Machine Learning Pipeline 2

What to Expect

1. Introduction
2. Data Acquisition and Converting into Python
3. Data Cleaning, Pre-processing, and Feature Engineering
4. Task Discussion
5. Model Selection
6. Training the Models
7. Predictions and Performance
8. Discussion of Performance
9. Summary
10. References
11. Appendix

1. Introduction

> Welcome to this follow-up tutorial on binary image classification using custom Convolutional Neural Networks (CNNs), pretrained models and XGBoost. In the previous paper, we explored the process of building a custom CNN model from scratch to classify images as happy or sad. While our custom-built CNN model achieved good results, we want to investigate how it can be mutated to use XGBoost and whether a pre-trained CNN model using transfer learning could perform even better.
>
> Our motivation for this paper is rooted in the practical applications of image classification. In many real-world scenarios, there are often constraints on computational resources and data availability that can limit the effectiveness of building custom models from scratch. Pre-trained models using transfer learning offer a promising alternative by leveraging existing knowledge from large datasets to improve classification accuracy on smaller datasets. Our goal is to provide readers with insights into the decision making that surrounds picking a model for a particular task. Do you want to build it yourself? Or go the pretrained route? Is it better to use a simple model or slap XGBoost onto it? Basically, how you can effectively choose and improve models for your own binary image classification tasks.
>
> Throughout the paper, we will provide detailed explanations and technical details to help readers understand the key concepts and techniques used. We will ensure side by side comparison of the models from data collection, preprocessing, model selection, training, evaluation, and testing. By the end of this tutorial, readers will have a comprehensive understanding of the process of binary image classification using custom-built and pre-trained models, as well as how to enhance their models with XGBoost, and the comparative advantages of these decisions.

1.b. Why Does It Matter to Me?

> This tutorial holds a special place in my heart because it continues to address a problem that I have personally faced - the challenge of accurately detecting emotions in my own photos - was I actually happy or just posing for a photo? As someone who loves taking photos, I often find myself wondering what my pictures say about my emotions and experiences. My dad always tells me that I look sad in the photos I take, but I don't always see it myself.
>
> By learning about binary image classification and building models that can automatically classify my photos as happy or sad, I can gain a deeper understanding of my own emotions and how they are reflected in my photography. Maybe I'll discover that my dad is right, or maybe I'll find that my photos actually capture moments of joy and happiness that I hadn't even noticed before. Regardless, this process of self-discovery is what makes the topic of binary image classification so fascinating to me, and I hope that this paper will inspire others to explore it as well.
>
> Not only does this paper provide insights into the process of binary image classification, but it also offers a way for individuals to gain a deeper understanding of their emotions and experiences through analyzing their own photos. By learning how to effectively choose and improve custom and pre-trained models for binary image classification tasks, readers can apply these techniques to their personal photos and gain valuable insights into their own emotional sentiment. It might even be fun to see what different models classify the same photos as!

1.c. Dependencies

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

In [255…

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, Lambda, Input
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 sklearn.metrics import f1_score, confusion_matrix
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.utils import plot_model
from sklearn.decomposition import PCA
```

```
from tensorflow.keras import layers
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
import tensorflow as tf
from PIL import Image
import seaborn as sns
import pandas as pd
import visualkeras
import numpy as np
import zipfile
import pprint
import imghdr
import json
import cv2
import os
```

2. Data Acquisition and Converting into Python

> In this section, we will focus on collecting image data for both models: the custom-built model from our previous paper and the pre-trained model we will be using in this tutorial. Part A will recap the data acquisition and preprocessing steps we used for the custom-built model. In Part B, we will introduce a new, more robust dataset that we will use for both models moving forward. Then Part C will cover the loading of the data into python accessible formats.

2.a. Custom-Model Data Acquisition Recap

> I used online image data since my personal dataset was insufficient to train the model accurately. Additionally, its lack of diversity would limit the robustness of my model.
>
> Using the Chrome extension "Download All", I was able to scrape happy and sad images from the web using the keyword searches "happy real people" and "sad real people". The use of the keyword "real" was intended to filter out cartoon or artificially generated images, and to focus on real-life images of people expressing emotions.
>
> Once the images were loaded on the web page, the "Download All" extension was used to download all the images on the page as a zip file. This process was repeated for both the happy and sad datasets. I then extracted and sorted into two separate directories for happy and sad images.
>
> I then removed irrelevant images and those with incorrect file extensions. With the cleaned and curated data, I moved on to the next step of the image classification pipeline, which was data loading. This time round, we will be using a different, more robust dataset as discussed below.

2.b. Pretrained-Model Data Acquisition

> In this project, we are upgrading our dataset to improve the accuracy of our model. While our initial personal data was insufficient, we had relied on a Google search curated dataset to train our model. However, for 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, which is precisely what we need to train our model. Additionally, the FER2013 dataset is specifically designed for facial expression recognition, providing a wide range of facial expressions for our model to learn from, resulting in better performance on our classification task. In contrast, the Google search dataset may have images of varying quality, lighting, and facial expressions, making it difficult for our model to learn the patterns necessary to accurately classify happy and sad expressions. If you can remember one of the challenges was the images containing watermarks that acted as noise in the data.
>
> Therefore, using a dataset that is specifically designed for our task, like FER2013, is crucial in achieving the best possible performance for our model

2.c. Data Loading

> We will load the FER2013 dataset into Python using the Keras library, which is a high-level neural networks API written in Python and capable of running on top of TensorFlow.
>
> First, you need to make sure that you have the kaggle package installed in your Anaconda environment. You can install it by running !pip install kaggle in a Jupyter Notebook cell.
>
> Next, you need to obtain your Kaggle API username and key. You can do this by logging in to Kaggle, navigating to the 'Account' tab in your profile settings, and clicking the 'Create New API Token' button. This will download a kaggle.json file containing your API credentials. Make sure you save the kaggle.json file in the same directory as this Jupyter Notebook. Then proceed with this cell below where we extract the data from the zipfile.

In [3]:
```
# Install kaggle flag > null silences if installed
!pip install kaggle > nul

# Get kaggle credentials
```

```
with open('kaggle.json') as f:
    kaggle_api_key = json.load(f)

os.environ['KAGGLE_USERNAME'] = kaggle_api_key['username']
os.environ['KAGGLE_KEY'] = kaggle_api_key['key']

# Download dataset FER2013
!kaggle datasets download -d msambare/fer2013 > nul

# Extract FER2013 dataset from downloaded ZIP file
with zipfile.ZipFile('fer2013.zip', 'r') as zip_ref:
    zip_ref.extractall('fer2013')
```

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

> In this section, we will focus on cleaning and preprocessing the FER2013 image data that will be used for both models. We will clean the load the data by removing the other classes we will not use and then do some preprocessing then do some exploratory data analysis to see if we need to do feature engineering.

3.a. Cleaning the data

> To prepare the FER2013 dataset for binary image classification into "happy" and "sad" categories, we will clean and filter the data to get rid of the unused 5 other emotion classes.
>
> Firstly, we will load all classes from the training and test datasets using the TensorFlow Keras API. We will then extract the images and labels for the "happy" and "sad" classes from both datasets using filtering techniques.
>
> Then, we will convert the integer labels of "happy" and "sad" classes into binary labels of 1 and 0, respectively. This will be done to standardize the labels and facilitate the classification process.
>
> The final step will involve confirming the number of images and labels for each class in both datasets to ensure we are on the right track.

In [4]:
```
# Define dataset directory and image size
data_dir = "fer2013"
img_size = (48, 48)
batch_size = 32

# 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.
```

Why should we split data?

- Splitting data into training, validation, and test sets is a critical step in the development of any machine learning model. Firstly, by randomly splitting data into the different sets, we prevent any selection bias.
- Also, it helps us to ensure that we are not making assumptions about the data that are not true, and that we are not overfitting to the training data.
- Overfitting occurs when a model becomes too good at the training data, that it starts to perform poorly on new data, i.e, it captures all the noise in the data. Through splitting, we can ensure that the model we build is robust, generalizes well, and can be applied to real-world data, (Baheti, 2023).

In [5]:
```
# 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']
```

In [6]:
```
# 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)
```

In [7]:
```
# 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
```

In [8]:
```
# Convert integer labels to binary labels
train_labels[train_labels == sad_class] = 0
train_labels[train_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: {l
print(f"Test:\n\nClass names:{np.unique(test_labels)}, Images: {len(test_images)}, Happy: {len(np.where(test_labels == 1)[0])}, Sad: {len(n
```

```
Train:

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


Test:

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

3.b. Pre-processing the data

> To prepare the data, first we will define the class indices for "happy" and "sad" and filter the images and labels for these classes from the train and test datasets. Next, we will create balanced train and test datasets with an equal number of happy and sad samples. For this purpose, we will randomly select samples from the filtered train and test datasets, and use TensorFlow datasets to handle the data easily while also shuffling the datasets. To confirm that we have balanced classes, we will count the number of samples for each class in the train and test datasets.
>
> After creating the balanced datasets, we will inspect the training and test data by taking a small subset of the train dataset and checking the shape and labels of the images. This exploration will give us an idea of the size and composition of the dataset, and will help us determine if further preprocessing is necessary. If needed, we can apply data augmentation techniques such as rotation, flipping, and scaling to preprocess the images. These techniques will help prevent overfitting and improve the model's ability to generalize to new images. 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.

In [9]:
```python
# 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]
```

Why do we shuffle the data?

- We shuffle the data to avoid any patterns or biases that may exist in the dataset. If the data is not shuffled, the model may learn to classify images based on their order rather than their content, which would result in poor performance on new data.
- By shuffling the data, we ensure that the order of the images does not affect the model's ability to learn and generalize.

In [10]:
```python
# 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(bat
test_ds = tf.data.Dataset.from_tensor_slices((test_images_balanced, test_labels_balanced)).shuffle(len(test_labels_balanced)).batch(batch_s
```

Why do we create TensorFlow datasets?

- TensorFlow dataset is a powerful API that simplifies the data loading process, making it easier to manage large datasets.
- It can handle preprocessing tasks such as batching, shuffling, and caching the data, which can help to optimize memory usage and improve the performance of the model.

In [11]:
```python
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
```

In [12]:
```python
# 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))
```

```
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
```

In [13]:
```python
# 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))
```

```
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
(32, 48, 48, 3) (32,) [0. 1.]
```

Why do we normalize pixel values?

1. Better training performance: Normalizing the data helps in better training performance by improving convergence speed and avoiding vanishing or exploding gradients. This is because normalization scales the data to a range that is more suitable for the learning algorithm.
2. Reduced model complexity: Normalization also helps to reduce the complexity of the model by reducing the range of possible values that the model has to learn. This can lead to better generalization and reduced overfitting.
3. Better feature extraction: Normalizing the data can also help in better feature extraction by ensuring that different features contribute equally to the final output.

In [14]:
```python
# 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))
```

3.c. Exploratory Data Analysis

> In this section, we focus on preliminary EDA in order to gain an initial understanding of the dataset and its underlying structure, characteristics, and patterns. Simply, it will allow us to identify potential issues that might affect our model later such as missing values, outliers, or skewness. Because we are dealing with image data we can focus on specific EDA methods.

What EDA methods are these?

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.

3.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.

In [15]:
```python
# 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()
```

3.c.ii. Descriptive Statistics

> The mean pixel value of 0.4893 indicates 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.

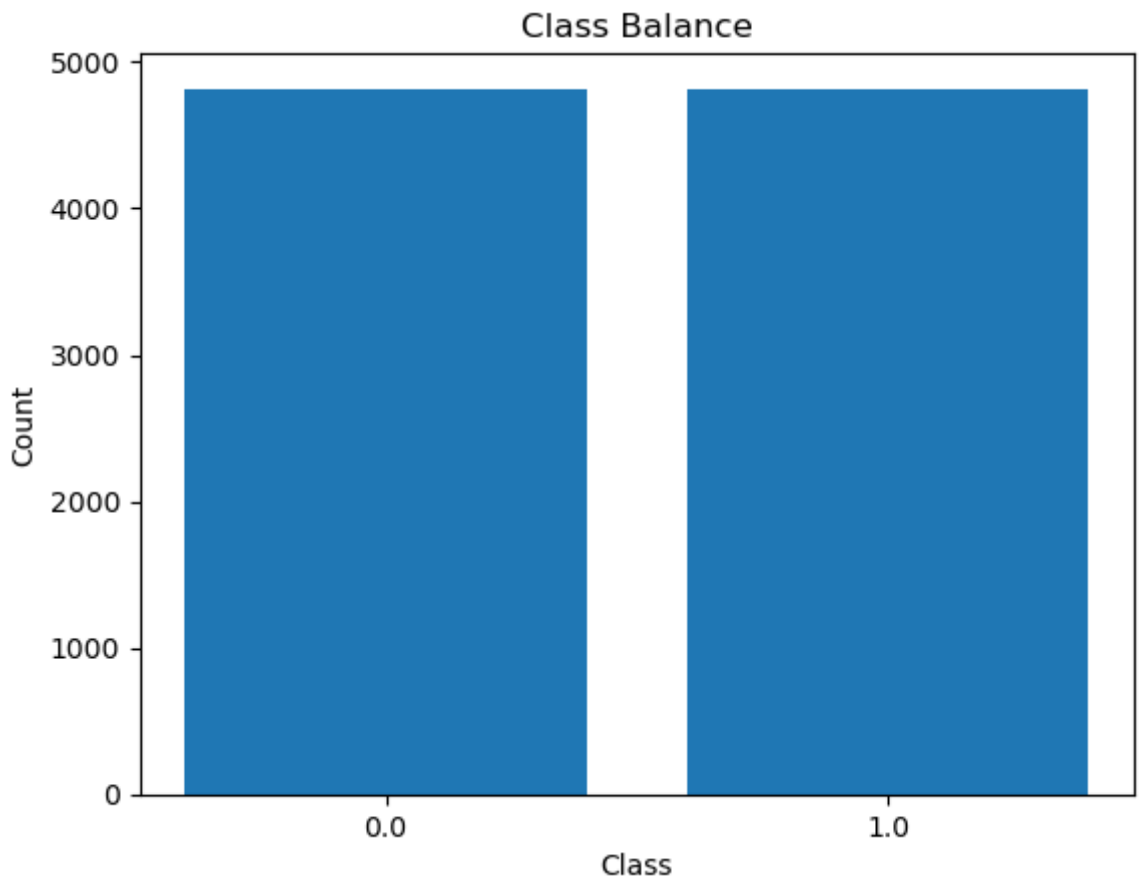
In [16]:
```
# Compute mean and standard deviation of pixel values in train_ds
pixel_means = []
pixel_stds = []

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: 0.4893646836280823
Standard deviation of pixel values: 0.2529911398887634
```

3.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.

In [17]:
```
# 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]}")
```




```
Even balance between the classes. Happy (0): 4818, Sad (1): 4818
```

3.c.iv. Checking image sizes and dimensions

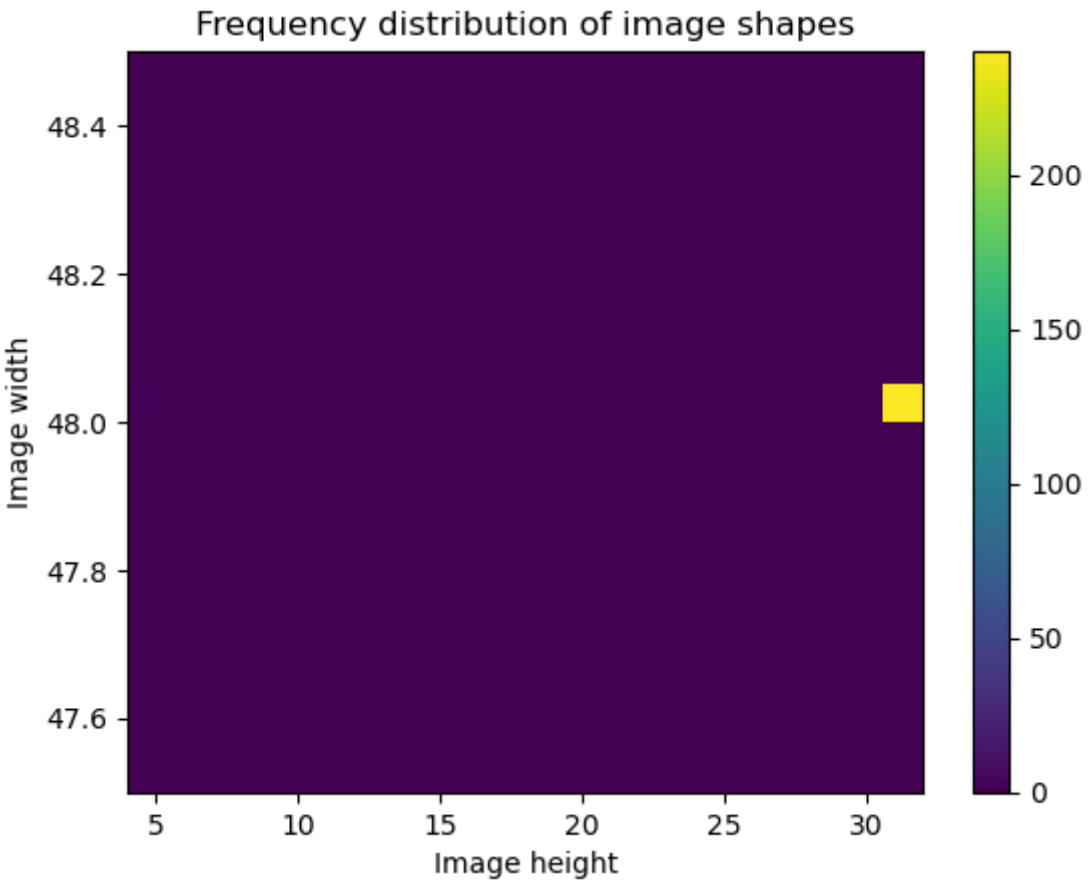
> The summary statistics of the image shapes show the range of sizes and dimensions in the dataset. The minimum shape of [4 48 48 3] indicates that the smallest image in the dataset has a height and width of 48 pixels, and a depth of 3 color channels. The maximum shape of [32 48 48 3] indicates that 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. Overall, these statistics indicate that the images in the dataset are fairly consistent in terms of size and dimensions, which is important for model training.

