National College of Ireland

Configuration Manual

MSc Research Project
MSc Data Analytics

Yogaraj Kori
Student ID: x23241365

School of Computing
National College of Ireland

Supervisor: Abid Yaqoob

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……. Yogaraj Kori……

Student ID: ……X23241365…….……

Programme: …. MSc In Data Analytics…… **Year:** ……2024…….
…..

Module: ……Research Project……

Lecturer: ……Abid Yaqoob……
Submission Due Date: ……12/12/2024……

Project Title: …… Fictional Face Generation using Adversarial Models……

Word Count: ……1383…… **Page Count:** ……11……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……Yogaraj Kori……

Date: ……12/12/2024……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Yogaraj Kori
Student ID: x23241365

1 Overview

This is a research manual for my project “Fictional Face Generation using Adversarial Models”. This will serve as a log and a manual for the technical procedure of development and further execution of the project. It contains data on the tools required, setup of environment and code configurations for this project. This manual also contains certain code required to configure the project.

2 Environment and Requirements

2.1 Environment

This project is run on Google Collab as it requires GPU for proper and fast executions of the code. You can turn the GPU on in google collab by changing the runtime type to T4 GPU with High RAM.

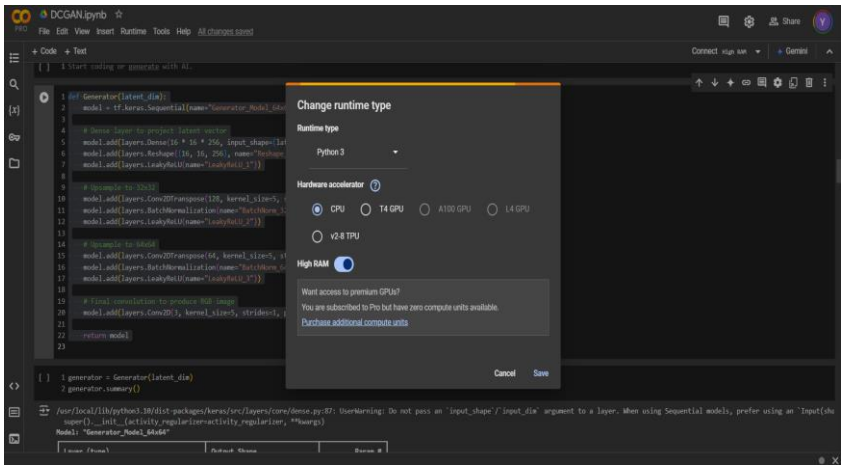

Fig 1: changing runtime type in Google Collab.

2.2 Hardware Requirements

The Hardware Requirements for this Research project requires GPU and CPU components following which the program can be executed.

- Operating System Requirements: Windows 10 or similar.
- Processor: Intel(R) Core (TM) i3 with 2.5GHz 1.8GHz
- Storage: 512 GB
- RAM: 8.0 GB

2.3 Software Requirements

This project was built and run on this software environment.

- Integrated Development Environment: Google Collab Python3
- Programming Language: Python 3
- Storage: Google drive.
- Other Tools: Notepad, SmartDraw, any Image Viewer

3 Setting up Environment on Google Collab

We have written our program in google collab, you can access google collab by simply going to the website and opening a new notebook, uploading and executing the code. However, the first piece of code you will need to run is to allow google collab to connect with google drive. The code to be used is shown below.

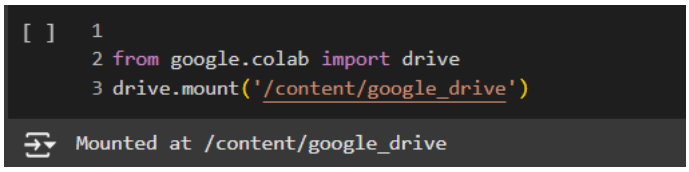

Fig 2: Connecting to google drive.

4 Data Collection.

We will be collecting the data from the data source, which is Kaggle using the Kaggle API, to get the API information you can go to the following website. https://www.kaggle.com/datasets/jessicali9530/celeba-dataset, the screenshot of the page is also shown below for your reference.

