National College of Ireland

# Configuration Manual

MSc Research Project

Data Analytics


## Karthika Nair

Student ID: 22105522

School of Computing

National College of Ireland


Supervisor: Shubham Subhnil

**National College of Ireland**

**MSc Project Submission Sheet**

**School of Computing**

| | | | |
|---|---|---|---|
| **Student Name:** | Karthika Nair | | |
| **Student ID:** | 22105522 | | |
| **Programme:** | MSc. Data Analytics | **Year:** | 2023 |
| **Module:** | MSc. Research Project | | |
| **Supervisor:** | Shubham Subhnil | | |
| **Submission Due Date:** | 14-12-2023 | | |
| **Project Title:** | Deepfake Detection: Comparison of Pretrained Xception and VGG16 Models - Configuration Manual | | |
| **Word Count:** | 1136 | **Page Count:** | 15 |

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

| | |
|---|---|
| **Signature:** | Karthika Nair |
| **Date:** | 14-12-2023 |

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST**

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

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

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |

# Deepfake Detection: Comparison of Pretrained Xception and VGG16 Models

# Configuration Manual

*Karthika Nair*

*22105522*

## 1. Introduction

This configuration manual will include all the steps that have been taken technically to achieve the goals of this research project, i.e., detection of deepfakes with a comparison of pretrained models such as Xception and VGG16 as suggested by (Kaushal, Singh, Negi, & Chhaukar, 2022). The following sections will encompass the system setup to aid the execution, exploratory data analysis procedures, the implementation of the model, and their evaluations. The code snippets will also be attached consecutively for a better understanding.

## 2. System Configuration – Software and Hardware Pre-requisites

This section will include the system configuration utilised for this study.

### 2.1 Hardware Configuration

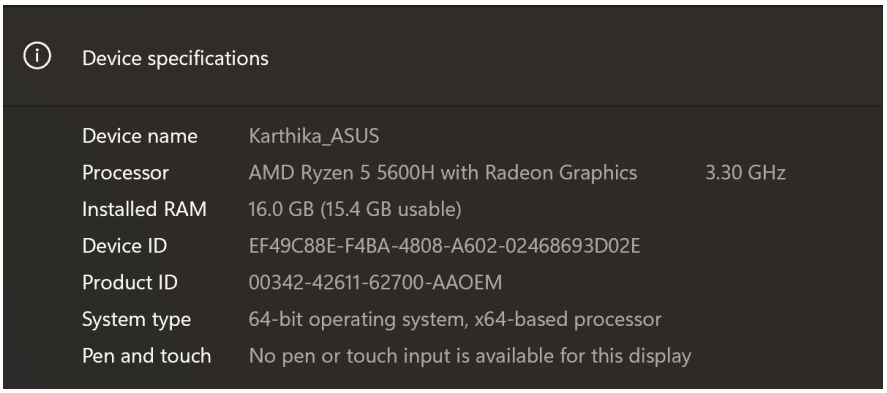



*Fig. 1 Hardware specifications*

### 2.2 Software Configuration

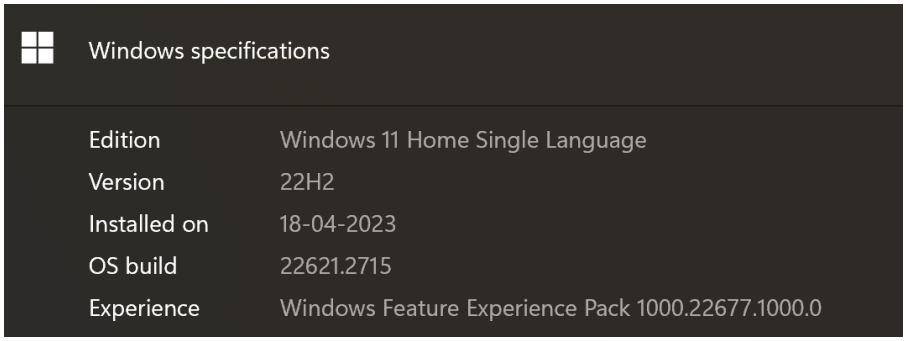



*Fig. 2 Software specifications*

### 2.3 Development Environment

Python programming language has been implemented by combining VS Code and other tools.