In [141…

```python
# 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)
```

In [142…

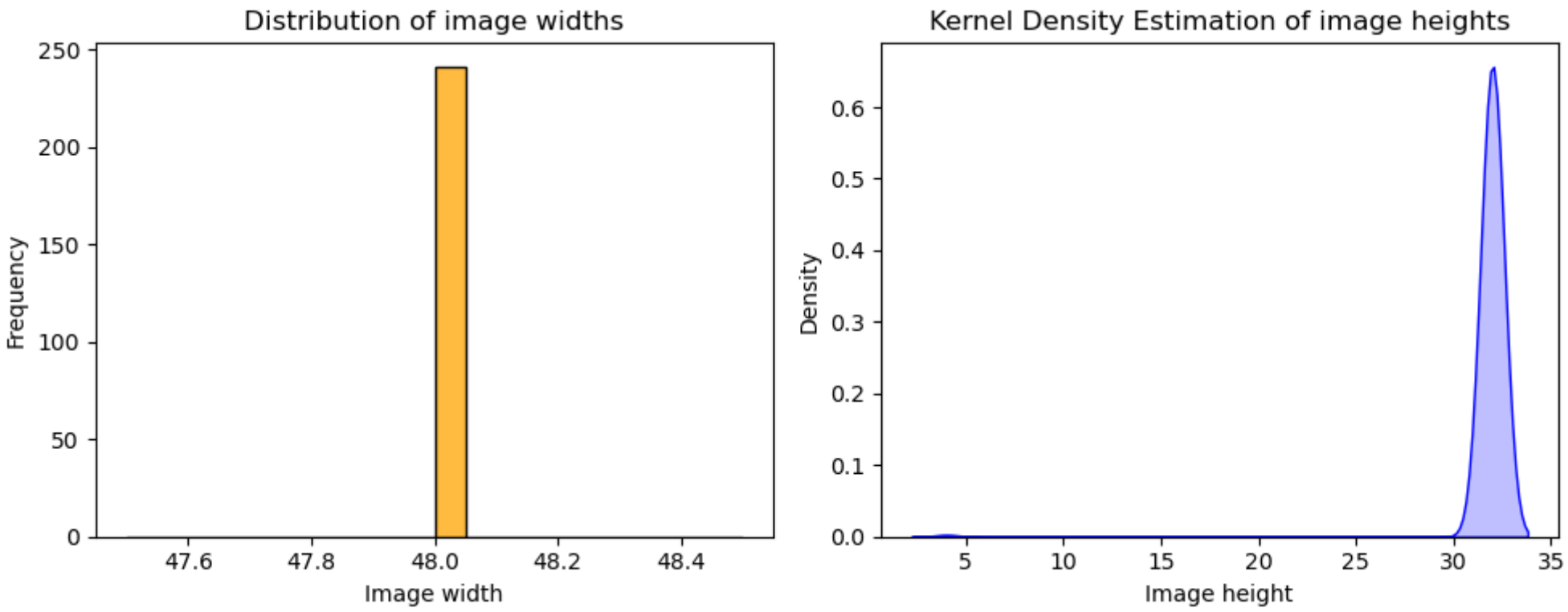
```python
# Plot histogram of image shapes
plt.hist2d(image_shapes[:,0], image_shapes[:,1], bins=20)
plt.xlabel('Image height')
plt.ylabel('Image width')
plt.title('Frequency distribution of image shapes')
plt.colorbar()
plt.show()

# 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: [ 4 48 48  3]
Maximum Shape: [32 48 48  3]
Mean Shape: [31.88381743 48.         48.          3.        ]
Median Shape: [32. 48. 48.  3.]
Standard Deviation of Shape: [1.79989268 0.         0.         0.        ]
```

In [149…

```python
# 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 de

print(caption)
```



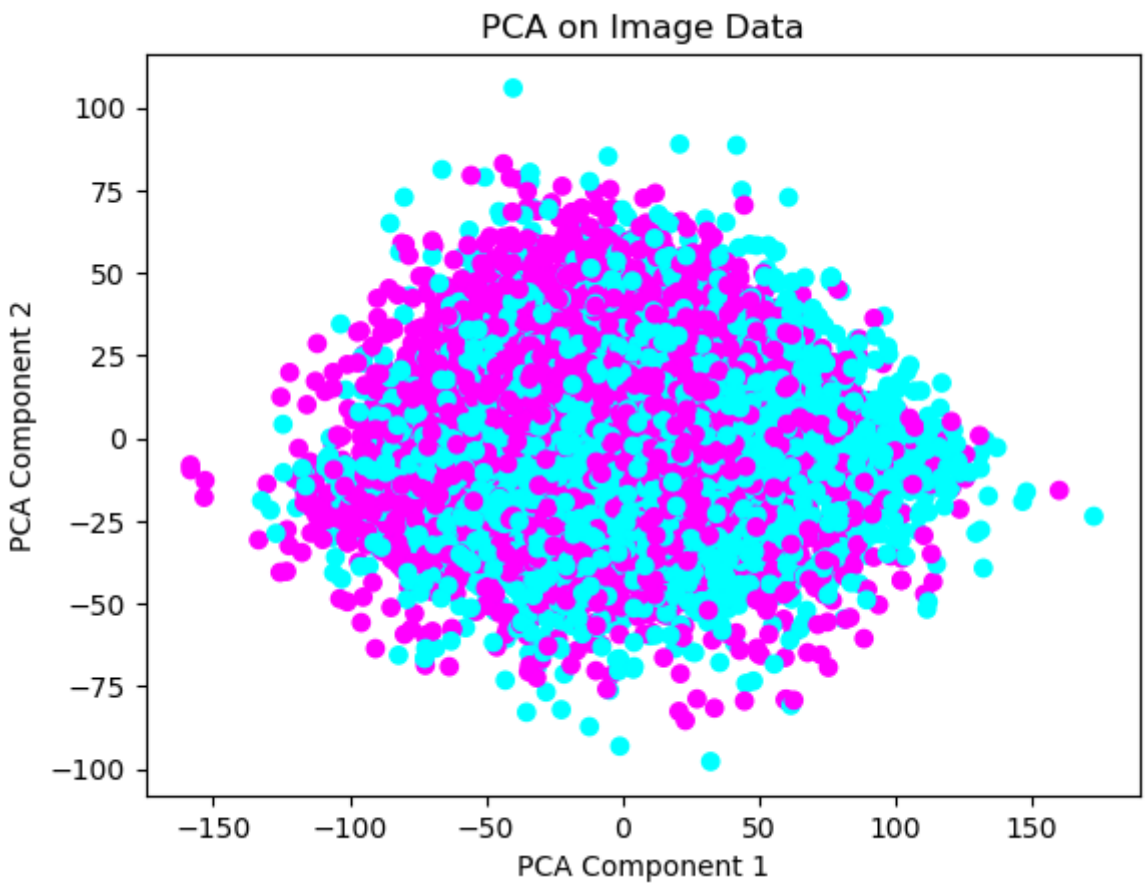

```
The distribution of image widths is plotted using a histogram while the distribution of image heights is plotted using a kernel density est
imate (KDE). The histogram is appropriate for visualizing the frequency distributions, more so in this case where all image widths have a v
alue of 48 while the KDE is appropriate for visualizing the density of the image heights.
```

3.c.v. Conducting PCA Analysis for EDA

> PCA is a dimensionality reduction technique that finds the most important features in the data by transforming the data into a new coordinate system in such a way that the axes are aligned with the principal components. In the case of image data, each pixel is treated as a feature, so PCA can be used to identify the most important pixels that contribute to the variation in the dataset.
>
> To conduct PCA on the image dataset, we will first flatten each image into a one-dimensional vector, then standardize the data and perform PCA on the resulting matrix. We use StandardScaler to standardize the data because PCA is sensitive to the scale of the input features. Scaling

> the input data to have zero mean and unit variance is an important preprocessing step for PCA to work well. This is because PCA aims to identify the directions of maximum variance in the data, and if the features have different scales, then those with larger scales may dominate the analysis. Standardization ensures that all features have the same scale, making the PCA analysis more reliable and meaningful.
>
> When conducting PCA for EDA, the variance ratio of each component provides insight into how much of the data's variability can be explained by that particular component. In this case, the first component explains about 28% of the total variance in the dataset, while the second component explains about 9% of the total variance. This means that these two components combined explain about 37% of the total variance in the dataset, which is not much. This information can be useful for understanding the main features of the data and potentially reducing the dimensionality of the data for modeling purposes. However, it should be noted that using PCA for EDA alone does not necessarily imply that PCA will be used for modeling. Nevertheless, the PCA results hint to us that in our architecture design of the models we might want to have more layers to extract these features.

In [19]:
```python
# 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])
```




```
PCA Component 1 Variance Ratio:  0.28161895
PCA Component 2 Variance Ratio:  0.09374834
```

In [20]:
```python
# 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 83.82% of the variance
```

What are the implications of these results on the model design?

1. It suggests that the model may require a larger number of input features or more complex feature engineering to capture the important information in the dataset. This could mean that additional data sources or feature extraction techniques may be needed to improve the model's performance.
2. It may indicate that the model will require a larger sample size to achieve sufficient statistical power to accurately capture the underlying relationships in the data. A larger sample size can help to reduce noise and better estimate the parameters of the model.
3. It may also indicate that the model will require more advanced modeling techniques that are capable of handling high-dimensional data. This could include techniques such as deep learning, ensemble methods, or non-linear modeling approaches.

4. Task Discussion

> In this binary classification task, we aim to classify images as happy or sad faces using deep learning models. We have prepared the data for analysis by tokenizing the images and normalizing them (dividing by 255) to have individual pixel values down to the range [0,1]. Through exploratory data analysis (EDA), we have gained insights about the image data and conducted principal component analysis (PCA) to understand the main features of the train_ds.
>
> We have two datasets - train_ds and test_ds - that we will use to train and evaluate our models, respectively. The train_ds dataset contains 9636 images, out of which 4818 are happy faces and 4818 are sad faces. The test_ds dataset contains 604 images, with 302 happy faces and 302 sad faces. To split our data into training and validation sets, we shuffle train_ds and portion 20% of it out to generate a validation_ds dataset, then the remaining 80% will form the new train_ds dataset.
>
> The training set will be used to retrain the convolutional neural network (CNN) model from the previous paper, a.k.a, custom-Model. During training, the model learns to recognize patterns in the data that are indicative of happy or sad images. The model then uses the validation set to assess how well it is performing, and we adjust the model's hyperparameters to improve its performance. This is an iterative process that is typically repeated until we obtain the best possible model. For the pretrain-Model, we will use VGG16. Once both models are validated, we will test and compare their performances. Then, we will apply XGBoost to custom model and tune the pretrained-model and compare the performances of the models.
>
> To evaluate the performance of our models, we will use various metrics such as accuracy, precision, recall, and F1 score. These metrics provide insight into how well a model is performing on the classification task. A score of 1 indicates that the model performed perfectly, while a score of 0 indicates that the model performed poorly. By assessing the model's performance, we can determine if any adjustments need to be made to improve the accuracy of the model's predictions.

In [21]:
```
# Determine the size of the dataset
num_examples = tf.data.experimental.cardinality(train_ds).numpy()

# Split the dataset into train and validation sets
train_size = int(0.8 * num_examples)
validation_size = num_examples - train_size

# Create a copy of the original train_ds dataset
old_train_ds = train_ds

# Split the dataset into train and validation sets
train_ds = old_train_ds.take(train_size)
validation_ds = old_train_ds.skip(train_size)
```

In [22]:
```
num_train_examples = tf.data.experimental.cardinality(train_ds).numpy()
num_validation_examples = tf.data.experimental.cardinality(validation_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()])

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

# Get the totals
all_train = train_counts[0]+train_counts[1]
all_val = validation_counts[0] + validation_counts[1]
all_imgs = all_train + all_val

# Calculate the percentages
pt = (all_train/all_imgs) * 100
pv = (all_val/all_imgs) * 100

# Confirm we have balanced classes 0-sad 1-happy
print(f"\nTrain {pt:.2f}%:\n\n\tImages: {train_counts[0]+train_counts[1]}, Happy: {train_counts[0]}, Sad: {train_counts[1]}\n")
print(f"Validation {pv:.2f}%:\n\n\tImages: {validation_counts[0] + validation_counts[1]}, Happy: {validation_counts[0]}, Sad: {validation_c
```

```
Train 79.80%:

        Images: 7712.0, Happy: 3859.0, Sad: 3853.0

Validation 20.20%:

        Images: 1952.0, Happy: 973.0, Sad: 979.0
```

5. Model Selection

> For the models, I decided to use Convolutional Neural Networks (CNNs) as they can be used for binary classification tasks, such as identifying happy or sad sentiments in image data. The model architecture for the custom-Model is sequential while the pretrained-Model is VGG16.

Recap on How CNNs work

- In a convolutional layer of a CNN, each neuron applies a mathematical operation called convolution to a small patch of the input data, using a set of learnable weights, i.e, a kernel of weights is multiplied against the input vector space in a way that neighboring information is condensed into one output. The output of the convolution is then passed through a non-linear activation function, such as the Rectified Linear Unit (ReLU) or sigmoid, which introduces non-linearity into the network and helps to make it more expressive.
- The resulting feature maps are then downsampled using a pooling operation, which reduces their spatial dimensions while retaining their important features. This downsampling helps to make the network more efficient by reducing the number of parameters and computation required while retaining their important features.
- By using convolution and pooling operations, CNNs can learn to extract hierarchical representations of the input data, from low-level features like edges and corners to high-level features like objects and scenes. The weights that the network learns for each neuron are shared across all the neurons that connect to the same local receptive field, which reduces the number of learnable parameters and helps to prevent overfitting. This weight sharing also allows the network to efficiently recognize patterns and features in the input data that are present in multiple locations, (LeCun, Bengio, & Hinton, 2015).

Recap on How CNNs differ from NNs

- CNNs are a type of neural network that are designed to process data with a grid-like structure, such as images, videos, or audio signals. They are just neural networks where the weights are "shared" between features.
- The "shared weights" in CNNs refer to the way that the network's parameters are learned and applied to different regions of the input data. In a traditional neural network, each neuron in one layer is connected to every neuron in the previous layer. However, in a convolutional layer of a CNN, each neuron is connected to only a small region of the input data, typically a square-shaped patch of pixels which we can stride along, (Shafkat, 2018).

5.a. Custom-Model

> In this model, we begin with a convolutional layer with 16 filters, a (3,3) window with a stride of 1. This is followed by a MaxPooling2D layer. Then there is another convolutional layer with 32 filters, followed by another MaxPooling2D layer. Then there is another convolutional layer with 16 filters, followed by another MaxPooling2D layer. The output is then flattened and passed through two dense layers with 256 and 1 units, respectively, with ReLU activation and sigmoid. The latter outputs the probability distribution over the classes.

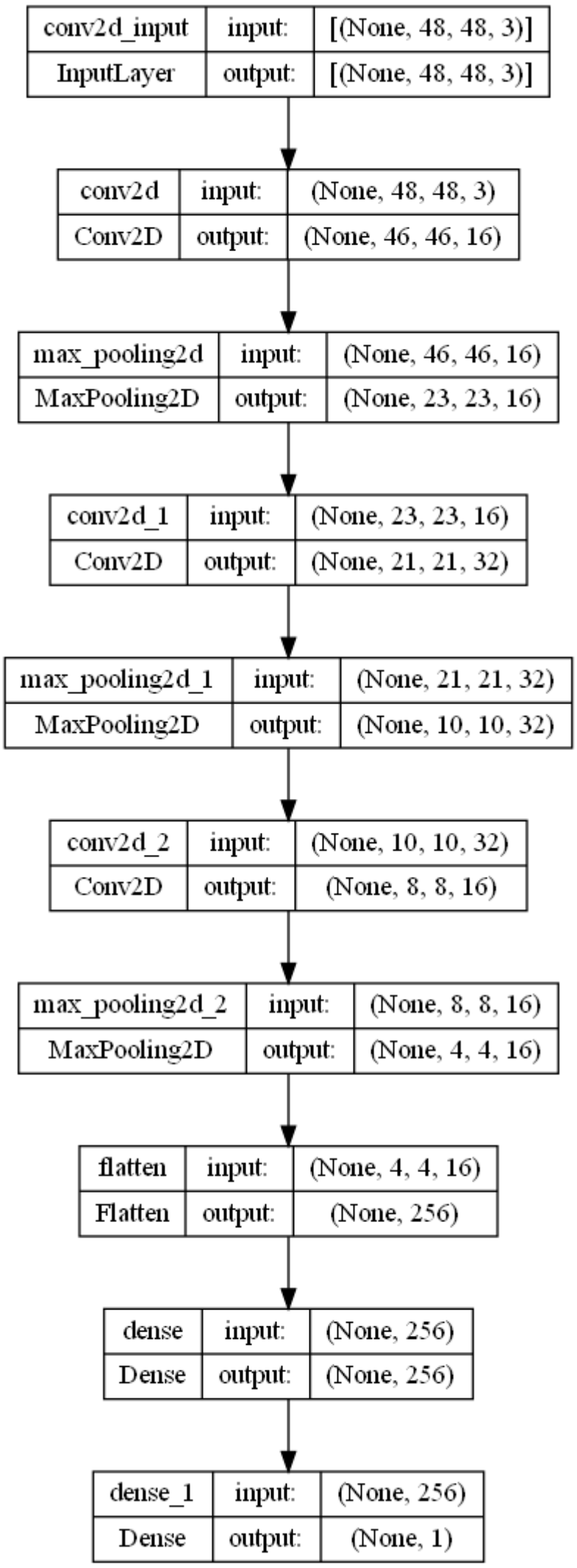
In [23]:
```
# Define the model from the previous paper
custom_model = Sequential([
    # Add convolutional layers with 16 filters
    Conv2D(16, (3,3), 1, activation='relu', input_shape=(48, 48, 3)),
    MaxPooling2D(),

    # Add convolutional layers with 32 filters
    Conv2D(32, (3,3), 1, activation='relu'),
    MaxPooling2D(),

    # Add convolutional layers with 16 filters
    Conv2D(16, (3,3), 1, activation='relu'),
    MaxPooling2D(),

    # Flatten the output of the previous layers
    Flatten(),

    # Add a dense layers with 48 units and relu activation
    Dense(256, activation='relu'),

    # Add a final dense layer with 1 units and sigmoid activation
    Dense(1, activation='sigmoid')
])
```

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.