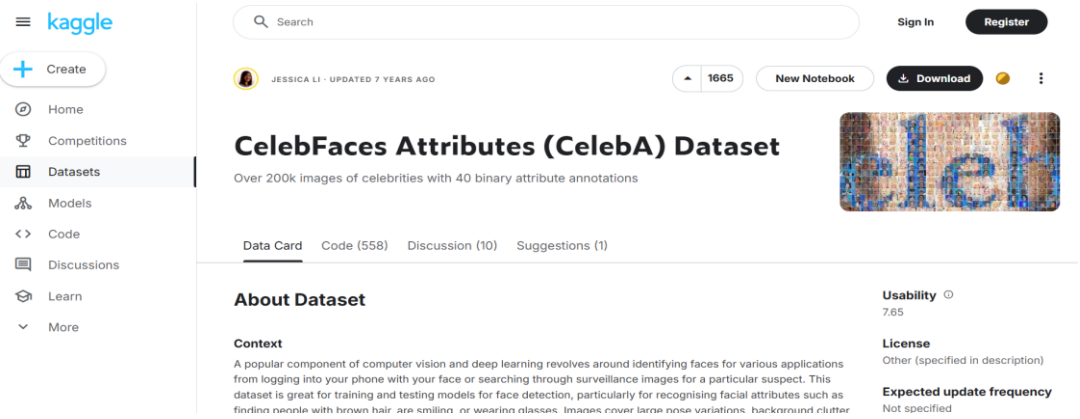

Fig 3: Kaggle data source.

4.1 Data Download:

We will be downloading the dataset using the API directly from the code, but we will first be requiring the path for the dataset. The code for this can be seen below.

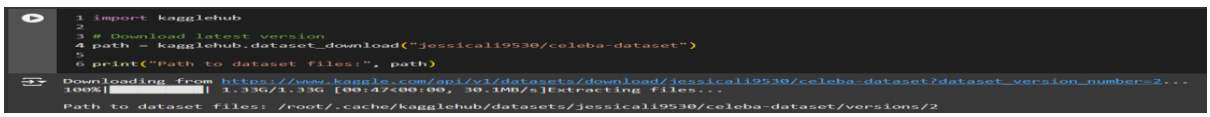

Fig 4: Kaggle data path collection.

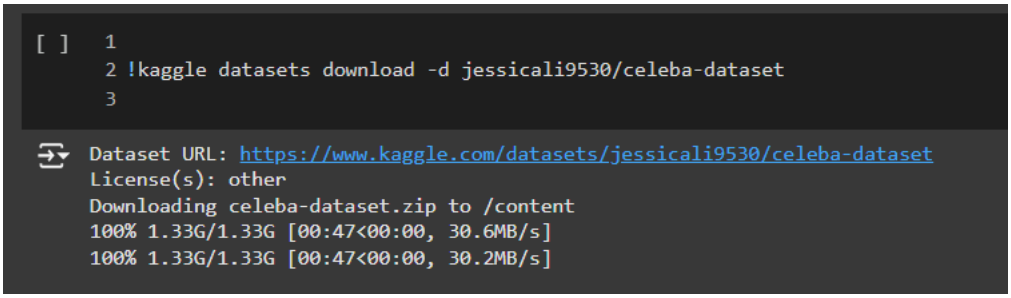

Fig 5: Kaggle data download

4.2 Data Extraction

The downloaded data will be in the form of a zip file which should then extracted into the google collab disk using the next code snippet.

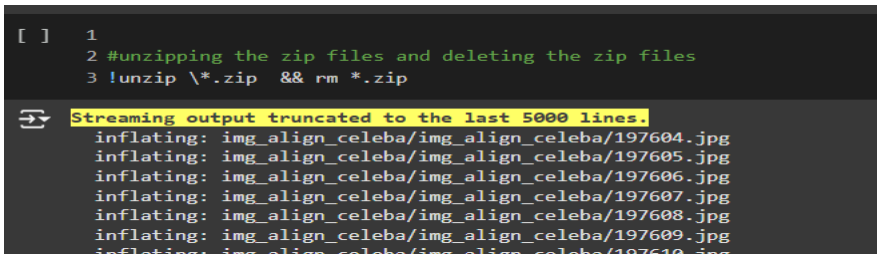

Fig 6: Kaggle data download

5 Preprocessing.

After Data collection, the next step is to preprocess the data but to first handle the data, we will need to download the required libraries for the project.

5.1 Required Libraries.

The following packages are required for the proper execution of the project.