- *Python Distribution*: Anaconda (Version - 23.1.0)

- *IDE*: Visual Studio Code (Version - 1.84.2)
  VS Code is used as a primary IDE as it enables a lightweight yet powerful environment for code editing, version control integration, and good support for package extensions.

- *Notebook Environment*: Jupyter Notebook (Version - 6.5.2)

Below mentioned are the Python libraries used: Numpy, Keras, Pandas, Matplotlib, Sklearn, Tensorflow (VS Code).

## 3. Datasets Used

The datasets used for this study have been collected from open source Kaggle which allows studying of deepfake detection.

Dataset source: Deep Fake Detection on Images and Videos | Kaggle

Dataset information:

- *faces_224.zip* – collection of faces extracted from the DFDC. All the images are of size 224 x 224.
- *metadata.csv* – list of filenames, label (REAL or FAKE), original filename.

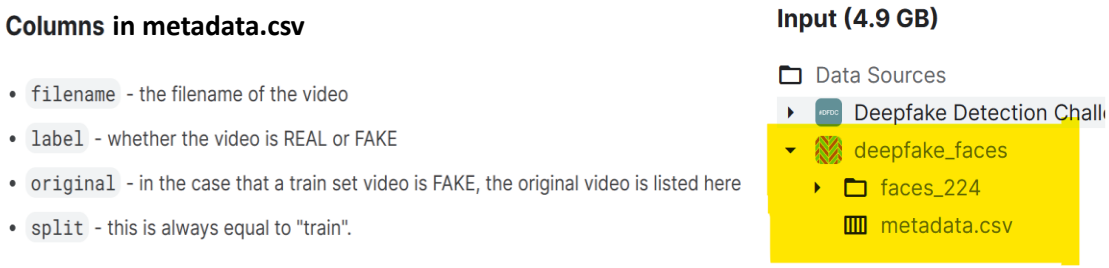



*Fig. 3 Dataset information*

## 4. Python packages and Libraries Used

The below table can be used as a reference to the Python libraries used in this study to achieve the development of a model that detects deepfakes and distinguishes between real and fake content accurately:

| Python Library Name | Description |
|---|---|
| Pandas | Ideal for working with data structures like DataFrame and Series |
| Numpy | Support for large, multi-dimensional arrays with mathematical operations on them |
| Sklearn | Used for data mining and data analysis including classification, regression, etc. |
| Tensorflow | Used for building and training deep learning models. |
| Matplotlib | 2D plotting library in Python used for data visualisations. |
| tensorflow.keras.applications | VGG16 |
| tensorflow.keras.layers | Conv2D |
| cv2 | Open-source computer vision library |

*Fig. 4 Python libraries and packages used*

```
import numpy as np
import pandas as pd
import sys
import sklearn
import tensorflow as tf

import cv2
import pandas as pd
import numpy as np

import plotly.graph_objs as go
from plotly.offline import iplot
from matplotlib import pyplot as plt
```

*Fig. 5 Import statements*

## 5. Data Pre-processing and Visualisation Code

- The function “extract_metadata()” is used to read the data within “metadata.csv” and the below code snippet shows a visualisation of how the information in the file looks like.

Visualisation

```
import os

def extract_metadata():
    return pd.read_csv("C:/Users/karth/PERSONAL/Data_Analytics/SEMESTER-3/Dataset/newDataset/metadata/metadata.csv")

meta_data=extract_metadata()
meta_data.head()
```

| | videoname | original_width | original_height | label | original |
|---|---|---|---|---|---|
| 0 | aznyksihgl.mp4 | 129 | 129 | FAKE | xnojggkrxt.mp4 |
| 1 | gkwmalrvcj.mp4 | 129 | 129 | FAKE | hqqmtxvbjj.mp4 |
| 2 | lxnqzocgaq.mp4 | 223 | 217 | FAKE | xjzkfqddyk.mp4 |
| 3 | itsbtrrelv.mp4 | 186 | 186 | FAKE | kqvepwqxfe.mp4 |
| 4 | ddvgrczjno.mp4 | 155 | 155 | FAKE | pluadmqqta.mp4 |

