ECS7022P: Computational Creativity Project

Project Title: Anime Face Generator GAN (AFGGAN)
Student Name: Nagarjuna Suryaji Mote
Student ID Number:210431715

HTML Link to Colab Notebook [required]:
https://colab.research.google.com/drive/1vIDj2FRikEWb-gw2zRZSQlN9pxKbvSob?usp=share_link
HTML Link to System Outputs [optional]:
https://drive.google.com/drive/folders/1EwwVhaIrrmUvcjtG5kOqqUTxeH1c4VlZ?usp=sharing.

Project Overview [10%]

This generative AI system creates Anime human faces using a deep Convolutional Generative Adversarial Network model. The system takes a random noise vector as an input and generates a coloured image of an anime drawing of a face of a human character. The user can select the image size/resolution of the image by a slider. This is applied to the image generated by the generator.

The system works by training a GAN on a database of Anime faces. The generator network takes a random noise vector as input and generates a face image, while the discriminator network tries to distinguish between real and generated images. The generator and discriminator are trained in a way that the generator tries to generate realistic faces to fool the discriminator, while the discriminator tries to classify whether an image is real or generated correctly. This competition between the two networks results in the generator learning to generate increasingly realistic images.

This project was motivated by the increasing interest in AI-generated art and websites like “This person does not exist”. This system is particularly interesting because of its potential in used in video games and character designs. This project also aimed to explore and understand the working and capabilities of GANs and demonstrate how they are used.

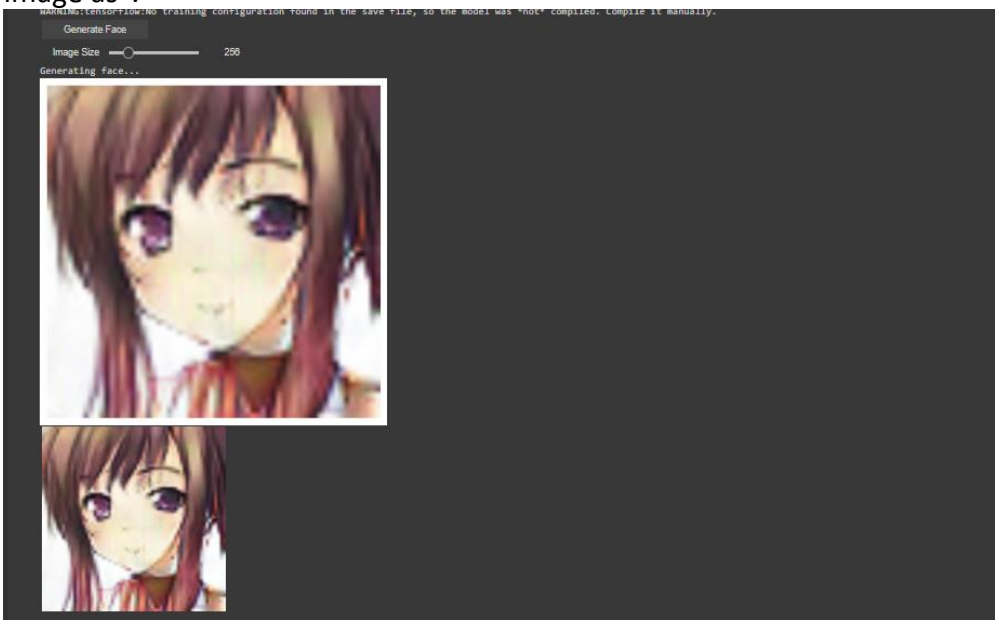
Generative Models [10%]

The generative neural model used in this project is a (deep convolutional) Generative Adversarial Network. This model has a generator which has five layers The first layer is a dense layer with 4*4*1024 nodes and ReLU activation function, followed by batch normalisation Transposed convolutional layers with 512, 256, and 128 filters make up the following three layers. Each layer utilises a ReLU activation function with batch normalisation and has a kernel size of 5 or 3 with a stride of 2. The produced image is then created by applying a transposed convolutional layer with three filters and a kernel size of three as the final layer. Discriminator takes in an image of size (64, 64,3) as input. It has 4 convolutional layers of 64,128,256,512 filters and 3,3,5,5 kernel sizes, respectively and LeakyRelU activation functions. The fifth and last layer is conv. Layer with 1 filter and kernel size of 4 followed by a flatten layer and sigmoid activation function.