- TensorFlow
- Keras
- Numpy
- Matplotlib
- CV2
- Tqdm
- keras.preprocessing.image
- skimage.metrics.structural_similarity
- scipy.linalg.sqrtm
- tensorflow.keras.applications.InceptionV3
- tensorflow.keras.mixed_precision
- imagio

```python
from google.colab import drive
drive.mount('/content/drive')

import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))
import keras
from keras import layers
import numpy as np
import matplotlib.pyplot as plt
import cv2
import os
from tqdm import tqdm
from keras.preprocessing.image import img_to_array
import time
import warnings

from tensorflow.keras import mixed_precision
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_global_policy(policy)
```

Fig 7: python Libraries Import

5.2 Flexible GPU application

Once the necessary libraries are called, we will need to make sure that the GPU RAM varies as the program might require more RAM for a code snippet and less RAM for another snippet. The below figure shows the code snippet to make the GPU flexible.

```python
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:

        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:

        print(e)
```

Fig 8: GPU configuration for memory Growth.

5.3 Preprocessing.

Now that we have flexible memory growth let us look at the preprocessing of the images. For preprocessing the images, we will need to convert all the images in the form of an 4-dimensional NumPy array and along with this save the path of each of the images. We will be using CV2 library to save the images in RGB format in the array the code for which can be seen below.

```python
def preprocess_img(img_path):
    img = cv2.imread(img_path)
    if img is None:
        raise ValueError(f"Image not found or unable to read: {img_path}")
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (SIZE, SIZE))
    img = img_to_array(img)
    img = img.astype('float32')
    img = (img - 127.5) / 127.5
    return img
```

Fig 9: Image preprocessing.

5.3 Saving the Image in a tf.records format.

Once we have pre-processed the images, we have saved the images in a tf.records format so that we do not have to load the images every time during model building, this also ensures the quick execution of the program and does not rely heavily on the GPU processing power.

```
def save_dataset(dataset, file_path):
    def serialize_example(image):
        feature = {
            'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[
                tf.io.encode_jpeg((image * 127.5 + 127.5).astype(np.uint8)).numpy()
            ]))
        }
        example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
        return example_proto.SerializeToString()

    with tf.io.TFRecordWriter(file_path) as writer:
        for image in dataset:
            serialized_image = serialize_example(image.numpy())
            writer.write(serialized_image)

# Preprocess and save the dataset
images = []
for img_name in tqdm(os.listdir(directory_path)[:num_imgs]):
    img_path = os.path.join(directory_path, img_name)
    img = preprocess_img(img_path)
    images.append(img)



100%|          | 3000/3000 [13:12<00:00,  3.79it/s]

images = np.array(images)
dataset = tf.data.Dataset.from_tensor_slices(np.array(images))

tfrecord_path = '/content/drive/MyDrive/preprocessed_dataset.tfrecord'
save_dataset(dataset, tfrecord_path)
print(f"Dataset saved to {tfrecord_path}")
```

Fig 10: Saving the images in tf.record format for faster model building.

5.4 Dataset Visualization.

Now that we have the preprocessed the data, let us visualize the data to understand what kind of images are stored in the dataset.

```
plt.figure(figsize = (6,6))

for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow((images[i]+1)/2)
    plt.axis('off')

plt.suptitle('Some Sample Images')
plt.show()
```

Fig 11:code for visualization of images.

6 Model Implementation.

For our project we have implemented 2 different GAN models which are implemented seperately in 2 different .ipynb files. First Let us look at the implmentation of the DCGAN.

6.1 DCGAN

The below code snippet shows the implementation of generator and discriminator for the DCGAN implementation.

```python
def Generator(latent_dim):
    model = tf.keras.Sequential(name="Generator_Model_64x64")

    # Dense layer to project latent vector
    model.add(layers.Dense(16 * 16 * 256, input_shape=(latent_dim,), name="Dense_Project"))
    model.add(layers.Reshape((16, 16, 256), name="Reshape_to_16x16x256"))
    model.add(layers.LeakyReLU(name="LeakyReLU_1"))

    # Upsample to 32x32
    model.add(layers.Conv2DTranspose(128, kernel_size=5, strides=2, padding='same', name="Upsample_to_32x32"))
    model.add(layers.BatchNormalization(name="BatchNorm_32x32"))
    model.add(layers.LeakyReLU(name="LeakyReLU_2"))

    # Upsample to 64x64
    model.add(layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding='same', name="Upsample_to_64x64"))
    model.add(layers.BatchNormalization(name="BatchNorm_64x64"))
    model.add(layers.LeakyReLU(name="LeakyReLU_3"))

    # Final convolution to produce RGB image
    model.add(layers.Conv2D(3, kernel_size=5, strides=1, padding='same', activation='tanh', name="Final_RGB_Output"))

    return model

```

Fig 12:code for generator.