*Fig. 6 Import statements*

- The dataset contains 95k images. To make the visualisation and modelling easier, we take a sample size of 16,000 real and fake images each.

```
deepfake_REAL = meta_data[meta_data["label"] == "REAL"]
deepfake_FAKE = meta_data[meta_data["label"] == "FAKE"]
sample_size = 8000

deepfake_REAL = deepfake_REAL.sample(sample_size, random_state=42)
deepfake_FAKE = deepfake_FAKE.sample(sample_size, random_state=42)

metaSample = pd.concat([deepfake_REAL, deepfake_FAKE])
```

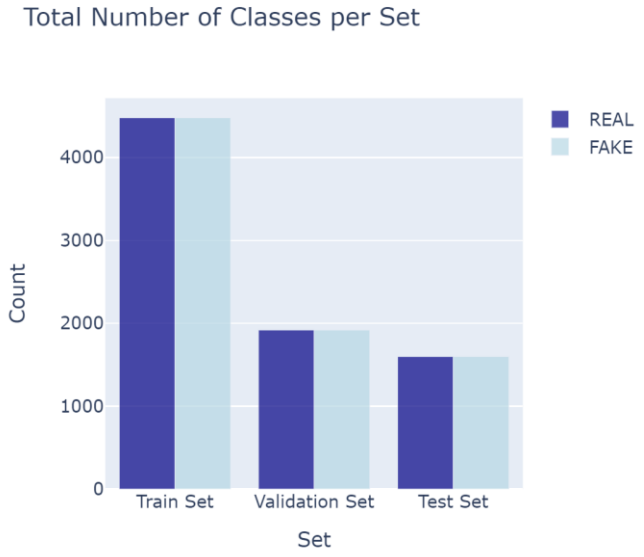
*Fig. 7 Sample size (16,000) for dataset*

- The dataset is then split into training, testing and validation sets.

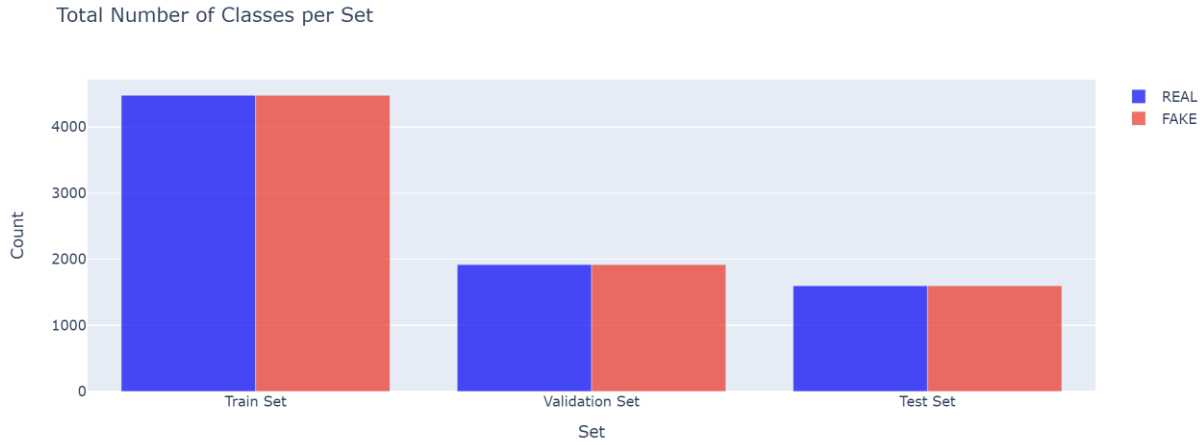
```
from sklearn.model_selection import train_test_split

Train_set, Test_set = train_test_split(metaSample,test_size=0.2,random_state=42,stratify=metaSample['label'])
Train_set, Validation_set  = train_test_split(Train_set,test_size=0.3,random_state=42,stratify=Train_set['label'])
```

*Fig. 8 Dataset splitting*

- Data visualization – Number of classes (Real/Fake) in each set




*(a)*




*(b)*

*Fig. 9 Classes per Set in (a) DeepfakeDetection_XceptionCode.ipynb*

*(b) DeepfakeDetection_VGG16Code.ipynb*

## 6. Model Implementation Code

The implementation for this research is carried out in two Jupyter Notebook Python files to compare two pretrained models' performances, namely Xception and VGG16. The model training using CNN as the baseline model is the same for both ipynb files. The change is implemented when model fitting is done.

