Generative Adversarial Networks: The Search for Emotion

CS156 - Machine Learning

Prof. P. Watson

TTh@6.30PM Hyderabad

21 / 04 / 2023

Table of Contents

Introduction

Background and Motivation

This paper is the logical culmination of my research series, building on the previous work where I used a Convolutional Neural Network (CNN) for binary image classification into happy and sad categories. By training a Generative Adversarial Network on the curated FER2013 dataset, I aim to take a step further in understanding the complexity of generating happy and sad faces. This research continues to address a problem that I have personally found interesting, the challenge of accurately detecting emotions in my own photos. By providing insights into the challenges and potential solutions for training Generative Adversarial Networks (GANs) on image datasets, I hope that this paper will inspire readers to explore how machine learning techniques can help them gain a deeper understanding of their own emotional sentiments.

Focus

Generative Adversarial Networks (GANs) have shown remarkable success in generating realistic images, but training them can be challenging. One hypothesis for this difficulty is that GANs struggle to settle on a stable distribution of data during training. In this paper, we aim to investigate this hypothesis by training a GAN on a curated FER2013 dataset for happy/sad photos. We investigate the potential of GANs to generate happy and sad faces. Through our experiments and analysis, we aim to provide insights into the challenges and potential solutions for training GANs on image datasets.

Data Acquisition and Loading

In this project, we will be using the FER2013 dataset, which is a widely used benchmark dataset for facial expression recognition. It contains over 35,000 grayscale images of faces with 48x48 pixel resolution, each labeled with one of seven emotion categories, including happiness

and sadness. It consists of 28,709 training images, 3,589 public testing images, and 3,589 private testing images.

The FER2013 dataset is ideal for our project because it provides a large number of labeled images of faces expressing different emotions, specifically happy/sad sentiments, which is precisely what we need to train our model. The FER2013 dataset was loaded into the Kaggle Notebook as shown in Appendix B.3, utilizing the search functionality. The data was then loaded using the TensorFlow Keras API into TensorFlow Datasets in batches of 128 images, and image sizes of (48x48) pixels.

Data Cleaning, Pre-processing, and Feature Engineering

Data Cleaning

To prepare the FER2013 dataset for training the GAN on "happy" and "sad" image categories, we cleaned and filtered the data to get rid of the unused 5 other emotion classes. We then extracted the images and labels for the "happy" and "sad" classes from both datasets using filtering techniques, converting the integer labels of the "happy" and "sad" classes into binary labels of 1 and 0, respectively. This was done to standardize the labels and facilitate the classification process. The final step involved confirming the number of images and labels for each class in both datasets to ensure a balance in the dataset.

Pre-processing

After creating the balanced datasets, we inspected the training and test data by taking a small subset of the training dataset and checking the shape and labels of the images. This exploration gave us an idea of the size and composition of the dataset and helped us determine if further preprocessing was necessary. It's worth noting that the FER2013 dataset was collected mainly from a single demographic group (university students in the United States), so data

augmentation might be especially important to ensure that the model can generalize well to other groups.

Feature Engineering

We conducted Exploratory Data Analysis to understand the data and identify potential issues that need to be addressed before modeling, consequently appropriately selecting features to engineer for the model.

Mean and Standard Deviation

The mean pixel value of 0.4893 indicate that the pixel values are centered around 0.5, which means that they are not too far from the neutral value of 0.5. The standard deviation of 0.2529 indicates that the pixel values are not too dispersed, and most of them are within a reasonable range of 0.25 above or below the mean. A higher standard deviation suggests that the pixel values are more spread out and diverse, while a lower standard deviation suggests that the pixel values are more similar to the mean. Normalization aims to transform the pixel values so that they have a mean of 0 and a standard deviation of 1. Since the mean and standard deviation of the pixel values in this dataset are already reasonable, it may not be necessary to further normalize the data.

Summary Statistics

The minimum shape of [4 48 48 3] indicates that the smallest image in the dataset has a height and width of 48 pixels and a depth of 3 color channels. The maximum shape of [32 48 48 3] indicates that the largest image in the dataset has a height and width of 48 pixels and a depth of 3 color channels. The mean shape of [31.91 48.00 48.00 3.00] indicates that the images in the dataset have an average height and width of almost 48 pixels and a depth of 3 color channels. The median shape of [32. 48. 48. 3.] is very close to the mean shape, indicating that the

distribution of image shapes is roughly symmetric. The standard deviation of shape of [1.61 0.00 0.00 0.00] indicates that the height of the images in the dataset varies slightly, but the width and color channels are consistent.

PCA Analysis

The PCA analysis of the image dataset shows that the first and second components explain about 28% and 9% of the total variance in the dataset, respectively. These two components combined explain about 37% of the total variance, which is not much. Only at PCA 50 Components is 83.82% of the variance explained. This implies that the data may have high complexity and that additional components may be needed to extract the main features of the data. The results also suggest that in designing the architecture of the models, it may be necessary to have more layers to extract these features. The PCA analysis provides insights into the main features of the data and can potentially help in reducing the dimensionality of the data for modeling purposes. However, it is important to note that using PCA for EDA alone does not necessarily imply that PCA will be used for modeling.

These EDA analyses indicate that the images in the dataset are fairly consistent in terms of size and dimensions, which is important for model training, and that feature engineering might not be required.

Task Discussion

This task will involve training a GAN on a curated FER2013 dataset of happy/sad photos. First, we preprocessed the dataset by resizing and cropping the images to a standard size of 48x48 pixels, converting them to grayscale, and normalizing the pixel values to be in the range [0, 1]. This standardization step was necessary because GANs are sensitive to the scale of the

input features, and normalizing the pixel values ensured that the network would be able to learn meaningful patterns in the data.

Next, we built the GAN architecture using the Keras API and trained it using the Adam optimizer with a learning rate of 0.0001 for the generator and 0.0002 for the discriminator, both with a batch size of 128. The generator network is designed to generate new images from random noise, and the discriminator network is trained to distinguish between real and fake images. The generator and discriminator are trained in an adversarial manner, with the generator trying to fool the discriminator and the discriminator trying to correctly classify the images.

During training, we monitored the loss of the discriminator and generator networks to assess their performance. We also saved the generator's output at regular intervals to visualize the progress of the training process. These output-generated images were ranked in order of increasing success. This evaluation will allow us to determine how well the GAN is able to capture the essential features of the happy/sad emotions and assess the overall quality of the generated images.

Model Selection

The choice of using GANs for the project was based on several factors. Firstly, GANs are capable of generating high-quality and realistic images with the same resolution as those of Variational Autoencoders (VAEs) and Stable Diffusion. Additionally, GANs offer the added advantage of allowing for the generation of diverse images by sampling from the latent space. This can be particularly useful in tasks such as artistic style transfer or image manipulation.

While VAEs can also be used for image generation, they may not produce results that are as visually appealing as those generated by GANs. On the other hand, Stable diffusion is a relatively new generative model that uses diffusion processes to generate high-quality images.

However, given its novelty and resource-intensive nature, it was not considered a viable option for the project. After careful consideration of these factors, GANs were deemed to be the most suitable model for the project.

The basics of GANs

The basic idea behind GANs is to use two neural networks, usually convolution neural networks (CNNs) to learn a distribution of data that is similar to a given training set, by training one network to generate data and another network to distinguish between real and fake data samples. Let's look at GANs' two neural networks:

Generator

The generator network takes a random noise vector z as input and generates fake data samples $G(z)$ that are meant to resemble real data samples from a training set X. For this case of generating images of faces with happy or sad expressions, the generator would take a random vector as input and produce an image of a face as output.

Mathematically, the generator can be represented by a function $G(z, \theta_g)$, where z is the random input vector, θ_g are the parameters of the generator neural network, and $G(z, \theta_g)$ is the output image generated by the generator.

Discriminator

The discriminator takes an input (either a real image from the training data or an image generated by the generator) and tries to determine whether it is a real image or a generated image. In our case, the discriminator would take an image of a face as input and produce a probability value as output, indicating whether it is a real or generated image. The output produced could also be a binary output $D(x)$ indicating whether the sample is real ($D(x) = 1$) or fake ($D(x) = 0$).

Mathematically, the discriminator can be represented by a function $D(x, \theta_d)$, where x is the input image, θ_d are the parameters of the discriminator neural network, and $D(x, \theta_d)$ is the probability that x is a real image.

The Training Loop

During training, the generator and discriminator are trained in alternating steps. The generator tries to generate images that the discriminator classifies as real, while the discriminator tries to correctly classify real images and generated images; the generator network is trained to minimize this objective function, while the discriminator network is trained to maximize it. The goal of the GAN training process is for the generator network to learn to generate data samples that are so similar to the real training data that the discriminator network cannot reliably distinguish between the two. The competitive dynamic created between the two networks can be formalized mathematically using the following objective function:

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

where:

- G is the generator network
- D is the discriminator network
- $p_{data}(x)$ is the distribution of the real training data
- $p_z(z)$ is the distribution of the random noise vector z
- $\mathbb{E}x \sim p_{data}(x)[\log D(x)]$ represents the expected value of the discriminator's output when it is presented with real training data
- $\mathbb{E}z \sim p_z(z)[\log(1 - D(G(z)))]$ represents the expected value of the discriminator's output when it is presented with fake data samples generated by the generator network.