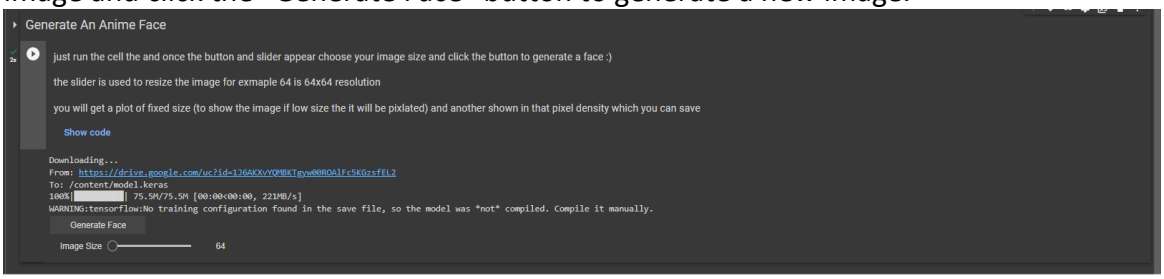
The Generator produces output from random noise, which is the input and the discriminator guesses if the image is real or fake. The generator tries to fool the discriminator, both networks improve. The loss function for the Generator is cross-entropy loss between real images and tensors of real images, whereas discriminator loss is the binary cross-entropy loss between the real output and a tensor of ones + the binary cross-entropy loss between the fake output and a tensor of zeros.

This model was trained on anime face pictures of the database "Anime Face Dataset" by Mckinsey666 on Kaggle.
The dataset is biased towards female characters that reflected in the outputs of the network. The model was trained around 225 epochs in batches of 125 the outputs were getting good around 90 epochs. The model struggles with noses and mouths

Process [15%]
The system loads in the trained model, the system takes a single input that is the desired size of the image chosen by the user using a slider. The system generates a random noise vector of size 128 and passes it to the loaded generator to generate an image. The generated image is then resized to the desired size using TensorFlow's image resizing function.
The output is displayed to the user in two ways. First is a plot of fixed size displayed using a metaplot displayed in the notebook. and another is an image in the pixel size chosen by the user. the user can save the image by right-clicking on the plot and selecting "Save image as".



The user can only interact with the system by means of a slider to choose the desired image and click the “Generate Face” button to generate a new image.



The system's data flow is comparatively straightforward. The system receives the user's input regarding the preferred image size. A random noise vector is created by the system, which then runs through the generator network and resizes the output to the required dimension. The user is then shown the output, and the user has the option of saving the image.

Example Outputs [10%]

Here are some example outputs more outputs can be found at the linked G_drive folder: https://drive.google.com/drive/folders/1EwwVhaIrrmUvcjtG5kOqqUTxeH1c4VlZ?usp=sharing.

Evaluation [20%]

Evaluating the model, the model is not well trained. The model struggles with noses and mouths and with eyes generated; outputs have eyes of different colours, and rarely the generated image would be mangled or have no eyes. The model also struggles with the symmetry of the faces. But on a positive note, this system can also generate images with different sizes, and the model has a high success rate of producing a nice anime character face image. But the generated images are simple and not that high resolution.
The model is relatively simple nothing fancy and the loss functions are relatively simple too.
From a deep learning perspective, the generator model has a higher desired loss.