The following code snippets depict how the model development is implemented for both models:

### 6.1 Model Training – Using CNN as baseline model

- The function pull_data(set_name) is responsible for retrieving the dataset (images) and assigning labels for each of the image files. The label for an image that is labelled as 'FAKE' is assigned 1 and 0 for 'REAL'.

Using CNN Model Training

```
def pull_dataset(set_name):
    images,labels=[],[]
    for (img, imclass) in zip(set_name['videoname'], set_name['label']):
        images.append(cv2.imread("C:/Users/karth/PERSONAL/Data_Analytics/SEMESTER-3/Dataset/newDataset/faces_jpg/"+img[:-4]+".jpg"))
        if(imclass=='FAKE'):
            labels.append(1)
        else:
            labels.append(0)

    return np.array(images),np.array(labels)
```

*Fig. 10 Code snippet for retrieving dataset as DataFrame through pull_dataset(set_name)*

- Next step is to split the dataset into training, testing, and validation sets. pull_dataset() assigns binary labels to each of the sets and splits the corresponding dataset to X_train, y_train, X_val, y_val, X_test, and y_test datasets.

```
X_train,y_train=pull_dataset(Train_set)
X_val,y_val=pull_dataset(Validation_set)
X_test,y_test=pull_dataset(Test_set)
```

*Fig. 11 Code snippet for splitting dataset*

- Model Training –

```
from functools import partial

tf.random.set_seed(42)
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                        activation="relu", kernel_initializer="he_normal")

model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[224, 224, 3]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=1, activation="sigmoid")
])
```

*Fig. 12 Model training using CNN*

- Model compile and summary –

```
model.compile(loss="binary_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 224, 224, 64)      9472

 max_pooling2d (MaxPooling2  (None, 112, 112, 64)      0
 D)

 conv2d_9 (Conv2D)           (None, 112, 112, 128)     73856

 conv2d_10 (Conv2D)          (None, 112, 112, 128)     147584

 max_pooling2d_1 (MaxPoolin  (None, 56, 56, 128)       0
 g2D)

 flatten (Flatten)           (None, 401408)            0

 dense_2 (Dense)             (None, 128)               51380352

 dropout (Dropout)           (None, 128)               0

 dense_3 (Dense)             (None, 64)                8256

 dropout_1 (Dropout)         (None, 64)                0
...
Total params: 51619585 (196.91 MB)
Trainable params: 51619585 (196.91 MB)
Non-trainable params: 0 (0.00 Byte)
```

*Fig. 13 Model compiling and summary output*

- Model Fitting –

The below code snippet is for the model fitting which runs for 10 epochs and 140 cycles. The dropout layers help in preventing overfitting. The dropout value considered for this training is 0.5.

```
history = model.fit(X_train, y_train, epochs=10,batch_size=64,
                    validation_data=(X_val, y_val))
```