Now let us look at the implementation of Discriminator for the DCGAN implementation.

```python
def Discriminator(SIZE):
    model = tf.keras.Sequential()
    model.add(layers.InputLayer(input_shape=(SIZE, SIZE, 3)))

    model.add(layers.Conv2D(64, 3, strides=2, padding='same'))
    model.add(tf.keras.layers.LeakyReLU())

    model.add(layers.Conv2D(128, 3, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2D(256, 3, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2D(512, 3, strides=2, padding='same'))  # Additional layer for downsampling
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Flatten())
    model.add(layers.Dense(1))  # No activation for `from_logits=True`

    return model

```

Fig 13: code for discriminator.

Now the most important part for the success of a GAN implementation is the optimizers for both the generator and discriminator which influences the learning rate of the model and the beta_1 and beta_2 momentum terms. The below figure shows the code snippet.

```python
#lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
#    initial_learning_rate=0.0003, decay_steps=1000, decay_rate=0.96, staircase=True)
optimizer_gen = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5,beta_2=0.5)
optimizer_dis = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5,beta_2=0.5)


cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits = True)
```

Fig 14: Optimizers for DCGAN.

Along with the optimizers, the next crucial part of the implementation is the calculation of the generator and the discriminator loss. The below code snippet shows the implementation of the functions for each loss.

```python
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output),fake_output)

def discriminator_loss(fake_output, real_output):
    fake_loss = cross_entropy(tf.zeros_like(fake_output),fake_output)
    real_loss = cross_entropy(tf.ones_like(real_output),real_output)
    return fake_loss + real_loss
```

Fig 15: Generator and Discriminator losses for DCGAN.

Now once we have implemented the loss functions as well, we will also need to train the model as such we have implemented the train steps function which will save the model execution along with implementing and updating the above mentioned losses.

```python
def train_steps(images,epoch):
    noise = tf.random.normal([batch_size, latent_dim])

    with tf.GradientTape() as disc_tape:
        generated_images = generator(noise)
        fake_output = discriminator(generated_images)
        real_output = discriminator(images)
        dis_loss = discriminator_loss(fake_output, real_output)

    gradient_of_discriminator = disc_tape.gradient(dis_loss, discriminator.trainable_variables)
    optimizer_dis.apply_gradients(zip(gradient_of_discriminator, discriminator.trainable_variables))

    noise = tf.random.normal([batch_size, latent_dim])
    with tf.GradientTape() as gen_tape:
        generated_images = generator(noise)
        fake_output = discriminator(generated_images)
        gen_loss = generator_loss(fake_output)

    checkpoint_path = "./checkpoints"
    if not os.path.exists(checkpoint_path):
      os.makedirs(checkpoint_path)
    generator.save(os.path.join(checkpoint_path, f"generator_epoch_{epoch + 1}.keras"))
    discriminator.save(os.path.join(checkpoint_path, f"discriminator_epoch_{epoch + 1}.keras"))

    gradient_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    optimizer_gen.apply_gradients(zip(gradient_of_generator, generator.trainable_variables))

    return {'gen_loss': gen_loss, 'disc_loss': dis_loss}
```

Fig 16: Train_steps Implementation.

The next step in the model implementation is to call all other functions and finally execute the model.

```python
import pandas as pd

def train_gan(epochs, dataset, output_dir, latent_dim=100, save_every=5, calculate_is_every=10, excel_file='losses.xlsx'):

    # Ensure output directory exists
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Create a list to store losses for each epoch
    losses_data = {'Epoch': [], 'Generator Loss': [], 'Discriminator Loss': []}

    for epoch in range(epochs):
        start = time.time()
        print("\nEpoch : {}".format(epoch + 1))

        # Iterate over the dataset and train the model
        for images in dataset:
            loss = train_steps(images,epoch)

        # Save the losses to the list
        losses_data['Epoch'].append(epoch + 1)
        losses_data['Generator Loss'].append(loss['gen_loss'].numpy())
        losses_data['Discriminator Loss'].append(loss['disc_loss'].numpy())

        if (epoch + 1) % save_every == 0:

          noise = np.random.normal(-1, 1, (1, latent_dim))  # Generate random noise for 1 image
          generated_image = generator(noise, training=False).numpy()  # Generate the image and convert to numpy
          generated_image = (generated_image[0] + 1) / 2  # Rescale from [-1, 1] to [0, 1]

          # Set up the plot for saving the image
          filename = os.path.join(output_dir, f"epoch_{epoch + 1}.png")
          plt.figure(figsize=(6, 6))
          plt.imshow(generated_image)
          plt.axis('off')  # Hide axes for a cleaner look
```

