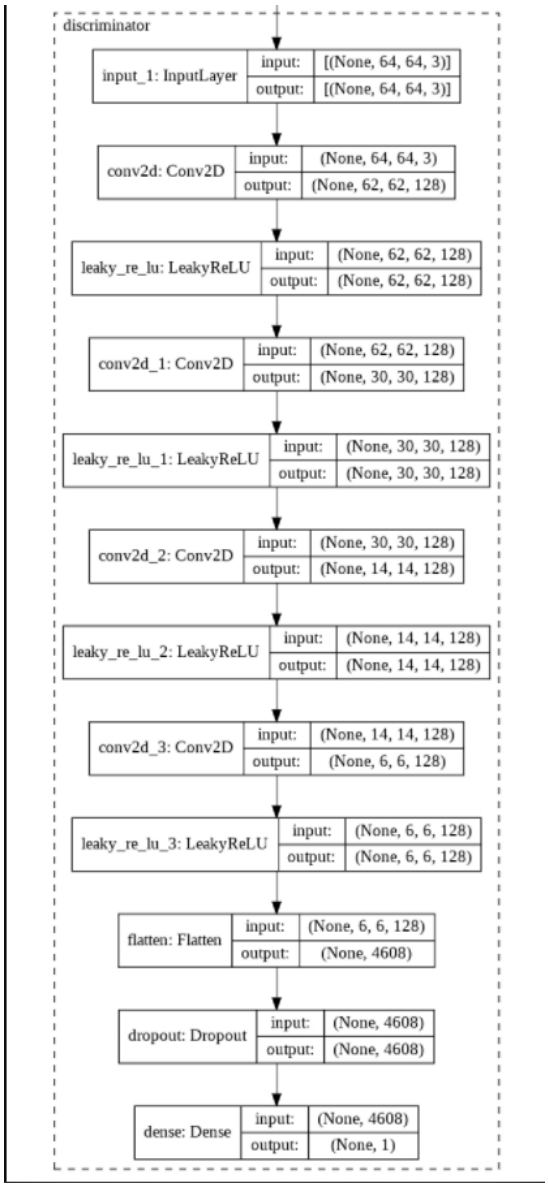


Fig - 9

6.2.1b: Hyper-parameters and optimizers

We used input size of 64*64*3 in discriminator and found that Root Mean Square Propagation(RMSprop) optimizer with Learning rate of 0.0005, clip_val 1.0 and decay rate of (10^-8) would be sufficient with loss function “Binary Cross Entropy ” which is also known as Log-loss.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Fig-10

6.2.2a Generator architecture

Generator Architecture is by far the Most challenging aspect of this project, we went over dozens of revisions, sometimes even overhauling entire architecture from scratch because we weren't satisfied with the results obtained.after, considering around 4 Versions with 8 revisions of Generator in each version. We were satisfied with a architecture which was giving us relatively good results under certain conditions and hyper-parameters.

Generator architecture is as follows:

1. Input with latent space of 100
2. A dense layer of 131072 neurons with Leaky ReLu activation
3. Reshape to 32*32*128
4. Convolution layer with 5*5 kernel with no padding and LRelu
5. De-convolution layer of 4*4 kernel with no padding and LRelu
6. 2 convolution layers of 5*5 kernel with no padding and LRelu activation
7. 1 convolution layer of 7*7 kernel with no padding and Tanh activation

```
def generator_model(latent_space,channels):
  i = Input(shape = (latent_space))

  x = Dense(128 * 32 * 32)(i)
  x = LeakyReLU()(x)
  x = Reshape((32, 32, 128))(x)
  #out_dim : 32 x 32 x 128

  x = Conv2D(256, 5, padding = "same")(x)
  x = LeakyReLU()(x)
  #out_dim : 32 x 32 x 256

  x = Conv2DTranspose(256, 4, strides = 2, padding = "same")(x)
  x = LeakyReLU()(x)
  #out_dim : 64 x 64 x 256

  x = Conv2D(256, 5, padding = "same")(x)
  x = LeakyReLU()(x)
  x = Conv2D(256, 5, padding = "same")(x)
  x = LeakyReLU()(x)
  #out_dim : 64 x 64 x 256

  x = Conv2D(channels, 7, activation = "tanh", padding = "same")(x)
  #out_dim : 64 x 64 x 3
  model = Model(i, x, name = "generator")
  return model
```