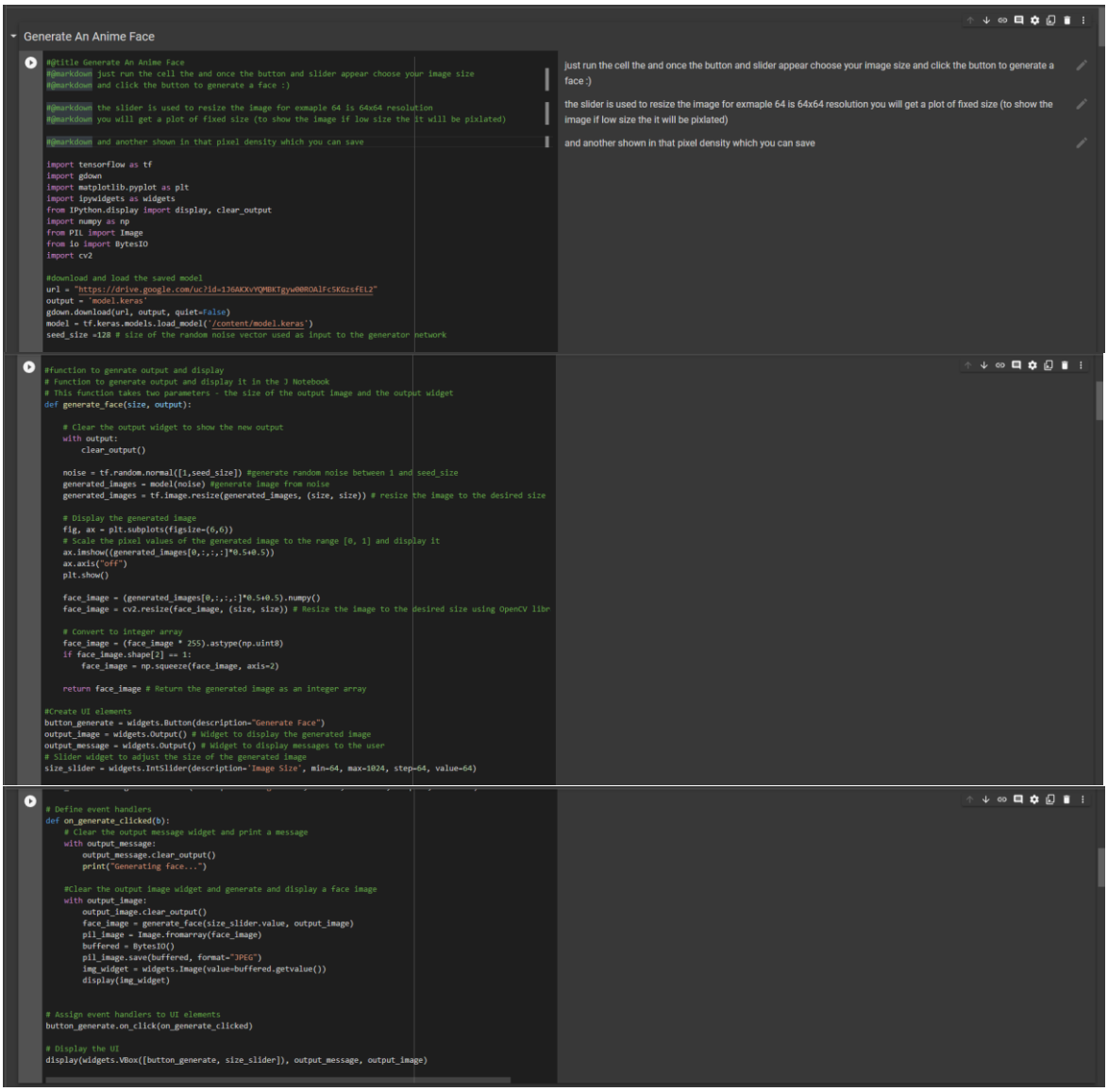
In terms of user interaction, the notebook provides a user interface where users can interact with the model by selecting the size of the generated image and clicking a button to generate a new anime face. The generated image is displayed in a widget, and users can save the image by right-clicking on it and selecting "Save Image As." There is no other way user can affect the system's output which is less than desirable in this age. But also, this can be taken in a positive way saying the system is more autonomous. And the learning curve to use this system is easy for a new user.

Value Added [20%]

Training one or more neural models rather than using pre-trained ones.

In this project, I have compiled and trained GAN Model on an Anime Faces dataset by Mckinsey666 on Kaggle. This model was trained for more than 200 epochs.
The generator part of the model is saved on my Gdrive in sharable access from where I have imported and loaded that model into my notebook by means of an URL and used it to produce the outputs. This is done in the first section of the notebook, and the code used for training is also in the notebook in the second section of the notebook.
There is also a UI for the user to adjust the parameter of the generated image.