Fig 17: Gan training function

Once we have implemented the GAN we can try to calculate the Inception score for the model. The below code snippet calculates the Inception score of the model.

```python
def calculate_inception_score(images, num_classes=1000, splits=10):
    model = InceptionV3(include_top=True, weights='imagenet')

    images = preprocess_images(images)

    preds = model.predict(images)

    split_scores = []
    n = preds.shape[0]
    split_size = n // splits

    for i in range(splits):
        part = preds[i * split_size: (i + 1) * split_size]


        marginal = np.mean(part, axis=0)

        scores = []
        for j in range(part.shape[0]):
            p_yx = part[j]
            kl_div = entropy(p_yx, marginal)
            scores.append(kl_div)


        split_scores.append(np.exp(np.mean(scores)))

    return np.mean(split_scores)

def preprocess_images(images):

    images_resized = tf.image.resize(images, (299, 299))

    return preprocess_input(images_resized.numpy())
```

Fig 18: Incetion Score calculation function

6.2 WGAN_GP

Most of the model implementation is similar to the model implementation of DCGAN, however we have implemented the model without utilizing the tf.records file and rather preprocess the images directly in the program, this allows us to test the model performances under various conditions and training capacities. Now first let us look at the generator development in the below figure.

```
def build_generator(noise_dim, output_channels=3, activation="tanh", alpha=0.2):
    inputs = layers.Input(shape=(noise_dim,), name="input")


    x = layers.Dense(4*4*512, use_bias=False)(inputs)

    x = layers.Reshape((4, 4, 512))(x)

    x = layers.Conv2DTranspose(512, (5, 5), strides=(2, 2), padding="same", use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Conv2DTranspose(256, (5, 5), strides=(2, 2), padding="same", use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Conv2DTranspose(128, (5, 5), strides=(2, 2), padding="same", use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding="same", use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha)(x)
    x = layers.Dropout(0.5)(x)

    x = layers.Conv2D(output_channels, (5, 5), strides=(1, 1), padding="same", activation=activation, use_bias=False, dtype='float32')(x)
    assert x.shape == (None, 64, 64, output_channels)

    model = tf.keras.Model(inputs=inputs, outputs=x)

    return model
```

Fig 19: Generator for WGAN_GP

Now let us look at the discriminator development in the following figure.

```
def build_discriminator(img_shape, activation='linear', alpha=0.2):
    inputs = layers.Input(shape=img_shape, name="input")

    x = layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', use_bias=False)(inputs)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same', use_bias=False)(x)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same', use_bias=False)(x)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Conv2D(512, (5, 5), strides=(2, 2), padding='same', use_bias=False)(x)
    x = layers.LeakyReLU(alpha)(x)

    x = layers.Flatten()(x)
    x = layers.Dropout(0.5)(x)

    x = layers.Dense(1, activation=activation, dtype='float32')(x)

    model = tf.keras.Model(inputs=inputs, outputs=x)
```