Loss functions

GANs use two loss functions: the generator loss and the discriminator loss. These loss functions are used to measure how well the generator and discriminator are performing, and they are minimized during training in order to improve the performance of the GAN.

Generator Loss: The generator loss measures how well the generator is able to create data that is similar to the training data. Specifically, the generator loss measures how well the discriminator is fooled by the generated data. The generator tries to minimize the generator loss. Mathematically, the generator loss is defined as below.

$$L_G = -E[log(D(G(z))]$$

where:

- $G(z)$ is the generated data
- $D(G(z))$ is the discriminator's estimate of the probability that $G(z)$ is real data

Discriminator Loss: The discriminator loss measures how well the discriminator is able to distinguish between the training data and the generated data. Specifically, the discriminator loss measures how well the discriminator is able to assign high probabilities to the real data and low probabilities to the generated data. The discriminator tries to minimize the discriminator loss. Mathematically, the discriminator loss is defined as below.

$$L_D = -E[log(D(x))] - E[log(1 - D(G(z)))]$$

where:

- x is a real data point from the training data
- $G(z)$ is a generated data point
- $D(x)$is the discriminator's estimate of the probability that x is real data
- $D(G(z))$ is the discriminator's estimate of the probability that $G(z)$ is real data
- $E[log(D(x))]$ is the expected value over all real data points x
- $E[log(1 - D(G(z)))]$ is the expected value over all possible values of z

Gradient descent

The generator and discriminator networks are trained simultaneously using gradient descent. The generator and discriminator loss functions are first computed using the current parameters of the networks. Then, the gradients of the loss functions with respect to the corresponding network parameters are computed using backpropagation. Finally, the network parameters are updated using the gradient descent update rule. This process is repeated iteratively until the loss functions converge or a stopping criterion is met. Mathematically, the gradient descent update rule can be expressed as below.

$$\theta = \theta - \alpha \times \nabla L(\theta)$$

where:

- θ is a vector of network parameters
- α is the learning rate
- $L(\theta)$ is the loss function
- $\nabla L(\theta)$ is the gradient of the loss function with respect to the network parameters.

Convolution Neural Networks

CNNs are used in both the generator and discriminator networks. The generator network typically takes a low-dimensional noise vector as input and maps it to a high-dimensional image. The discriminator network takes an image as input and produces a scalar output indicating whether the image is real or fake. By using CNNs in the generator and discriminator networks, GANs are able to generate high-quality images that are visually indistinguishable from real images.

The Convolution Operation

The convolutional layers in the networks are able to effectively capture the spatial relationships in the input data, allowing the networks to learn to generate realistic images with features such as happy or sad expressions. Mathematically, the convolution operation can be expressed as below.

$$h(i,j) = \sum_k \sum_l x(i-k, j-l) * w(k,l)$$

where:

- $h(i,j)$ is the output of the convolution operation at position (i,j)
- $x(i-k, j-l)$ is the input value at position $(i-k, j-l)$
- $w(k,l)$ is the value of the filter at position (k,l)

The Upsampling Transpose Operation

Upsampling is a technique used in GANs to increase the resolution of generated images. The process involves taking a low-resolution image and transforming it into a high-resolution image. This is achieved by adding more pixels to the original image using interpolation methods. Upsampling is a key technique in GANs as it helps to improve the quality and clarity of generated images, making them more realistic and visually appealing.

Mathematically, upsampling involves increasing the size of the feature maps in the generator network. This is done by inserting a layer that increases the size of the feature maps, followed by a convolutional layer that applies filters to the feature maps. One common upsampling technique used in GANs is the transposed convolutional layer, also known as the deconvolutional layer. The output of the transposed convolutional layer is a feature map with a

larger spatial dimension than the input. The transposed convolution operation can be expressed as shown below.

$$ConvTranspose(x, W, b)i, j = \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} W_{k,l} \cdot x_{i-k,j-l} + b$$

where:

- x is the input image
- W is the filter weights
- b is the bias term
- K and L are the dimensions of the filter.

Model Training

Training involves the alternating training of the two neural networks. The generator learns to create new data samples that are similar to the real data samples, while the discriminator learns to distinguish between real and fake data samples. The training process is performed in a loop (epoch) that alternates between training the generator and the discriminator. During each epoch, the generator creates new fake data samples, and the discriminator distinguishes between the real and fake data samples. The gradients of the loss functions with respect to the network weights are computed, and the weights are updated using an optimizer.

Model Architecture

The generator architecture consists of a fully connected layer, followed by several transposed convolutional layers that upsample the data to the desired size. The discriminator architecture consists of several convolutional layers followed by a fully connected layer. Both models use leaky ReLU activation functions and dropout regularization. The GAN model is trained using the binary cross-entropy loss function. The Adam optimizer is used to optimize the generator and discriminator weights.

Model Iterations

Through our research, we are informed that the number of training epochs and the architecture design are two critical factors that can significantly impact the performance of a Generative Adversarial Network (GAN). Various combinations of epoch training numbers and architecture designs were employed in the model training process in a bid to find a concoction for realistic sentiment-embedded images.

Model Iterations	
Combination	**Expectation**
Fewer Epochs with Simple Architecture	If we use a simple architecture with a small number of training epochs, the GAN may produce low-quality and blurry images. The generator will not have enough time to learn the underlying distribution of the data and generate high-quality samples.
More Epochs with Simple Architecture	If we use a simple architecture with a large number of training epochs, the GAN may generate better-quality images. However, there is a risk of overfitting, which can lead to the generator producing the same set of images repeatedly.
Fewer Epochs with Complex Architecture	If we use a complex architecture with a small number of training epochs, the GAN may produce noisy and inconsistent results. The generator will struggle to learn the complex distribution of the data, and the discriminator may not be able to distinguish between real and fake images.

More Epochs with Complex Architecture	If we use a complex architecture with a large number of training epochs, the GAN may generate high-quality images that are visually appealing and realistic. The generator will have enough time to learn the underlying distribution of the data and generate high-quality samples. However, training a GAN with a complex architecture can be computationally expensive and time-consuming.
Pretrained Generator with Fewer Epochs	If we use a pre-trained generator with a small number of training epochs, the GAN may generate high-quality images faster. The trained generator will have already learned the underlying distribution of the data, and the GAN can focus on improving the discriminator.
Pretrained Generator with More Epochs	If we use a pre-trained generator with a large number of training epochs, the GAN may produce high-quality images faster, and the generator will not need as many iterations to learn the underlying distribution of the data. However, there is a risk of overfitting, and the generator may not be able to generate diverse images.

Table 1. Comparison of GANs with varying epoch training numbers and architecture designs

Model Performance

As expected, finding the right set-up to make the GAN work was a challenging task. In this section, we evaluate the performance of the model based on two datasets: a public dataset and a private dataset.

Curated FER2013 Dataset

Despite training variations of the model, the resultant generated pictures were of low quality. Figure 1 below shows the assorted best-generated images, indicating a correlation between high epochs and high model complexity.

Figure 1. Varying epoch number (y-axis) and model architecture complexity (x-axis) corroborate the expectations in Table 1.

Private Dataset

This dataset was created from 5 personal photos with data augmentation applied to increase the size (30) and diversity of the dataset. The data was batched and used to train the GAN model to observe the effect of the dataset on the model architecture that generated the images with the best resolution. From the findings, it seems that extracting features from the dataset was a challenge despite 50 epochs of training as seen in Figure 2 below. It would be interesting to observe if the model significantly improves with a higher number of epochs but this was subject to limited computational resources, as at this point in the research more than 30 hrs of GPU compute time (above the allotted figure) had been exhausted.

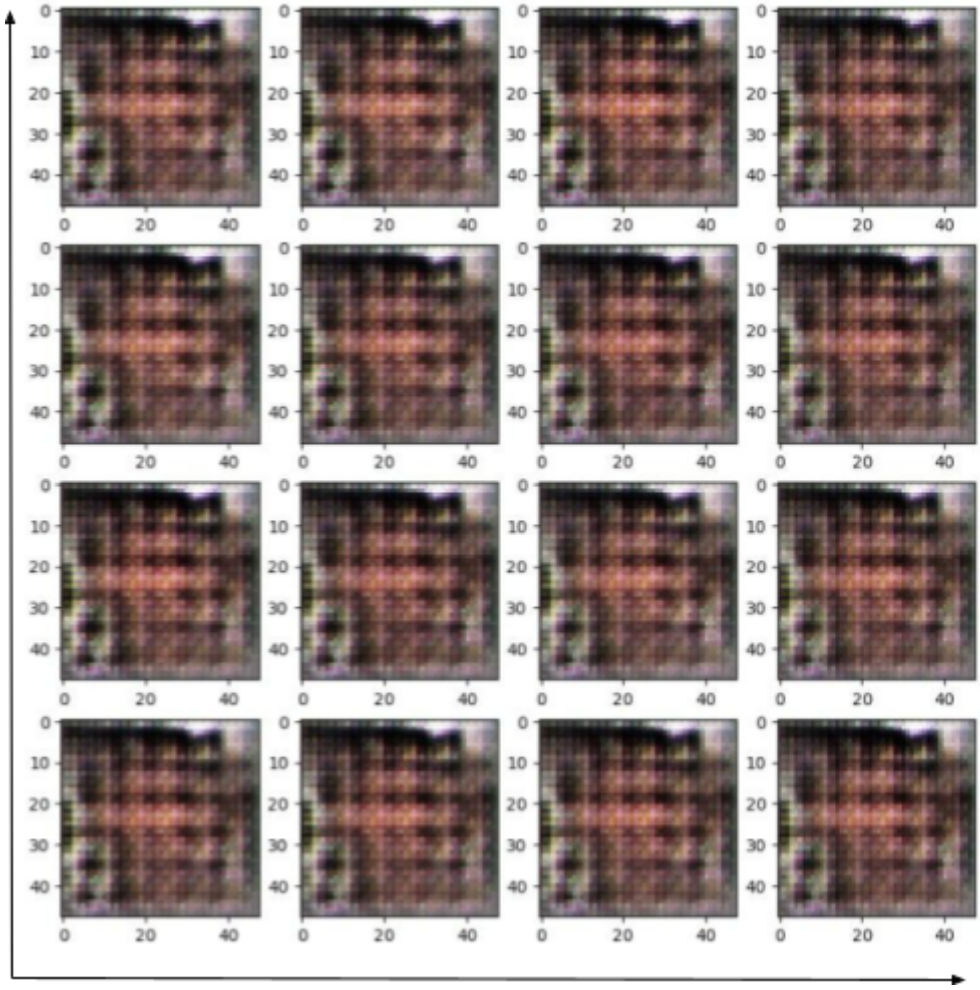


Figure 2. Best-performing model architecture output generated for 50 epochs.