Fig -11

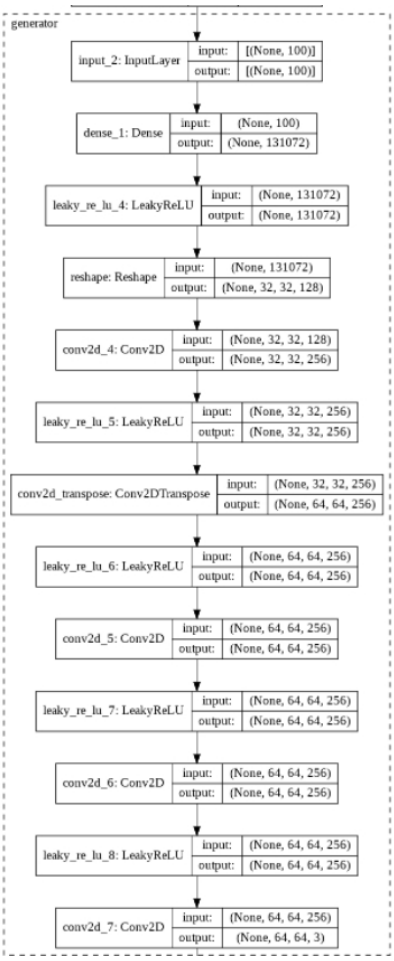


Fig -12

6.2.2b: Hyper-parameters and optimizers

Before we compile Gen, we have wrapped the whole network to Discriminator and made a combined model (c_model) and compiled it using loss: Binary cross Entropy, optimizer RMSprop with learning rate of 0.0005, clip val 1.0 and decay rate of (10^-8). C_model is combined model of Discriminator and Generator

```
dis = discriminator_model(img_size = (H,W,D))
gen = generator_model(latent_space,D)

dis_opti = RMSprop(lr = 0.0005, clipvalue = 1.0, decay = 1e-8)
dis.compile(optimizer = dis_opti, loss = "binary_crossentropy")

dis.trainable = False

gan_in = Input(shape = (latent_space,))
gan_out = dis(gen(gan_in))
c_model = Model(gan_in,gan_out)

gan_opti = RMSprop(lr = 0.0005, clipvalue = 1.0, decay = 1e-8)
c_model.compile(optimizer = gan_opti, loss = "binary_crossentropy")
```