Fig 20: discriminator for WGAN_GP

Now we will be implementing the WGAN_GP as a Object oriented programming code by utilizing the classes and inheritance to implement the code. The first class of the model is the code to implement the GAN itself, which building the model which calls the generator and the discriminator, compiles the model, add instance noises and finally calculate the gradient penalty.

```python
class WGAN_GP(tf.keras.models.Model):
    def __init__(self, discriminator, generator, noise_dim, discriminator_extra_steps=5, gp_weight=10.0):
        super(WGAN_GP, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.noise_dim = noise_dim
        self.discriminator_extra_steps = discriminator_extra_steps
        self.gp_weight = gp_weight

    def compile(self, discriminator_opt, generator_opt, discriminator_loss, generator_loss, **kwargs):
        super(WGAN_GP, self).compile(**kwargs)
        self.discriminator_opt = discriminator_opt
        self.generator_opt = generator_opt
        self.discriminator_loss = discriminator_loss
        self.generator_loss = generator_loss

    def add_instance_noise(self, x, stddev=0.1):
        noise = tf.random.normal(tf.shape(x), mean=0.0, stddev=stddev, dtype=x.dtype)
        return x + noise

    def gradient_penalty(self, real_samples, fake_samples, discriminator):
        batch_size = tf.shape(real_samples)[0]
        epsilon = tf.random.uniform([batch_size, 1, 1, 1], minval=0, maxval=1)
        interpolated_samples = epsilon * real_samples + (1 - epsilon) * fake_samples

        with tf.GradientTape() as tape:
            tape.watch(interpolated_samples)
            logits = discriminator(interpolated_samples, training=True)

        gradients = tape.gradient(logits, interpolated_samples)
        gradients_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1, 2, 3]))
        gradient_penalty = tf.reduce_mean((gradients_norm - 1.0) ** 2)
        return gradient_penalty
```

Fig 21: train class 1 for WGAN_GP

```python
    def train_step(self, real_samples):
        batch_size = tf.shape(real_samples)[0]
        noise = tf.random.normal([batch_size, self.noise_dim])
        gps = []

        for _ in range(self.discriminator_extra_steps):
            with tf.GradientTape() as tape:
                fake_samples = self.generator(noise, training=True)
                pred_real = self.discriminator(real_samples, training=True)
                pred_fake = self.discriminator(fake_samples, training=True)

                real_samples = self.add_instance_noise(real_samples)
                fake_samples = self.add_instance_noise(fake_samples)

                gp = self.gradient_penalty(real_samples, fake_samples, self.discriminator)
                gps.append(gp)

                disc_loss = self.discriminator_loss(pred_real, pred_fake) + gp * self.gp_weight

            grads = tape.gradient(disc_loss, self.discriminator.trainable_variables)
            self.discriminator_opt.apply_gradients(zip(grads, self.discriminator.trainable_variables))

        with tf.GradientTape() as tape:
            fake_samples = self.generator(noise, training=True)
            pred_fake = self.discriminator(fake_samples, training=True)
            gen_loss = self.generator_loss(pred_fake)

        grads = tape.gradient(gen_loss, self.generator.trainable_variables)
        self.generator_opt.apply_gradients(zip(grads, self.generator.trainable_variables))

        self.compiled_metrics.update_state(real_samples, fake_samples)

        results = {m.name: m.result() for m in self.metrics}
        results.update({"d_loss": disc_loss, "g_loss": gen_loss, "gp": tf.reduce_mean(gps)})
        return results
```

Fig 22: train class 2 for WGAN_GP

Now the next class for the model implementation is the LRScheduler model which creates the model results for each epoch and if the generator or the discriminator loss is high, we will be applying gradient loss on the model output.

```python
class LRSheduler(tf.keras.callbacks.Callback):
    """Learning rate scheduler for WGAN-GP"""
    def __init__(self, decay_epochs: int, tb_callback=None, min_lr: float=0.00001):
        super(LRSheduler, self).__init__()
        self.decay_epochs = decay_epochs
        self.min_lr = min_lr
        self.tb_callback = tb_callback
        self.compiled = False

    def on_epoch_end(self, epoch, logs=None):
        if not self.compiled:
            self.generator_lr = self.model.generator_opt.learning_rate.numpy()
            self.discriminator_lr = self.model.discriminator_opt.learning_rate.numpy()
            self.compiled = True

        if epoch < self.decay_epochs:
            new_g_lr = max(self.generator_lr * (1 - (epoch / self.decay_epochs)), self.min_lr)
            self.model.generator_opt.learning_rate.assign(new_g_lr)
            new_d_lr = max(self.discriminator_lr * (1 - (epoch / self.decay_epochs)), self.min_lr)
            self.model.discriminator_opt.learning_rate.assign(new_d_lr)
            print(f"Learning rate generator: {new_g_lr}, discriminator: {new_d_lr}")


            if self.tb_callback is not None:
                writer = self.tb_callback._writers.get('train')
                with writer.as_default():
                    tf.summary.scalar('generator_lr', data=new_g_lr, step=epoch)
                    tf.summary.scalar('discriminator_lr', data=new_d_lr, step=epoch)
                    writer.flush()
```

