**GENERATING ART USING GAN**

Submitted in partial fulfilment of the requirements of the degree of

BACHELOR OF COMPUTER ENGINEERING

# by

Roll No 16 Zenil Gosher Moodle id: 19102045

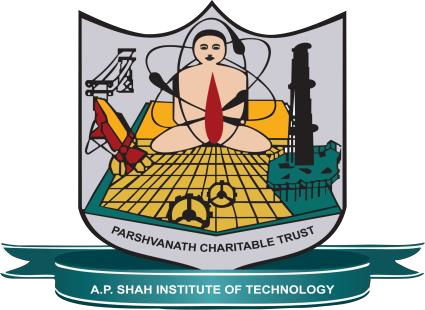
Roll no: 24 Jayesh Jain Moodle id: 19102021

Roll no: 36 Hardika Lalwani Moodle id: 20202003

Roll no: 77 Jainam Zaveri Moodle id: 20202007

Guide:

Prof. Brinal Colaco



Department of Computer Engineering

A. P. SHAH INSTITUTE OF TECHNOLOGY, THANE

(2021-2022)



# CERTIFICATE

# 

This is to certify that the Mini Project 2B entitled “**Generating art using GAN**” is a bonafide work of **“Zenil Gosher (**19102045)**, Jayesh Jain (**19102021)**, Hardika Lalwani (20202003), Jainam Zaveri (20202007)”** submitted to the University of Mumbai in partial ulfilment of the requirement for the award of the degree of **Bachelor of Engineering** in **Computer Engineering**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Name and Sign)

Guide

\_\_\_\_\_\_\_\_\_\_\_\_\_

(Name and Sign)

Head of Department

# 

# **Project Report Approval for TE**

This Mini project report entitled “**Generating art using Gann**” by **“Zenil Gosher (19102045), Jayesh Jain (19102021), Hardika Lalwani (20202003), Jainam Zaveri (20202007)”** is approved for the degree of ***Bachelor of Engineering*** in ***Computer Engineering***, ***2021-22***.

Examiner Name Signature 1.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:

Place: Thane

**Declaration**

We declare that this written submission represents our ideas in our own words and where others’ ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed. -

(Signature)

**Zenil Gosher (19102045)**

(Signature)

**Jayesh Jain (19102021)**

(Signature)

**Hardika Lalwani (20202003)**

(signature)

**Jainam Zaveri (20202007)**

Date:

Place: Thane

**Abstract**

CycleGAN incorporates a novel unpaired image-to-image translation technique that can be used for style transfer. CNNs have been usually used with paired datasets. This approach is a very human way of unsupervised learning because the model learns what not to do and then improves itself over time. The loss functions evaluate the accuracy or the similarity between real and fake images. CycleGAN provides us the ability to perceive the plausible what-if situation? Like how it was possible for us to translate input images into a Monet-esque paintings even after a century. With our implementation we were able to see, how a Monet painting would look today. It gives us the ability to perceive different styles, through the vision of the artist.

**CONTENTS**

1 Introduction ........................………………………………………………………………1

2 Literature Survey ………………...………………………....…….………………………2

1. Problem Statement.………………………...…...………………………………….……...4
2. Objective and Scope....….………………………………………………..……………….5
3. Experimental Setup.…...…………………………………………..………………………6
   1. Hardware Requirement
   2. Software Requirement
4. System Design……………………………………………………………………………7
   1. UML Diagram
   2. Algorithm/Process (with Expected input and outputs)
5. Implementation.….…………...………………………………………………………….11
   1. Code
   2. Screenshots
6. Result………....………………...……………………….………………….……………0
7. Conclusion………………………………………………………...………….………….0

References ............................................................................................................................0

Annexture 1- Project Planning (Using Gantt chart) ……..………………………………..0

LIST OF FIGURES

1. **Fig 1 Output 1** ...............................................................................11

2.Fig 2 Output 2 .................................................................................12

3 Fig 3 ……………………………………………………………….13

LIST OF TABLES

1.1 name of table ................................................................................................................0

2.1 name of table ................................................................................................................0

LIST OF GRAPHS

1.1 name of graph................................................................................................................0

2.2 name of graph……………………………... ................................................................0

**CHAPTER 1:**

1. **Introduction**

Generative modelling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset. Generative Adversarial Networks or GANs, however, use neural networks for a very different purpose: Generative modelling. Generative Adversarial Networks belong to the set of generative models. These models are capable of generating new data completely on their own.

A GAN consists of at least two neural networks: a generator model and a discriminator model. The generator is a neural network that creates the images. We should generate images in the style of the artist This generator is trained using a discriminator. The two models will work against each other, with the generator trying to trick the discriminator, and the discriminator trying to accurately classify the real vs. generated images.

Our project demonstrates unpaired image to image translation using conditional GAN's, as described in Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, also known as CycleGAN. proposes a method that can capture the characteristics of one image domain and figure out how these characteristics could be translated into another image domain, all in the absence of any paired training examples. Cycle GAN uses a cycle consistency loss to enable training without the need for paired data. In other words, it can translate from one domain to another without a one-to-one mapping between the source and target domain. This opens up the possibility to do a lot of interesting tasks like photo-enhancement, image colorization, style transfer, etc. All you need is the source and the target dataset which simply encompasses a directory of images

**CHAPTER 2:**

1. **Literature Survey**

Ian Goodfellow, the inventor of GAN proposed a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to ½ everywhere. In the case where G and D are defined by multilayer perceptron’s, the entire system can be trained with backpropagation.