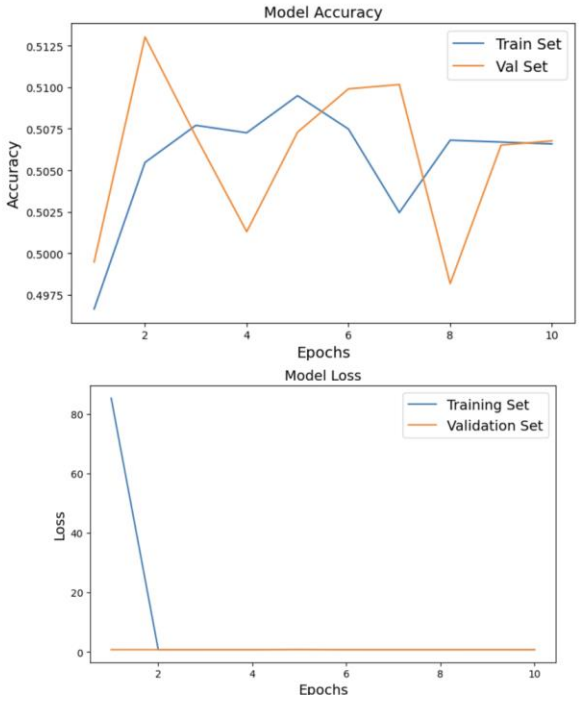
```
Epoch 1/10
140/140 [==============================] - 1209s 9s/step - loss: 85.2157 - accuracy: 0.4967 - val_loss: 0.6932 - val_accuracy: 0.4995
Epoch 2/10
140/140 [==============================] - 1219s 9s/step - loss: 0.6928 - accuracy: 0.5055 - val_loss: 0.6929 - val_accuracy: 0.5130
Epoch 3/10
140/140 [==============================] - 1430s 10s/step - loss: 0.6945 - accuracy: 0.5077 - val_loss: 0.6931 - val_accuracy: 0.5070
Epoch 4/10
140/140 [==============================] - 1284s 9s/step - loss: 0.6994 - accuracy: 0.5073 - val_loss: 0.6930 - val_accuracy: 0.5013
Epoch 5/10
140/140 [==============================] - 1088s 8s/step - loss: 0.7517 - accuracy: 0.5095 - val_loss: 0.6930 - val_accuracy: 0.5073
Epoch 6/10
140/140 [==============================] - 875s 6s/step - loss: 0.6931 - accuracy: 0.5075 - val_loss: 0.6953 - val_accuracy: 0.5099
Epoch 7/10
140/140 [==============================] - 896s 6s/step - loss: 0.6928 - accuracy: 0.5025 - val_loss: 0.6924 - val_accuracy: 0.5102
Epoch 8/10
140/140 [==============================] - 883s 6s/step - loss: 0.6940 - accuracy: 0.5068 - val_loss: 0.6921 - val_accuracy: 0.4982
Epoch 9/10
140/140 [==============================] - 867s 6s/step - loss: 0.7084 - accuracy: 0.5067 - val_loss: 0.6932 - val_accuracy: 0.5065
Epoch 10/10
140/140 [==============================] - 869s 6s/step - loss: 0.6959 - accuracy: 0.5066 - val_loss: 0.6938 - val_accuracy: 0.5068
```

*Fig. 14 Model fitting*

- Model Performance and Evaluation –

The performance of CNN model is tested and an accuracy of 50% is observed. This value is not essentially the best and that's why a pretrained model is chosen for finetuning. This will be explored in the next section.

```
acc = history.history['accuracy']
valAccuracy = history.history['val_accuracy']
loss = history.history['loss']
valLoss = history.history['val_loss']
epochsRange = range(1, len(history.epoch) + 1)

plt.figure(figsize=(15,5))

plt.subplot(1, 2, 1)
plt.plot(epochsRange, acc, label='Train Set')
plt.plot(epochsRange, valAccuracy, label='Val Set')
plt.legend(loc="best")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochsRange, loss, label='Training Set')
plt.plot(epochsRange, valLoss, label='Validation Set')
plt.legend(loc="best")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Model Loss')

plt.tight_layout()
```




```
score = model.evaluate(X_test, y_test)
```

```
100/100 [==============================] - 246s 2s/step - loss: 122.7304 - accuracy: 0.5000
```

*Fig. 15 Model performance*

### 6.2 Model Training – Finetuning using Xception pretrained model

- Using TensorFlow, datasets are generated from training, validation and testing sets.

# Using Pretrained Model - Xception

```
# creating datasets from training, validation and testing sets using TensorFlow

train_set_tensor=tf.data.Dataset.from_tensor_slices((X_train,y_train))
valid_set_tensor=tf.data.Dataset.from_tensor_slices((X_val,y_val))
test_set_tensor=tf.data.Dataset.from_tensor_slices((X_test,y_test))
```

*Fig. 16 Dataset generation using TensorFlow*

- Data Augmentation –

```
tf.keras.backend.clear_session()

batch_size = 32
preprocess = tf.keras.applications.xception.preprocess_input
train_set = train_set_tensor.map(lambda X, y: (preprocess(tf.cast(X, tf.float32)), y))
train_set = train_set.shuffle(1000, seed=42).batch(batch_size).prefetch(1)
valid_set = valid_set_tensor.map(lambda X, y: (preprocess(tf.cast(X, tf.float32)), y)).batch(batch_size)
test_set = test_set_tensor.map(lambda X, y: (preprocess(tf.cast(X, tf.float32)), y)).batch(batch_size)
```