Discussion and Conclusion

Our experiments demonstrate the image generation process with GANs using varying model architectures, number of epochs, and image quality. We have observed the GAN is able to generate images that display features such as eyes, nose, and mouth, as well as a background form. This implies that the GAN is able to learn important features from the data, despite the limitations of the dataset. However, we also found that the GAN is very sensitive to the setup and parameters used during training, particularly in relation to the number of epochs, the complexity of the model, and the size and quality of the dataset. In some cases, we observed that the model failed to converge or produced very low-quality images, with little or no human facial features, indicating that the GAN was not able to learn the underlying patterns in the data.

The difficulty in training GANs lies in their two-part architecture, where the generator and discriminator networks are trained iteratively. This means that any instability or non-convergence in one network can negatively affect the performance of the other network. Additionally, the size and complexity of the model, as well as the quality and size of the dataset, can also impact the GAN's ability to learn and generalize to new data.

In conclusion, our experiments highlight the challenges involved in training GANs and the importance of careful parameter tuning to ensure stable convergence and good performance. Despite these challenges, our results demonstrate the potential of GANs for generating realistic and high-quality images from low-quality datasets. Moving forward, further research is needed to explore how GAN performance can be improved, including exploring new architectures and training strategies that can help to overcome some of the challenges we have highlighted. GANs are a powerful image generation tool with the right setup.

Executive Summary

This project explores the use of Generative Adversarial Networks (GANs) to generate realistic images of human faces. The GAN was trained on both a public and private dataset, with different combinations of the model architecture and training epochs. Despite encountering some challenges in finding the optimal setup, the GAN was able to produce images that demonstrated the emergence of facial features such as eyes, nose, and mouth, as well as a background form. The results showed that GANs can struggle to reach a stable state, particularly when working with complex models or larger datasets. However, with careful consideration of these factors, GAN performance can be improved. This study provides insights into the potential of GANs for image generation and highlights the importance of selecting appropriate model configurations and training strategies.

References

Normalized Nerd (2021). The Math Behind Generative Adversarial Networks Clearly Explained! [Video Resource]. Youtube. https://www.youtube.com/watch?v=Gib_kiXgnvA

Renotte, N. (2022). Build a Generative Adversarial Neural Network with Tensorflow and Python Deep Learning Projects. [Video Resource]. YouTube. https://www.youtube.com/watch?v=AALBGpLbj6Q

Appendix

A. AI Tools Declaration

ChatGPT was used in certain sections of the assignment using the prompt, “Are there any features in this text which should be removed to make it more concise?” The aim was to improve the composition of the said sections.

B. Kaggle NoteBook

Scroll below to interact with the code appendix document.

B. Kaggle Notebook

1. Dependencies

> This section brings all the packages into one block for ease of maintainance.

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, Lambda, Input, Conv2DTranspose, Reshape, LeakyReLU, Up
from tensorflow.keras.losses import BinaryCrossentropy, CategoricalCrossentropy
from tensorflow.keras.metrics import Precision, Recall, BinaryAccuracy
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential, load_model, Model
from tensorflow.keras.preprocessing.image import array_to_img
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import StandardScaler
from scipy.signal import convolve2d, correlate2d
from tensorflow.keras.applications import VGG16
from tensorflow.keras.callbacks import Callback
from tensorflow.keras.utils import plot_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import backend as K
from sklearn.decomposition import PCA
from tensorflow.keras import layers
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
import tensorflow as tf
from scipy import stats
from PIL import Image
import seaborn as sns
import pandas as pd
import numpy as np
import zipfile
import pprint
import imghdr
import json
import cv2
import os
```

2. Data Cleaning, Pre-processing, and Feature Engineering

2.a. Cleaning and Loading the data

```
# Define dataset directory and image size
data_dir = "/kaggle/input/fer2013"
img_size = (48, 48)
batch_size = 128

# Load all classes from the training dataset
train = tf.keras.preprocessing.image_dataset_from_directory(
    os.path.join(data_dir, "train"),
    labels="inferred",
    label_mode="int",
    class_names=None,
    batch_size=batch_size,
    image_size=img_size,
    shuffle=True,
    seed=42,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    smart_resize=False,
)

# Load all classes from the test
test = tf.keras.preprocessing.image_dataset_from_directory(
    os.path.join(data_dir, "test"),
    labels="inferred",
    label_mode="int",
    class_names=None,
    batch_size=batch_size,
    image_size=img_size,
    shuffle=True,
    seed=42,
    validation_split=None,
```

```
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    smart_resize=False,
)
```

```
Found 28709 files belonging to 7 classes.
Found 7178 files belonging to 7 classes.
```

```
# Print the class names of train
print("Train class names:", "\n\n", train.class_names, "\n\n")

# Print the class names of test
print("Test class names:", "\n\n", test.class_names)
```

```
Train class names:

 ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad', 'surprise']


Test class names:

 ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad', 'surprise']
```

```
# Define the class indices for "happy" and "sad"
happy_class = 3
sad_class = 5

# Filter the images and labels for the "happy" and "sad" classes from the train dataset
train_images = []
train_labels = []
for images, labels in train:
    happy_idx = np.where(labels == happy_class)[0]
    sad_idx = np.where(labels == sad_class)[0]
    indices = np.concatenate((happy_idx, sad_idx))
    train_images.append(images.numpy()[indices])
    train_labels.append(labels.numpy()[indices])

train_images = np.concatenate(train_images)
train_labels = np.concatenate(train_labels)

# Filter the images and labels for the "happy" and "sad" classes from the test dataset
test_images = []
test_labels = []
for images, labels in test:
    happy_idx = np.where(labels == happy_class)[0]
    sad_idx = np.where(labels == sad_class)[0]
    indices = np.concatenate((happy_idx, sad_idx))
    test_images.append(images.numpy()[indices])
    test_labels.append(labels.numpy()[indices])

test_images = np.concatenate(test_images)
test_labels = np.concatenate(test_labels)

# Confirm we have the right classes 3-sad 5-happy
print(f"Train:\n\nClass names:{np.unique(train_labels)}, Images: {len(train_images)}\n\n")
print(f"Test:\n\nClass names:{np.unique(test_labels)}, Images: {len(test_images)}")
```

```
Train:

Class names:[3 5], Images: 12045


Test:

Class names:[3 5], Images: 3021
```

```
# Convert integer labels to binary labels
train_labels[train_labels == sad_class] = 0
train_labels[train_labels == happy_class] = 1
test_labels[test_labels == sad_class] = 0
test_labels[test_labels == happy_class] = 1

# Confirm we have the right classes 0-sad 1-happy
print(f"Train:\n\nClass names:{np.unique(train_labels)}, Images: {len(train_images)}, Happy: {len(np.where(train_labels == 1)[0])}, Sad:
print(f"Test:\n\nClass names:{np.unique(test_labels)}, Images: {len(test_images)}, Happy: {len(np.where(test_labels == 1)[0])}, Sad: {ler
```

```
Train:

Class names:[0 1], Images: 12045, Happy: 7215, Sad: 4830
```

```
Test:

Class names:[0 1], Images: 3021, Happy: 1774, Sad: 1247
```

2.b. Pre-processing the data

```
# Define the class indices for "happy" and "sad"
happy_class = 1
sad_class = 0

# Filter the images and labels for the "happy" and "sad" classes from the train dataset
happy_train_idx = np.where(train_labels == happy_class)[0]
sad_train_idx = np.where(train_labels == sad_class)[0]

# Calculate the number of samples for each class in the train dataset
num_happy_train = len(happy_train_idx)
num_sad_train = len(sad_train_idx)
num_total_train = num_happy_train + num_sad_train

# Determine the number of samples to include in the train and validation sets
train_size = int(0.8 * num_total_train)
val_size = num_total_train - train_size

# Create a balanced train dataset with equal number of happy and sad samples
train_happy_idx = np.random.choice(happy_train_idx, size=train_size//2, replace=False)
train_sad_idx = np.random.choice(sad_train_idx, size=train_size//2, replace=False)
train_images_balanced = np.concatenate([train_images[train_happy_idx], train_images[train_sad_idx]])
train_labels_balanced = np.concatenate([np.ones(len(train_happy_idx)), np.zeros(len(train_sad_idx))])