In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistinguishable from the genuine articles.

Many problems in image processing, graphics, and vision involve translating an input image into a corresponding output image. These problems are often treated with application-specific algorithms, even though the setting is always the same: map pixels to pixels. Conditional adversarial nets are a general-purpose solution that appears to work well on a wide variety of these problems. Phillip Isola’s paper on Image-to-Image Translation with Conditional Adversarial Networks proposes a framework that simplifies a very basic problem. In analogy to automatic language translation, we define automatic image-to-image translation as the task of translating one possible representation of a scene into another, given sufficient training data (see Figure 1). Traditionally, each of these tasks has been tackled with separate, special-purpose machinery (e.g., [16, 25, 20, 9, 11, 53, 33, 39, 18, 58, 62]), despite the fact that the setting is always the same: predict pixels from pixels is always the same: predict pixels from pixels.

CNNs learn to minimize a loss function – an objective that scores the quality of results – and although the learning process is automatic, a lot of manual effort still goes into designing effective losses. In other words, we still have to tell CNN what we wish it to minimize.

GANs learn a loss that tries to classify if the output image is real or fake, while simultaneously training a generative model to minimize this loss. Blurry images will not be tolerated since they look obviously fake. Because GANs learn a loss that adapts to the data, they can be applied to a multitude of tasks that traditionally would require very different kinds of loss functions

We explore GANs in the conditional setting. Just as GANs learn a generative model of data, conditional GANs (cGANs) learn a conditional generative model . This makes cGANs suitable for image-to-image translation tasks, where we condition on an input image and generate a corresponding output image.

**CHAPTER 3:**

1. **Problem Statement**

Generating stylistically inspired artwork of modern settings by implementing a GAN, based on the works of legendary artists.

**CHAPTER 4:**

1. **Objective and Scope**

**Objectives:**

* We present a method that can learn to do the same: capturing special characteristics of one image collection and figuring out how these characteristics could be translated into the other image collection, all in the absence of any paired training examples.
* Capturing the styles of legendary artists like Picasso, Monet etc. and giving modern images their signature touch.

**Scope:**

* It is possible to create modern artworks by leveraging the styles of legacy artists.
* With artists there’s always a feeling of what they would do next if they had more time. By studying the individual styles of the artists, it is possible to create what they would paint if they were present now and create new masterpieces.
* We can actually see how they would portray a modern setting in their style.

**CHAPTER 5:**

1. **Experimental Setup**
2. Hardware Requirements:

* **Windows**
* **GPU and TPU**

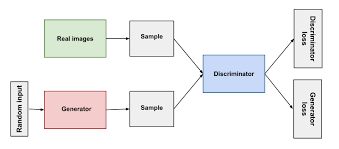
1. Software Requirement:

* **PyTorch:** PyTorch is an open-source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing,
* **Tensorflow:** TensorFlow is an end-to-end open-source platform for machine learning.
* **Numpy:** NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.
* **Pandas:** pandas is a software library written for the Python programming language for data manipulation and analysis
* **Matplotlib:** Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
* **Google colab:** Colab notebooks allow you to combine executable code and rich text in a single document, along with images, HTML, LaTeX and more.

**CHAPTER 6:**

1. **System Design**

 Data flow diagram



**Discriminator:**

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying. The discriminator's training data comes from two sources: Real data instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.

Fake data instances created by the generator. The discriminator uses these instances as negative examples during training.

**Generator:**

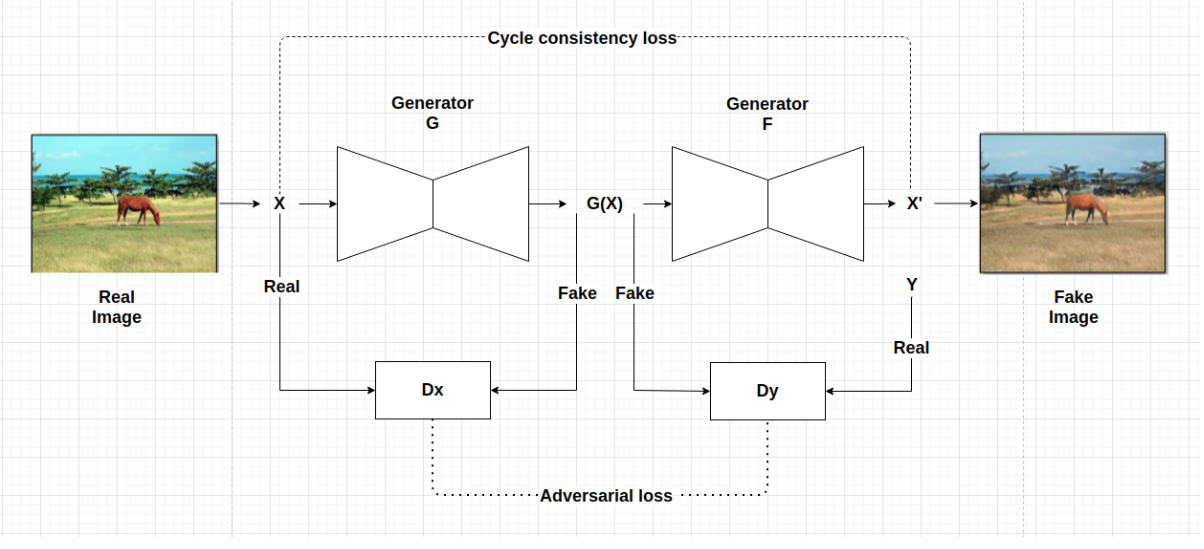
The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real. Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes: random input generator network, which transforms the random input into a data instance discriminator network, which classifies the generated data discriminator output generator loss, which penalizes the generator for failing to fool the discriminator.