Fig -13

CHAPTER 7:ARCHTECTURE DIAGRAMS

7.1 project flow diagram

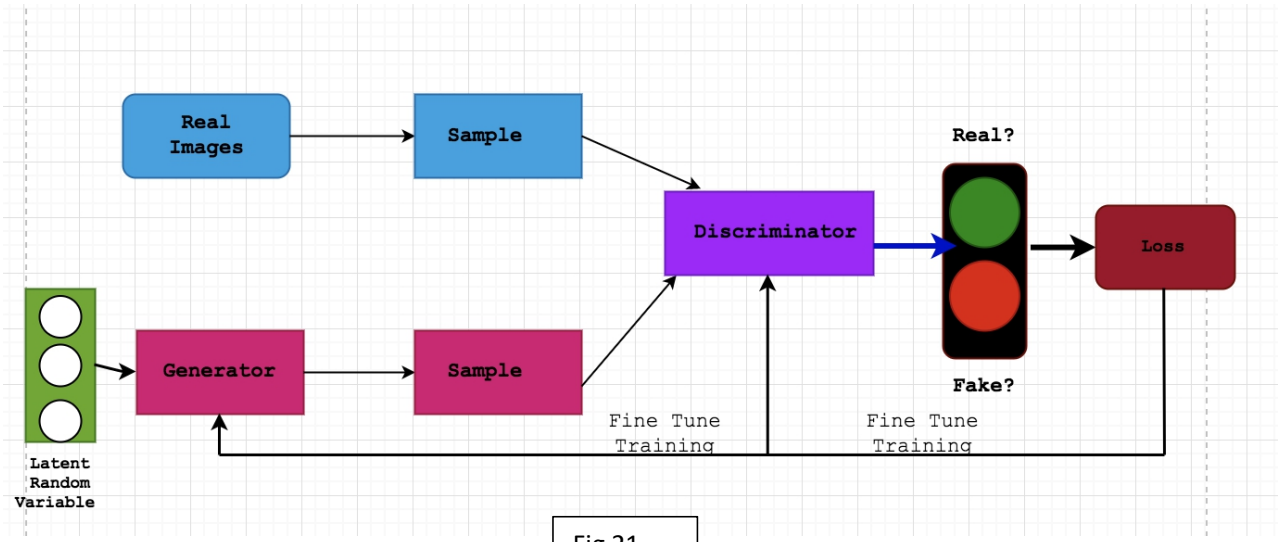


Fig 21

7.2 Data Flow Diagram

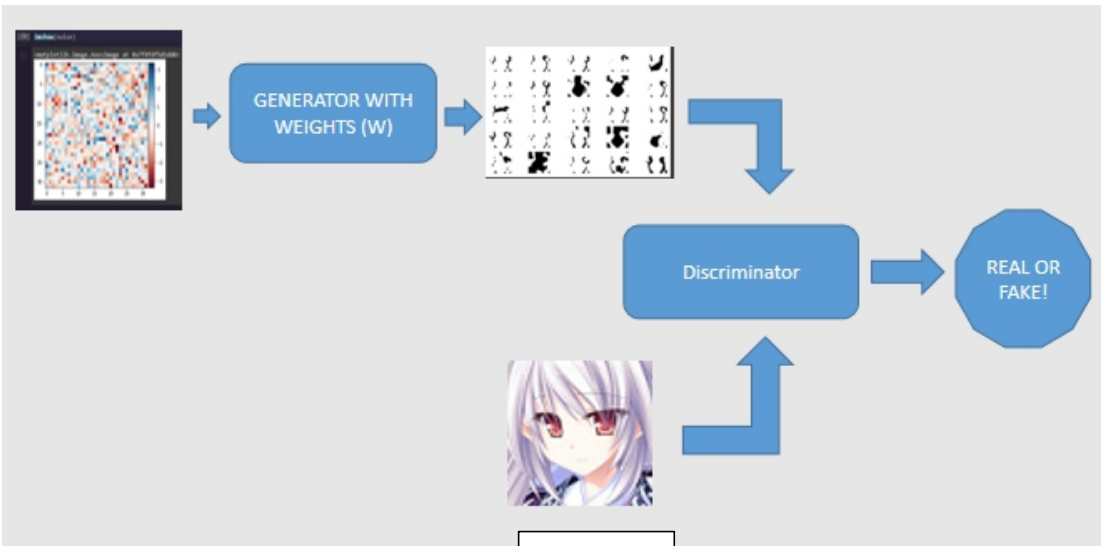


Fig 22

CHAPTER 8 :EVALUATION METRICS

8.1 Log-loss

For GANs there are no Evaluation metrics since they rely on empirical visual inspection to determine the quality of image generated using a particular architecture. Although these are not strictly Evaluation metrics, we believe these do count towards “evaluating” the model.

Loss measure - log loss:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Binary Cross-Entropy / Log Loss

Fig - 23

Binary_cross_entropy is a loss function that is used in binary classification tasks. These are tasks that answer a question with only two choices (yes or no, A or B, 0 or 1, left or right).