Colab Notebook Code [15%]



Generate An Anime Face

```
#@title Generate An Anime Face
#@markdown just run the cell the and once the button and slider appear choose your image size
#@markdown and click the button to generate a face :)

#@markdown the slider is used to resize the image for exmaple 64 is 64x64 resolution
#@markdown you will get a plot of fixed size (to show the image if low size the it will be pixlated)

#@markdown and another shown in that pixel density which you can save

import tensorflow as tf
import gdown
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display, clear_output
import numpy as np
from PIL import Image
from io import BytesIO
import cv2

#download and load the saved model
url = "https://drive.google.com/uc?id=13GAKXvYQMBKTgyw0ORQAlFc5KGzsfEL2"
output = 'model.keras'
gdown.download(url, output, quiet=False)
model = tf.keras.models.load_model('/content/model.keras')
seed_size =128 # size of the random noise vector used as input to the generator network
```

just run the cell the and once the button and slider appear choose your image size and click the button to generate a face :)

the slider is used to resize the image for exmaple 64 is 64x64 resolution you will get a plot of fixed size (to show the image if low size the it will be pixlated)

and another shown in that pixel density which you can save

```
#function to genrate output and display
# Function to generate output and display it in the J Notebook
# This function takes two parameters - the size of the output image and the output widget
def generate_face(size, output):

    # Clear the output widget to show the new output
    with output:
        clear_output()

    noise = tf.random.normal([1,seed_size]) #generate random noise between 1 and seed_size
    generated_images = model(noise) #generate image from noise
    generated_images = tf.image.resize(generated_images, (size, size)) # resize the image to the desired size

    # Display the generated image
    fig, ax = plt.subplots(figsize=(6,6))
    # Scale the pixel values of the generated image to the range [0, 1] and display it
    ax.imshow((generated_images[0,:,:,:]*0.5+0.5))
    ax.axis("off")
    plt.show()

    face_image = (generated_images[0,:,:,:]*0.5+0.5).numpy()
    face_image = cv2.resize(face_image, (size, size)) # Resize the image to the desired size using OpenCV libr

    # Convert to integer array
    face_image = (face_image * 255).astype(np.uint8)
    if face_image.shape[2] == 1:
        face_image = np.squeeze(face_image, axis=2)

    return face_image # Return the generated image as an integer array

#Create UI elements
button_generate = widgets.Button(description="Generate Face")
output_image = widgets.Output() # Widget to display the generated image
output_message = widgets.Output() # Widget to display messages to the user
# Slider widget to adjust the size of the generated image
size_slider = widgets.IntSlider(description='Image Size', min=64, max=1024, step=64, value=64)
```

```
# Define event handlers
def on_generate_clicked(b):
    # Clear the output message widget and print a message
    with output_message:
        output_message.clear_output()
        print("Generating face...")

    #Clear the output image widget and generate and display a face image
    with output_image:
        output_image.clear_output()
        face_image = generate_face(size_slider.value, output_image)
        pil_image = Image.fromarray(face_image)
        buffered = BytesIO()
        pil_image.save(buffered, format="JPEG")
        img_widget = widgets.Image(value=buffered.getvalue())
        display(img_widget)


# Assign event handlers to UI elements
button_generate.on_click(on_generate_clicked)

# Display the UI
display(widgets.VBox([button_generate, size_slider]), output_message, output_image)
```

Bellow are the cells with the code used for tranning the model

GPU INFO

```
[ ] #@title GPU INFO
    gpu_info = !nvidia-smi
    gpu_info = '\n'.join(gpu_info)
    if gpu_info.find('failed') >= 0:
      print('Not connected to a GPU')
    else:
      print(gpu_info)
```

Imports

```
[ ] #@title Imports

    from google.colab import output
    import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras.layers import Input, Reshape, Dropout, Dense
    from tensorflow.keras.layers import Flatten, BatchNormalization
    from tensorflow.keras.layers import Activation
    from tensorflow.keras.layers import LeakyReLU , ReLU, PReLU
    from tensorflow.keras.layers import Conv2D ,Conv2DTranspose
    from tensorflow.keras.models import Sequential, Model, load_model
    from tensorflow.keras.optimizers import Adam
    import numpy as np
    from PIL import Image
    import os
    import matplotlib.pyplot as plt
    import cv2
    import urllib.request
```