**Loss functions:**

The generator tries to minimize the following function while the discriminator tries to maximize it:   Ex[log(D(x))]+Ez[log(1−D(G(z)))]

In this function:

* D(x) is the discriminator's estimate of the probability that real data instance x is real.
* Ex is the expected value over all real data instances.
* G(z) is the generator's output when given noise z.
* D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
* Ez is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
* The formula derives from the cross-entropy between the real and generated distributions.
  1. **UML Diagram**

****

We use dual generator in continuous to translate x to y and next restrain back y to x . This architecture create a latent space to transform image without predefined paired image. Cycle consistency loss function is applied to keep distribution of output of dual generator process not far from input.

* 1. **Algorithm/Process (with Expected input and outputs)**

There are 2 generators (G and F) and 2 discriminators (X and Y) being trained here.

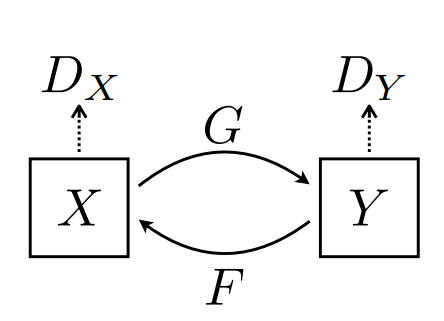
* Generator G learns to transform image X to image Y.

(G:X−>Y)

* Generator F learns to transform image Y to image X.

(F:Y−>X)

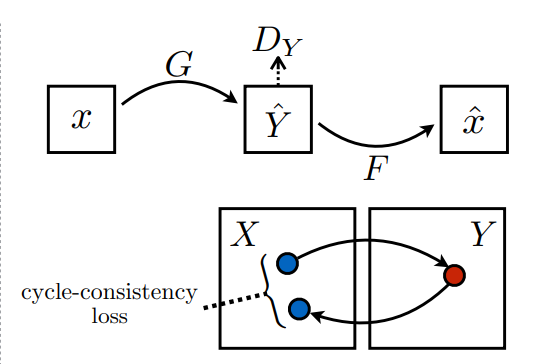
* Discriminator D\_X learns to differentiate between image X and generated image X (F(Y)).
* Discriminator D\_Y learns to differentiate between image Y and generated image Y (G(X)).



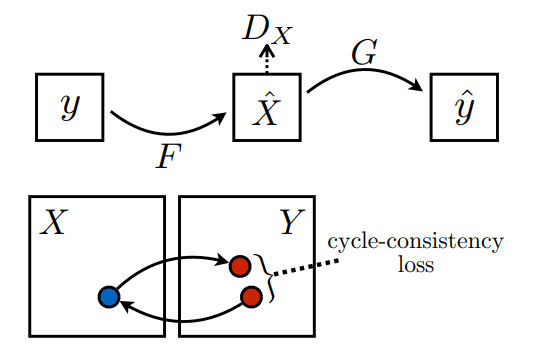
To further regularize the mappings, we introduce two cycle consistency losses that capture the intuition that if

we translate from one domain to the other and back again we should arrive at where we started:  1.forward cycle-consistency

loss: x → G(x) → F(G(x)) ≈ x



2. backward cycle-consistency loss: y → F(y) → G(F(y)) ≈ y



**CHAPTER 7:**

1. **Implementation**
   1. **Code**

**Importing Required Libraries**

from \_\_future\_\_ import absolute\_import, division, print\_function

import sys

print("Python version")

print (sys.version)

print("Version info.")

print (sys.version\_info)

*#!{sys.executable} -m pip install --upgrade pip*

*#!{sys.executable} -m pip install numpy*

*#!{sys.executable} -m pip install pandas*

*#!{sys.executable} -m pip install tensorflow\_addons*

*#!pip install PyDrive*

*#! gdown --id 11dzPv9LxegCDJTNE0RVCJD9hi2sxjQb4*

*#! mkdir ~/.kaggle*

*#! cp kaggle.json ~/.kaggle/*

*#! chmod 600 ~/.kaggle/kaggle.json*

*#! kaggle competitions download -c gan-getting-started*

*#! unzip gan-getting-started*

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

import tensorflow\_addons as tfa

import matplotlib.pyplot as plt

import numpy as np

import re

import os

import gzip, pickle

from scipy import linalg

import pathlib

import urllib

import warnings

from tqdm import tqdm

import os.path

from os import path

from PIL import Image

try:

from kaggle\_datasets import KaggleDatasets

except:

pass

import PIL

import torch

import torch.nn as nn

import torch.nn.functional as F

import torchvision

from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter

import imageio

from torch.autograd import Variable

from torch.nn.functional import adaptive\_avg\_pool2d

from functools import reduce

import cv2

import pandas as pd

import shutil

try:

from torchvision.models.utils import load\_state\_dict\_from\_url

except **ImportError**:

from torch.utils.model\_zoo import load\_url as load\_state\_dict\_from\_url

# **Connecting to the TPU**

In [2]:

try:

*#Get the TPU address*

tpu\_resolver = tf.distribute.cluster\_resolver.TPUClusterResolver()

print('Running on TPU ', tpu\_resolver.master())

except **ValueError**:

tpu\_resolver = None

if tpu\_resolver:

*#Connecting to the TPU*

tf.config.experimental\_connect\_to\_cluster(tpu\_resolver)

*#TPU initialisazion*

tf.tpu.experimental.initialize\_tpu\_system(tpu\_resolver)

*#TPU's trategy instanciation*

strategy = tf.distribute.TPUStrategy(tpu\_resolver)

else:

strategy = tf.distribute.get\_strategy()

*#Show replica number*

print("REPLICAS: ", strategy.num\_replicas\_in\_sync)

*#Set AUTOTUNE variable for optimization*

AUTOTUNE = tf.data.experimental.AUTOTUNE

# **CycleGAN Building**

Convolution layers building function

In [9]:

OUTPUT\_CHANNELS = 3

def down\_sample(filters, size, apply\_instancenorm=True):

initializer = tf.random\_normal\_initializer(0., 0.02)

gamma\_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

layer = keras.Sequential()

layer.add(layers.Conv2D(filters, size, strides=2, padding='same', kernel\_initializer=initializer, use\_bias=False))

if apply\_instancenorm:

layer.add(tfa.layers.InstanceNormalization(gamma\_initializer=gamma\_init))

layer.add(layers.LeakyReLU())

return layer

Transpose convolution layers building function

In [10]:

def up\_sample(filters, size, apply\_dropout=False):

initializer = tf.random\_normal\_initializer(0., 0.02)

gamma\_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

layer = keras.Sequential()

layer.add(layers.Conv2DTranspose(filters, size, strides=2, padding='same', kernel\_initializer=initializer,use\_bias=False))

layer.add(tfa.layers.InstanceNormalization(gamma\_initializer=gamma\_init))

if apply\_dropout:

layer.add(layers.Dropout(0.5))

layer.add(layers.ReLU())

return layer

Generator building function

In [11]:

def Generator():

inputs = layers.Input(shape=[256,256,3])

down\_stack = [

down\_sample(64, 4, apply\_instancenorm=False),

down\_sample(128, 4),

down\_sample(256, 4),

down\_sample(512, 4),

down\_sample(512, 4),

down\_sample(512, 4),

down\_sample(512, 4),

down\_sample(512, 4),

]

up\_stack = [

up\_sample(512, 4, apply\_dropout=True),

up\_sample(512, 4, apply\_dropout=True),

up\_sample(512, 4, apply\_dropout=True),

up\_sample(512, 4),

up\_sample(256, 4),

up\_sample(128, 4),

up\_sample(64, 4),

]

initializer = tf.random\_normal\_initializer(0., 0.02)

last = layers.Conv2DTranspose(3, 4, strides=2, padding='same', kernel\_initializer=initializer, activation='tanh')

x = inputs

*# Downsampling through the model*

skips = []

for down **in** down\_stack:

x = down(x)

skips.append(x)

skips = reversed(skips[:-1])

*# Upsampling and establishing the skip connections*

for up, skip **in** zip(up\_stack, skips):

x = up(x)

x = layers.Concatenate()([x, skip])

x = last(x)

return keras.Model(inputs=inputs, outputs=x)

def Discriminator():

initializer = tf.random\_normal\_initializer(0., 0.02)

gamma\_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

inp = layers.Input(shape=[256, 256, 3], name='input\_image')

x = inp

down1 = down\_sample(64, 4, False)(x)

down2 = down\_sample(128, 4)(down1)

down3 = down\_sample(256, 4)(down2)

zero\_pad1 = layers.ZeroPadding2D()(down3)

conv = layers.Conv2D(512, 4, strides=1, kernel\_initializer=initializer, use\_bias=False)(zero\_pad1)

norm1 = tfa.layers.InstanceNormalization(gamma\_initializer=gamma\_init)(conv)

leaky\_relu = layers.LeakyReLU()(norm1)

zero\_pad2 = layers.ZeroPadding2D()(leaky\_relu)

return tf.keras.Model(inputs=inp, outputs=zero\_pad2)

def DHead():

initializer = tf.random\_normal\_initializer(0., 0.02)

inp = layers.Input(shape=[33, 33, 512], name='input\_image')

x = inp

last = layers.Conv2D(1, 4, strides=1, kernel\_initializer=initializer)(x) *# (size, 30, 30, 1)*

return tf.keras.Model(inputs=inp, outputs=last)

Discriminator building function for real photo

In [14]:

def DiscriminatorP():

initializer = tf.random\_normal\_initializer(0., 0.02)

gamma\_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

inp = layers.Input(shape=[256, 256, 3], name='input\_image')

x = inp

down1 = down\_sample(64, 4, False)(x)

down2 = down\_sample(128, 4)(down1)

down3 = down\_sample(256, 4)(down2)

zero\_pad1 = layers.ZeroPadding2D()(down3)

conv = layers.Conv2D(512, 4, strides=1, kernel\_initializer=initializer, use\_bias=False)(zero\_pad1)

norm1 = tfa.layers.InstanceNormalization(gamma\_initializer=gamma\_init)(conv)

leaky\_relu = layers.LeakyReLU()(norm1)

zero\_pad2 = layers.ZeroPadding2D()(leaky\_relu)

last = layers.Conv2D(1, 4, strides=1, kernel\_initializer=initializer)(zero\_pad2)

return tf.keras.Model(inputs=inp, outputs=last)