To Make GAN work, previously it is stated that both NN need to be in a Zero sum training. This is visual representation of zero sum training

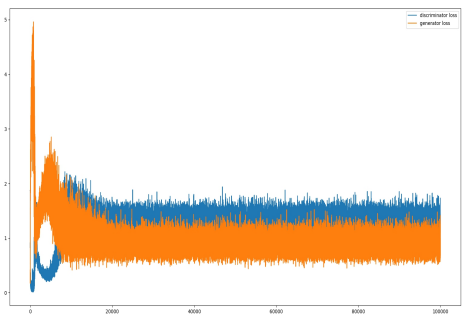


Fig -24

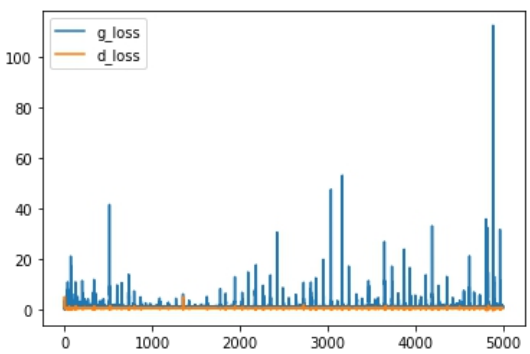


fig 25

here, blue is Discriminator Loss and orange is Gen loss. Notice how they negate each other. This is also known as Nash Equilibrium.

8.2 Results

Initially, the results were very disappointing as most of version 1 models fail to maintain nash equilibrium and fail to learn these patterns.we saw promising results from version 3 as these models learned silhouette of a face. After overhauling architecture in version 4, we saw extremely promising results as they were able to maintain nash equilibrium for a long time. But, they suffered from random inputs after often features of portrait in images were different and "deformed" at worse.

Version 1 results:

Notes: underfitting, generator not complex enough, reduce Learning rate,optimizer used : Adam

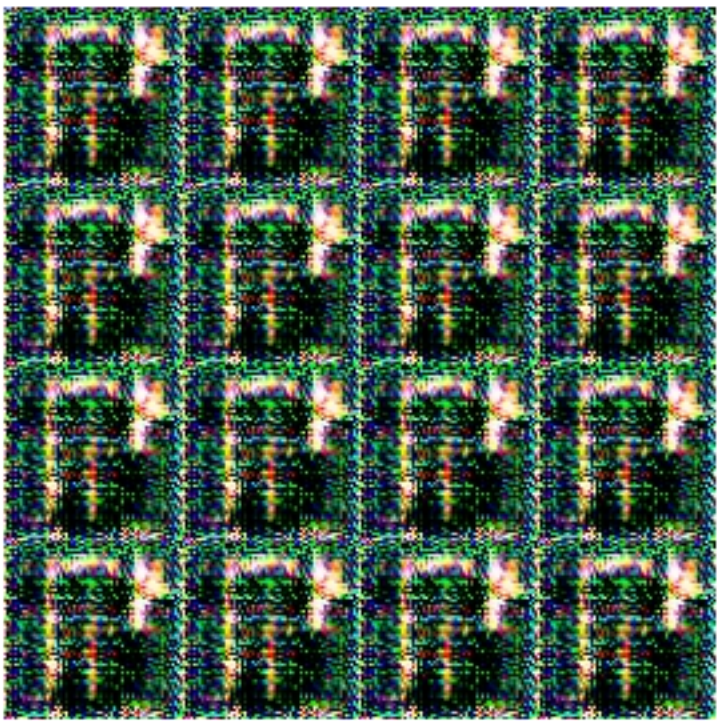


Fig 26

Version 2 results:

Notes: still not good, but it learned relative position of face. Increase Model complexity, lower Learning rate, switch optimizer to RMSprop