Data (Kaggle)

```
[ ] #@title  Data (Kaggle)
    #@markdown "Need a kaggle.jason i.e. your API token uploded in the colab notebook instance to download the dat
    !pip install kaggle
    !mkdir ~/.kaggle
    !cp kaggle.json ~/.kaggle/
    ! chmod 600 ~/.kaggle/kaggle.json
    !kaggle datasets download soumikrakshit/anime-faces
    !unzip anime-faces.zip
    output.clear()
```

"Need a kaggle.jason i.e. your API token uploded in the colab notebook instance to download the data from kaggle"

Mount Drive

```
[ ] #@title Mount Drive
    from google.colab import drive
    drive.mount('/content/drive')
```

Reading and Prepossing / Normalizing data

```
[ ] #@title Reading and Preprossing / Normalizing data
    data_path = "/content/data/data" #@param   default_path="/content/data/data"
    batch_size =  125 #@param {type:"number"}
    img_size = 64 #@param {type:"number"}
    colour_mode = 'rgb'

    train_set = tf.keras.preprocessing.image_dataset_from_directory(
        data_path,
        label_mode=None,
        color_mode=colour_mode,
        batch_size=batch_size,
        image_size=(img_size, img_size),
        shuffle=True
    )
    #to normalizes each pixel value in the images to be between -1 and 1 for traning
    train_set = train_set.map(lambda x: ((x/127.5)-1))
```

data_path: "/content/data/data"

batch_size: 125

img_size: 64

Show_images training set

```
[ ] #@title Show_images   training set
    ncols = 10 #@param {type:"slider", min:1, max:16, step:1}
    nrows = 10 #@param {type:"slider", min:1, max:16, step:1}
    def plot_images(ds, nrows=nrows, ncols=ncols):
        plt.figure(figsize=(10, 10))
        ds = ds.unbatch().take(nrows*ncols)
        index = 1
        for image in ds:
            plt.subplot(nrows, ncols, index)
            plt.imshow(image)
            plt.axis('off')
            index += 1
        plt.subplots_adjust(wspace=0.1, hspace=0.1)
        plt.show()

    plot_images(train_set)
```

ncols: 10

nrows: 10

strides

```
[ ] #@title strides
    n_stride = 2 #stride value for convolution layer
    #n_layers_stride = 2

    init = tf.keras.initializers.RandomNormal(stddev=0.02)  # initializer for the weights of the layers. with standard deviation of 0.02
    cross_entropy = tf.keras.losses.BinaryCrossentropy()
```

Generator

```
[ ] #@title Generator

    #function to build the generator
    def build_generator(seed_size):
        model = Sequential()
        #first layer dense
        model.add(Dense(4*4*1024,kernel_initializer=init,input_dim=seed_size))
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(ReLU())
        model.add(Reshape((4,4,1024)))
        #OP_shape = 4,4,1024

        #3 convolutional layers,
        model.add(Conv2DTranspose(512,kernel_size=5,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(ReLU())

        model.add(Conv2DTranspose(256,kernel_size=5,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(ReLU())

        model.add(Conv2DTranspose(128,kernel_size=3,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(ReLU())

        #final convolutional layer with tanh
        model.add(Conv2DTranspose(3,kernel_size=3,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(Activation('tanh'))
        #OP_shape =  64,64,3

        return model
```