CycleGAN building class definition

In [15]:

class **CycleGan**(keras.Model):

*#Executed function at the instanciation time*

def \_\_init\_\_(

self,

monet\_generator,

photo\_generator,

monet\_discriminator,

photo\_discriminator,

dhead1,

dhead2,

lambda\_cycle=3,

lambda\_id=3,

):

super(CycleGan, self).\_\_init\_\_()

self.m\_gen = monet\_generator

self.p\_gen = photo\_generator

self.m\_disc = monet\_discriminator

self.p\_disc = photo\_discriminator

self.lambda\_cycle = lambda\_cycle

self.lambda\_id = lambda\_id

self.dhead1 = dhead1

self.dhead2 = dhead2

def compile(

self,

m\_gen\_optimizer,

p\_gen\_optimizer,

m\_disc\_optimizer,

p\_disc\_optimizer,

gen\_loss\_fn1,

gen\_loss\_fn2,

disc\_loss\_fn1,

disc\_loss\_fn2,

cycle\_loss\_fn,

identity\_loss\_fn,

aug\_fn,

):

super(CycleGan, self).compile()

self.m\_gen\_optimizer = m\_gen\_optimizer

self.p\_gen\_optimizer = p\_gen\_optimizer

self.m\_disc\_optimizer = m\_disc\_optimizer

self.p\_disc\_optimizer = p\_disc\_optimizer

self.gen\_loss\_fn1 = gen\_loss\_fn1

self.gen\_loss\_fn2 = gen\_loss\_fn2

self.disc\_loss\_fn1 = disc\_loss\_fn1

self.disc\_loss\_fn2 = disc\_loss\_fn2

self.cycle\_loss\_fn = cycle\_loss\_fn

self.identity\_loss\_fn = identity\_loss\_fn

self.aug\_fn = aug\_fn

self.step\_num = 0

def train\_step(self, batch\_data):

real\_monet, real\_photo = batch\_data

batch\_size = tf.shape(real\_monet)[0]

with tf.GradientTape(persistent=True) as tape:

*# photo to monet back to photo*

fake\_monet = self.m\_gen(real\_photo, training=True)

cycled\_photo = self.p\_gen(fake\_monet, training=True)

*# monet to photo back to monet*

fake\_photo = self.p\_gen(real\_monet, training=True)

cycled\_monet = self.m\_gen(fake\_photo, training=True)

*# generating itself*

same\_monet = self.m\_gen(real\_monet, training=True)

same\_photo = self.p\_gen(real\_photo, training=True)

*# Diffaugment*

both\_monet = tf.concat([real\_monet, fake\_monet], axis=0)

aug\_monet = self.aug\_fn(both\_monet)

aug\_real\_monet = aug\_monet[:batch\_size]

aug\_fake\_monet = aug\_monet[batch\_size:]

*# two-objective discriminator*

disc\_fake\_monet1 = self.dhead1(self.m\_disc(aug\_fake\_monet, training=True), training=True)

disc\_real\_monet1 = self.dhead1(self.m\_disc(aug\_real\_monet, training=True), training=True)

disc\_fake\_monet2 = self.dhead2(self.m\_disc(aug\_fake\_monet, training=True), training=True)

disc\_real\_monet2 = self.dhead2(self.m\_disc(aug\_real\_monet, training=True), training=True)

monet\_gen\_loss1 = self.gen\_loss\_fn1(disc\_fake\_monet1)

monet\_disc\_loss1 = self.disc\_loss\_fn1(disc\_real\_monet1, disc\_fake\_monet1)

monet\_gen\_loss2 = self.gen\_loss\_fn2(disc\_fake\_monet2)

monet\_disc\_loss2 = self.disc\_loss\_fn2(disc\_real\_monet2, disc\_fake\_monet2)

monet\_gen\_loss = (monet\_gen\_loss1 + monet\_gen\_loss2) \* 0.4

monet\_disc\_loss = monet\_disc\_loss1 + monet\_disc\_loss2

*# discriminator used to check, inputing real images*

disc\_real\_photo = self.p\_disc(real\_photo, training=True)

*# discriminator used to check, inputing fake images*

disc\_fake\_photo = self.p\_disc(fake\_photo, training=True)

*# evaluates generator loss*

photo\_gen\_loss = self.gen\_loss\_fn1(disc\_fake\_photo)

*# evaluates discriminator loss*

photo\_disc\_loss = self.disc\_loss\_fn1(disc\_real\_photo, disc\_fake\_photo)

*# evaluates total generator loss*

total\_cycle\_loss = self.cycle\_loss\_fn(real\_monet, cycled\_monet, self.lambda\_cycle/ tf.cast(batch\_size,tf.float32)) + self.cycle\_loss\_fn(real\_photo, cycled\_photo, self.lambda\_cycle/ tf.cast(batch\_size,tf.float32))

*# evaluates total generator loss*

total\_monet\_gen\_loss = monet\_gen\_loss + total\_cycle\_loss + self.identity\_loss\_fn(real\_monet, same\_monet, self.lambda\_id / tf.cast(batch\_size,tf.float32))

total\_photo\_gen\_loss = photo\_gen\_loss + total\_cycle\_loss + self.identity\_loss\_fn(real\_photo, same\_photo, self.lambda\_id/ tf.cast(batch\_size,tf.float32))