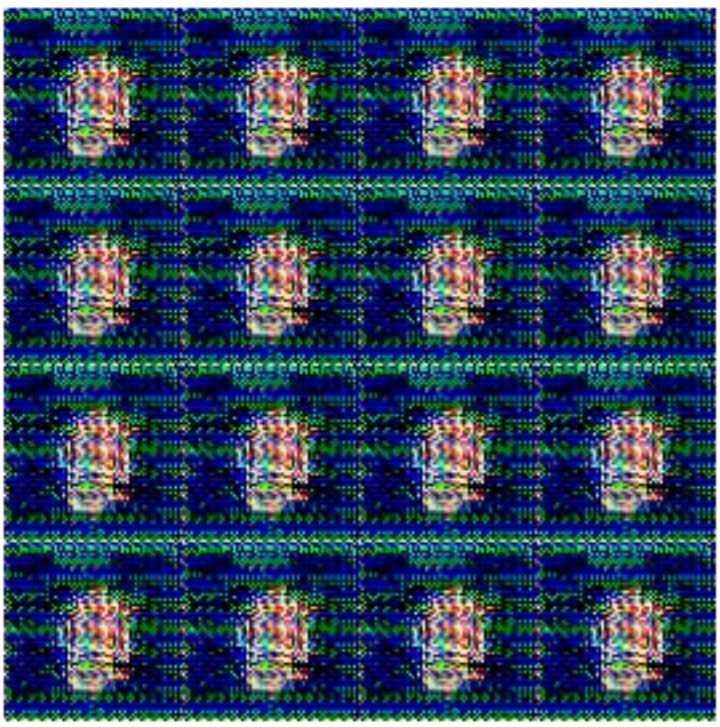


Fig 27

Version 3 results:

Notes : underfitting mitigated, facial structure is partially learned

Use decay learning rate, with initial rate of 0.01

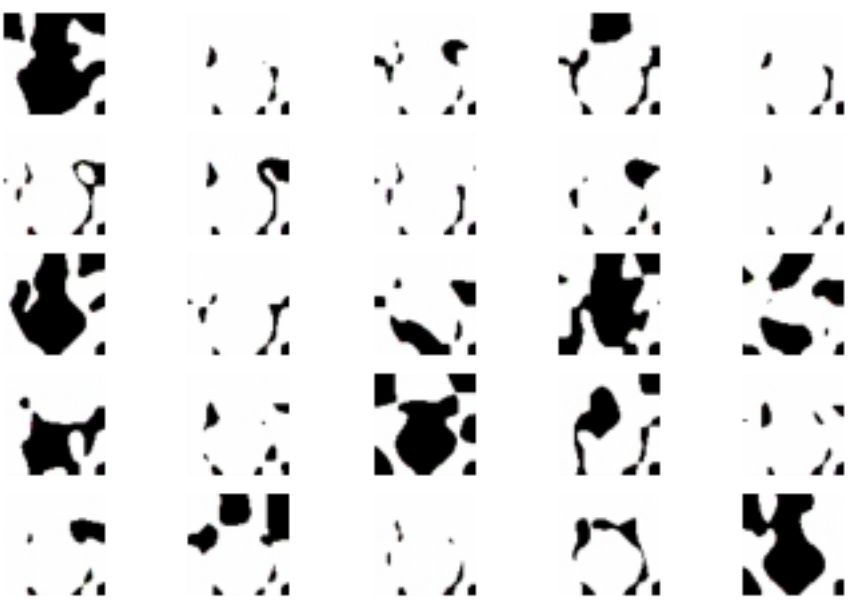


Fig 28

Version 4 results:

Notes: complexity is sufficient, but requires tweaking of layers, learning rate 0.0001, RMSprop.

Fig 29

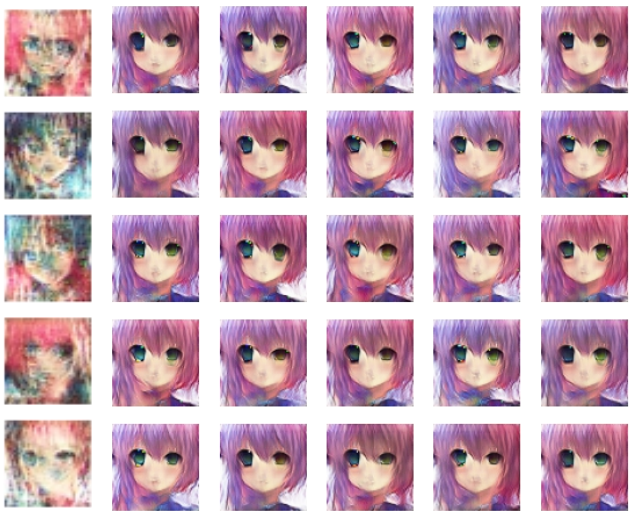


Fig 30

Version 5 results:

Notes: increased complexity, learning rate 0.0005, RMSprop, normalized , still has minor flaws as marked, but results are satisfactory.