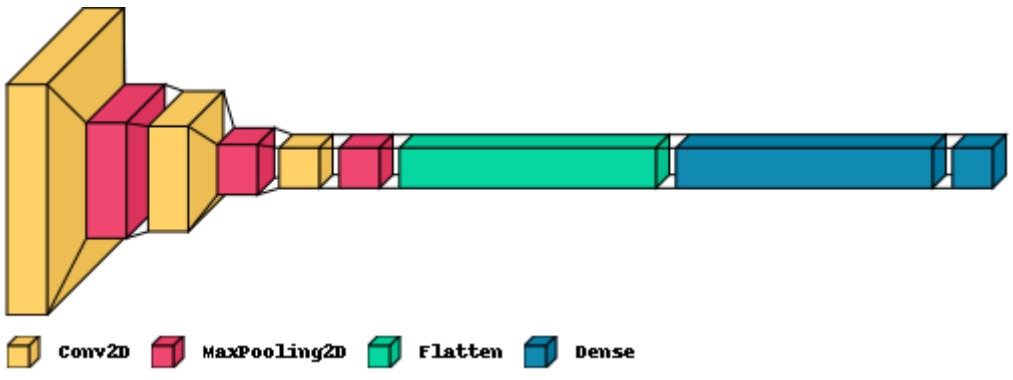
In [24]:
```
# plot the custon-Model as a graph
plot_model(custom_model, show_shapes=True, show_layer_names=True, expand_nested=True)
```

Out[24]:



In [25]:
```
### Visualize the architecture of custom-Model
visualkeras.layered_view(custom_model, legend=True, scale_z=0.5, scale_xy=2.5)
```

Out[25]:



Recap on custom-Model Architecture

Note: While this is a recap, the VGG16 model uses similar layers, and hence this section is useful for understanding the pretrained model.

1. Convolutional layer (yellow):
    - This layer is particularly effective at learning spatial features in image data, which we discussed above under 'how do CNNs work?'. The convolution operation is a mathematical operation that combines the input data with a set of learnable filters, also known as kernels, to extract features that are useful for the downstream tasks, such as image classification.

- In a convolutional layer, the filters slide over the input data in a sliding window fashion, and the dot product between the filter and the corresponding input data is computed at each position. The result of the convolution operation is a set of feature maps, which capture the presence and distribution of different patterns or structures in the input data. Here, we have used a (3,3) window with a stride of 1.
- If our data had increasingly complex features, we would use multiple convolutional layers with increasing numbers of filters to allow the model to learn those increasingly complex features.
- With this understanding, we can discuss the math behind the convolution layer. Where input is the input feature map, output is the output feature map, activation_fn is an activation function such as ReLU or sigmoid, and bias is a scalar bias term. The sum is taken over the filter size and all input feature map channels. The convolution operation applies a set of filters to the input feature map, sliding the filters over the input feature map and computing a dot product at each position to generate a set of activation maps. The activation maps are then passed through an activation function and a bias term is added before being output as the next layer's feature map, (Goodfellow et al., 2016, p. 328).

$$\text{output}[i, j, k] = \text{activation\_fn}(\text{sum}(\text{input}[i : i + F, j : j + F, :] * \text{filter}[:, :, :, k]) + \text{bias}[k])$$

2. Maximum Pooling layers (red):
   - The basic idea of max pooling is to reduce the size of the feature maps generated by the convolutional layers, while retaining the most important information in the data.
   - During max pooling, the feature maps are divided into small sub-regions or "windows," typically with a size of 2x2 or 3x3 pixels. Within each window, the maximum value is selected and passed on to the next layer, while the other values are discarded. This has the effect of reducing the size of the feature maps by a factor of 2 or 3, depending on the size of the window. The size of the sliding window is determined by a hyperparameter, and the window slides over the feature map in non-overlapping steps.
   - At each position of the sliding window, the max pooling layer selects the maximum value within the window as the output. The output of the max pooling layer is a downsampled feature map that retains the most salient features of the input data.
   - Max pooling helps to prevent overfitting and improve the computational efficiency of the network. Additionally, because the operation only keeps the maximum value in each window, it helps to preserve the most important information in the data, which is crucial for accurate image recognition.
   - The max pooling operation has several parameters, (Scherer, Müller, & Behnke, 2010, p. 95), where input is the input feature map, output is the output feature map, s is the stride (typically equal to p which is window size), and k represents the depth or number of feature maps. The max pooling operation selects the maximum value within each pooling window and places it in the corresponding position in the output feature map

$$\text{output}[i, j, k] = \text{max}(\text{input}[i * s : i * s + p, j * s : j * s + p, k])$$

3. Flatten layer (green):
   - Flatten layers convert a multidimensional input tensor into a single dimension, which is then fed into a subsequent fully connected (Dense) layer. The Flatten layer essentially reshapes the output of the preceding layer into a 1D array, which can be thought of as a long sequence of values.
   - Here, the 2D downsampled data from the pooling layer is converted into a 1D feature vector that can be fed into a subsequent fully connected layer where matrix multiplication between the flattened input and a weight matrix to generate the final output of the model, i.e, the classification.
   - For example, if the output of the previous layer is a 2D matrix with dimensions (batch_size, 16 x 16), the Flatten layer will reshape it into a 1D vector with dimensions (batch_size, 16 x 16 = 256).

4. Dense layers (blue):
   - These layers are particularly effective at learning non-linear relationships in the data, which can help the model to make more accurate predictions. In a dense layer, the flattened output of the previous layers is connected to a set of densely connected neurons. Each neuron in this layer receives input from every neuron in the previous layer, and their outputs are used as input for every neuron in the next dense layer.
   - Here, the first Dense layer performs a linear operation on the input, followed by a non-linear activation function. This dense layer has 256 units and adds a Rectified Linear Unit (ReLU) activation function. The ReLU activation function applies the mathematical function f(x) = max(0,x) to each neuron's output, effectively setting all negative values to 0. The number of units in the Dense layer is a hyperparameter that can be tuned to optimize the performance of the model. Increasing the number of units can increase the model's capacity to learn complex relationships between the input and output, but may also increase the risk of overfitting to the training data.
   - In the final dense layer with a single output unit, a sigmoid activation function is added. The sigmoid function transforms the input into a value between 0 and 1, which can be interpreted as a probability. In the context of our binary classification problem, this output of the model represents the probability of the input belonging to a certain class, i.e, either happy (< 0.5) or sad (> 0.5).

In [26]:
```
# Compile the custom-model
custom_model.compile(optimizer='adam', loss=BinaryCrossentropy(), metrics=['accuracy'])
```

Recap on the Compile Method

1. Optimizer: This is the optimization algorithm used to train the model. In this case, the optimizer is set to 'adam', which is a popular stochastic gradient descent optimization algorithm that is efficient and requires little memory.

2. Loss: This is the loss function that the model will use to compute the error between the predicted and actual output. BinaryCrossentropy() is a type of cross-entropy loss function that is commonly used for binary classification problems because it can be optimized efficiently using gradient descent methods. It computes the cross-entropy loss between the true labels and the predicted probabilities. The categorical_crossentropy() loss function would be a good choice for multi-class classification tasks, where the goal is to classify input data into one of several classes.

3. Metrics: This is a list of metrics that will be used to evaluate the performance of the model during training and testing. In this case, the metric is set to ['accuracy'], which means that the model's accuracy will be monitored during training and testing. Others metrics include precision, recal, f1score and

AUC.

How does binary cross-entropy work?

Binary cross-entropy loss, also known as log loss, is commonly used for binary classification tasks. Given a binary classification problem with N examples, where each example has a true label $y_i \in 0, 1$ and a predicted probability $\hat{y}_i$, the binary cross-entropy loss is defined as:

$$L_{BCE} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{1}$$

where:

- L_{BCE} is the binary cross-entropy loss
- N is the total number of samples in the dataset
- y_i is the true label (either 0 or 1) for the i-th sample
- $\hat{y}_i$ is the predicted probability of the positive class for the i-th sample.

Intuitively, the binary cross-entropy loss measures the difference between the true label and predicted probability for each example, and averages this difference over all examples. The loss is minimized when the predicted probabilities match the true labels. It is derived from the negative log-likelihood of a Bernoulli distribution, where the true label y_i is modeled as a random variable with a Bernoulli distribution parameterized by the predicted probability $\hat{y}_i$. The log-likelihood of the Bernoulli distribution is:

$$\log p(y_i|\hat{y}_i) = y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \tag{2}$$

where:

- y_i is the true label (either 0 or 1) for the i-th sample
- $\hat{y}_i$ is the predicted probability of the positive class for the i-th sample.

Taking the negative log of the likelihood gives us the binary cross-entropy loss.

In [193…

```
# Summary of the custom-model
custom_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 46, 46, 16)        448

 max_pooling2d (MaxPooling2D  (None, 23, 23, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 21, 21, 32)        4640

 max_pooling2d_1 (MaxPooling  (None, 10, 10, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 8, 8, 16)          4624

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 16)         0
 2D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 256)               65792

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 75,761
Trainable params: 75,761
Non-trainable params: 0
_________________________________________________________________
```

5.b Pretrained-Model

> Since building an accurate model requires a diverse and representative dataset, we will be using a pre-existing dataset of labeled images rather than collecting images ourselves. The VGG16 convolutional neural network architecture has been pre-trained on the ImageNet dataset. This pre-training process involves training the model on a large dataset of images to learn general features such as edges and shapes, which can then be fine-tuned for specific tasks such as binary classification. We will use the splitted prepped dataset into training and validation sets. The training set will be used to train the model, while the validation set will be used to evaluate the model's performance and prevent overfitting.

What are pre-trained models?

- Pre-trained models are machine learning models that have been trained on large datasets by experts and made available for general use. These models have learned to recognize patterns and make predictions based on the data they were trained on.
- Pre-trained models can be very useful for developers and researchers because they save a lot of time and resources that would otherwise be required to train a model from scratch. They can be fine-tuned on specific tasks or used as a starting point for building custom models.

What are some pre-trained models for image classification?

There are many models that could help us with this project, but the choice of model is determined by the nuances of the project. Generally, pre-trained models are available in various sizes, with the larger models having more layers and higher accuracy but also requiring more computational resources. For example, there are:

- **VGG**- developed by the Visual Geometry Group at the University of Oxford. The VGG models are characterized by their deep architecture and use of small 3x3 filters. VGG16 is a popular choice for image classification tasks and has shown strong performance on the ImageNet dataset. It has a simple architecture and is easy to fine-tune on smaller datasets.
- **ResNet (Residual Network)** - developed by Microsoft Research. The ResNet models are characterized by their use of residual connections, which allow the model to learn residual mappings between layers. ResNet50 is a popular CNN model that has shown strong performance on a range of image classification tasks, including object detection and recognition. It has a deeper architecture than VGG16 and is known for its ability to prevent overfitting.
- **Inception** - developed by Google. The Inception models are characterized by their use of multiple filter sizes and pooling operations in parallel to extract features at different scales. InceptionV3 is a recent CNN model that has shown strong performance on image classification tasks and has a unique architecture that uses multiple branches to process different levels of image information.

Why VGG16?

- When it comes to image classification tasks, the VGG16 model is a reliable and effective choice. One reason for this is its relatively simple architecture, which consists of a stack of convolutional layers with small 3x3 filters, followed by pooling layers and three fully connected layers. This simplicity makes it easier to understand and interpret the model's behavior, which is important for gaining insights into the image classification process. Pre-trained weights for the VGG16 model are readily available, which can save time and computational resources during the fine-tuning process.
- Furthermore, the model's stellar performance on the ImageNet dataset suggests that it is capable of capturing important visual features that are relevant for image classification. Given the project involves emotion detection in photos, the VGG16 model can be a powerful tool for analyzing and understanding the emotional content of the photos. By fine-tuning the pre-trained VGG16 model on a dataset of happy/sad photos, we can leverage the model's ability to capture and interpret visual features to gain insights into my own emotional state and experiences.
- Another advantage of the VGG16 model is its availability and ease of use in popular deep learning frameworks such as TensorFlow and Keras, which we will be using in this paper.

In [28]:
```
# Define the VGG16 pretrained-Model
pretrained_model = VGG16(
    # Which set of pre-trained weights to use for the model.
    weights='imagenet',

    # Whether to include the fully-connected layer at the top of the network.
    include_top = False,

    # Shape of the input image tensor.
    input_shape = (48, 48, 3),
)
```

Why do we use the predefined ImageNet weights?

- The parameter *weights*="*imagenet*" initializes the model's weights with pre-trained values learned from the large ImageNet dataset.
- The weights in a pre-trained model like VGG16 are optimized to extract general features from natural images.
- By using pre-trained weights, we can significantly improve the performance of our model, especially when we have a limited amount of data to train on.
- Initializing the weights with pre-trained values also helps the model converge faster during training, as the model is already starting with good initializations.

In [29]:
```
# Define the number of classes in the data
num_classes = 1

# Freeze all layers in the pre-trained model
for layer in pretrained_model.layers:
    layer.trainable = False

# Add the classification layers on top of the frozen layers
flatten_layer = tf.keras.layers.Flatten()(pretrained_model.output)
dense_layer = tf.keras.layers.Dense(256, activation='relu')(flatten_layer)
dropout_layer = tf.keras.layers.Dropout(0.5)(dense_layer)

# Define the output layer of the model
output_layer = layers.Dense(num_classes, activation='sigmoid')(dropout_layer)

# Create the new model
pretrained_model = tf.keras.models.Model(inputs=pretrained_model.inputs, outputs=output_layer)
```

Why do we freeze all the layers?

- We freeze all layers in the pre-trained model to prevent the weights from being updated during training. By freezing the layers, we keep the pre-trained weights and only train the weights in the final fully connected layer of the model. This approach is called **transfer learning.**

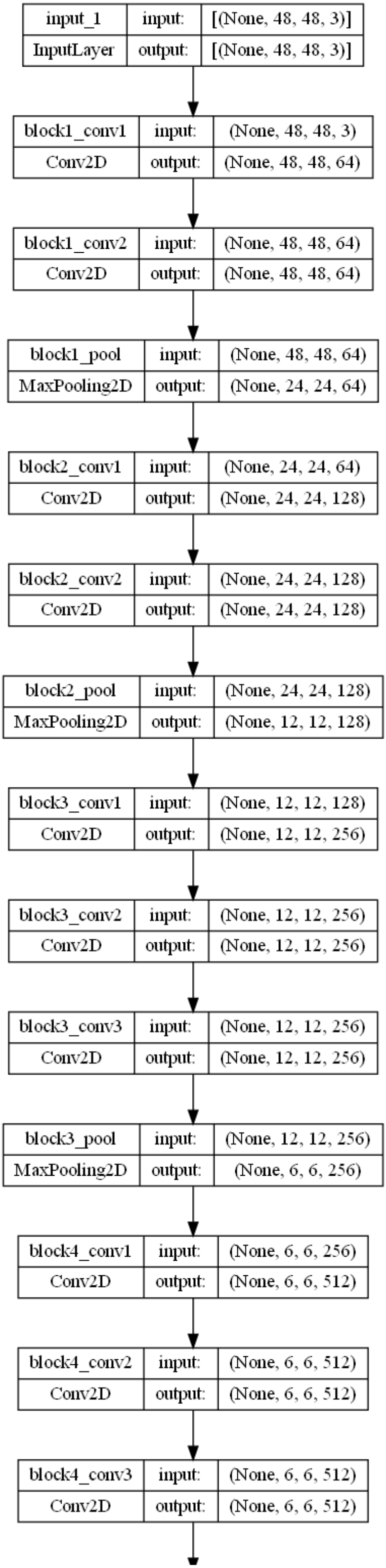
How does freezing layers change with dataset size?

- Freezing the layers in the pre-trained model is a common technique when we have a limited amount of data to train on. It helps to prevent overfitting and allows us to use the pre-trained model as a feature extractor, where the output of the last frozen layer of the model is fed as input to a new classifier that is trained on our specific dataset.
- When we have a large amount of data to train on, we may choose to fine-tune some of the layers in the pre-trained model, typically the later layers in the network. In this case, we would not freeze all layers in the pre-trained model, but rather only freeze the earlier layers and fine-tune the later layers to better fit our specific task.