*# Calculate the gradients for generator and discriminator*

monet\_generator\_gradients = tape.gradient(total\_monet\_gen\_loss,self.m\_gen.trainable\_variables)

photo\_generator\_gradients = tape.gradient(total\_photo\_gen\_loss,self.p\_gen.trainable\_variables)

monet\_discriminator\_gradients = tape.gradient(monet\_disc\_loss,

self.m\_disc.trainable\_variables)

photo\_discriminator\_gradients = tape.gradient(photo\_disc\_loss,

self.p\_disc.trainable\_variables)

*# Heads gradients*

monet\_head\_gradients = tape.gradient(monet\_disc\_loss1,

self.dhead1.trainable\_variables)

self.m\_disc\_optimizer.apply\_gradients(zip(monet\_head\_gradients,

self.dhead1.trainable\_variables))

monet\_head\_gradients = tape.gradient(monet\_disc\_loss2,

self.dhead2.trainable\_variables)

self.m\_disc\_optimizer.apply\_gradients(zip(monet\_head\_gradients,

self.dhead2.trainable\_variables))

*# Apply the gradients to the optimizer*

self.m\_gen\_optimizer.apply\_gradients(zip(monet\_generator\_gradients,

self.m\_gen.trainable\_variables))

self.p\_gen\_optimizer.apply\_gradients(zip(photo\_generator\_gradients,

self.p\_gen.trainable\_variables))

self.m\_disc\_optimizer.apply\_gradients(zip(monet\_discriminator\_gradients,

self.m\_disc.trainable\_variables))

self.p\_disc\_optimizer.apply\_gradients(zip(photo\_discriminator\_gradients,

self.p\_disc.trainable\_variables))

return {

'total\_monet\_gen\_loss': total\_monet\_gen\_loss,

'total\_photo\_gen\_loss': total\_photo\_gen\_loss,

'monet\_disc\_loss': monet\_disc\_loss,

'monet\_disc\_loss1':monet\_disc\_loss1,

'monet\_disc\_loss2':monet\_disc\_loss2,

'photo\_disc\_loss': photo\_disc\_loss,

}

# **Data-Augmentation functions**

In [20]:

with strategy.scope():

*#General Data-Augmentation calling function*

def DiffAugment(x, policy='', channels\_first=False):

if policy:

if channels\_first:

x = tf.transpose(x, [0, 2, 3, 1])

for p **in** policy.split(','):

for f **in** AUGMENT\_FNS[p]:

x = f(x)

if channels\_first:

x = tf.transpose(x, [0, 3, 1, 2])

return x

*#Random modification of brightness*

def rand\_brightness(x):

magnitude = tf.random.uniform([tf.shape(x)[0], 1, 1, 1]) - 0.5

x = x + magnitude

return x

*#Random modification of saturation*

def rand\_saturation(x):

magnitude = tf.random.uniform([tf.shape(x)[0], 1, 1, 1]) \* 2

x\_mean = tf.reduce\_sum(x, axis=3, keepdims=True) \* 0.3333333333333333333

x = (x - x\_mean) \* magnitude + x\_mean

return x

*#Random modification of contraste*

def rand\_contrast(x):

magnitude = tf.random.uniform([tf.shape(x)[0], 1, 1, 1]) + 0.5

x\_mean = tf.reduce\_sum(x, axis=[1, 2, 3], keepdims=True) \* 5.086e-6

x = (x - x\_mean) \* magnitude + x\_mean

return x

*#Image random translation*

def rand\_translation(x, ratio=0.125):

batch\_size = tf.shape(x)[0]

image\_size = tf.shape(x)[1:3]

shift = tf.cast(tf.cast(image\_size, tf.float32) \* ratio + 0.5, tf.int32)

translation\_x = tf.random.uniform([batch\_size, 1], -shift[0], shift[0] + 1, dtype=tf.int32)

translation\_y = tf.random.uniform([batch\_size, 1], -shift[1], shift[1] + 1, dtype=tf.int32)

grid\_x = tf.clip\_by\_value(tf.expand\_dims(tf.range(image\_size[0], dtype=tf.int32), 0) + translation\_x + 1, 0, image\_size[0] + 1)

grid\_y = tf.clip\_by\_value(tf.expand\_dims(tf.range(image\_size[1], dtype=tf.int32), 0) + translation\_y + 1, 0, image\_size[1] + 1)

x = tf.gather\_nd(tf.pad(x, [[0, 0], [1, 1], [0, 0], [0, 0]]), tf.expand\_dims(grid\_x, -1), batch\_dims=1)

x = tf.transpose(tf.gather\_nd(tf.pad(tf.transpose(x, [0, 2, 1, 3]), [[0, 0], [1, 1], [0, 0], [0, 0]]), tf.expand\_dims(grid\_y, -1), batch\_dims=1), [0, 2, 1, 3])

return x

*#Image random cutout*

def rand\_cutout(x, ratio=0.5):

batch\_size = tf.shape(x)[0]

image\_size = tf.shape(x)[1:3]

cutout\_size = tf.cast(tf.cast(image\_size, tf.float32) \* ratio + 0.5, tf.int32)

offset\_x = tf.random.uniform([tf.shape(x)[0], 1, 1], maxval=image\_size[0] + (1 - cutout\_size[0] % 2), dtype=tf.int32)