Fig 31

CHAPTER 9: MODULE EXPLAINATION AND CODE

9.1 Training the GAN

This section goes into detail of the custom train loop we designed for this GAN to provide tangible results.

9.1.0 hyper parameters for train_loop

We define some variable/Hyper parameters to govern the loop and also to experiment to find best combination of settings required for this experiment.

1.Epochs : 5,000

2.Batch_size : 64

3.Index = 0

4.Learning rate : 0.0005

5.Log_rate : 500

6.Weight_rate : 10,000

9.1.1 Preprocessing/Sampling

9.1.1.1 sampling noise

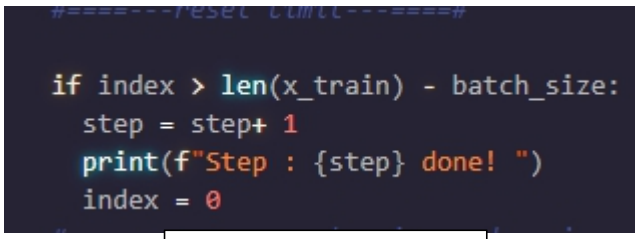
We sample 64 random numbers from normal distribution and store it in array of size (64*100) using

Numpy.random.normal(size = (batch_size, latent_space))

We use this noise to train both discriminator and Generator.

9.1.1.2 sampling data

We sample images in an iterative fashion using index to keep track of image indices and using step to reset index to avoid “out of index” error. A “step” is a counter we use to represent whenever c_model has gone through whole 86,000 images ONCE.



```
#-------reset limit-------#

if index > len(x_train) - batch_size:
  step = step+ 1
  print(f"Step : {step} done! ")
  index = 0
```

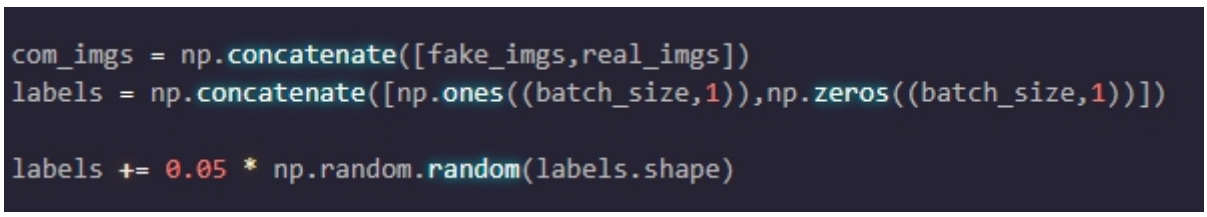
Fig :14

9.1.1.3 combining data/making labels

We combine sampled images, both fake and real and combine them and assign labels

Real images : 1

Fake images : 0



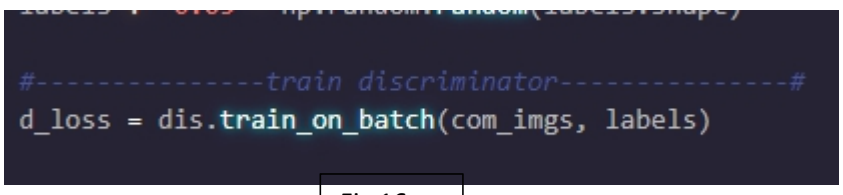
```
com_imgs = np.concatenate([fake_imgs,real_imgs])
labels = np.concatenate([np.ones((batch_size,1)),np.zeros((batch_size,1))])

labels += 0.05 * np.random.random(labels.shape)
```

Fig 15

9.1.2 Training NN

9.1.2.1 training discriminator

Discriminator is trained on (128*64*64*1) dimension Tensor which consists of 64 real images and 64 fake images and (128*1) refering to labels of images.



```
#---------------train discriminator---------------#
d_loss = dis.train_on_batch(com_imgs, labels)
```