# Shuffle the train dataset
train_shuffle_idx = np.random.permutation(len(train_labels_balanced))
train_images_balanced = train_images_balanced[train_shuffle_idx]
train_labels_balanced = train_labels_balanced[train_shuffle_idx]

# Filter the images and labels for the "happy" and "sad" classes from the test dataset
happy_test_idx = np.where(test_labels == happy_class)[0]
sad_test_idx = np.where(test_labels == sad_class)[0]

# Calculate the number of samples for each class in the test dataset
num_happy_test = len(happy_test_idx)
num_sad_test = len(sad_test_idx)
num_total_test = num_happy_test + num_sad_test

# Determine the number of samples to include in the test set
test_size = num_total_test // 5

# Create a balanced test dataset with equal number of happy and sad samples
test_happy_idx = np.random.choice(happy_test_idx, size=test_size//2, replace=False)
test_sad_idx = np.random.choice(sad_test_idx, size=test_size//2, replace=False)
test_images_balanced = np.concatenate([test_images[test_happy_idx], test_images[test_sad_idx]])
test_labels_balanced = np.concatenate([np.ones(len(test_happy_idx)), np.zeros(len(test_sad_idx))])

# Shuffle the test dataset
test_shuffle_idx = np.random.permutation(len(test_labels_balanced))
test_images_balanced = test_images_balanced[test_shuffle_idx]
test_labels_balanced = test_labels_balanced[test_shuffle_idx]

print("Completed")
```

```
Completed
```

```
# Create TensorFlow datasets to easy data handling
train_ds = tf.data.Dataset.from_tensor_slices((train_images_balanced, train_labels_balanced)).shuffle(len(train_labels_balanced)).batch(b
test_ds = tf.data.Dataset.from_tensor_slices((test_images_balanced, test_labels_balanced)).shuffle(len(test_labels_balanced)).batch(batch

print("Completed")
```

```
Completed
```

```
num_train_examples = tf.data.experimental.cardinality(train_ds).numpy()
num_test_examples = tf.data.experimental.cardinality(test_ds).numpy()

train_counts = np.zeros(2)
for images, labels in train_ds:
    train_counts += np.array([tf.math.count_nonzero(labels == 0).numpy(), tf.math.count_nonzero(labels == 1).numpy()])

test_counts = np.zeros(2)
for images, labels in test_ds:
    test_counts += np.array([tf.math.count_nonzero(labels == 0).numpy(), tf.math.count_nonzero(labels == 1).numpy()])
```

```
# Confirm we have balanced classes 0-sad 1-happy
print(f"\nTrain:\n\n\tImages: {train_counts[0]+train_counts[1]}, Happy: {train_counts[0]}, Sad: {train_counts[1]}\n")
print(f"Test:\n\n\tImages: {test_counts[0] + test_counts[1]}, Happy: {test_counts[0]}, Sad: {test_counts[1]}\n")
```

```
Train:

        Images: 9636.0, Happy: 4818.0, Sad: 4818.0

Test:

        Images: 604.0, Happy: 302.0, Sad: 302.0
```

```
# Shuffle and inspect training data
train_ds = train_ds.shuffle(10000)
sample_train = train_ds.take(10)
for images, labels in sample_train:
    print(images.shape, labels.shape, np.unique(labels))
```

```
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
```

```
# Shuffle and inspect testing data
test_ds = test_ds.shuffle(10000)
sample_test = test_ds.take(10)
for images, labels in sample_test:
    print(images.shape, labels.shape, np.unique(labels))
```

```
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(128, 48, 48, 3) (128,) [0. 1.]
(92, 48, 48, 3) (92,) [0. 1.]
```

```
# Normalize the pixel values between 0 and 1
train_ds = train_ds.map(lambda x, y: (tf.cast(x, tf.float32) / 255.0, y))
test_ds = test_ds.map(lambda x, y: (tf.cast(x, tf.float32) / 255.0, y))

print("Completed")
```

```
Completed
```

2.c. Exploratory Data Analysis

1. Visualizing the images in the dataset to gain a better understanding of the data.
2. Calculating the mean and standard deviation of the pixel values in the images to normalize the data for better training performance.
3. Checking the class balance to ensure that the dataset is not biased towards one class or the other.
4. Checking the size and dimensions of the images to ensure they are consistent and fit for model training.
5. Conducting principal component analysis (PCA) to understand the main features of the data.

2.c.i. Visualizing sample images

We will plot some sample images from the dataset to visualize the data and ensure that it was loaded correctly. We will plot 10 images along with their corresponding labels.

```
# Define class labels
class_names = ['Sad', 'Happy']

# Take 10 samples from train_ds
sample_train = train_ds.take(10)

# Create a figure with 2 rows and 5 columns
fig, axes = plt.subplots(2, 5, figsize=(10, 5))

# Loop over each sample and plot it
for i, (image, label) in enumerate(sample_train):
    ax = axes[i // 5, i % 5]
    ax.imshow(image[0])
    ax.set_title(class_names[int(label[0])])
```

```
    ax.axis('off')

# Display the figure
plt.show()
```

2.c.ii. Descriptive Statistics

```
# Compute mean and standard deviation of pixel values in train_ds
pixel_means = []
pixel_stds = []

for images, labels in train_ds:
    pixel_means.append(tf.math.reduce_mean(images, axis=(0, 1, 2)))
    pixel_stds.append(tf.math.reduce_std(images, axis=(0, 1, 2)))

mean = tf.reduce_mean(pixel_means, axis=0)
std = tf.reduce_mean(pixel_stds, axis=0)

print(f"\nMean pixel value: {mean[0].numpy()}\n")
print(f"Standard deviation of pixel values: {std[0].numpy()}")
```

```
Mean pixel value: 0.49013814330101013

Standard deviation of pixel values: 0.25501903891563416
```

2.c.iii. Visualizing the class balance

> The visualization confirms that we properly balanced the train dataset with an even balance between the classes. Happy has 4818 images and Sad has 4818 images.

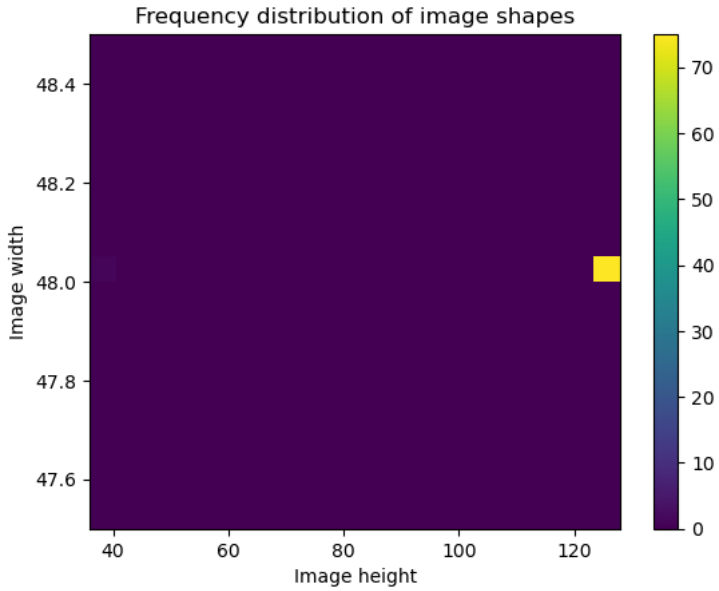
```
# Get the labels and count the number of images per class
labels = np.concatenate([y for x, y in train_ds], axis=0)
counts = np.unique(labels, return_counts=True)

# Plot the class distribution
plt.bar(np.arange(len(counts[0])), counts[1])
plt.xticks(np.arange(len(counts[0])), counts[0])
plt.xlabel('Class')
plt.ylabel('Count')
plt.title('Class Balance')
plt.show()

# print summary of histogram
print(f"Even balance between the classes. Happy (0): {counts[1][0]}, Sad (1): {counts[1][1]}")
```

2.c.iv. Checking image sizes and dimensions

```
# Compile all the image shapes
image_shapes = []

for image, _ in train_ds:
    image_shapes.append(image.shape)

# Convert list of shape tuples to numpy array
image_shapes = np.stack(image_shapes)

print("Completed")
```

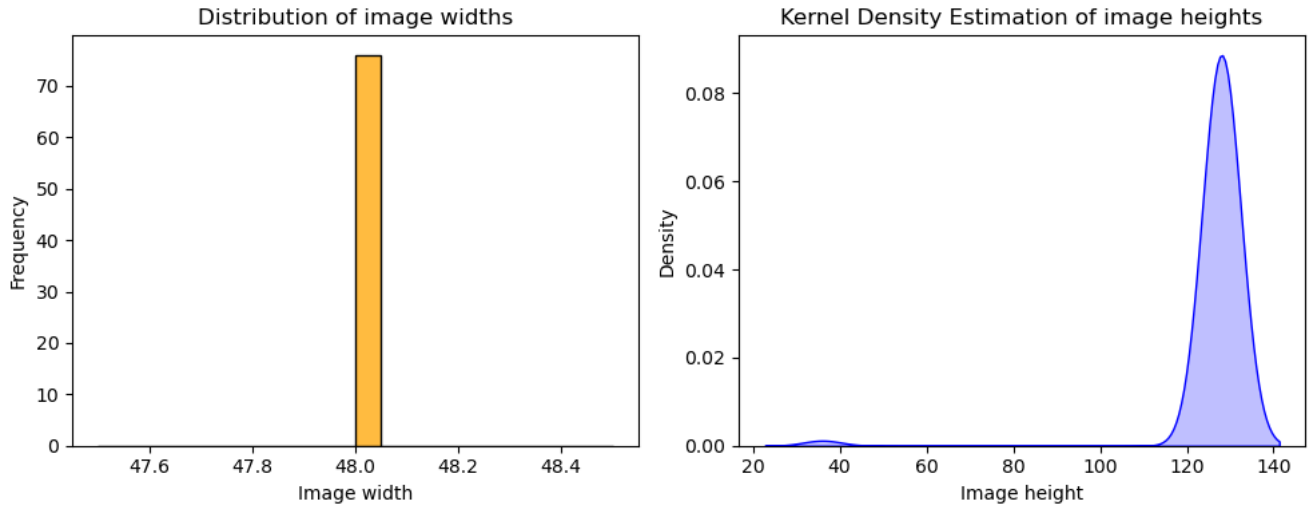
```
Completed
```