Discriminator

```
[ ] #@title Discriminator

    #function to build the discriminator
    def build_discriminator(image_length,image_channels):

        model = Sequential()
        #first Conv layer input size = 64,64,3
        #input size = output size
        model.add(Conv2D(64,kernel_size=3,strides=n_stride,padding='same',use_bias=False,input_shape=(image_length,image_length,image_channels),kernel_initializer=init)) #input size = output size
        model.add(LeakyReLU(alpha=0.2)) #LeakyReLU with a negative slope of 0.2, which helps to prevent the vanishing gradient problem during training.

        # second Conv layer
        model.add(Conv2D(128,kernel_size=3,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(LeakyReLU(alpha=0.2)) #LeakyReLU with a negative slope of 0.2, which helps to prevent the vanishing gradient problem during training.

         # third Conv layer
        model.add(Conv2D(256,kernel_size=5,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(LeakyReLU(alpha=0.2)) #LeakyReLU with a negative slope of 0.2, which helps to prevent the vanishing gradient problem during training.

         # fourth Conv layer
        model.add(Conv2D(512,kernel_size=5,strides=n_stride,padding='same',use_bias=False,kernel_initializer=init)) #input size = output size
        model.add(BatchNormalization()) #to stabilize and speed up training.
        model.add(LeakyReLU(alpha=0.2)) #LeakyReLU with a negative slope of 0.2, which helps to prevent the vanishing gradient problem during training.

        #final Conv Layer
        model.add(Conv2D(1,kernel_size=4,strides=1,padding='valid',use_bias=False,kernel_initializer=init))  #output size < the input size.
        model.add(Flatten()) #flattens the output of the previous layer into a 1D tensor
        model.add(Activation('sigmoid')) #maps the output to a probability value between 0 and 1.

        return model
```

GAN

```
[ ] #@title GAN
    class GAN(keras.Model):


        def __init__(self,seed_size,image_length,image_channels,**kwargs):

            super(GAN,self).__init__(**kwargs)
            # Initialize the generator and discriminator
            self.generator = build_generator(seed_size)
            self.discriminator = build_discriminator(image_length,image_channels)
            self.seed_size = seed_size


        def generator_loss(self,fake_output):
          # The generator loss  - the binary cross-entropy loss between the fake output and a tensor of a fake outputs
            return cross_entropy(tf.ones_like(fake_output), fake_output)


        def discriminator_loss(self,real_output, fake_output,smooth=0.1):
            real_loss = cross_entropy(tf.ones_like(real_output)*(1-smooth), real_output)  #binary cross-entropy loss between the real output and a tensor of real output ,
            fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output) #the binary cross-entropy loss between the fake output and a tensor of zeros
            total_loss = real_loss + fake_loss
            return total_loss

        def compile(self,generator_optimizer,discriminator_optimizer):
            # Compile the GAN model with the given optimizers for the generator and discriminator
            super(GAN, self).compile()
            self.generator_optimizer = generator_optimizer
            self.discriminator_optimizer = discriminator_optimizer


        @tf.function
        def train_step(self,data):
            # Get the batch size of the input data and generate a random seed of that size
            batch_size = tf.shape(data)[0]
            seed = tf.random.normal(shape=(batch_size,self.seed_size))

            with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
                generated_image = self.generator(seed, training = True)  # Generate a batch of fake images using the generator network

                # Get the discriminator outputs for the real and fake images
                real_output = self.discriminator(data,training = True)
                fake_output = self.discriminator(generated_image,training = True)

                # Calculate the generator and discriminator losses
                gen_loss = self.generator_loss(fake_output)
                disc_loss = self.discriminator_loss(real_output,fake_output)

                # Calculate the gradients of the generator and discriminator losses with respect to the trainable variables of their respective networks
                generator_grad = gen_tape.gradient(gen_loss,self.generator.trainable_variables)
                discriminator_grad = disc_tape.gradient(disc_loss,self.discriminator.trainable_variables)

                # Apply the gradients to update the trainable variables of the generator and discriminator networks
                self.generator_optimizer.apply_gradients(zip(generator_grad,self.generator.trainable_variables))
                self.discriminator_optimizer.apply_gradients(zip(discriminator_grad,self.discriminator.trainable_variables))

            # Return generator and discriminator losses for the current batch
            return {
                "generator loss": gen_loss,
                "discriminator_loss": disc_loss
            }
```