*Fig. 17 Data Augmentation*

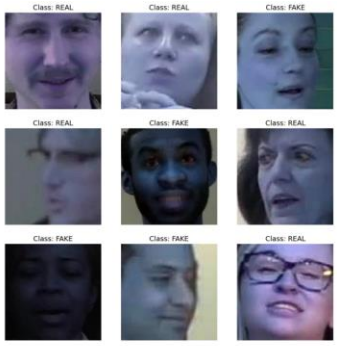
- Image Visualisation –

```
plt.figure(figsize=(12, 12))
for X_batch, y_batch in valid_set.take(1):
    for index in range(9):
        plt.subplot(3, 3, index + 1)
        plt.imshow((X_batch[index] + 1) / 2)
        if(y_batch[index]==1):
            classt='FAKE'
        else:
            classt='REAL'
        plt.title(f"Class: {classt}")
        plt.axis("off")

plt.show()
```

```
augmented_data = tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", seed=42),
    tf.keras.layers.RandomRotation(factor=0.05, seed=42)
])
```

```
plt.figure(figsize=(12, 12))
for X_batch, y_batch in valid_set.take(1):
    augmented_X_batch = augmented_data(X_batch, training=True)
    for index in range(9):
        plt.subplot(3, 3, index + 1)
        plt.imshow(np.clip((augmented_X_batch[index] + 1) / 2, 0, 1))
        if(y_batch[index]==1):
            classt='FAKE'
        else:
            classt='REAL'
        plt.title(f"Class: {classt}")
        plt.axis("off")

plt.show()
```

*(a)*



*(b)*

*(c)*

*Fig. 18 (a) Image visualisation code snippet with data augmentation, (b)Visualisation before data augmentation (c) Visualisation after data augmentation*

- Finetuning and final Model training

The top layers of the pretrained model are removed and replaced with the original task that we need it to be assigned to do. Following that, the model is trained again but with a learning rate of 0.1. The trainability of the top layers of the model are stopped.

```
# removing top layers of the pretrained model to discard original model's
# head to replace it with the specific task I need to assign

tf.random.set_seed(42)
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
output = tf.keras.layers.Dense(1, activation="sigmoid")(avg)
model = tf.keras.Model(inputs=base_model.input, outputs=output)
```

```
for layer in base_model.layers:
    layer.trainable = False
```

```
# training the model epoch wise; weights are unchanged as that of the base model's weights

optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="binary_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10, callbacks=[early_stopping, lr_scheduler])
```

```
Epoch 1/10
280/280 [==============================] - 394s 1s/step - loss: 0.6800 - accuracy: 0.5593 - val_loss: 0.6695 - val_accuracy: 0.5833 -
Epoch 2/10
280/280 [==============================] - 384s 1s/step - loss: 0.6608 - accuracy: 0.6075 - val_loss: 0.6600 - val_accuracy: 0.6010 -
Epoch 3/10
280/280 [==============================] - 376s 1s/step - loss: 0.6530 - accuracy: 0.6218 - val_loss: 0.6552 - val_accuracy: 0.6047 -
Epoch 4/10
280/280 [==============================] - 381s 1s/step - loss: 0.6471 - accuracy: 0.6254 - val_loss: 0.6509 - val_accuracy: 0.6159 -
Epoch 5/10
280/280 [==============================] - 380s 1s/step - loss: 0.6435 - accuracy: 0.6326 - val_loss: 0.6482 - val_accuracy: 0.6177 -
Epoch 6/10
280/280 [==============================] - 388s 1s/step - loss: 0.6397 - accuracy: 0.6402 - val_loss: 0.6457 - val_accuracy: 0.6211 -
Epoch 7/10
280/280 [==============================] - 390s 1s/step - loss: 0.6365 - accuracy: 0.6406 - val_loss: 0.6447 - val_accuracy: 0.6313 -
Epoch 8/10
280/280 [==============================] - 389s 1s/step - loss: 0.6348 - accuracy: 0.6440 - val_loss: 0.6424 - val_accuracy: 0.6250 -
Epoch 9/10
280/280 [==============================] - 388s 1s/step - loss: 0.6329 - accuracy: 0.6431 - val_loss: 0.6414 - val_accuracy: 0.6302 -
Epoch 10/10
280/280 [==============================] - 387s 1s/step - loss: 0.6310 - accuracy: 0.6444 - val_loss: 0.6402 - val_accuracy: 0.6294 -
```

*Fig. 19 Model training with learning rate as 0.1 and top layers of the pretrained model removed*