```
# Plot histogram of image shapes
plt.hist2d(image_shapes[:,0], image_shapes[:,1], bins=20)
plt.xlabel('Image height')
plt.ylabel('Image width')
plt.title('Frequency distribution of image shapes')
plt.colorbar()
plt.show()

# convert the list to a numpy array for easier computation
image_shapes = np.array(image_shapes)

print("Summary Statistics of Image Shapes:")
print(f"Minimum Shape: {np.min(image_shapes, axis=0)}")
print(f"Maximum Shape: {np.max(image_shapes, axis=0)}")
print(f"Mean Shape: {np.mean(image_shapes, axis=0)}")
print(f"Median Shape: {np.median(image_shapes, axis=0)}")
print(f"Standard Deviation of Shape: {np.std(image_shapes, axis=0)}")
```




```
Summary Statistics of Image Shapes:
Minimum Shape: [36 48 48  3]
Maximum Shape: [128  48  48   3]
Mean Shape: [126.78947368  48.          48.           3.        ]
Median Shape: [128.  48.  48.   3.]
Standard Deviation of Shape: [10.48346541  0.          0.          0.        ]
```

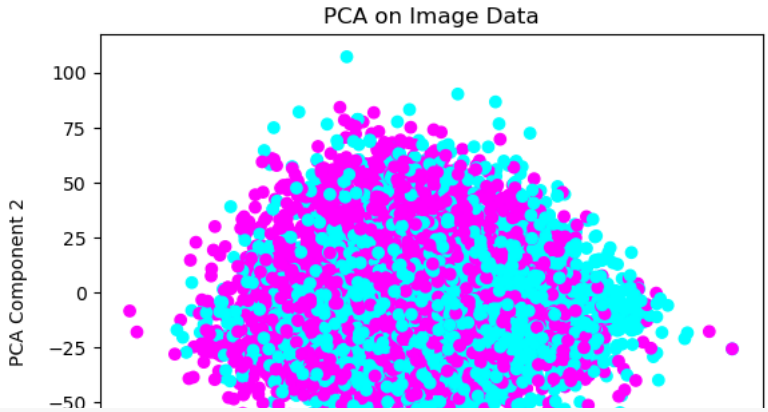
```
# Plot histogram of image widths
fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(10,4))
sns.histplot(x=image_shapes[:,1], bins=20, ax=ax0, color='orange')
ax0.set_xlabel('Image width')
ax0.set_ylabel('Frequency')
ax0.set_title('Distribution of image widths')

# Plot kernel density estimate of image heights
sns.kdeplot(x=image_shapes[:,0], ax=ax1, color='b', fill=True)
ax1.set_xlabel('Image height')
ax1.set_ylabel('Density')
ax1.set_title('Kernel Density Estimation of image heights')

plt.tight_layout()
plt.show()

caption="The distribution of image widths is plotted using a histogram while the distribution of image heights is plotted using a kernel

print(caption)
```




```
The distribution of image widths is plotted using a histogram while the distribution of image heights is plotted using a kernel den
```

2.c.v. Conducting PCA Analysis for EDA

```
# Reshape images to vectors
train_images_flat = train_images_balanced.reshape(-1, np.prod(train_images_balanced.shape[1:]))
test_images_flat = test_images_balanced.reshape(-1, np.prod(test_images_balanced.shape[1:]))

# Standardize data
scaler = StandardScaler()
train_images_scaled = scaler.fit_transform(train_images_flat)
test_images_scaled = scaler.transform(test_images_flat)

# Perform PCA
pca = PCA(n_components=2)
train_images_pca = pca.fit_transform(train_images_scaled)

# Plot PCA components
plt.scatter(train_images_pca[:, 0], train_images_pca[:, 1], c=train_labels_balanced, cmap='cool')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.title('PCA on Image Data')
plt.show()

# Get variance ratios of PCA components
print("PCA Component 1 Variance Ratio: ", pca.explained_variance_ratio_[0])
print("PCA Component 2 Variance Ratio: ", pca.explained_variance_ratio_[1])
```




```
# Enter number of components to see difference
components = 50
print_all = False

# Redo PCA to see how many more components are needed
pca = PCA(n_components=components)
train_images_pca = pca.fit_transform(train_images_scaled)

# Print summary
print(f"\nPCA {components} Components explains {sum(pca.explained_variance_ratio_)*100:.2f}% of the variance\n")

# Get variance ratios of PCA components
if print_all:
    for i in range(len(pca.explained_variance_ratio_)):
        print(f"PCA Component {i+1} Variance Ratio: {pca.explained_variance_ratio_[i]}")
```

```
PCA 50 Components explains 83.89% of the variance
```

3. Model Selection

> For the models, I decided to use Convolutional Neural Networks (CNNs) for both the generator and discriminator.

Recap on sequential models

- Having a sequential model means that the layers are stacked on top of each other, i.e, a linear stack of layers. Each layer in the network is connected to the preceding and following layers, and information flows through the layers in a sequential order. This makes sequential models well-suited for processing sequential data, such as time series data or natural language data, where the order of the input data is important.
- In fact, we have seen sequential models before when we worked with feedforward neural netwroks which consisted of an input layer, one or more hidden layers, and an output layer. In a feedforward network, information flows forward from the input layer to the output layer, with no feedback connections.

3.a. Generator

```
def build_generator():
    model = Sequential()

    model.add(Dense(256 * 6 * 6, input_dim=256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((6, 6, 256)))

    # Upsampling block 2
    model.add(layers.Conv2DTranspose(128, 5, padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))

    # Upsampling block 3
    model.add(layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))

    # Upsampling block 4
    model.add(layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))

    # Upsampling block 5
    model.add(layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))

    model.add(Conv2D(3, (3,3), padding='same', activation='sigmoid'))
```

```
    return model
```

```
generator = build_generator()
```

```
generator.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 9216)              2368512

 leaky_re_lu (LeakyReLU)     (None, 9216)              0

 reshape (Reshape)           (None, 6, 6, 256)         0

 conv2d_transpose (Conv2DTra  (None, 6, 6, 128)        819328
 nspose)

 leaky_re_lu_1 (LeakyReLU)   (None, 6, 6, 128)         0

 conv2d_transpose_1 (Conv2DT  (None, 12, 12, 128)      262272
 ranspose)

 leaky_re_lu_2 (LeakyReLU)   (None, 12, 12, 128)       0

 conv2d_transpose_2 (Conv2DT  (None, 24, 24, 128)      262272
 ranspose)

 leaky_re_lu_3 (LeakyReLU)   (None, 24, 24, 128)       0

 conv2d_transpose_3 (Conv2DT  (None, 48, 48, 128)      262272
 ranspose)

 leaky_re_lu_4 (LeakyReLU)   (None, 48, 48, 128)       0

 conv2d (Conv2D)             (None, 48, 48, 3)         3459

=================================================================
Total params: 3,978,115
Trainable params: 3,978,115
Non-trainable params: 0
_________________________________________________________________
```

3.b. Discriminator

```
def build_discriminator():
    model = Sequential()

    # First Conv Block
    model.add(Conv2D(64, (3,3), padding='same', input_shape=(48,48,3)))
    model.add(LeakyReLU(alpha=0.2))

    # Second Conv Block
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    # Third Conv Block
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    # Fourth Conv Block
    model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    # Flatten
    model.add(Flatten())