Callbacks

```
    #@title Callbacks
    class callbacks(keras.callbacks.Callback):

        def __init__(self,noise,margin,num_rows,num_cols,**kwargs):
            super(keras.callbacks.Callback,self).__init__(**kwargs)
            # Initialize callback variables
            self.noise = noise         #Noise input used to generate images
            self.margin = margin       #Margin between generated images
            self.num_rows = num_rows   #Number of rows of generated images to display
            self.num_cols = num_cols   #Number of columns of generated images to display

        def on_epoch_end(self, epoch, logs=None):
            # Create a blank image array to store the generated images
            image_array = np.full((
                self.margin + (self.num_rows * (64 + self.margin)),    #Height of the image array
                self.margin + (self.num_cols * (64 + self.margin)),    # Width of the image array
                3), # 3 color channels for the image array
                255, dtype=np.uint8)  #Set the background color to white

            generated_images = self.model.generator.predict(self.noise) # Generate images using the generator model
            generated_images = 0.5 * generated_images + 0.5  # Scale the pixel values of the generated images to be between 0 and 1

            image_count = 0   # Counter for keeping track of the current image being generated
            # Iterate through the rows and columns of the image array and add the generated images
            for row in range(self.num_rows):
                for col in range(self.num_cols):
                    # Calculate the position of the current image in the image array
                    r = row * (64 + 16) + self.margin
                    c = col * (64 + 16) + self.margin
                    image_array[r:r + 64, c:c + 64] = generated_images[image_count] * 255  # Add the current generated image to the image array
                    image_count += 1   # Increment image count


            output_path = 'Traning_images' #path where the images will be saved
            #if the path doesn't exist then make that folder
            if not os.path.exists(output_path):
                os.makedirs(output_path)
            # Save the generated image array as a PNG file
            filename = os.path.join(output_path, f"train-{epoch+1}.png")
            im = Image.fromarray(image_array)
            im.show(filename)  #display traning  image for that epoch
            #if epoch % 20 == 0:
              #im.save(filename)


    class checkpoint_callback(keras.callbacks.Callback):

        def __init__(self,**kwargs):
            super(keras.callbacks.Callback, self).__init__(**kwargs)
        #uncomment two lines below if you want to save the training weights after every epoch
        #def on_epoch_end(self, epoch, logs=None):
            #self.model.generator.save_weights("/content/drive/MyDrive/data/generator_weights.h5")
            #self.model.discriminator.save_weights("/content/drive/MyDrive/data/discriminator_weights.h5")
```

parameters

```
#@title parameters
image_length = 64  #@param    # image length in pixels
image_channels = 3 #@param # number of color channels (3 = RGB)(1 = greyscale but have to do appropriate chang
batch_size = 128 #@param
seed_size = 128 #@param

NUM_ROWS = 4  #the number of rows in the grid of images
NUM_COLS = 7 #the number of columns in the grid of images.
MARGIN = 16 #the number of pixels of margin between each image in the grid.

fixed_seed = tf.random.normal(shape=(NUM_ROWS * NUM_COLS, seed_size)) #fixed random seed for generating exampl
GEN_LR = 0.0004 #@param {type:"number"} #Generator Learning Rate
DISC_LR = 0.0005 #@param {type:"number"} #Discriminator Learning Rate

N_EPOCS = 1 #@param {type:"number"} #number of training epochs
```

image_length: 64

image_channels: 3

batch_size: 128

seed_size: 128

GEN_LR: 0.0004

DISC_LR: 0.0005

N_EPOCS: 1

initlize Optimizers and Compile GAN

```
[ ] #@title initlize Optimizers and Compile GAN
generator_optimizer = Adam(learning_rate=GEN_LR,beta_1=0.5) #Initialize optimizer for the generator using Adam with a learning rate of GEN_LR and beta1 of 0.5.
discriminator_optimizer = Adam(learning_rate=DISC_LR,beta_1=0.5) #Initialize optimizer for the discriminator using Adam with a learning rate of DISC_LR and beta1 of 0.5.

gan = GAN(seed_size,image_length,image_channels) #initialize the GAN model
gan.compile(generator_optimizer,discriminator_optimizer) #Compile the GAN model with given optimizers for the generator and discriminator.
```