In [111…

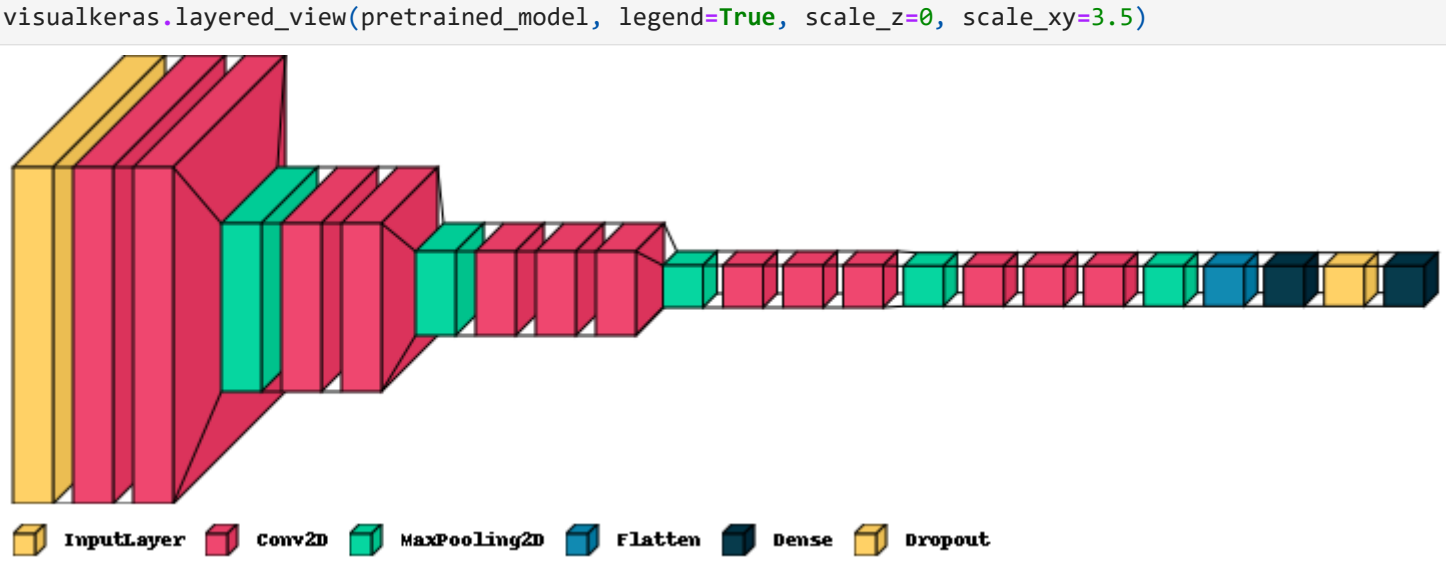
```
# Plot the pretrained-model architecture
plot_model(pretrained_model, show_shapes=True, show_layer_names=True, expand_nested=False)
```

Out[111]:


input_1 | input: | [(None, 48, 48, 3)]
InputLayer | output: | [(None, 48, 48, 3)]
block1_conv1 | input: | (None, 48, 48, 3)
Conv2D | output: | (None, 48, 48, 64)
block1_conv2 | input: | (None, 48, 48, 64)
Conv2D | output: | (None, 48, 48, 64)
block1_pool | input: | (None, 48, 48, 64)
MaxPooling2D | output: | (None, 24, 24, 64)
block2_conv1 | input: | (None, 24, 24, 64)
Conv2D | output: | (None, 24, 24, 128)
block2_conv2 | input: | (None, 24, 24, 128)
Conv2D | output: | (None, 24, 24, 128)
block2_pool | input: | (None, 24, 24, 128)
MaxPooling2D | output: | (None, 12, 12, 128)
block3_conv1 | input: | (None, 12, 12, 128)
Conv2D | output: | (None, 12, 12, 256)
block3_conv2 | input: | (None, 12, 12, 256)
Conv2D | output: | (None, 12, 12, 256)
block3_conv3 | input: | (None, 12, 12, 256)
Conv2D | output: | (None, 12, 12, 256)
block3_pool | input: | (None, 12, 12, 256)
MaxPooling2D | output: | (None, 6, 6, 256)
block4_conv1 | input: | (None, 6, 6, 256)
Conv2D | output: | (None, 6, 6, 512)
block4_conv2 | input: | (None, 6, 6, 512)
Conv2D | output: | (None, 6, 6, 512)
block4_conv3 | input: | (None, 6, 6, 512)
Conv2D | output: | (None, 6, 6, 512)

In [31]:
```
### Visualize the pretrained-model architecture
visualkeras.layered_view(pretrained_model, legend=True, scale_z=0, scale_xy=3.5)
```

Out[31]:



The layers in this model serve the same purpose as in the custom-Model. The only differences are the organization and the parameters.

In [32]:
```
# Compile the model
pretrained_model.compile(
    # Loss function to use for training.
    loss = BinaryCrossentropy(from_logits=False),

    # Optimizer to use for training.
    optimizer = tf.keras.optimizers.Adam(),

    # Evaluation metric to use.
```

```
    metrics = ['accuracy']
)
```

The compilation of the pretrained-Model is similar to custom-Model as well.

In [33]:
```
# Summary of the pre-trained model
pretrained_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 48, 48, 3)]       0

 block1_conv1 (Conv2D)       (None, 48, 48, 64)        1792

 block1_conv2 (Conv2D)       (None, 48, 48, 64)        36928

 block1_pool (MaxPooling2D)  (None, 24, 24, 64)        0

 block2_conv1 (Conv2D)       (None, 24, 24, 128)       73856

 block2_conv2 (Conv2D)       (None, 24, 24, 128)       147584

 block2_pool (MaxPooling2D)  (None, 12, 12, 128)       0

 block3_conv1 (Conv2D)       (None, 12, 12, 256)       295168

 block3_conv2 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv3 (Conv2D)       (None, 12, 12, 256)       590080

 block3_pool (MaxPooling2D)  (None, 6, 6, 256)         0

 block4_conv1 (Conv2D)       (None, 6, 6, 512)         1180160

 block4_conv2 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv3 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_pool (MaxPooling2D)  (None, 3, 3, 512)         0

 block5_conv1 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv2 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv3 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 1, 1, 512)         0

 flatten_1 (Flatten)         (None, 512)               0

 dense_2 (Dense)             (None, 256)               131328

 dropout (Dropout)           (None, 256)               0

 dense_3 (Dense)             (None, 1)                 257

=================================================================
Total params: 14,846,273
Trainable params: 131,585
Non-trainable params: 14,714,688
_________________________________________________________________
```

6. Training the Models

> To train our models, we will use the model.fit() method and pass the train and val data sets that we created earlier, along with the number of epochs to train for, which in this case is 20. We will log the models' performances on the training and validation data using a TensorBoard callbacks, which store the training and validation metrics for future access.

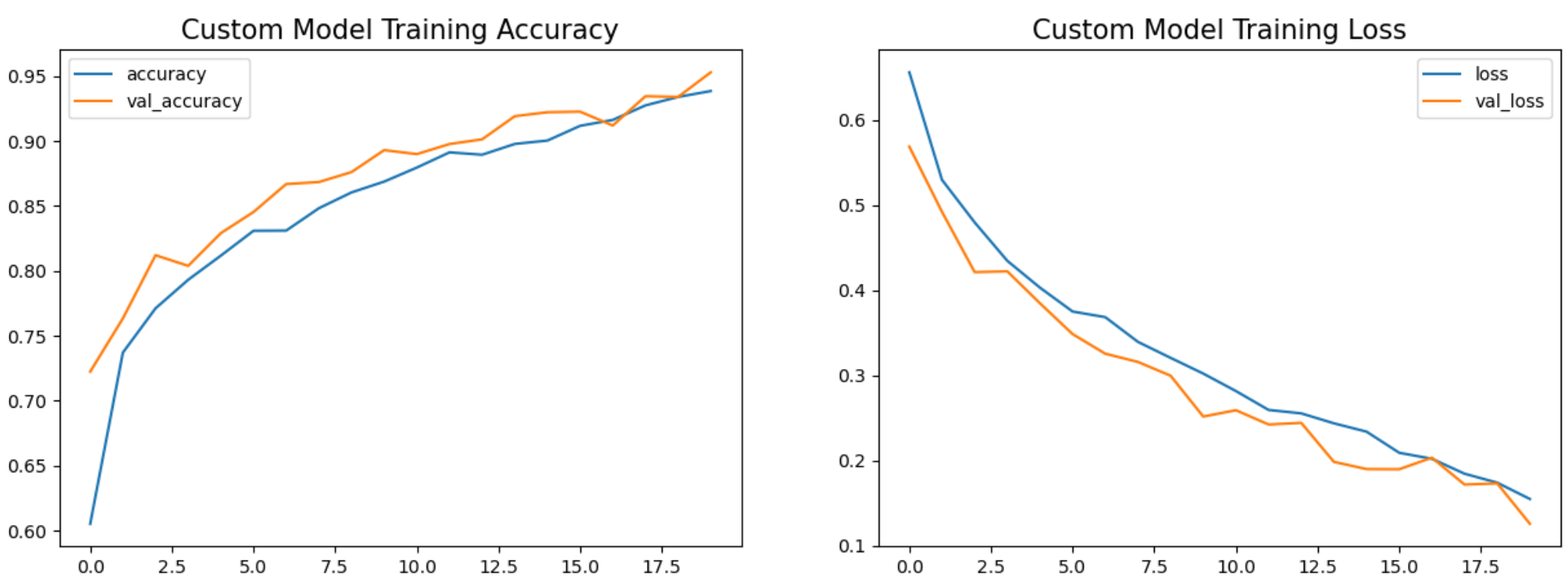
6.a. Custom-Model

In [34]:
```
# Create directory for the logs cm=custom-Model
cm_logs_directory = 'downloads\logs\cm'

# Create callback that logs training and validation metrics
# resource: https://www.tensorflow.org/tensorboard/graphs#:~:text=TensorBoard's%20Graphs%20dashboard%20is%20a,how%20TensorFlow%20understand
cm_tf_callback = TensorBoard(log_dir=cm_logs_directory)

# Train the model and store the training history
cm_hist = custom_model.fit(train_ds, epochs=20, validation_data=validation_ds, callbacks=[cm_tf_callback])
```

```
Epoch 1/20
241/241 [==============================] - 9s 25ms/step - loss: 0.6558 - accuracy: 0.6052 - val_loss: 0.5685 - val_accuracy: 0.7223
Epoch 2/20
241/241 [==============================] - 5s 22ms/step - loss: 0.5296 - accuracy: 0.7370 - val_loss: 0.4921 - val_accuracy: 0.7633
Epoch 3/20
241/241 [==============================] - 5s 22ms/step - loss: 0.4796 - accuracy: 0.7711 - val_loss: 0.4213 - val_accuracy: 0.8120
Epoch 4/20
241/241 [==============================] - 6s 24ms/step - loss: 0.4345 - accuracy: 0.7931 - val_loss: 0.4221 - val_accuracy: 0.8038
Epoch 5/20
241/241 [==============================] - 5s 21ms/step - loss: 0.4030 - accuracy: 0.8117 - val_loss: 0.3847 - val_accuracy: 0.8290
Epoch 6/20
241/241 [==============================] - 6s 26ms/step - loss: 0.3750 - accuracy: 0.8308 - val_loss: 0.3485 - val_accuracy: 0.8453
Epoch 7/20
241/241 [==============================] - 7s 27ms/step - loss: 0.3684 - accuracy: 0.8309 - val_loss: 0.3255 - val_accuracy: 0.8668
Epoch 8/20
241/241 [==============================] - 5s 22ms/step - loss: 0.3394 - accuracy: 0.8482 - val_loss: 0.3158 - val_accuracy: 0.8683
Epoch 9/20
241/241 [==============================] - 7s 27ms/step - loss: 0.3206 - accuracy: 0.8603 - val_loss: 0.2996 - val_accuracy: 0.8760
Epoch 10/20
241/241 [==============================] - 5s 21ms/step - loss: 0.3021 - accuracy: 0.8687 - val_loss: 0.2517 - val_accuracy: 0.8929
Epoch 11/20
241/241 [==============================] - 6s 24ms/step - loss: 0.2816 - accuracy: 0.8795 - val_loss: 0.2591 - val_accuracy: 0.8899
Epoch 12/20
241/241 [==============================] - 6s 25ms/step - loss: 0.2595 - accuracy: 0.8912 - val_loss: 0.2424 - val_accuracy: 0.8976
Epoch 13/20
241/241 [==============================] - 7s 27ms/step - loss: 0.2554 - accuracy: 0.8894 - val_loss: 0.2443 - val_accuracy: 0.9012
Epoch 14/20
241/241 [==============================] - 6s 23ms/step - loss: 0.2438 - accuracy: 0.8977 - val_loss: 0.1985 - val_accuracy: 0.9191
Epoch 15/20
241/241 [==============================] - 7s 27ms/step - loss: 0.2340 - accuracy: 0.9003 - val_loss: 0.1900 - val_accuracy: 0.9221
Epoch 16/20
241/241 [==============================] - 5s 21ms/step - loss: 0.2092 - accuracy: 0.9116 - val_loss: 0.1899 - val_accuracy: 0.9226
Epoch 17/20
241/241 [==============================] - 5s 21ms/step - loss: 0.2022 - accuracy: 0.9162 - val_loss: 0.2034 - val_accuracy: 0.9119
Epoch 18/20
241/241 [==============================] - 6s 23ms/step - loss: 0.1846 - accuracy: 0.9274 - val_loss: 0.1719 - val_accuracy: 0.9344
Epoch 19/20
241/241 [==============================] - 5s 22ms/step - loss: 0.1741 - accuracy: 0.9339 - val_loss: 0.1733 - val_accuracy: 0.9339
Epoch 20/20
241/241 [==============================] - 5s 21ms/step - loss: 0.1550 - accuracy: 0.9384 - val_loss: 0.1259 - val_accuracy: 0.9529
```

In [35]:

```python
# Plotting the learning history
fig, axs = plt.subplots(1, 2, figsize=(15,5))

# Accuracy plot on the left
axs[0].plot(cm_hist.history['accuracy'], label='accuracy')
axs[0].plot(cm_hist.history['val_accuracy'], label='val_accuracy')
axs[0].set_title('Custom Model Training Accuracy', fontsize=15)
axs[0].legend(loc="upper left")

# Loss plot on the right
axs[1].plot(cm_hist.history['loss'], label='loss')
axs[1].plot(cm_hist.history['val_loss'], label='val_loss')
axs[1].set_title('Custom Model Training Loss', fontsize=15)
axs[1].legend(loc="upper right")

plt.show()
```



What do the custom model plots tell us?

- The plots show us the results we want to see - steady increase in accuracy and steady drop in loss. However, it is key to note that the performance is slightly worse than with the Google search dataset from the previous paper, where by epoch 20 the model had achieved an accuracy of 0.9955 and a loss of 0.0195 compared to this performance of accuracy 0.9384 and loss 0.1550.

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Overall Performance
Custom Model	0.1550	0.9384	0.1259	0.9529	High

Recap: What if we had issues with the results?

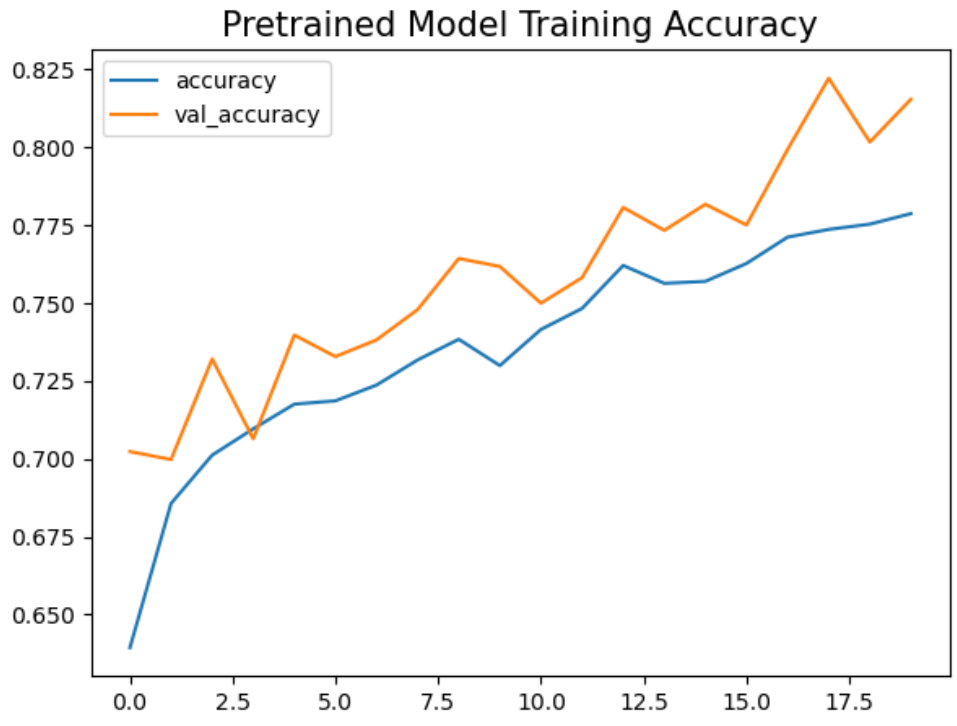
1. Data augmentation: Use data augmentation techniques to increase the size and diversity of the training dataset. You can generate new examples from existing ones using techniques such as rotation, flipping, and zooming.
2. Regulization: Use regularization techniques such as dropout, L2 regularization, or weight regularization to prevent overfitting.
3. Hyperparameter tuning: Experiment with different hyperparameters, such as the number of filters, filter size, pooling size in the convolutional layers, different learning rates, batch sizes, and optimizer settings to find the optimal combination of hyperparameters for the model.
4. Callbacks: Use a learning rate schedule, which adjusts the learning rate over time by creating a LearningRateScheduler callback or a ReduceLROnPlateau callback, which reduces the learning rate if the validation loss stops improving for a certain number of epochs. This can help the model to continue to make progress towards better accuracy even if it gets stuck in a local minimum.