    # Fully connected layer
    model.add(Dropout(0.4))

    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    return model
```

```
discriminator = build_discriminator()
```

```
discriminator.summary()
```

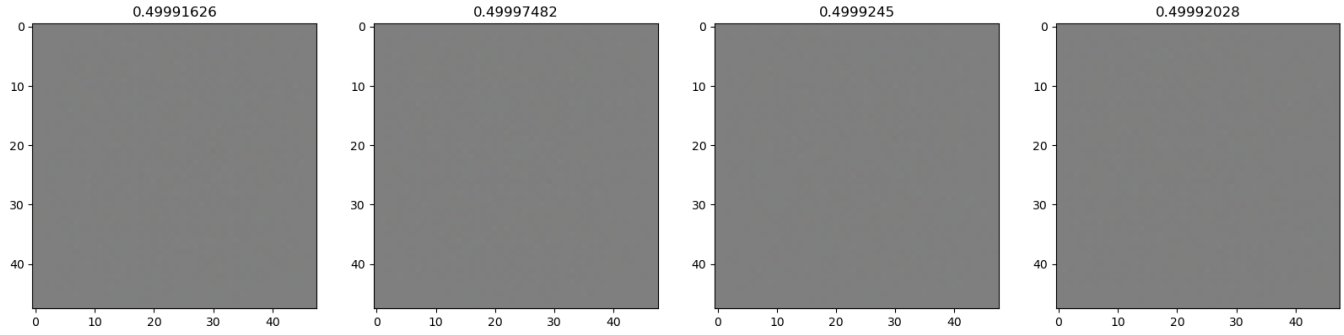
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_1 (Conv2D)           (None, 48, 48, 64)        1792

 leaky_re_lu_5 (LeakyReLU)   (None, 48, 48, 64)        0

 conv2d_2 (Conv2D)           (None, 24, 24, 128)       73856

 leaky_re_lu_6 (LeakyReLU)   (None, 24, 24, 128)       0

 conv2d_3 (Conv2D)           (None, 12, 12, 128)       147584

 leaky_re_lu_7 (LeakyReLU)   (None, 12, 12, 128)       0

 conv2d_4 (Conv2D)           (None, 6, 6, 256)         295168

 leaky_re_lu_8 (LeakyReLU)   (None, 6, 6, 256)         0

 flatten (Flatten)           (None, 9216)              0

 dropout (Dropout)           (None, 9216)              0

 dense_1 (Dense)             (None, 1)                 9217

=================================================================
Total params: 527,617
Trainable params: 527,617
Non-trainable params: 0
_________________________________________________________________
```

3.c. Sanity Checkpoint

Let's confirm that the models are working as intended before moving on to training.

```
# Sample 4 generated images to test generator
images = generator.predict(tf.random.normal((4, 256, 1)))
labels = discriminator.predict(images)

# Visualize output
fig, ax = plt.subplots(ncols=4, figsize=(20,20))
for idx, img in enumerate(images):
    # Appending the image label as the plot title
    ax[idx].imshow(np.squeeze(img))
    ax[idx].title.set_text(labels[idx][0])
```

```
1/1 [==============================] - 0s 497ms/step
1/1 [==============================] - 0s 103ms/step
```



4. Training the Models

> To train our models, we will use a custom class and callback to ensure we are alternating between the generator and discriminator training.

```
class EmotionGAN(Model):
    def __init__(self, generator, discriminator, *args, **kwargs):
        # Pass through args and kwargs to base class
        super().__init__(*args, **kwargs)

        # Create attributes for gen and disc
        self.generator = generator
```

```
        self.discriminator = discriminator

    def compile(self, g_opt, d_opt, g_loss, d_loss, *args, **kwargs):
        # Compile with base class
        super().compile(*args, **kwargs)

        # Create attributes for losses and optimizers
        self.g_opt = g_opt
        self.d_opt = d_opt
        self.g_loss = g_loss
        self.d_loss = d_loss

    def train_step(self, batch):
        # Get the data
        real_images, real_labels = batch
        fake_images = self.generator(tf.random.normal((128, 256, 1)), training=False)

        # Train the discriminator
        with tf.GradientTape() as d_tape:
            # Pass the real and fake images to the discriminator model
            yhat_real = self.discriminator(real_images, training=True)
            yhat_fake = self.discriminator(fake_images, training=True)
            yhat_realfake = tf.concat([yhat_real, yhat_fake], axis=0)

            # Create labels for real and fakes images
            y_realfake = tf.concat([tf.zeros_like(yhat_real), tf.ones_like(yhat_fake)], axis=0)

            # Add some noise to the TRUE outputs
            noise_real = 0.15*tf.random.uniform(tf.shape(yhat_real))
            noise_fake = -0.15*tf.random.uniform(tf.shape(yhat_fake))
            y_realfake += tf.concat([noise_real, noise_fake], axis=0)

            # Calculate loss - BINARYCROSS
            total_d_loss = self.d_loss(y_realfake, yhat_realfake)

        # Apply backpropagation - nn learn
        dgrad = d_tape.gradient(total_d_loss, self.discriminator.trainable_variables)
        self.d_opt.apply_gradients(zip(dgrad, self.discriminator.trainable_variables))

        # Train the generator
        with tf.GradientTape() as g_tape:
            # Generate some new images
            gen_images = self.generator(tf.random.normal((128, 256, 1)), training=True)

            # Create the predicted labels
            predicted_labels = self.discriminator(gen_images, training=False)

            # Calculate loss - trick to training to fake out the discriminator
            total_g_loss = self.g_loss(tf.zeros_like(predicted_labels), predicted_labels)

        # Apply backprop
        ggrad = g_tape.gradient(total_g_loss, self.generator.trainable_variables)
        self.g_opt.apply_gradients(zip(ggrad, self.generator.trainable_variables))

        return {"d_loss":total_d_loss, "g_loss":total_g_loss}

    def call(self, inputs, training=None, mask=None):
        x = self.generator(inputs, training=training)
        return self.discriminator(x, training=training)

print('Completed')
```

```
 Completed
```

```
# Create instance of subclassed model
emoGan = EmotionGAN(generator, discriminator)

# Setup optimizers and loss functions
g_opt = Adam(learning_rate=0.0001)
d_opt = Adam(learning_rate=0.0002)
g_loss = BinaryCrossentropy()
d_loss = BinaryCrossentropy()

# Compile the model
emoGan.compile(g_opt, d_opt, g_loss, d_loss)

# Create directory for the logs emo-Model
logs_dir = 'kaggle/working/logs/emo/'
os.makedirs(logs_dir, exist_ok=True)
emo_logs_directory = 'kaggle/working/logs/emo'
```

```
class CustomCallback(Callback):
    def __init__(self, num_img=3, latent_dim=256):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.uniform((self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
        for i in range(self.num_img):
            img = array_to_img(generated_images[i])
            img.save(os.path.join(emo_logs_directory, f'generated_img_{epoch}_{i}.png'))

print('Completed')
```

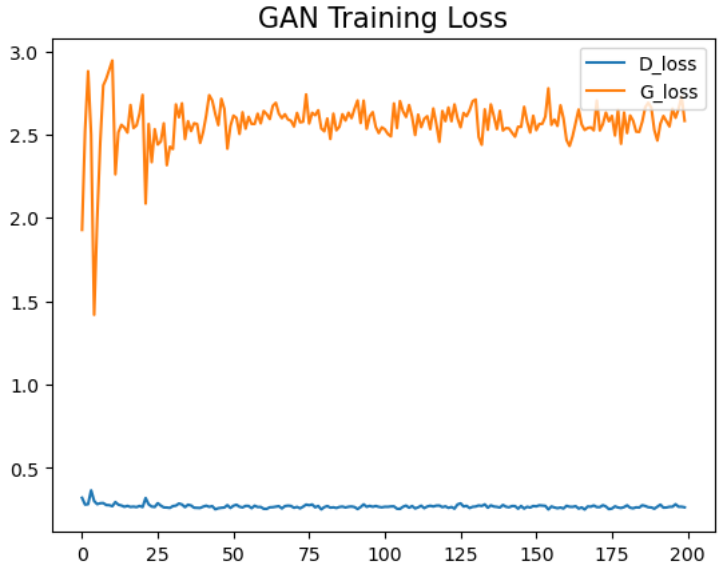
```
Completed
```

```
# Train the model and store the training history
emo_hist = emoGan.fit(train_ds, epochs=200, callbacks=[CustomCallback()])
```

```
Epoch 1/200
76/76 [==============================] - 20s 207ms/step - d_loss: 0.5437 - g_loss: 1.1834
Epoch 2/200
76/76 [==============================] - 16s 212ms/step - d_loss: 0.3761 - g_loss: 1.8356
Epoch 3/200
76/76 [==============================] - 16s 214ms/step - d_loss: 0.2892 - g_loss: 2.4611
Epoch 4/200
76/76 [==============================] - 16s 206ms/step - d_loss: 0.2781 - g_loss: 2.5407
Epoch 5/200
76/76 [==============================] - 16s 207ms/step - d_loss: 0.5194 - g_loss: 2.5558
Epoch 6/200
76/76 [==============================] - 16s 209ms/step - d_loss: 0.5614 - g_loss: 2.8887
Epoch 7/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.2908 - g_loss: 2.6940
Epoch 8/200
76/76 [==============================] - 16s 212ms/step - d_loss: 0.2954 - g_loss: 2.6456
Epoch 9/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.2792 - g_loss: 2.6376
Epoch 10/200
76/76 [==============================] - 16s 208ms/step - d_loss: 0.4395 - g_loss: 2.3233
Epoch 11/200
76/76 [==============================] - 16s 208ms/step - d_loss: 0.3992 - g_loss: 1.9964
Epoch 12/200
76/76 [==============================] - 16s 209ms/step - d_loss: 0.4203 - g_loss: 1.8082
Epoch 13/200
76/76 [==============================] - 16s 209ms/step - d_loss: 0.3747 - g_loss: 2.1140
Epoch 14/200
76/76 [==============================] - 16s 211ms/step - d_loss: 0.5173 - g_loss: 1.8780
Epoch 15/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.4960 - g_loss: 1.9839
Epoch 16/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.4517 - g_loss: 2.1683
Epoch 17/200
76/76 [==============================] - 16s 208ms/step - d_loss: 0.4141 - g_loss: 2.0311
Epoch 18/200
76/76 [==============================] - 16s 211ms/step - d_loss: 0.5145 - g_loss: 1.8436
Epoch 19/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.4358 - g_loss: 1.6691
Epoch 20/200
76/76 [==============================] - 16s 208ms/step - d_loss: 0.5473 - g_loss: 1.7822
Epoch 21/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.5321 - g_loss: 1.3222
Epoch 22/200
76/76 [==============================] - 16s 212ms/step - d_loss: 0.4803 - g_loss: 1.6499
Epoch 23/200
76/76 [==============================] - 16s 208ms/step - d_loss: 0.5026 - g_loss: 1.5346
Epoch 24/200
76/76 [==============================] - 16s 209ms/step - d_loss: 0.4871 - g_loss: 1.6678
Epoch 25/200
76/76 [==============================] - 16s 209ms/step - d_loss: 0.5443 - g_loss: 1.3987
Epoch 26/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.5159 - g_loss: 1.8217
Epoch 27/200
76/76 [==============================] - 16s 209ms/step - d_loss: 0.4288 - g_loss: 1.8133
Epoch 28/200
76/76 [==============================] - 16s 210ms/step - d_loss: 0.3788 - g_loss: 2.0443
Epoch 29/200
76/76 [==============================] - 16s 212ms/step - d_loss: 0.5821 - g_loss: 1.8895
```

```
# Plotting the learning history
plt.plot(emo_hist.history['d_loss'], label='D_loss')
plt.plot(emo_hist.history['g_loss'], label='G_loss')
plt.title('GAN Training Loss', fontsize=15)
plt.legend(loc="upper right")
```