load model weights from URL

```
[ ] #@title load model weights from URL

# set the URL of the generator and discriminator weights file
url_gen = "https://drive.google.com/uc?export=download&id=1YyhIjmY2OCJgWg0uz6FlQ5yYZu_Ql7S5" #@param
url_disc = "https://drive.google.com/uc?export=download&id=1pl5fhB6BynXu6VVObU0V5jxiEa2uZkJv" #@param

# set the filename of the generator and discriminator weights files
g = "generator_weights.h5"
d = "discriminator_weights.h5"

# download the generator weights file from the URL and save it to the filename
urllib.request.urlretrieve(url_gen, g)

# load the weights from the generator weights file into the GAN model's generator
gan.generator.load_weights(g)

# download the discriminator weights file from the URL and save it to the filename
urllib.request.urlretrieve(url_disc, d)

# load the weights from the discriminator weights file into the GAN model's discriminator
gan.discriminator.load_weights(d)
```

url_gen: "https://drive.google.com/uc?export=download&id=1YyhIjmY2OCJgWg0uz6FlQ5yYZu_Ql7S5"

url_disc: "https://drive.google.com/uc?export=download&id=1pl5fhB6BynXu6VVObU0V5jxiEa2uZkJv"

Training

load weights from G_drive

```
[ ] #@title load weights from G_drive

#gan.generator.load_weights("/content/generator_weights.h5")
#gan.discriminator.load_weights("/content/discriminator_weights.h5")
#gan.generator.load_weights("/content/drive/MyDrive/data/generator_weights.h5")
#gan.discriminator.load_weights("/content/drive/MyDrive/data/discriminator_weights.h5")

#op = gan.fit(train_set,epochs=N_EPOCS,batch_size=batch_size)
```

Traning

```
[ ] #@title Traning
#train GAN

#comment out call backs if you dont want to call call back class
op = gan.fit(train_set,epochs=N_EPOCS,batch_size=batch_size
             ,callbacks=[callbacks(noise=fixed_seed,num_rows=NUM_ROWS,num_cols=NUM_COLS,margin=MARGIN), checkpoint_callback()]
             )
```

test genertating many outputs

```
[ ] #@title test genertating many outputs
#@markdown this cell genrates faces using generator
#@markdown set the number of images to be generated using a slider and run the cell to get the output
how_many_outputs_to_gen = 64 #@param {type:"slider", min:1, max:64, step:1}
def generate_faces():

    noise = tf.random.normal([how_many_outputs_to_gen,seed_size]) # generate random noise
    generated_images = gan.generator(noise) # use the generator to generate images from the random noise

    fig = plt.figure(figsize=(12,12))  #create a figure to display the generated images
    for i in range(generated_images.shape[0]):
        plt.subplot(12,12,i+1) #add a subplot for each generated image
        #display the generated image and adjust the range of pixel values to be between 0 and 1
        plt.imshow((generated_images[i,:,:,:]*0.5+0.5))
        plt.axis("off")  #turn off the axes for the subplot
    plt.show() #show the figure with the generated images

generate_faces() #call the function to generate random faces
```

this cell genrates faces using generator set the number of images to be generated using a slider and run the cell to get the output

how_many_outputs_to_gen: 64

Build just the genertor and load weights

```
[ ] #@title Build just the genertor and load weights

#@markdown must run the code wher the imports, Strides and generator sections above before running this

seed_size = 128
model = build_generator(seed_size)
url_g = "https://drive.google.com/uc?export=download&id=1YyhIjmY2OCJgWg0uz6FlQ5yYZu_Ql7S5" #@param
g = "generator_weights.h5"
urllib.request.urlretrieve(url_g, g)
model.load_weights(g)

def generate_face():

    noise = tf.random.normal([1,seed_size])
    generated_images = model(noise)

    fig = plt.figure(figsize=(12,12))
    for i in range(generated_images.shape[0]):
        plt.subplot(2,2,i+1)
        plt.imshow((generated_images[i,:,:,:]*0.5+0.5))
        plt.axis("off")
    plt.show()
    plt.savefig("face.png")

#model.save('/content/drive/MyDrive/Saved_models/AF.keras')
```

must run the code wher the imports, Strides and generator sections above before running this

url_g: "https://drive.google.com/uc?export=download&id=1YyhIjmY2OCJgWg0uz6FlQ5yYZu_Ql7S5"

GET _Output

```
[ ] #@title GET _Output
generate_face()
```

refrence : the code for traning i.e functions for building generator , discimenetor and GAN/callbacks is basaed on https://github.com/yashy3nugu/Anime-Face-GAN.