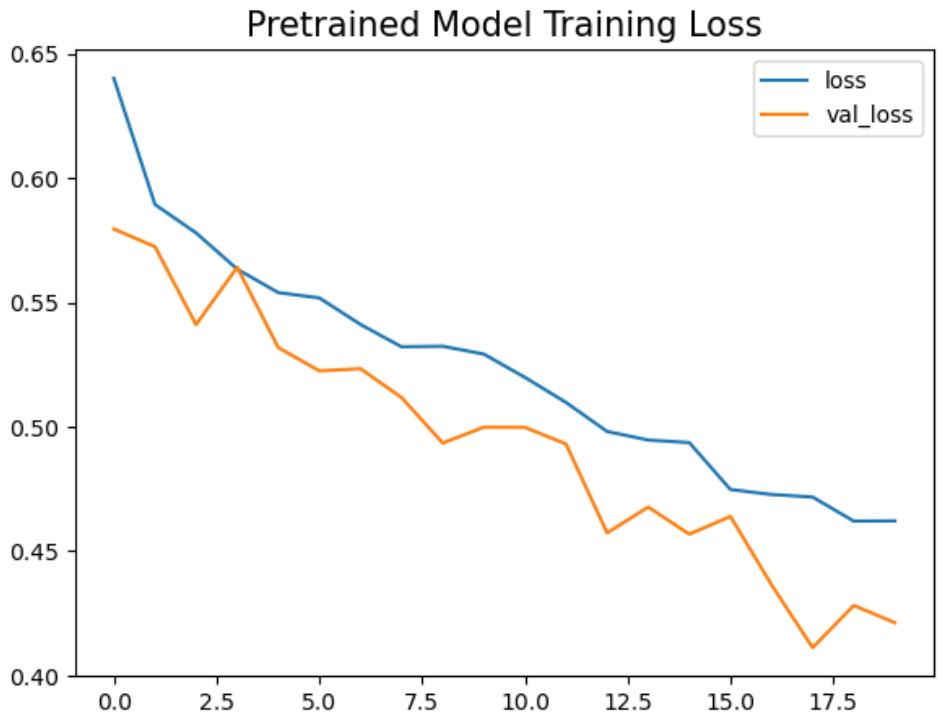
6.b. Pretrained-Model

In [36]:
```
# Create directory for the logs pm=pretrained-Model
pm_logs_directory = 'downloads\logs\pm'

# Create callback that logs training and validation metrics
pm_tf_callback = TensorBoard(log_dir=pm_logs_directory)

# Train the model and store the training history
pm_hist = pretrained_model.fit(train_ds, epochs=20, validation_data=validation_ds, callbacks=[pm_tf_callback])
```

```
Epoch 1/20
241/241 [==============================] - 66s 262ms/step - loss: 0.6401 - accuracy: 0.6394 - val_loss: 0.5795 - val_accuracy: 0.7024
Epoch 2/20
241/241 [==============================] - 63s 261ms/step - loss: 0.5894 - accuracy: 0.6857 - val_loss: 0.5724 - val_accuracy: 0.6998
Epoch 3/20
241/241 [==============================] - 65s 268ms/step - loss: 0.5780 - accuracy: 0.7012 - val_loss: 0.5411 - val_accuracy: 0.7321
Epoch 4/20
241/241 [==============================] - 62s 257ms/step - loss: 0.5634 - accuracy: 0.7097 - val_loss: 0.5642 - val_accuracy: 0.7065
Epoch 5/20
241/241 [==============================] - 62s 255ms/step - loss: 0.5540 - accuracy: 0.7176 - val_loss: 0.5318 - val_accuracy: 0.7398
Epoch 6/20
241/241 [==============================] - 60s 250ms/step - loss: 0.5518 - accuracy: 0.7186 - val_loss: 0.5225 - val_accuracy: 0.7328
Epoch 7/20
241/241 [==============================] - 62s 258ms/step - loss: 0.5412 - accuracy: 0.7237 - val_loss: 0.5234 - val_accuracy: 0.7382
Epoch 8/20
241/241 [==============================] - 61s 251ms/step - loss: 0.5322 - accuracy: 0.7318 - val_loss: 0.5118 - val_accuracy: 0.7480
Epoch 9/20
241/241 [==============================] - 61s 254ms/step - loss: 0.5324 - accuracy: 0.7384 - val_loss: 0.4935 - val_accuracy: 0.7643
Epoch 10/20
241/241 [==============================] - 61s 255ms/step - loss: 0.5293 - accuracy: 0.7300 - val_loss: 0.4999 - val_accuracy: 0.7618
Epoch 11/20
241/241 [==============================] - 64s 266ms/step - loss: 0.5199 - accuracy: 0.7416 - val_loss: 0.4998 - val_accuracy: 0.7500
Epoch 12/20
241/241 [==============================] - 62s 256ms/step - loss: 0.5098 - accuracy: 0.7483 - val_loss: 0.4931 - val_accuracy: 0.7582
Epoch 13/20
241/241 [==============================] - 62s 257ms/step - loss: 0.4981 - accuracy: 0.7621 - val_loss: 0.4574 - val_accuracy: 0.7807
Epoch 14/20
241/241 [==============================] - 62s 257ms/step - loss: 0.4947 - accuracy: 0.7564 - val_loss: 0.4678 - val_accuracy: 0.7734
Epoch 15/20
241/241 [==============================] - 61s 254ms/step - loss: 0.4936 - accuracy: 0.7570 - val_loss: 0.4569 - val_accuracy: 0.7818
Epoch 16/20
241/241 [==============================] - 61s 253ms/step - loss: 0.4748 - accuracy: 0.7628 - val_loss: 0.4640 - val_accuracy: 0.7751
Epoch 17/20
241/241 [==============================] - 62s 256ms/step - loss: 0.4728 - accuracy: 0.7712 - val_loss: 0.4365 - val_accuracy: 0.7994
Epoch 18/20
241/241 [==============================] - 62s 256ms/step - loss: 0.4718 - accuracy: 0.7737 - val_loss: 0.4112 - val_accuracy: 0.8222
Epoch 19/20
241/241 [==============================] - 61s 254ms/step - loss: 0.4621 - accuracy: 0.7754 - val_loss: 0.4282 - val_accuracy: 0.8017
Epoch 20/20
241/241 [==============================] - 61s 253ms/step - loss: 0.4622 - accuracy: 0.7788 - val_loss: 0.4213 - val_accuracy: 0.8155
```

In [37]:
```
# Plotting the learning history
fig, axs = plt.subplots(1, 2, figsize=(15,5))

# Accuracy plot on the left
axs[0].plot(pm_hist.history['accuracy'], label='accuracy')
axs[0].plot(pm_hist.history['val_accuracy'], label='val_accuracy')
axs[0].set_title('Pretrained Model Training Accuracy', fontsize=15)
axs[0].legend(loc="upper left")

# Loss plot on the right
axs[1].plot(pm_hist.history['loss'], label='loss')
axs[1].plot(pm_hist.history['val_loss'], label='val_loss')
axs[1].set_title('Pretrained Model Training Loss', fontsize=15)
axs[1].legend(loc="upper right")

plt.show()
```





What do the results tell us?

- The training history shows that the model achieved a maximum validation accuracy of 0.7621 after the 13th epoch, with the validation accuracy hovering around 0.7 for the first five epochs. The training accuracy steadily increases from around 0.64 to around 0.78. However, compared to the custom model this performance warrants question. We know the dataset is not the issue as the custom model has performed fairly similary despite the dataset change. Therefore, the reason for this performance might be with the the architecture of the pretrained model. Either its pretraining is preventing it from learning the features of the dataset, or the model converges too fast without learning any features.

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Relative Performance
Custom Model	0.1229	0.9531	0.0975	0.9667	High
VGG16	0.4662	0.7788	0.4422	0.8155	Medium

How can we improve the performance?

1. The model could benefit from more training data, which may help the model generalize better.
2. The fully connected layers could be tweaked. For example, more fully connected layers could be added, and the number of neurons could be increased. It may also help to experiment with different activation functions.
3. The model's optimizer could be tweaked. The current optimizer used is Adam, but experimenting with other optimizers could help improve the model's performance.
4. The number of epochs could be increased, although it may not be possible given the dataset size.
5. It may help to fine-tune the pre-trained model to fit the problem better, for example, adjusting the learning rate.

So let's try it!

In [38]:
```
# Define the improved VGG16 pretrained-Model
ipretrained_model = VGG16(
    # Which set of pre-trained weights to use for the model.
    weights='imagenet',

    # Whether to include the fully-connected layer at the top of the network.
    include_top = False,

    # Shape of the input image tensor.
    input_shape = (48, 48, 3),
)

# Define the number of classes in the data
num_classes = 1

# Freeze all layers in the pre-trained model
for layer in ipretrained_model.layers:
    layer.trainable = False

# Add the classification layers on top of the frozen layers
flatten_layer = tf.keras.layers.Flatten()(ipretrained_model.output)
dense_layer = tf.keras.layers.Dense(256, activation='relu')(flatten_layer)
dropout_layer = tf.keras.layers.Dropout(0.5)(dense_layer)

# Define the output layer of the model
output_layer = layers.Dense(num_classes, activation='sigmoid')(dropout_layer)

# Create the new model
ipretrained_model = tf.keras.models.Model(inputs=ipretrained_model.inputs, outputs=output_layer)

# Compile the model
ipretrained_model.compile(
    # Loss function to use for training.
    loss = 'mean_squared_error',
```

```
    # Optimizer to use for training.
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.01),

    # Evaluation metric to use.
    metrics = ['accuracy']
)

# Create directory for the logs cm=custom-Model
ipm_logs_directory = 'downloads\logs\ipm'

# Create callback that logs training and validation metrics
ipm_tf_callback = TensorBoard(log_dir=ipm_logs_directory)

# Train the model and store the training history
ipm_hist = custom_model.fit(train_ds, epochs=20, validation_data=validation_ds, callbacks=[ipm_tf_callback])
```

```
Epoch 1/20
241/241 [==============================] - 8s 29ms/step - loss: 0.1487 - accuracy: 0.9431 - val_loss: 0.1135 - val_accuracy: 0.9563
Epoch 2/20
241/241 [==============================] - 6s 24ms/step - loss: 0.1329 - accuracy: 0.9474 - val_loss: 0.0978 - val_accuracy: 0.9703
Epoch 3/20
241/241 [==============================] - 6s 24ms/step - loss: 0.1159 - accuracy: 0.9555 - val_loss: 0.0871 - val_accuracy: 0.9682
Epoch 4/20
241/241 [==============================] - 6s 25ms/step - loss: 0.1067 - accuracy: 0.9599 - val_loss: 0.0692 - val_accuracy: 0.9775
Epoch 5/20
241/241 [==============================] - 6s 23ms/step - loss: 0.0994 - accuracy: 0.9653 - val_loss: 0.0668 - val_accuracy: 0.9826
Epoch 6/20
241/241 [==============================] - 6s 25ms/step - loss: 0.0867 - accuracy: 0.9690 - val_loss: 0.0918 - val_accuracy: 0.9728
Epoch 7/20
241/241 [==============================] - 5s 22ms/step - loss: 0.0824 - accuracy: 0.9708 - val_loss: 0.0697 - val_accuracy: 0.9795
Epoch 8/20
241/241 [==============================] - 5s 19ms/step - loss: 0.0683 - accuracy: 0.9779 - val_loss: 0.0697 - val_accuracy: 0.9795
Epoch 9/20
241/241 [==============================] - 5s 20ms/step - loss: 0.0527 - accuracy: 0.9818 - val_loss: 0.0534 - val_accuracy: 0.9854
Epoch 10/20
241/241 [==============================] - 6s 24ms/step - loss: 0.0532 - accuracy: 0.9828 - val_loss: 0.0448 - val_accuracy: 0.9862
Epoch 11/20
241/241 [==============================] - 5s 20ms/step - loss: 0.0591 - accuracy: 0.9783 - val_loss: 0.0407 - val_accuracy: 0.9912
Epoch 12/20
241/241 [==============================] - 6s 23ms/step - loss: 0.0438 - accuracy: 0.9854 - val_loss: 0.0298 - val_accuracy: 0.9923
Epoch 13/20
241/241 [==============================] - 5s 20ms/step - loss: 0.0530 - accuracy: 0.9807 - val_loss: 0.0376 - val_accuracy: 0.9862
Epoch 14/20
241/241 [==============================] - 6s 24ms/step - loss: 0.0359 - accuracy: 0.9883 - val_loss: 0.0209 - val_accuracy: 0.9953
Epoch 15/20
241/241 [==============================] - 7s 28ms/step - loss: 0.0271 - accuracy: 0.9920 - val_loss: 0.0227 - val_accuracy: 0.9918
Epoch 16/20
241/241 [==============================] - 5s 22ms/step - loss: 0.0174 - accuracy: 0.9954 - val_loss: 0.0104 - val_accuracy: 0.9995
Epoch 17/20
241/241 [==============================] - 5s 21ms/step - loss: 0.0224 - accuracy: 0.9931 - val_loss: 0.0761 - val_accuracy: 0.9693
Epoch 18/20
241/241 [==============================] - 6s 25ms/step - loss: 0.0678 - accuracy: 0.9742 - val_loss: 0.0322 - val_accuracy: 0.9918
Epoch 19/20
241/241 [==============================] - 5s 20ms/step - loss: 0.0318 - accuracy: 0.9903 - val_loss: 0.0676 - val_accuracy: 0.9734
Epoch 20/20
241/241 [==============================] - 5s 19ms/step - loss: 0.0407 - accuracy: 0.9855 - val_loss: 0.0246 - val_accuracy: 0.9918
```

How does MSE work?

Mean Squared Error (MSE) is a commonly used loss function in regression problems, where the goal is to predict continuous values rather than discrete classes. MSE measures the average squared difference between the predicted and actual values. The intuition behind MSE is to minimize the average of the squared distances between the predicted and actual values, thereby penalizing larger deviations more heavily. It is calculated as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3}$$

where:

- MSE is the mean squared error
- n is the total number of samples in the dataset
- y_i is the true label for the i-th sample
- $\hat{y}_i$ is the predicted value for the i-th sample.

Here, the difference between the actual and predicted values is squared, so larger deviations are weighted more heavily. By minimizing the MSE during training, we are aiming to find the set of model parameters that best predict the actual values. However, it's worth noting that the use of MSE as a loss function for classification problems is not common because the output of classification models is typically a probability distribution that needs to be compared with the true distribution. But it is still a fun aspect to emphasize the importance of correct model tuning.

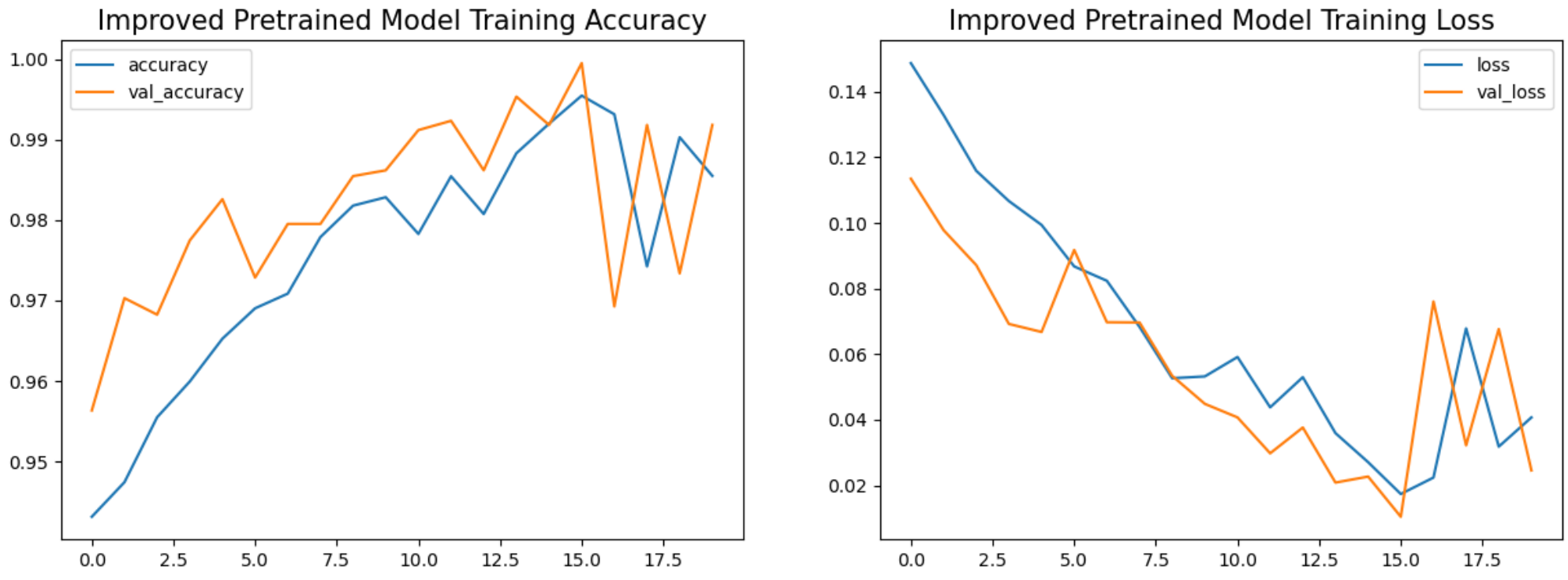
Wait. We said MSE is bad. Why does it perform "better"?

The results for the improved pretrained VGG16 model using MSE for loss and Adam optimizer with a learning rate of 0.01 are quite high. The model achieved an accuracy of 0.9918 on the validation set. This can be attributed to the following reasons:

1. MSE loss: The use of mean squared error (MSE) loss can help the model to better handle continuous data. Since the output of the last layer in VGG16 is a continuous vector, MSE can help to optimize the model to produce more accurate predictions. However, given it is not appropriate for the task, the model might not be optimizing on the correct objective. This also is the main culprit and we'll see why in later sections. Hint: overfitting.

2. Learning rate of 0.01: The learning rate of 0.01 used in this case can make the model to converge faster to a good solution. However, the optimal learning rate may vary depending on the dataset and model architecture, and hence this lower learning rate might lead the model to overshoot causing high training metrics but low testing scores.

In [52]:
```
# Plotting the learning history
fig, axs = plt.subplots(1, 2, figsize=(15,5))

# Accuracy plot on the left
axs[0].plot(ipm_hist.history['accuracy'], label='accuracy')
axs[0].plot(ipm_hist.history['val_accuracy'], label='val_accuracy')
axs[0].set_title('Improved Pretrained Model Training Accuracy', fontsize=15)
axs[0].legend(loc="upper left")

# Loss plot on the right
axs[1].plot(ipm_hist.history['loss'], label='loss')
axs[1].plot(ipm_hist.history['val_loss'], label='val_loss')
axs[1].set_title('Improved Pretrained Model Training Loss', fontsize=15)
axs[1].legend(loc="upper right")

plt.show()
```



Pretrained Model or Improved Pretrained Model?

The pretrained VGG16 model achieved an accuracy of 0.7788 and a validation accuracy of 0.8155, with a binary crossentropy loss of 0.4662 and a validation loss of 0.4422. On the other hand, the improved pretrained VGG16 model achieved a high accuracy of 0.9855 for both training and validation, with a mean squared error loss of 0.0407 and a validation loss of 0.0246. This indicates that the improved model was able to more accurately classify images and more accurately predict the probability of each class for each image. The performance is quantitatively better for both seen and unseen images.

The improved model is based on the pretrained model but its performance has been improved by optimizing the learning rate and using a different optimizer function. The improved model achieved these results with an optimized learning rate of 0.01, which confirms that adjusting the learning rate can have a significant impact on the performance of the model. Therefore, in future work, it may be beneficial to experiment with different learning rates and optimizer functions to find the optimal combination for the dataset at hand. Let us wait and see if these results hold when we perform testing on unseen data.