offset\_y = tf.random.uniform([tf.shape(x)[0], 1, 1], maxval=image\_size[1] + (1 - cutout\_size[1] % 2), dtype=tf.int32)

grid\_batch, grid\_x, grid\_y = tf.meshgrid(tf.range(batch\_size, dtype=tf.int32), tf.range(cutout\_size[0], dtype=tf.int32), tf.range(cutout\_size[1], dtype=tf.int32), indexing='ij')

cutout\_grid = tf.stack([grid\_batch, grid\_x + offset\_x - cutout\_size[0] // 2, grid\_y + offset\_y - cutout\_size[1] // 2], axis=-1)

mask\_shape = tf.stack([batch\_size, image\_size[0], image\_size[1]])

cutout\_grid = tf.maximum(cutout\_grid, 0)

cutout\_grid = tf.minimum(cutout\_grid, tf.reshape(mask\_shape - 1, [1, 1, 1, 3]))

mask = tf.maximum(1 - tf.scatter\_nd(cutout\_grid, tf.ones([batch\_size, cutout\_size[0], cutout\_size[1]], dtype=tf.float32), mask\_shape), 0)

x = x \* tf.expand\_dims(mask, axis=3)

return x

*#Dictionary of data augmentation methods*

AUGMENT\_FNS = {

'color': [rand\_brightness, rand\_saturation, rand\_contrast],

'translation': [rand\_translation],

'cutout': [rand\_cutout],

}

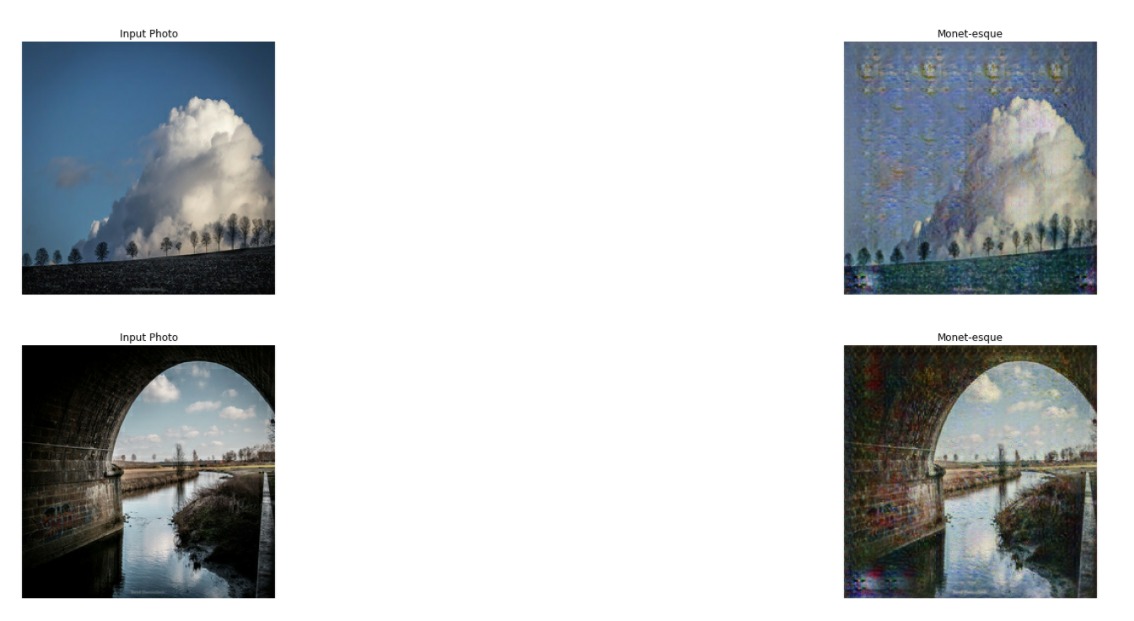
def aug\_fn(image):

return DiffAugment(image,"color,translation,cutout")