Fig 23: LRScheduler for WGAN_GP

At the end , we will be creating instances of all the classes and finally call the train class with the instances of all other classes.

```
callback = ResultsCallback(noise_dim=noise_dim, output_path=model_path)
tb_callback = TensorBoard(log_dir=model_path + '/logs')
lr_scheduler = LRSheduler(decay_epochs=100, tb_callback=tb_callback)
results_callback = ResultsCallbackWithLossPlot(
    noise_dim=noise_dim,
    output_path=f'/content/google_drive/MyDrive/Colab Notebooks/',
    examples_to_generate=16,
    grid_size=(4, 4),
    spacing=5,
    gif_size=(416, 416),
    duration=0.1,
    save_model=True
)


fid_callback = FIDCallback(
    real_images=real_image_sample,
    noise_dim=noise_dim,
    output_path=f'/content/google_drive/MyDrive/Colab Notebooks/'
)


ssim_callback = SSIMCallback(
    real_images=real_image_sample,
    noise_dim=noise_dim,
    output_path=f'/content/google_drive/MyDrive/Colab Notebooks/'
)


inception_score_callback = InceptionScoreCallback(
    real_images=real_image_sample,
    noise_dim=noise_dim,
    output_path=f'/content/google_drive/MyDrive/Colab Notebooks/',
    n_split=10
)

gan = WGAN_GP(discriminator, generator, noise_dim, discriminator_extra_steps=5)
gan.compile(discriminator_optimizer, generator_optimizer, discriminator_w_loss, generator_w_loss)
gan.fit(train_generator, epochs=10, callbacks=[callback, tb_callback, lr_scheduler, results_callback, fid_callback, ssim_callback, inception_score_callback])

```

Fig 24: train class call with all instances for WGAN_GP

7 Model Implementation.

Now that we have implemented all the models we will need to calculate Inception Score, Fretchet Inception score and Structural similarity Index, which we will be calculating using the following classes.

```python
from scipy.linalg import sqrtm
from tensorflow.keras.applications import InceptionV3
import numpy as np
import tensorflow as tf

def calculate_fid(real_images, generated_images):

    inception_model = InceptionV3(include_top=False, pooling='avg', input_shape=(299, 299, 3))

    def preprocess_images(images):

        images_resized = tf.image.resize(images, (299, 299))
        return tf.keras.applications.inception_v3.preprocess_input(images_resized)


    real_images = preprocess_images(real_images)
    generated_images = preprocess_images(generated_images)


    real_features = inception_model.predict(real_images, batch_size=32)
    fake_features = inception_model.predict(generated_images, batch_size=32)


    mu_real = np.mean(real_features, axis=0)
    sigma_real = np.cov(real_features, rowvar=False)

    mu_fake = np.mean(fake_features, axis=0)
    sigma_fake = np.cov(fake_features, rowvar=False)
    diff = mu_real - mu_fake
    covmean = sqrtm(sigma_real @ sigma_fake + np.eye(sigma_real.shape[0]) * 1e-6)

    if np.iscomplexobj(covmean):
        covmean = covmean.real

    fid = np.sum(diff**2) + np.trace(sigma_real + sigma_fake - 2 * covmean)
    return fid

```

Fig 25: FID score calculation for WGAN_GP

Similarly we can implement SSIM and IS similar to the way we implemented for DCGAN.

8 Testing.

We have trained the model under various conditions including increasing and reducing the training size, different implementations of the models and for different resolutions of images and different batch sizes, from our various testing we believe this is the best model implementation and provides the most optimal performance for the model.

References

Jagad Nabil Tuah Imanda, Bachtiar, F., & Achmad Ridok. (2023). Application of Deep Convolutional Generative Adversarial Networks to Generate Pose Invariant Facial Image Synthesis Data. Jurnal RESTI (Rekayasa Sistem Dan Teknologi Informasi), 7(5), 1049 - 1055. https://doi.org/10.29207/resti.v7i5.5112

Li S, Dutta V, He X, Matsumaru T. Deep Learning Based One-Class Detection System for Fake Faces Generated by GAN Network. Sensors. 2022; 22(20):7767. https://doi.org/10.3390/s22207767.

Reddy, Shirisha. (2024). Unveiling Spoofing Attempts: A DCGAN-based Approach to Enhance Face Spoof Detection in Biometric Authentication.