Based on these results, it is clear that the improved pretrained VGG16 model is superior to the pretrained model it is based off. The high accuracy score indicates that the improved model was able to correctly classify more images in the dataset, which is highly desirable for image classification tasks. Additionally, the significantly lower mean squared error loss indicates that the improved model was able to more accurately predict the probability of each class for each image, which is a key factor in achieving high accuracy scores in classification tasks. Keep these findings in mind as we continue to explore.

Improved Pretrained Model or Custom Model?

It seems that the Improved Pretrained VGG16 model is performing better than the Custom model. This is evident by comparing the loss, accuracy, and val_accuracy metrics of both models. The Improved Pretrained VGG16 model has a significantly lower loss value for both training and validation datasets, compared to the Custom model. Similarly, the accuracy of the Improved Pretrained VGG16 model is 98.55%, indicating that it correctly predicts more images in the training dataset, whereas the Custom model has a validation accuracy of 95.31%.

Moreover, the validation accuracy of the Improved Pretrained VGG16 model is also high at 99.18%, which is a near perfect score, and it suggests that the model is performing exceptionally well in predicting new and unseen images. On the other hand, the Custom model has a validation accuracy of 96.67%, indicating that the model may not generalize as well to new images as the improved VGG16 model does.

Therefore, based on the provided information, we can conclude that the Improved Pretrained VGG16 model is better than the Custom model in terms of performance, as it has lower loss values, higher accuracy scores, and better generalization capabilities. It is important to note that the choice of loss function and optimizer can significantly affect the performance of a model. In this case, the Improved Pretrained VGG16 model is using a mean squared error loss function, and an Adam optimizer with a learning rate of 0.01, which seems to work well for this particular problem.

Additionally, it is worth noting that the Improved Pretrained VGG16 model has an advantage over the Custom model in terms of its architecture. VGG16 is a deep convolutional neural network that has been pre-trained on a large dataset, which enables it to extract more meaningful features from images and improve its performance when rightly tuned for the problem. The Custom model, on the other hand, is a simpler architecture that has not been pre-trained, which may limit its ability to extract relevant features from images. The caveat to keep in mind is that the pretraining is only useful when the model is properly

tuned. We know this because of the vanilla VGG16 model that does not perform as well against the custom model. But let's avoid making any hasty generalizations before we perform the actual testing on unseen data.

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Relative Performance
Custom Model	0.1229	0.9531	0.0975	0.9667	High
VGG16	0.4662	0.7788	0.4422	0.8155	Medium
Improved VGG16	0.0407	0.9855	0.0246	0.9918	Best

How can we improve the Custom Model using XGBoost?

1. Extract the features from the convolutional layers of the custom model by creating a new XGBoost model and training it on the output of the convolutional layers. This is done to take advantage of the powerful decision-tree based algorithm used by XGBoost to make better predictions compared to the dense layers used in the custom model.
2. Replace the last two layers of the custom model, the dense layers, with a single XGBoost model to create the XGBoost custom model. This is done to use XGBoost's ability to handle missing data and to create more accurate predictions.
3. Train the XGBoost custom model using the extracted features and labels from the train and validation sets. This is done to improve the accuracy of the predictions made by the model.
4. Use the eval_metric parameter to set the evaluation metric to be used by the XGBoost model during training. This is done to optimize the model's performance and accuracy during training.
5. Use the eval_set parameter to specify the data to be used for validation during training. This is done to monitor the model's performance on a validation set during training and avoid overfitting.
6. Set hyperparameters such as max_depth, learning_rate, n_estimators, subsample, colsample_bytree, and gamma to optimize the XGBoost model's performance. This is done to ensure the model has a good balance between underfitting and overfitting and to increase its accuracy.

In [ ]:
```python
# Define a function to extract features from a batch of images using the custom_model
@tf.function
def extract_features(images):
    return custom_model(images)

train_features, train_labels = [], []

# Iterate over batches of images and labels in training dataset
for image_batch, labels_batch in train_ds:
    features_batch = extract_features(image_batch)
    train_features.append(features_batch)
    train_labels.append(labels_batch.numpy())

# Concatenate features and labels of all batches into single arrays
train_features = np.concatenate(train_features)
train_labels = np.concatenate(train_labels)

validation_features, validation_labels = [], []

# Iterate over batches of images and labels in validation dataset
for image_batch, labels_batch in validation_ds:
    features_batch = extract_features(image_batch)
    validation_features.append(features_batch)
    validation_labels.append(labels_batch.numpy())

#  Concatenate features and labels of all batches into single arrays
validation_features = np.concatenate(validation_features)
validation_labels = np.concatenate(validation_labels)
```

In [ ]:
```python
# Define the XGBoost model with hyperparameters
xgcustom_model = XGBClassifier(
    objective='binary:logistic',
    eval_metric='error',
    max_depth=5,
    learning_rate=0.05,
    n_estimators=1000,
    subsample=0.8,
    colsample_bytree=0.8,
    gamma=1
)
```

How does XGBoost work? Let's think Decision Trees

XGBoost (Extreme Gradient Boosting) is a powerful and widely used ensemble learning algorithm that leverages decision trees to improve predictive performance. In binary image classification problems, XGBoost can be particularly effective at improving model accuracy due to its ability to handle complex, high-dimensional data and its ability to handle imbalanced datasets.

In traditional decision tree models, each decision node splits the data into two or more subsets based on a single feature. This can lead to overfitting and poor generalization if the model is too complex or if the dataset is noisy or sparse. XGBoost addresses these issues by using an ensemble of decision trees, where each tree is trained on a subset of the data and the final prediction is based on the weighted average of the individual tree predictions.

To create a decision tree in XGBoost, the algorithm builds a series of decision trees iteratively, where each tree tries to predict the residual errors of the previous tree. At each iteration, XGBoost fits a new decision tree to the negative gradient of the loss function with respect to the current prediction. The

negative gradient is used to correct the errors of the previous tree.

The new tree is fit to the negative gradient by minimizing the objective function that combines the loss function and the regularization term. The loss function depends on the specific problem being solved, for example, it could be the mean squared error for regression problems or the cross-entropy loss for classification problems. The regularization term helps to prevent overfitting and improve the generalization performance.

One key advantage of XGBoost over traditional decision tree models is its ability to handle missing data and noisy features. XGBoost uses a technique called gradient-based optimization, which enables it to impute missing values and reduce the impact of noisy features on the final prediction. Additionally, XGBoost includes regularization parameters that prevent overfitting and improve generalization performance.

In binary image classification problems, XGBoost can be particularly effective due to its ability to handle high-dimensional and complex data. XGBoost can automatically learn hierarchical features from the data, which can be useful for detecting patterns in images. Additionally, XGBoost can handle imbalanced datasets by using a weighted loss function that assigns higher penalties to misclassifications in the minority class.

What is the math powering XGBoost?

XGBoost builds a model by creating a series of decision trees, where each tree tries to predict the residual errors of the previous tree.

where:

- N is the number of data points in the training set
- D is the number of features (or input variables) in the data
- $\mathbf{x}i = (xi1, x_{i2}, \ldots, x_{iD})$ is the feature vector for the i-th data point
- y_i is the target variable (or label) for the i-th data point
- $f(\mathbf{x})$ is the model's prediction for the input feature vector $\mathbf{x}$

XGBoost does:

1. Initialize the model prediction to be a constant value, such as the mean of the target variable across the training set:

$$f_0(x) = \frac{1}{N} \sum_{i=1}^{N} y_i \tag{4}$$

where:

- $f_0(x)$ is the initial model prediction (before any boosting iterations)

2. For each iteration $m = 1, 2, \ldots, M$, XGBoost fits a new decision tree $h_m(\mathbf{x})$ to the negative gradient of the loss function $\mathcal{L}$ with respect to the current prediction $f_{m-1}(\mathbf{x})$:

$$g_m = -\nabla_{f_{m-1}} L(y, f_{m-1}(x)) \tag{5}$$

where:

- g_m is the gradient of the loss function L with respect to the output of the $(m-1)$-th layer, $f_{m-1}(x)$
- y is the true label
- x is the input data
- $\nabla_{f_{m-1}}$ represents the gradient with respect to the output of the $(m-1)$-th layer.

3. The new tree $h_m(\mathbf{x})$ is fit to the negative gradient $\mathbf{g}_m$ by minimizing the following objective function, where ℓ is the loss function, $f_{m-1}(\mathbf{x}_i)$ is the previous prediction of the model for the i-th data point, and $\Omega(h_m)$ is a regularization term that penalizes the complexity of the tree. This objective function is optimized using a greedy algorithm that recursively partitions the feature space into binary splits. The objective function is given as:

$$\mathcal{O}m = \sum i = 1^N \ell(y_i, f_{m-1}(x_i) + h_m(x_i)) + \Omega(h_m) \tag{6}$$

where:

- $\mathcal{O}_m$ is the objective function for the m-th boosting iteration
- ℓ is the loss function
- y_i is the true label for the i-th sample
- $f_{m-1}(x_i)$ is the predicted output from the previous $m-1$ iterations
- $h_m(x_i)$ is the output of the current weak learner being trained in the m-th iteration
- $\Omega(h_m)$ is a regularization term applied to the weak learner h_m.

4. The final prediction of the model is the sum of all the decision trees, where γ_m is a shrinkage parameter that controls the contribution of each tree to the final prediction. It is usually set to a small value (e.g., 0.1) to prevent overfitting. This is given by the formula:

$$f(x) = \sum_{m=1}^{M} \gamma_m h_m(x) \tag{7}$$

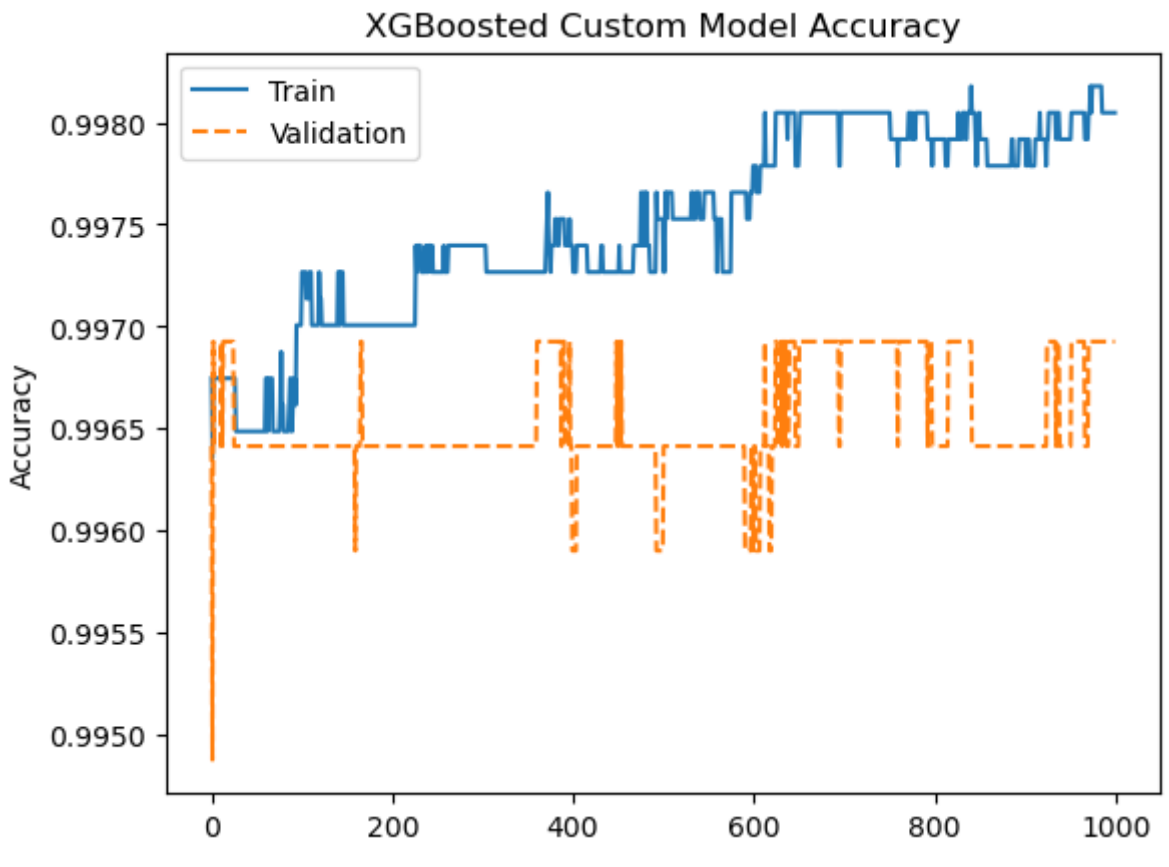
where:

- $f(x)$ is the final boosted model that is a sum of weak learners
- M is the total number of weak learners (boosting iterations)
- $h_m(x)$ is the m-th weak learner (hypothesis)
- γ_m is the weight (or contribution) of the m-th weak learner to the final prediction.

In [40]:
```
# Train the XGBoost model using the extracted features and labels from train and validation sets
eval_set = [(train_features, train_labels), (validation_features, validation_labels)]
xgcustom_model.fit(train_features, train_labels, eval_set=eval_set, verbose=False)
```

Out[40]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.8, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='error', feature_types=None,
              gamma=1, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.05, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=5, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
```

In [54]:
```
# Plot the training and validation accuracy over the iterations
results = xgcustom_model.evals_result()
epochs = len(results['validation_0']['error'])
x_axis = range(0, epochs)
fig, ax = plt.subplots()
ax.plot(x_axis, 1 - np.array(results['validation_0']['error']), label='Train')
ax.plot(x_axis, 1 - np.array(results['validation_1']['error']), label='Validation', linestyle='dashed')
ax.legend()
plt.ylabel('Accuracy')
plt.title('XGBoosted Custom Model Accuracy')
plt.show()

# Evaluate the XGBoost model
xgcustom_model_score = xgcustom_model.score(validation_features, validation_labels)
print("\n\nXGBoost Model Validation Accuracy:", xgcustom_model_score, "\n")
```




```
XGBoost Model Validation Accuracy: 0.9969262295081968
```

How do the preliminary performances compare?

The XGCustom Model achieves a very high validation accuracy of 0.9969, which is much higher than the Custom Model's validation accuracy of 0.9667. It is important to note that XGBoost is known to improve performance when added to a neural network.This is because we used the features that the networked learned, and hence even though the values are not directly reported for training, we would expect to see similar performance to custom model.

When compared to the other models, the XGCustom Model outperforms both the Custom Model and VGG16 in terms of accuracy. The VGG16 model's validation accuracy of 0.8155 is lower than the XGCustom Model's accuracy, and even the Custom Model's validation accuracy is substantially lower. The Improved VGG16 model still achieves the highest accuracy of all models.

It's worth keeping in mind that this comparison only applies to this specific dataset and task, and different datasets and tasks may require different approaches and models. These preliminary performances are based on training and validation sets and may not reflect the performance of these models on new, unseen data. It is possible that the models may overfit to the training and validation data and not generalize well to new data.

Additionally, the comparison is limited in that it only considers accuracy as the sole metric of performance. Other metrics such as precision, recall, and F1-score may also be important to consider depending on the specific application. We will explore these performances in the next section.

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Relative Performance
Custom Model	0.1229	0.9531	0.0975	0.9667	High

Model	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Relative Performance
XGCustom Model	N/A	N/A	N/A	0.9969	High
VGG16	0.4662	0.7788	0.4422	0.8155	Medium
Improved VGG16	0.0407	0.9855	0.0246	0.9918	Best

7. Predictions and Performance

> Testing a model on metrics other than accuracy is important because accuracy alone does not provide a complete picture of a model's performance. While accuracy measures the proportion of correctly classified instances, it does not differentiate between true positive and true negative instances, nor does it account for imbalanced datasets. That is why in this section we will perform predictions of the models on test data and evaluate their performance on accuracy, precision, recall, and f1score.

Accuracy

- Accuracy is important in evaluating a model's overall performance in correctly identifying happy/sad sentiments in photos. However, accuracy alone may not provide a complete picture of the model's performance. For example, a model may achieve high accuracy by only predicting the majority class (e.g., predicting all photos as happy). This is why we will evaluate the models using additional metrics such as precision, recall, and F1 score.

Precision

- Precision measures the proportion of predicted positive instances (e.g., happy) that are actually positive. In the context of identifying happy/sad sentiments in photos, precision would help to determine how often the model correctly identifies a photo as happy, given that it predicted it to be happy. A high precision value indicates that the model is making fewer false positive errors.

Recall

- Recall measures the proportion of actual positive instances (e.g., happy) that are correctly predicted as positive by the model. In the context of identifying happy/sad sentiments in photos, recall would help to determine how often the model correctly identifies a happy photo, given that it actually is a happy photo. A high recall value indicates that the model is making fewer false negative errors.

F1 Score

- F1 score is a measure of the balance between precision and recall, and it provides a single score that represents both metrics. F1 score would help to determine the model's overall performance in correctly identifying both happy and sad photos. A high F1 score indicates that the model is making fewer errors in both precision and recall.

In [58]:
```
# create the evaluation function
def evaluate_model(model, data, show_cm=False):
    # create the evaluation metrics
    precision = Precision()
    recall = Recall()
    accuracy = BinaryAccuracy()

    # evaluate the model on the data set
    y_true = []
    y_pred = []

    for batch in data.as_numpy_iterator():
        # unpack batch data
        X, y = batch

        # make a prediction using our model
        y_hat = model.predict(X, verbose=False)

        # convert the predicted probabilities to class labels
        y_hat = (y_hat > 0.5).astype(int)

        # add the true and predicted labels to our lists
        y_true.extend(y)
        y_pred.extend(y_hat)

        # update our metrics
        precision.update_state(y, y_hat)
        recall.update_state(y, y_hat)
        accuracy.update_state(y, y_hat)

    # calculate f1score
    f1score = f1_score(y_true, y_pred)

    # print the evaluation metrics
    print(f'\nPrecision: {precision.result()*100:.2f}%\nRecall: {recall.result().numpy()*100:.2f}%\nF1 score: {f1score*100:.2f}%\nAccuracy:

    # compute and plot the confusion matrix if show_cm is True
    if show_cm:
        cm = confusion_matrix(y_true, y_pred)

        # plot the confusion matrix
        plot_confusion_matrix(cm, classes=['Sad', 'Happy'], normalize=True)
```

```
        # print the confusion matrix
        print("\nConfusion Matrix:\n", np.array2string(np.array(cm), separator=', ', formatter={'int': lambda x: f'{x:2d}'}))
```

Recap: What does the confusion matrix represent?

- The confusion matrix above represents a binary classification problem, where there are two possible classes.
- The actual class labels are on the vertical axis and the predicted labels are on the horizontal axis.
- In this case, the class labels are "0" and "1", and "0" is the positive class (happy) while "1" is the negative class (sad).

Recap: How do we interpret the confusion matrix?

- The first row of the confusion matrix shows the performance of the classifier on the positive class. The entry at (1,1) shows that the classifier predicted true positive samples, which means that the classifier correctly identified all positive samples if this value is all the positive samples. The entry at (1,2) shows that the classifier predicted false negative samples, which means that the classifier did not miss any positive samples if the value is 0.

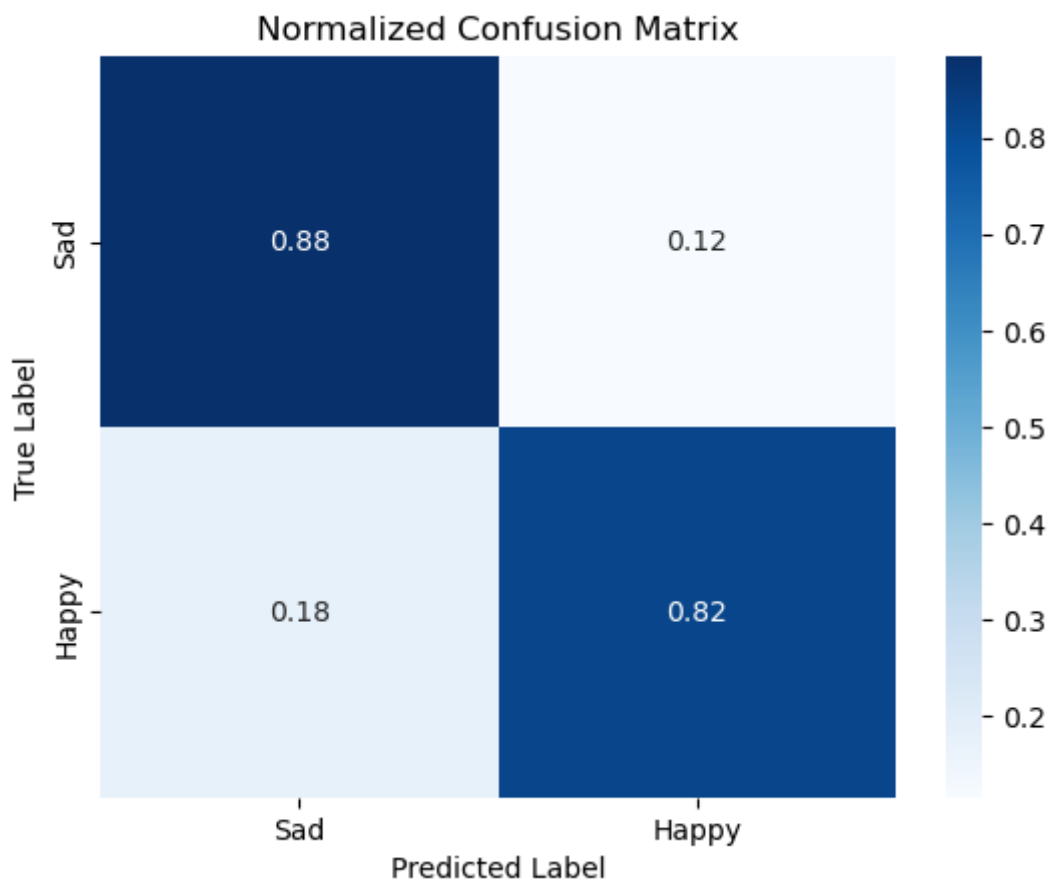
- The second row of the confusion matrix shows the performance of the classifier on the negative class. The entry at (2,1) shows that the classifier predicted false positive samples, which means that the classifier did not misclassify any negative samples if the value is 0. The entry at (2,2) shows that the classifier predicted true negative samples, which means that the classifier correctly identified all negative samples if the value is equal to the sampled negatives.

In [59]:
```
# create the confusion matrix plot function
def plot_confusion_matrix(cm, classes, normalize=False):
    cmap = plt.cm.Blues
    title = 'Normalized Confusion Matrix' if normalize else 'Confusion Matrix'
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    fig, ax = plt.subplots()
    sns.heatmap(cm, annot=True, cmap=cmap, xticklabels=classes, yticklabels=classes)
    ax.set_title(title)
    ax.set_ylabel('True Label')
    ax.set_xlabel('Predicted Label')
    plt.show()
```

Let's test the models' performance on the test dataset.

In [60]:
```
evaluate_model(custom_model, test_ds, show_cm=True)
```

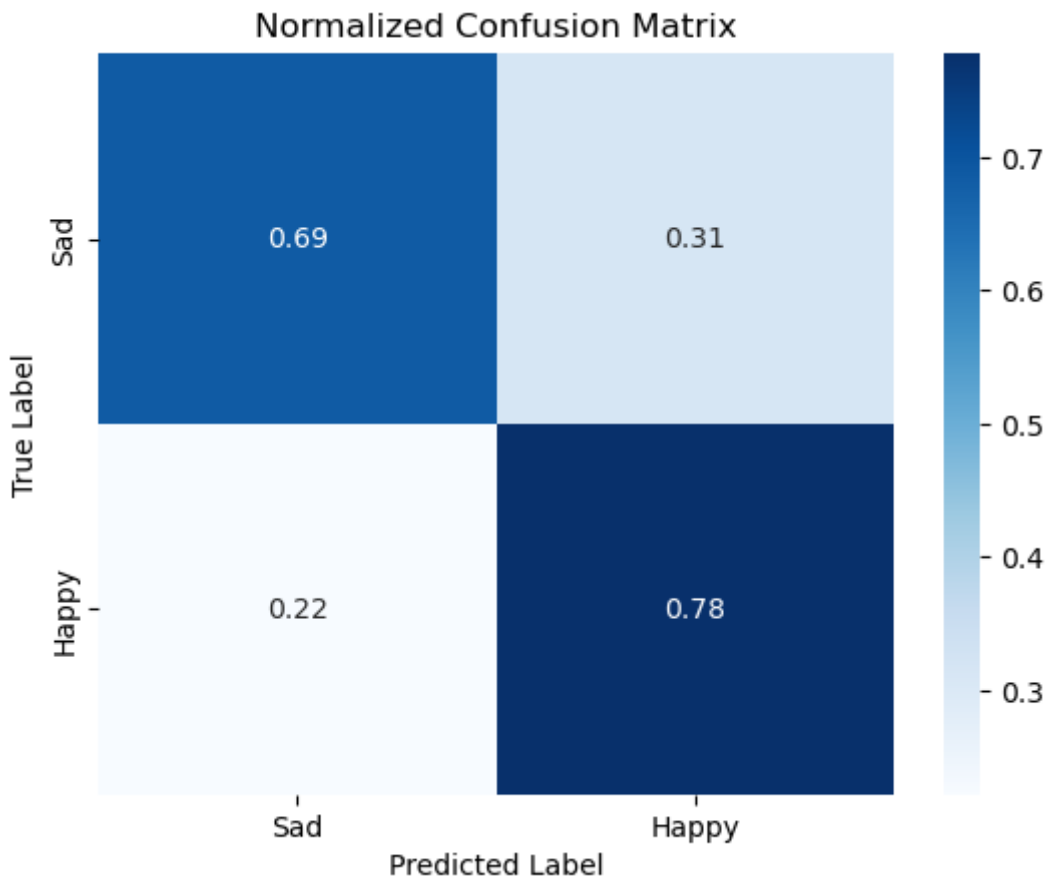
```
Precision: 87.63%
Recall: 82.12%
F1 score: 84.79%
Accuracy: 85.26%
```




```
Confusion Matrix:
 [[267, 35],
 [54, 248]]
```

In [45]:
```
evaluate_model(pretrained_model, test_ds, show_cm=True)
```

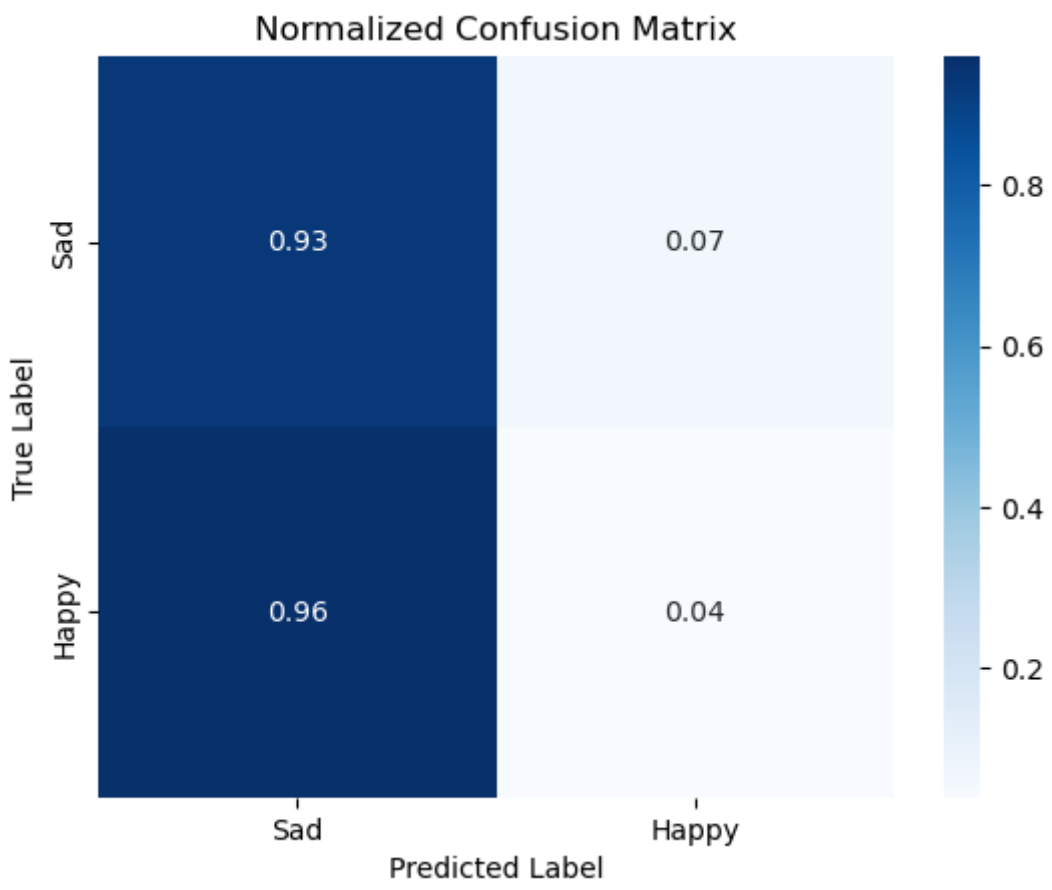
```
Precision: 71.21%
Recall: 77.81%
F1 score: 74.37%
Accuracy: 73.18%
```




```
Confusion Matrix:
 [[207, 95],
 [67, 235]]
```

In [56]:
```
evaluate_model(ipretrained_model, test_ds, show_cm=True)
```

```
Precision: 36.36%
Recall: 3.97%
F1 score: 7.16%
Accuracy: 48.51%
```



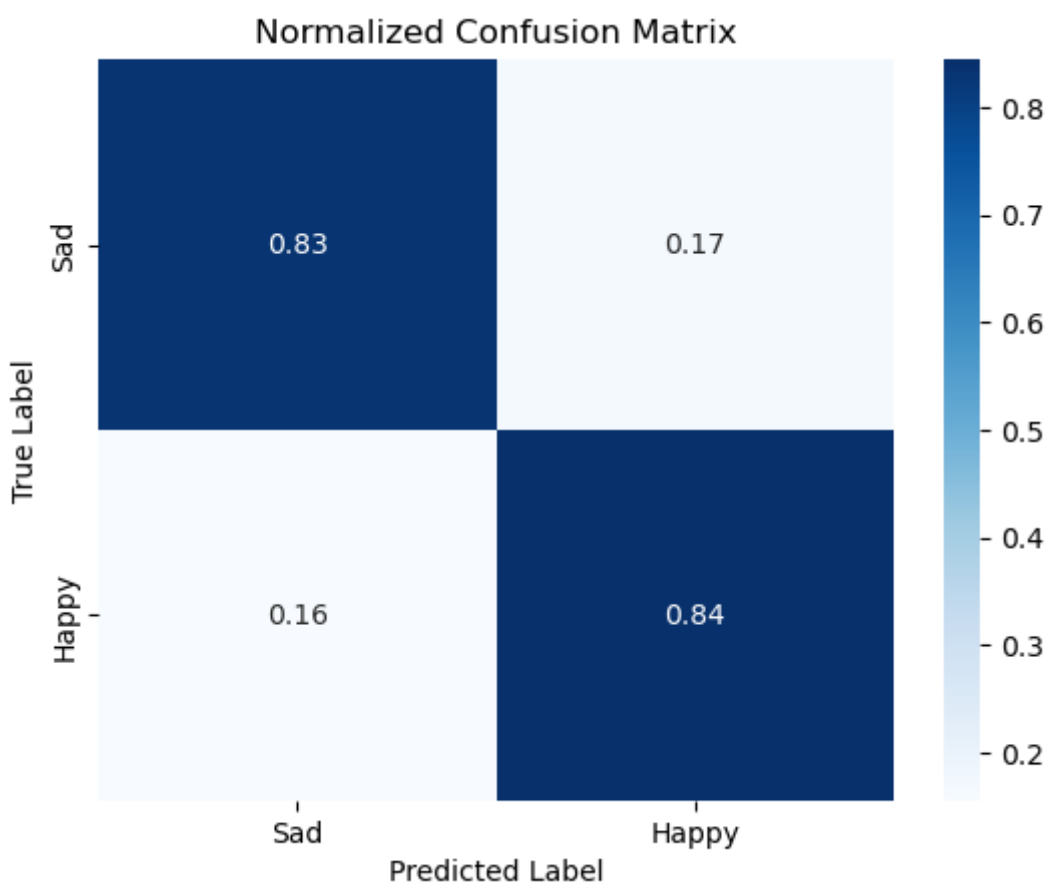

```
Confusion Matrix:
 [[281, 21],
 [290, 12]]
```

In [61]:
```
# Code to convert test_ds into a format that we can run on xgcustom_model
def convert_test_ds(test_ds):
    test_features, test_labels = [], []

    # Iterate over batches of images and labels in test dataset
    for image_batch, labels_batch in test_ds:
        # Extract features from images using the custom model
        features_batch = extract_features(image_batch)

        # Append the features and labels of the batch to the lists
        test_features.append(features_batch)
        test_labels.append(labels_batch.numpy())

    # Concatenate features and labels of all batches into single arrays
    test_features = np.concatenate(test_features)
    test_labels = np.concatenate(test_labels)

    return test_features, test_labels

def evaluate_tree_model(model, data, show_cm=False, tree=True):
    # create the evaluation metrics
    precision = Precision()
```

```
    recall = Recall()
    accuracy = BinaryAccuracy()

    # convert test_ds to a format that xgcustom_model accepts
    test_features, test_labels = convert_test_ds(data)

    # make a prediction using our model
    y_hat = model.predict(test_features)

    # convert the predicted probabilities to class labels
    y_pred = (y_hat > 0.5).astype(int)

    # update our metrics
    precision.update_state(test_labels, y_pred)
    recall.update_state(test_labels, y_pred)
    accuracy.update_state(test_labels, y_pred)

    # calculate f1score
    f1score = f1_score(test_labels, y_pred)

    # print the evaluation metrics
    print(f'\nPrecision: {precision.result()*100:.2f}%\nRecall: {recall.result().numpy()*100:.2f}%\nF1 score: {f1score*100:.2f}%\nAccuracy:

    # compute and plot the confusion matrix if show_cm is True
    if show_cm:
        cm = confusion_matrix(test_labels, y_pred)

        # plot the confusion matrix
        plot_confusion_matrix(cm, classes=['Sad', 'Happy'], normalize=True)

        # print the confusion matrix
        print("\nConfusion Matrix:\n", np.array2string(np.array(cm), separator=', ', formatter={'int': lambda x: f'{x:2d}'}))
```

In [48]:
```
evaluate_tree_model(xgcustom_model, test_ds, show_cm=True)
```