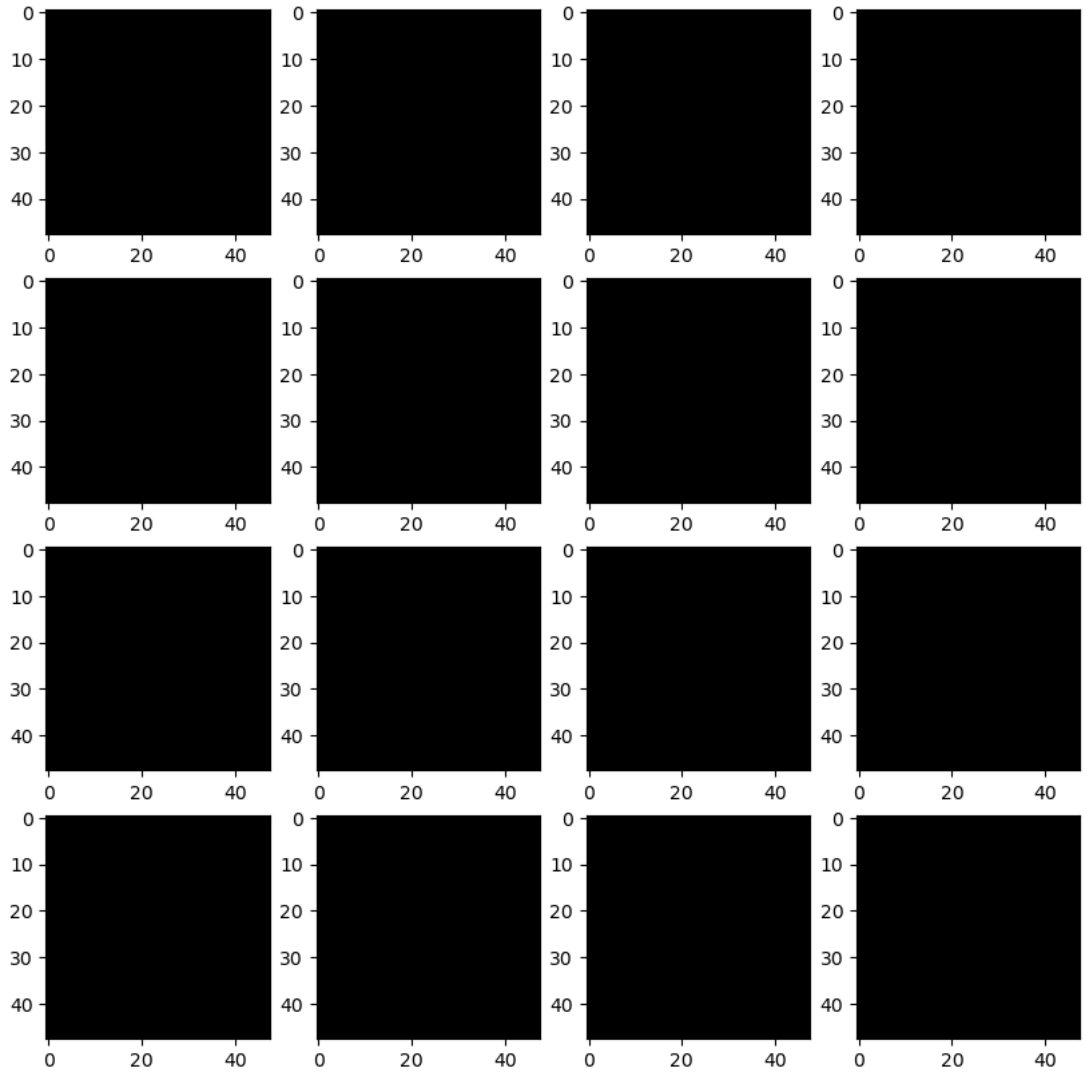
```
plt.show()
```




```
imgs = generator.predict(tf.random.normal((16, 256, 1)))
#imgs = (imgs + 1) / 2.0
```

```
1/1 [==============================] - 0s 23ms/step
```

```
fig, ax = plt.subplots(ncols=4, nrows=4, figsize=(10,10))
for r in range(4):
    for c in range(4):
        ax[r][c].imshow(imgs[(r+1)*(c+1)-1])
```



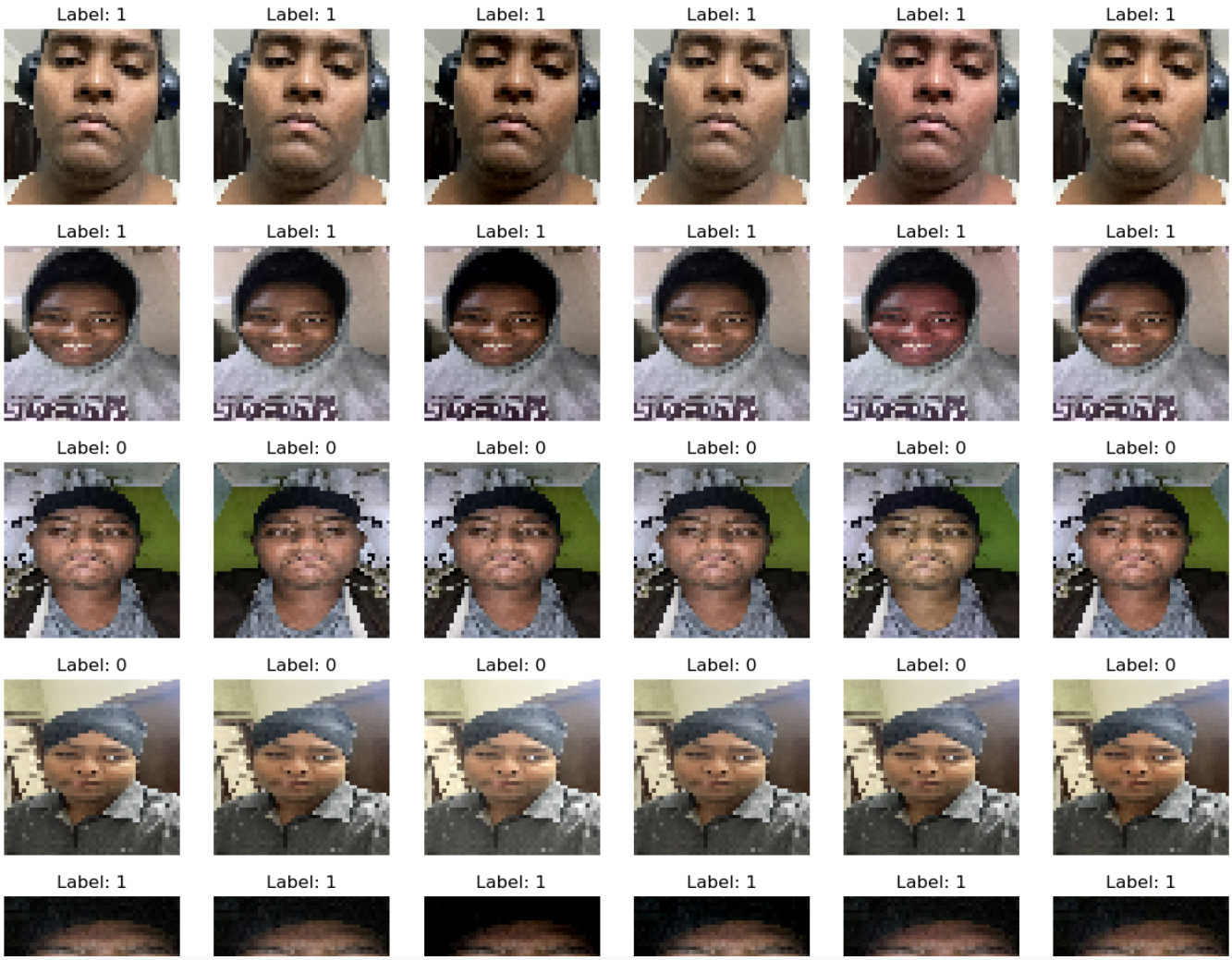
Actual results not displayed above, notebook refreshed and out of GPU. Cannot retrain model.