- Model Evaluation before making the top layers of the model trainable–

```
model.evaluate(test_set)
```

```
100/100 [==============================] - 106s 1s/step - loss: 0.6381 - accuracy: 0.6294

[0.6381250619888306, 0.6293749809265137]
```

*Fig. 20 Model evaluation result before making top layers trainable*

- Model Evaluation after making top layers trainable and lowering the learning rate to 0.05-

```
for layer in base_model.layers[56:]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.05, momentum=0.9)
model.compile(loss="binary_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10)
```

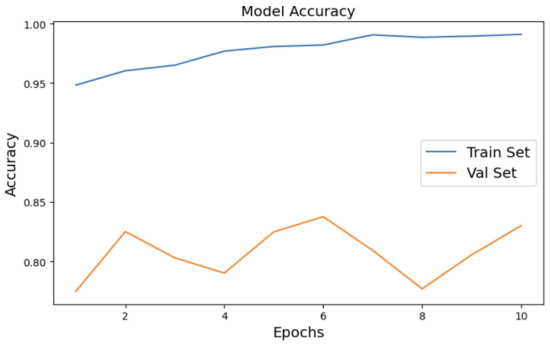
```
Epoch 1/10
280/280 [==============================] - 917s 3s/step - loss: 0.1363 - accuracy: 0.9482 - val_loss: 0.7384 - val_accuracy: 0.7747
Epoch 2/10
280/280 [==============================] - 890s 3s/step - loss: 0.1051 - accuracy: 0.9603 - val_loss: 0.6623 - val_accuracy: 0.8250
Epoch 3/10
280/280 [==============================] - 903s 3s/step - loss: 0.0908 - accuracy: 0.9650 - val_loss: 0.6876 - val_accuracy: 0.8029
Epoch 4/10
280/280 [==============================] - 897s 3s/step - loss: 0.0662 - accuracy: 0.9768 - val_loss: 0.8378 - val_accuracy: 0.7901
Epoch 5/10
280/280 [==============================] - 888s 3s/step - loss: 0.0538 - accuracy: 0.9807 - val_loss: 0.6800 - val_accuracy: 0.8247
Epoch 6/10
280/280 [==============================] - 880s 3s/step - loss: 0.0572 - accuracy: 0.9819 - val_loss: 0.5832 - val_accuracy: 0.8375
Epoch 7/10
280/280 [==============================] - 875s 3s/step - loss: 0.0307 - accuracy: 0.9905 - val_loss: 0.8224 - val_accuracy: 0.8091
Epoch 8/10
280/280 [==============================] - 928s 3s/step - loss: 0.0298 - accuracy: 0.9884 - val_loss: 1.1255 - val_accuracy: 0.7768
Epoch 9/10
280/280 [==============================] - 951s 3s/step - loss: 0.0294 - accuracy: 0.9894 - val_loss: 0.8627 - val_accuracy: 0.8055
Epoch 10/10
280/280 [==============================] - 1461s 5s/step - loss: 0.0261 - accuracy: 0.9908 - val_loss: 0.7600 - val_accuracy: 0.8299
```

*Fig. 21 Model evaluation result after making top layers trainable*

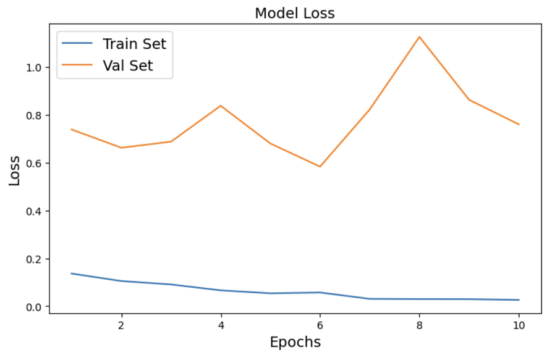
- Final Model Performance

```
# plot model performance
acc = history.history['accuracy']
loss = history.history['loss']
valLoss = history.history['val_loss']
valAccuracy = history.history['val_accuracy']

epochsRange = range(1, len(history.epoch) + 1)

plt.figure(figsize=(15,5))

plt.subplot(1, 2, 1)
plt.plot(epochsRange, acc, label='Train Set')
plt.plot(epochsRange, valAccuracy, label='Val Set')
plt.legend(loc="best")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochsRange, loss, label='Train Set')
plt.plot(epochsRange, valLoss, label='Val Set')
plt.legend(loc="best")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Model Loss')

plt.tight_layout()
plt.show()
```