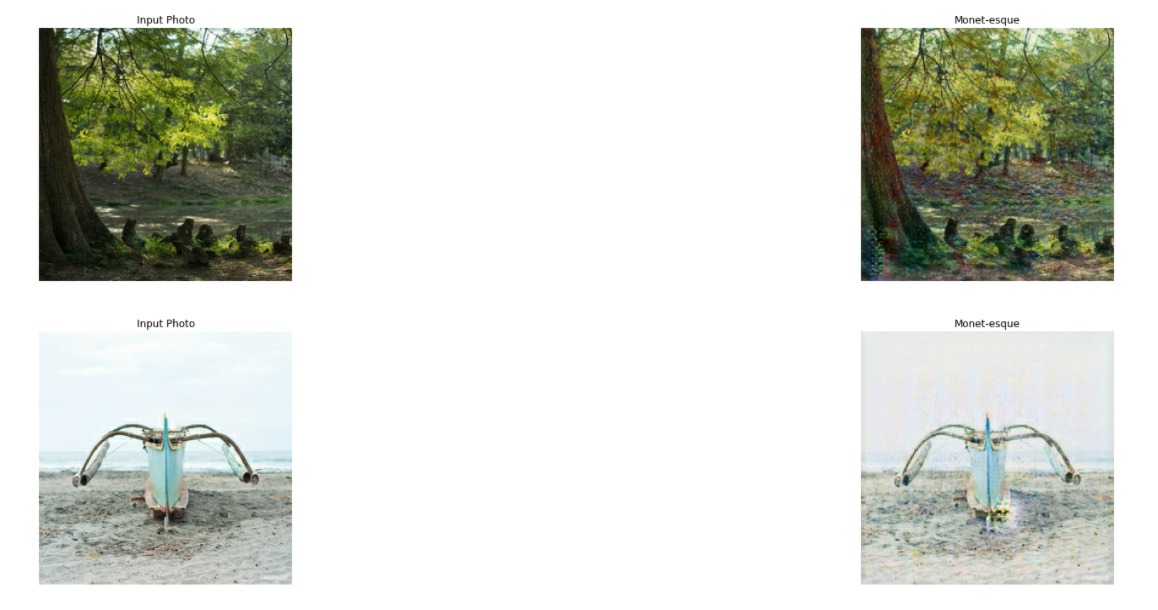
**CHAPTER 8:**

1. **Result**

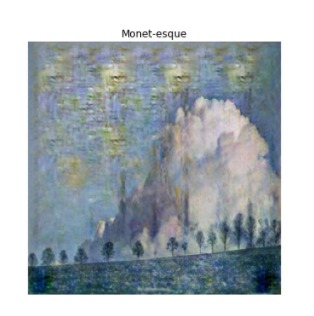
If a discriminator achieves 50% accuracy it means that it cannot tell the difference between the real image and a fake image. With our implementation we have achieved 63% accuracy in generating a fake Monet image . Thus, we have translated the respective input images into the artistic style of Monet incorporating a style transfer using CycleGAN. The following results are based on 200 epochs and the output image becomes sharper and more clear with subsequent epochs which signifies that our model is improving with time.



**Fig 1 Output 1**



**Fig 2 Output 2**



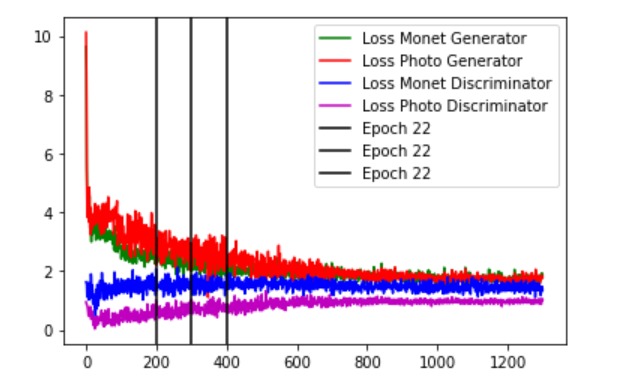






**Fig 3 output 3**

400 epochs



10/10 [==============================] - 3s 284ms/step - total\_monet\_gen\_loss: 1.7741 - total\_photo\_gen\_loss: 1.7287 - monet\_disc\_loss: 1.4030 - monet\_disc\_loss1: 0.7781 - monet\_disc\_loss2: 0.6249 - photo\_disc\_loss: 0.9613 Epoch 200/200

Loss Photo Generator has the highest delta because when the generator first starts generating images are obviously fake and need a lot of improvement. The loss function penalizes the generator for being wrong and updates the weights. This improves the image quality over time which is shown by subsequent epochs.

There is convergence after about 800 epochs showing that the discriminator can’t tell the difference between original and generated images.

**CHAPTER 9:**

1. **Conclusion**

The results of our implementation suggest that conditional adversarial networks are a promising approach for many image to-image translation tasks, especially those involving highly structured graphical outputs. These networks learn a loss adapted to the task and data at hand, which makes them applicable in a wide variety of settings. Generating a stylistically inspired painting built by training a dataset of given paintings is possible by translating a given input image through CycleGAN. CycleGAN architecture has the ability to create perceptions into something tangible. CycleGAN, as well as any GAN-based method, is fundamentally hallucinating part of the content it creates. Its outputs are predictions of "what might it look like if ..." and the predictions, though plausible, may largely differ from the ground truth. CycleGAN should only be used with great care and calibration in domains where critical decisions are to be taken based on its output

**References:**

[1] Neural information processing system Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, et al. Generative adversarial nets. In: Advances ins. 2014. p. 2672–80.

[2] Radford, Alec et al. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” *CoRR* abs/1511.06434 (2016): n. pag.

[3] Jun-Yan Zhu\*, Taesung Park\*, Phillip Isola, and Alexei A. Efros. "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks", in IEEE International Conference on Computer Vision (ICCV), 2017.

[4] Martin Heusel et al. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. 2018. arXiv: 1706.08500v6 [cs.LG]

[5]<https://www.christies.com/features/A-collaboration-between-two-artists-one-human-one-a-machine-9332-1.aspx>

[6] Learning image-to-image translation using paired and unpaired training samples

Soumya Tripathy, Juho Kannala, Esa Rahtu https://doi.org/10.48550/arXiv.1805.03189

[7] Y. Aytar, L. Castrejon, C. Vondrick, H. Pirsiavash, and A. Torralba. Cross-modal scene networks. PAMI, 2016. 3

[8] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks. In CVPR, 2017. 3

[9] R. W. Brislin. Back-translation for cross-cultural research. Journal of cross-cultural psychology, 1(3):185–216, 1970. 2, 3

[10] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In CVPR, 2016. 2, 5, 6, 18

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In CVPR, 2009. 8, 13, 18

[12] E. L. Denton, S. Chintala, R. Fergus, et al. Deep generative image models using a laplacian pyramid of adversarial networks. In NIPS, 2015. 2

**Annexture 1- Project Planning (Using Gantt chart)**