Fig 16

9.1.2.2 training Generator

Generator is trained on noise to produce fake images and use discriminator as judge to assess the pattern that is generated by generator. It takes in tensor of random inputs of shape (64*100) with label 1 to produce (64*64*64*3) images

```
#---------------train generator-------------------#
noise = np.random.normal(size = (batch_size,latent_space))
f_labels = np.zeros((batch_size,1))
g_loss = c_model.train_on_batch(noise,f_labels)
```

Fig 17

9.1.3 logging loss and weights

9.1.3.1 loss logs

We need to be able asses the performance graphs and be able to identify regions of mode collapse wherein generator overfits features of Discriminator which results in bad results. We used lists to maintain record losses to identify time and parameters where mode collapse occurs and we rectify this in next test.

```
#=======-----some logging-----=======#
index = index + batch_size
g_losses.append(g_loss)
d_losses.append(d_loss)
#=======---reset Limit---=======#
```

Fig 18

9.1.3.2 model weight logs

Training GAN is very taxing on GPU, thus it might take upto 10-12 hrs to train one model in interest on time we used checkpoints where model weights are saved at every 10,000 epochs to be more fault tolerant and resume from middle of training if Python 3.8.5 kernel dies due to load.

```
if epoch % 10000 == 0:
  c_model.save_weights(f"/content/weights/gan_weigths_{epoch}.h5")
  print(f"super_rare_find UwU Weights saved!")
```

Fig 19

9.1.4 Sampling Result images

after every 500 epoch we save loss and produce images to inspect any mode collapse and training process. This is done by plotting the images produced by generator(5*64*64*3) by using random input(25*100) using matplotlib.pyplot.subplots() .

```
def sample_imgs(epoch):
  rows,cols = 5,5
  noise = np.random.normal(size = (rows*cols,latent_space))
  gen_imgs = gen.predict(noise)
  fig,axes = plt.subplots(rows,cols)
  index = 0
  for i in range(rows):
    for j in range(cols):
      temp_img = resize(gen_imgs[index],(H,W))
      axes[i,j].imshow((temp_img*255).astype("uint8"))
      axes[i,j].axis("off")
      index = index + 1
  plt.show()
  plt.savefig(f"/content/gan_images/gan_{epoch}.png")
```

Fig 20

CHAPTER 10: CONCLUSION

We explore the automatic creation of the anime characters in this work. By combining a clean dataset and several practicable GAN training strategies, we successfully build a model which can generate realistic facial images of anime characters.This proposed model does produce tangible results as intended. But, there is a lot of room for improvement, mainly using LOGAN or SRGAN.There still remain some issues for us for further investigations. One direction is how to improve the GAN model when class labels in the training data are not evenly distributed.

References

REFERENCE PAPER
[1] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. arXiv preprint arXiv:1701.04862, 2017.
[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. arXiv preprint arXiv:1701.07875, 2017.
Sanjeev Arora, Rong Ge, Yingyu Liang, Tengyu Ma, and Yi Zhang. Generalization and equilibrium in generative adversarial nets (gans). arXiv preprint arXiv:1703.00573, 2017.
Marc G Bellemare, Ivo Danihelka, Will Dabney, Shakir Mohamed, Balaji Lakshminarayanan, Stephan Hoyer, and Rémi Munos. The cramer distance as a solution to biased wasserstein gradients. arXiv preprint arXiv:1705.10743, 2017.
Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Advances in neural information processing systems, pages 2672–2680, 2014.
Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. arXiv preprint arXiv:1704.00028, 2017.
Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a nash equilibrium. arXiv preprint arXiv:1706.08500, 2017.
Hiroshiba. Girl friend factory. http://qiita.com/Hiroshiba/items/d5749d8896613e6f0b48, 2016.
Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional

adversarial networks. arXiv preprint arXiv:1611.07004, 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

Naveen Kodali, Jacob Abernethy, James Hays, and Zsolt Kira. How to train your dragan. arXiv preprint arXiv:1705.07215, 2017.