(a)




(b)




(c)

*Fig. 22 Final Model Performance*

The performance of this final model achieves an accuracy of 84% indicating that the model's effectiveness of detecting deepfakes is good. The figure below shows the result.

```
model.evaluate(test_set)
```

```
100/100 [==============================] - 144s 1s/step - loss: 0.6804 - accuracy: 0.8462

[0.6803543567657471, 0.8462499976158142]
```

*Fig. 23 Final Model Evaluation*

### 6.3 Model Training – Finetuning using VGG16 pretrained model

For the pretrained VGG16 model, the same procedures have been followed which generates similar results until I used VGG16 for finetuning. Hence, the following sections will include only the results of the finetuning using VGG16 to avoid repetitiveness.

After finetuning of top layers of VGG16 model, model performance jumps to 64%, now we make the top layers trainable

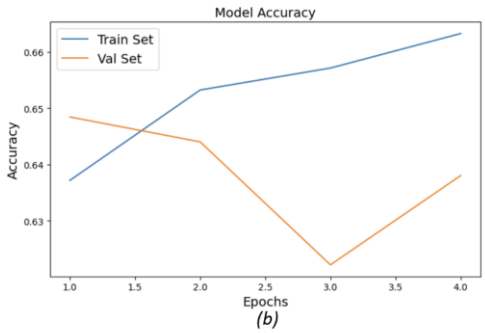
```
for layer in base_model.layers[56:]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="binary_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10, callbacks=[early_stopping, lr_scheduler])
```

*Fig. 24 Training pretrained VGG16 model by making the top layers trainable again*

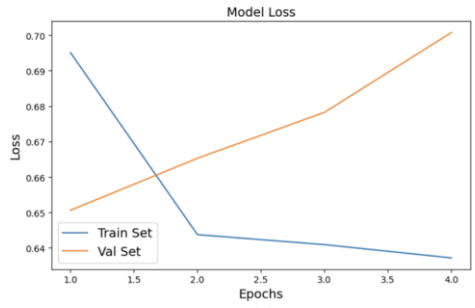
- Final Model Performance

```
# plot model performance
acc = history.history['accuracy']
loss = history.history['loss']
valLoss = history.history['val_loss']
valAccuracy = history.history['val_accuracy']

epochsRange = range(1, len(history.epoch) + 1)

plt.figure(figsize=(15,5))

plt.subplot(1, 2, 1)
plt.plot(epochsRange, acc, label='Train Set')
plt.plot(epochsRange, valAccuracy, label='Val Set')
plt.legend(loc="best")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochsRange, loss, label='Train Set')
plt.plot(epochsRange, valLoss, label='Val Set')
plt.legend(loc="best")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Model Loss')

plt.tight_layout()
plt.show()
```

(a)




(b)




(c)

*Fig. 25 Final performance of the pretrained VGG16 model*

## 7. Model Comparison

```
model.evaluate(test_set)
```

```
100/100 [==============================] - 144s 1s/step - loss: 0.6804 - accuracy: 0.8462

[0.6803543567657471, 0.8462499976158142]
```

*Fig. 26 Evaluation metrics of Xception model*

```
from sklearn.metrics import classification_report

import seaborn as sns

y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)
report = classification_report(y_test, y_pred)
print(report)
```

✓ 7m 10.6s

```
100/100 [==============================] - 430s 4s/step
              precision    recall  f1-score   support

           0       0.60      0.72      0.65      1600
           1       0.65      0.51      0.57      1600

    accuracy                           0.62      3200
   macro avg       0.62      0.62      0.61      3200
weighted avg       0.62      0.62      0.61      3200
```

*Fig. 27 Evaluation metrics of VGG16 model*

The study reveals that using CNN as the baseline model and pretrained Xception model for finetuning is the best model with a better accuracy for the detection of deepfakes when compared to pretrained VGG16 model for finetuning.

## References

Kaushal, A., Singh, S., Negi, S., & Chhaukar, S. (2022). A Comparative Study on Deepfake Detection Algorithms. *IEEE, 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, 854-860.