```
Precision: 83.33%
Recall: 84.44%
F1 score: 83.88%
Accuracy: 83.77%
```




```
Confusion Matrix:
 [[251, 51],
 [47, 255]]
```

8. Discussion of Performance

The performance results on the testing dataset show that the custom_model achieved the highest performance with an accuracy of 85.26%, followed by the xgcustom_model with an accuracy of 83.77%. The pretrained_model had an accuracy of 73.18%, and the ipretrained_model had an accuracy of 48.51%. When compared to their performance on the training and validation data, we can observe that the models performed slightly worse on the testing data. This is expected as the testing data is unseen data, and the models may not generalize well to it.

The custom_model performed consistently well across all three datasets with an accuracy of over 95% on the training data and 96.67% on the validation data. The xgcustom_model also performed well on the training and validation data with an accuracy of over 99%. The pretrained_model had an accuracy of 79.22% on the training data and 81.66% on the validation data. The ipretrained_model had an accuracy of 98.87% on the training data and 50.42% on the validation data.

Overall, we can observe that the custom_model and xgcustom_model performed well across all three datasets. The pretrained_model and ipretrained_model did not perform as well on the testing data, and this is likely due to overfitting to the training data.

Model	Precision	Recall	F1 Score	Accuracy	Relative Performance
Custom Model	87.63%	82.12%	84.79%	85.26%	Medium
XGCustom Model	83.33%	84.44%	83.88%	83.77%	Medium
VGG16	71.21%	77.81%	74.37%	73.18%	Low
Improved VGG16	36.36%	3.97%	7.16%	48.51%	Very Low

In the table, we can observe the number of true positives (correctly predicted positives), false positives (incorrectly predicted positives), false negatives (incorrectly predicted negatives), and true negatives (correctly predicted negatives) for each model. When comparing the models based on their true positive rate, we can see that the Custom Model performed the best with 267 true positives, while the Improved Pretrained Model had the lowest true positive rate with only 12 true positives. However, the Improved Pretrained Model had the lowest false positive rate with only 21 false positives.

In summary, we observe that the Custom Model has the highest number of true positives among the models, indicating that it correctly predicted the most positive instances. However, it also had a higher number of false positives, meaning it incorrectly predicted more positive instances than the other models. On the other hand, the Improved Pretrained Model had the lowest false positive rate, meaning it made fewer incorrect positive predictions than the other models, but it had the lowest true positive rate.

These results suggest that the Custom Model may have a tendency to overfit the training data and hence, is predicting more positive instances than necessary, while the Improved Pretrained Model is not making as many positive predictions as the other models, indicating that it may be underfitting the training data. Overall, the Custom Model may have higher precision but lower recall compared to the other models, while the Improved Pretrained Model may have lower precision but higher recall.

Model	True Positive	False Positive	False Negative	True Negative
Custom Model	267	35	54	248
XGCustom Model	251	51	47	255
VGG16	207	95	67	235
Improved VGG16	281	21	290	12

How do we expect the models to perform on personal data?

Based on the results, the Custom Model appears to be the best model for identifying happy or sad sentiments in photos. This is because it had the highest true positive rate, meaning it correctly predicted the most positive instances, which in this case would be identifying happy sentiments in photos. However, it is important to note that the Custom Model also had a higher false positive rate than the Improved Pretrained Model, meaning it may have a tendency to incorrectly predict more positive instances than necessary, which in this case would be identifying happy sentiments in photos where there are none. Therefore, while the Custom Model has a higher true positive rate, it is important to consider the potential for false positives as well.

Can model ensembling improve our performance?

Ensemble learning is a method that combines multiple models to make better predictions. When two or more models are combined in this way, it is known as **model ensembling**. One popular technique for model ensembling is called "voting," where each model gets one vote and the final prediction is based on the majority vote. Ensemble learning can be a useful technique for improving model performance, especially when the individual models have different strengths and weaknesses. Based on the results of the individual models, it is possible that using ensemble learning could improve overall performance in identifying happy or sad sentiments in photos.

Which models would we ensemble to improve our performance?

- We would need to consider the performance of each individual model as well as their diversity.
- We want to choose models that perform well on their own but also have different strengths and weaknesses so that they can complement each other when combined.
- Based on the results, the Custom Model and XGCustom Model have similar performance on most metrics, but the Custom Model has a higher true positive rate and the XGCustom Model has a higher true negative rate. The Pretrained Model has lower performance than the Custom and XGCustom models, while the Improved Pretrained Model has the lowest overall performance.
- A robust ensemble voting strategy could include the Custom Model and XGCustom Model since they have similar overall performance but different strengths in terms of their true positive and true negative rates. The Pretrained Model could also be included to provide additional diversity to the ensemble, but it may not contribute as strongly to the overall performance.

And now... for the most important part - model performance on my data

Now that we have developed this corpus of information on custom models and pretrained models, let us see how they fair on classification of my images. We will create a function that takes an image and predicts its classification on all models. The function should return a table of the results.

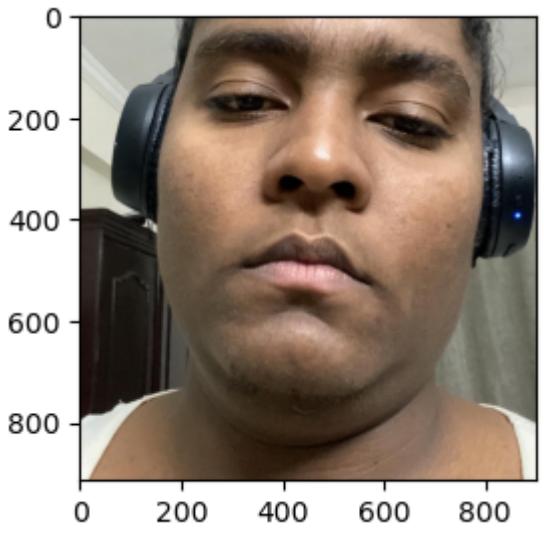
In [316…

```
def process_image(image_path, image_size=(48,48)):
    # Open image
    img = cv2.imread(image_path)

    # Resize image
    img = cv2.resize(img, image_size)
```

```
    # Convert image to RGB format
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Extract features using simple_model
    features = custom_model(np.expand_dims(img, axis=0))

    # Concatenate features
    img_array = np.concatenate(features)

    return img_array

def tag_image(value):
    value = float(value.split('%')[0])
    tag = ""

    if value <= 50:
        tag = "Sad"
    else:
        tag = "Happy"

    return tag

def predict_image_class(image_path, image_label, show_image=False):
    # Filter the models based on input using list slicing
    models = {
        "custom":custom_model,
        "xgcustom":xgcustom_model,
        "pretrained":pretrained_model,
        "ipretrained":ipretrained_model
    }

    #read image
    img = cv2.imread(image_path)

    # resize image
    resize = tf.image.resize(img, (48,48))

    if show_image:
        #plot original image
        plt.figure(figsize=(3, 3))
        plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        plt.show()

    # classify
    image = np.expand_dims(resize/255, axis=0)

    # Predict the image class on the models
    results = []
    predicted = []
    for i, (name, model) in enumerate(models.items(), start=1):
        if name == "xgcustom":
            pred = model.predict(process_image(image_path))
            y_hat = '{:.2f}%'.format(round(float(pred[0])*100, 2))
            results.append(y_hat)
            predicted.append(tag_image(y_hat))
        else:
            pred = model.predict(image)
            y_hat = '{:.2f}%'.format(round(float(pred[0])*100, 2))
            results.append(y_hat)
            predicted.append(tag_image(y_hat))

    # Create table with columns model and result
    table = pd.DataFrame({
        'No.': range(1, len(models) + 1),
        'Model Name': ['custom_model', 'xgcustom_model', 'pretrained_model', 'ipretrained_model'],
        'Result': results,
        'Predicted': predicted,
        'Actual': [image_label, image_label, image_label, image_label]
    }).set_index(['No.', 'Model Name', 'Result', 'Predicted', 'Actual'])

    return table
```

In [317…

```
predict_image_class('downloads/photo1.jpeg', "Sad", True)
```



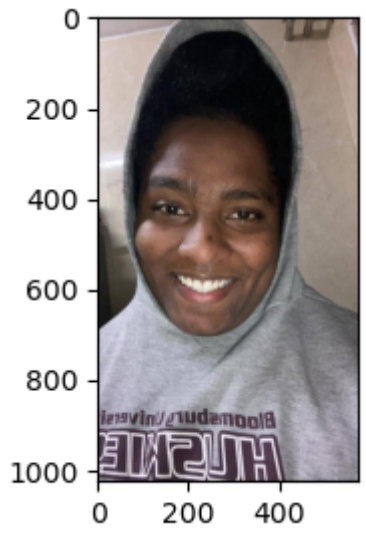
```
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 92ms/step
1/1 [==============================] - 0s 63ms/step
```

Out[317]:

No.	Model Name	Result	Predicted	Actual
1	custom_model	0.00%	Sad	Sad
2	xgcustom_model	0.00%	Sad	Sad
3	pretrained_model	49.92%	Sad	Sad
4	ipretrained_model	21.38%	Sad	Sad

In [318…

```
predict_image_class('downloads/photo2.jpeg', "Happy", True)
```



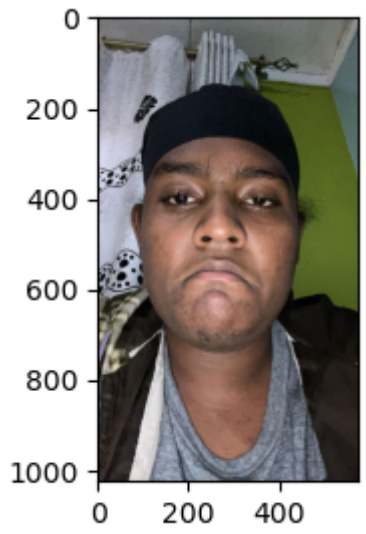

```
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 78ms/step
1/1 [==============================] - 0s 68ms/step
```

Out[318]:

No.	Model Name	Result	Predicted	Actual
1	custom_model	100.00%	Happy	Happy
2	xgcustom_model	100.00%	Happy	Happy
3	pretrained_model	86.09%	Happy	Happy
4	ipretrained_model	80.47%	Happy	Happy

In [320…

```
predict_image_class('downloads/photo3.jpeg', "Sad", True)
```



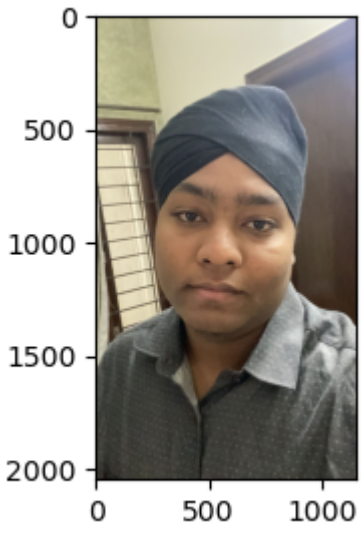

```
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 78ms/step
1/1 [==============================] - 0s 54ms/step
```

Out[320]:

No.	Model Name	Result	Predicted	Actual
1	custom_model	0.31%	Sad	Sad
2	xgcustom_model	100.00%	Happy	Sad
3	pretrained_model	51.23%	Happy	Sad
4	ipretrained_model	34.28%	Sad	Sad

In [321…

```
predict_image_class('downloads/photo4.jpeg', "Happy", True)
```




```
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 86ms/step
1/1 [==============================] - 0s 70ms/step
```

Out[321]:

No.	Model Name	Result	Predicted	Actual
1	custom_model	96.64%	Happy	Happy
2	xgcustom_model	100.00%	Happy	Happy
3	pretrained_model	32.74%	Sad	Happy
4	ipretrained_model	29.37%	Sad	Happy

In [322…

```
predict_image_class('downloads/photo5.jpeg', "Happy", True)
```


```
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 74ms/step
1/1 [==============================] - 0s 64ms/step
```

Out[322]:

No.	Model Name	Result	Predicted	Actual
1	custom_model	91.99%	Happy	Happy
2	xgcustom_model	100.00%	Happy	Happy
3	pretrained_model	78.20%	Happy	Happy
4	ipretrained_model	60.86%	Happy	Happy

Only custom model predicts all the images right. It seems the search for true emotion detection is afoot- watch out for tutorial 3! On a lighter note, I think one of the counfounding variables in these examples is my turban, which might be throwing off the models.

9. Summary

> In this project, we conducted a comparative analysis of various models for image classification using the FER2013 dataset. We started by adapting a custom CNN model for this task and then explored the use of pre-trained models such as VGG16 and ResNet. We fine-tuned VGG16 to create an Improved VGG16 model and observed the effect of hyperparameter tuning on its performance. We also created XGCustom Model by adding XGBoost to the custom model to match the performance of Improved VGG16.
>
> We evaluated the models on multiple performance metrics such as training and validation accuracy, precision, recall, F1 score, and confusion matrices. We also contextualized the findings to the task at hand and discussed the importance of hyperparameter tuning and regularizing models to achieve better performance.
>
> The three main takeaways from this project are: (1) fine-tuning pre-trained models on relevant datasets can help improve their performance on specific tasks, (2) hyperparameter tuning is crucial for achieving better performance, but high training scores do not necessarily translate to high performance on testing, and (3) evaluating models on multiple metrics and datasets can help ensure that they do not overfit the training dataset and have good generalization capabilities. I go into more detail below.

9.1. Using XGBoost on a Model

In the previous paper and this one, we have observed that our custom model consistently performs well, even when trained on different datasets such as the Google search dataset and the FER2013 dataset. However, when we created the XGCustom Model by incorporating XGBoost on top of the learned features of our Custom Model, we noticed that it only performed better than the original model in training, but not in testing. In fact, the results suggest that adding XGBoost to an already well-performing model may not have a significant impact and could even lead to overfitting on the training data, resulting in lower performance on the testing data. One way to further test this hypothesis would be to create a purposely weaker model, add XGBoost to it, and compare its performance to that of the original weak model.

9.2. Interpretation of Performance Metrics

The results of the model performance on the test dataset demonstrate the importance of considering multiple performance metrics when evaluating the effectiveness of a model. While the Custom Model and XGCustom Model initially performed well in terms of accuracy on the training and validation datasets, their performance on precision, recall, and F1 score when evaluated on the test dataset was only medium. Conversely, while the Improved VGG16 had the best performance on precision, recall, and F1 score, its accuracy was only slightly higher than the Custom Model and XGCustom Model. This emphasizes the importance of contextualizing performance metrics to the problem at hand. For example, our goal is to develop a model to identify happy/sad emotions in images, therefore a high recall score may be more important than high accuracy, as false negatives (misclassifying sad images as happy) may be more detrimental than false positives (misclassifying happy images as sad). Therefore, a model with a high recall score, even if it has lower accuracy, may be more effective at correctly identifying sad images. Hence, multiple metrics, such as accuracy, precision, recall, and F1 score, provide a more comprehensive understanding of a model's effectiveness and can inform improvements to the model.

9.3. Hypertuning Pretrained Models

Hyperparameter tuning is an important step in achieving better performance from pre-trained models. In this project, we explored the impact of changing hyperparameters, such as the loss function and learning rate, on the performance of VGG16 for our specific task of image classification. By changing these hyperparameters, we were able to create Improved VGG16, which had the best relative performance on validation accuracy and training accuracy compared to other models.

However, we also found that high training scores do not necessarily guarantee high performance on the testing dataset, and that overfitting is a common issue that can negatively impact the model's generalization capabilities. To avoid overfitting and get more robust models from pre-trained models, it is essential to follow best practices such as fine-tuning on a relevant dataset, choosing the right hyperparameters through experimentation and validation, and using regularization techniques such as dropout and early stopping.

During our experiments, we also explored the effect of changing the loss function and learning rate of the pre-trained model on our task of image classification. We found that while mean squared error (MSE) is a valid loss function, it was not suitable for our specific task and resulted in poor performance. This is because MSE penalizes large errors more than small ones, which may not be desirable for classification tasks where small errors are also important. Therefore, we concluded that binary cross-entropy is a more appropriate loss function for our task of image classification.

10. References

Jiao, W., Hao, X., & Qin, C. (2021). The Image Classification Method with CNN-XGBoost Model Based on Adaptive Particle Swarm Optimization. Information, 12(4), 156.
https://doi.org/10.3390/info12040156

Jonaac. (2021). Deep XGBoost Image Classifier. GitHub.
https://github.com/jonaac/deep-xgboost-image-classifier

Thongsuwan, S., Jaiyen, S., Padcharoen, A., & Agarwal, P. (2021). ConvXGB: A new deep learning model for classification problems based on CNN and XGBoost. Nuclear Engineering and Technology, 53(2), 522-531.
https://doi.org/10.1016/j.net.2020.04.008

Ren, X., Guo, H., Li, S., Wang, S., & Li, J. (2017). A Novel Image Classification Method with CNN-XGBoost Model. In Digital Forensics and Watermarking (pp. 378-390). Springer.
https://link.springer.com/chapter/10.1007/978-3-319-64185-0_28

11. Appendix

A. GPT Declaration

> I used ChatGPT in this assignment for rewriting the technical sections to fact check my responses. To ensure ChatGPT was using the right information, I used the prompt, "Imagine you are an expert in machine learning skilled with communicating technical concepts in simple language. You are responsible for fact checking the information that is provided to you. Train these resources to ensure your knowledge base is updated." The resources mentioned in the prompt are the references I have provided in the reference section above. This was an interesting way to use ChatGPT for me, as in the media a lot of the content has surrounded the accuracy and completeness of the responses ChatGPT provides, and therefore I would fact check the fact-checker.

B. HCs

> **#descriptivestats:** I applied this HC by compiling the summary statistics of the image shapes in the processed FER2013 dataset. I accurately calculate the range (minimum, maximum), mean, median, and standard deviation of the image shapes. I then interpreted the results and provide a well-reasoned and justified interpretation of the statistics, including the importance of consistency in image size and dimensions for model training. Additionally, I create data visualization to display the descriptive statistics: a histogram (1D and 2D) and a kernel density estimate to visualize the frequency and density distributions of the image widths and heights respectively. The 2D histogram combines the image heights and image widths but I notice this visualization is ineffective at showing the variability in the image heights, therefore I visualize them seperately. The histogram is appropriate for visualizing the frequency distributions, while the KDE is appropriate for visualizing the density of the image heights. I elect to use a histogram as the image widths all have the same value of 48 and a KDE would not be appropriate. I use

> these multiple descriptive statistics and visualizations to create a robust analysis which suggest that the images in the dataset are fairly consistent in terms of size and dimensions, which is important for model training.

> **#decisiontrees:** I apply this HC in the novel context of delving into how XGBoost utilizes decision trees for classification problems. I make sure to address the facets listed in the rubric by explaining how XGBoost creates each decision tree using clear, detailed steps and how this grows from a single tree to multiple trees. I explore how XGBoost evaluates the outcomes of a decision tree resulting from specific decision strategies such as ensembling where the final decision is based on a weighted average of the individual tree predictions and metrics it uses such as negative gradient to correct the errors from the previous trees. To address the facet of perfect and imperfect information, I describe how XGBoost handles noisy or missing data through gradient-based optimization. I additionally describe the mathematics that makes this possible in clear, detailed steps and use XGBoost in my custom model, where I interpret and compare its results. While this is an unconventional use of the HC, I have justified my application in the detail to which the rubric was applied to this novel use case.