4.b. Private Dataset

```
def augment_images(image_paths):

    # Define the augmentation pipeline
    def augment(image):
        image1 = tf.image.random_flip_left_right(image)
        image2 = tf.image.random_brightness(image, max_delta=0.1)
        image3 = tf.image.random_contrast(image, lower=0.9, upper=1.1)
        image4 = tf.image.random_hue(image, max_delta=0.05)
        image5 = tf.image.random_saturation(image, lower=0.9, upper=1.1)

        image1 = tf.image.resize(image1, (48, 48))
        image2 = tf.image.resize(image2, (48, 48))
        image3 = tf.image.resize(image3, (48, 48))
        image4 = tf.image.resize(image4, (48, 48))
        image5 = tf.image.resize(image5, (48, 48))

        images = [image, image1, image2, image3, image4, image5]

        label = tf.random.uniform([], minval=0, maxval=2, dtype=tf.int32)
        labels = [label] * 6

        # Plot each image and label
        fig, axs = plt.subplots(nrows=1, ncols=6, figsize=(15,15))
        for i in range(6):
            axs[i].imshow(images[i])
            axs[i].set_title(f"Label: {labels[i]}")
            axs[i].axis('off')
        plt.show()

        return images, labels

    # Load the images from the paths
    images = []
    labels = []
    for path in image_paths:
        img = cv2.imread(path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (48, 48)) # Resize to 48x48x3
        img = img.astype('float32') / 255.0 # Normalize between 0 and 1
        img, label = augment(img)
        if len(img) > 0:
            images.extend(img)
            labels.extend(label)

    print(len(images), len(labels))

    # Convert the images to a TensorFlow dataset
    ds = tf.data.Dataset.from_tensor_slices((images, labels)).shuffle(len(labels)).batch(1)

    return ds


print('Completed')
```

```
Completed
```

```
image_paths = ["/kaggle/input/finished/photo1.jpeg","/kaggle/input/finished/photo2.jpeg","/kaggle/input/finished/photo3.jpeg","/kaggle/ir

train_pd = augment_images(image_paths)
```



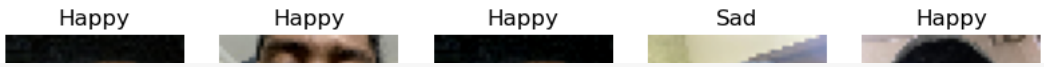

```
# Inspect personal training data
train_pd
```

```
<BatchDataset element_spec=(TensorSpec(shape=(None, 48, 48, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,),
dtype=tf.int32, name=None))>
```

```
# Define class labels
class_names = ['Sad', 'Happy']

# Create a figure with 2 rows and 5 columns
fig, axes = plt.subplots(2, 5, figsize=(10, 5))

# Loop over each sample and plot it
for i, (image, label) in enumerate(train_pd):
    ax = axes[i // 5, i % 5]
    ax.imshow(image[0])
    ax.set_title(class_names[int(label[0])])
    ax.axis('off')

# Display the figure
plt.show()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_27/3088022792.py in <module>
      7 # Loop over each sample and plot it
      8 for i, (image, label) in enumerate(train_pd):
----> 9     ax = axes[i // 5, i % 5]
     10     ax.imshow(image[0])
     11     ax.set_title(class_names[int(label[0])])

IndexError: index 2 is out of bounds for axis 0 with size 2
```

SEARCH STACK OVERFLOW




```
# Create instance of subclassed model
emoGanPD = EmotionGAN(generator, discriminator)

# Setup optimizers and loss functions
g_opt = Adam(learning_rate=0.0001)
d_opt = Adam(learning_rate=0.0002)
g_loss = BinaryCrossentropy()
d_loss = BinaryCrossentropy()

# Compile the model
emoGanPD.compile(g_opt, d_opt, g_loss, d_loss)

# Create directory for the logs emo-Model
logs_dir = 'kaggle/working/logs/emoPD/'
os.makedirs(logs_dir, exist_ok=True)
emo_logs_directory = 'kaggle/working/logs/emoPD'

class CustomCallback(Callback):
    def __init__(self, num_img=3, latent_dim=256):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.uniform((self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
        for i in range(self.num_img):
            img = array_to_img(generated_images[i])
            img.save(os.path.join(emo_logs_directory, f'generated_img_{epoch}_{i}.png'))

print('Completed')
```

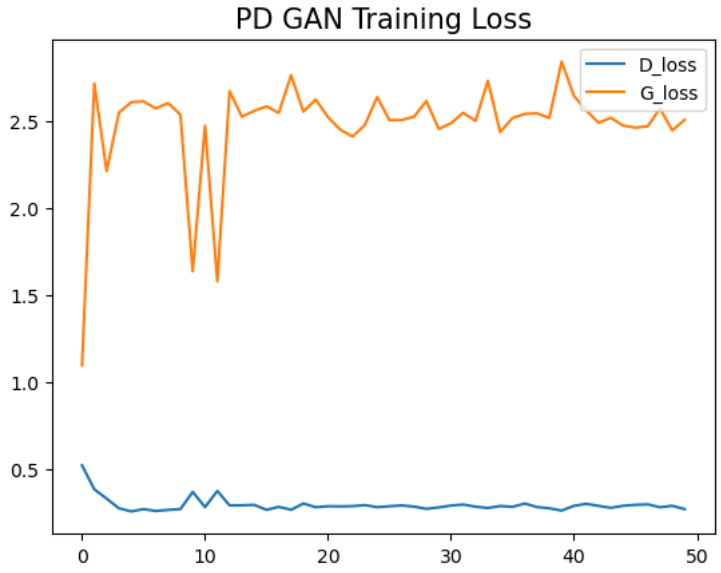
```
Completed
```

```
# Train the model and store the training history
emoPD_hist = emoGanPD.fit(train_pd, epochs=50, callbacks=[CustomCallback()])
```

```
Epoch 1/50
30/30 [==============================] - 189s 6s/step - d_loss: 0.4011 - g_loss: 2.0174
Epoch 2/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.4209 - g_loss: 2.3450
Epoch 3/50
30/30 [==============================] - 180s 6s/step - d_loss: 0.3649 - g_loss: 2.3427
Epoch 4/50
30/30 [==============================] - 185s 6s/step - d_loss: 0.2813 - g_loss: 2.4860
Epoch 5/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2645 - g_loss: 2.6142
Epoch 6/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2663 - g_loss: 2.5897
Epoch 7/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.2655 - g_loss: 2.5993
Epoch 8/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.2664 - g_loss: 2.5867
Epoch 9/50
30/30 [==============================] - 185s 6s/step - d_loss: 0.2659 - g_loss: 2.5994
Epoch 10/50
30/30 [==============================] - 188s 6s/step - d_loss: 0.3212 - g_loss: 2.9450
Epoch 11/50
30/30 [==============================] - 187s 6s/step - d_loss: 0.2784 - g_loss: 2.6918
Epoch 12/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.3363 - g_loss: 2.9064
Epoch 13/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2979 - g_loss: 2.5899
Epoch 14/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2973 - g_loss: 2.4131
Epoch 15/50
30/30 [==============================] - 185s 6s/step - d_loss: 0.2909 - g_loss: 2.4998
Epoch 16/50
```

```
30/30 [==============================] - 186s 6s/step - d_loss: 0.2699 - g_loss: 2.5859
Epoch 17/50
30/30 [==============================] - 185s 6s/step - d_loss: 0.2688 - g_loss: 2.5878
Epoch 18/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2647 - g_loss: 2.6524
Epoch 19/50
30/30 [==============================] - 187s 6s/step - d_loss: 0.2873 - g_loss: 2.8704
Epoch 20/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2833 - g_loss: 2.5527
Epoch 21/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.2897 - g_loss: 2.5491
Epoch 22/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.2838 - g_loss: 2.5196
Epoch 23/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2831 - g_loss: 2.5021
Epoch 24/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2836 - g_loss: 2.5127
Epoch 25/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.2829 - g_loss: 2.5209
Epoch 26/50
30/30 [==============================] - 185s 6s/step - d_loss: 0.2822 - g_loss: 2.5301
Epoch 27/50
30/30 [==============================] - 184s 6s/step - d_loss: 0.2827 - g_loss: 2.5165
Epoch 28/50
30/30 [==============================] - 186s 6s/step - d_loss: 0.2839 - g_loss: 2.5167
Epoch 29/50
```

```
# Plotting the PD learning history
plt.plot(emoPD_hist.history['d_loss'], label='D_loss')
plt.plot(emoPD_hist.history['g_loss'], label='G_loss')
plt.title('PD GAN Training Loss', fontsize=15)
plt.legend(loc="upper right")

plt.show()
```




```
imgs = emoGanPD.generator.predict(tf.random.normal((16, 256, 1)))

fig, ax = plt.subplots(ncols=4, nrows=4, figsize=(10,10))
for r in range(4):
    for c in range(4):
        ax[r][c].imshow(imgs[(r+1)*(c+1)-1])
```

```
1/1 [==============================] - 0s 209ms/step
```