

# MODEL.PY

```
import numpy as np
import pandas as pd
from keras.applications.mobilenet import MobileNet
from tensorflow.keras.applications import MobileNetV2, VGG16, ResNet50, InceptionV3
from tensorflow.keras.layers import Input, GlobalAveragePooling2D, Dense,
BatchNormalization, Dropout
from tensorflow.keras.models import Model
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import LearningRateScheduler
import os
import cv2
# Define paths
real = "D:\\delete\\DeepFake-Detection\\archive\\real_and_fake_face\\training_real"
fake = "D:\\delete\\DeepFake-Detection\\archive\\real_and_fake_face\\training_fake"
dataset_path = "D:\\delete\\DeepFake-Detection\\archive\\real_and_fake_face"
# Data Augmentation
data_with_aug = ImageDataGenerator(
    horizontal_flip=True,
    vertical_flip=False,
    rescale=1./255,
    validation_split=0.2
)
train = data_with_aug.flow_from_directory(
    dataset_path,
    class_mode="binary",
    target_size=(224, 224),
    batch_size=32,
    subset="training"
)
val = data_with_aug.flow_from_directory(
    dataset_path,
    class_mode="binary",
    target_size=(224, 224),
```

```

    batch_size=32,
    subset="validation"
)

# Model Building (MobileNetV2 example)
input_layer = Input(shape=(224, 224, 3))
# Initialize MobileNetV2 without top layers
mnet = MobileNetV2(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
x = mnet(input_layer)
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
x = Dense(128, activation="relu")(x)
x = Dropout(0.1)(x)
output = Dense(2, activation="softmax")(x)
model = Model(inputs=input_layer, outputs=output)
# Freeze MobileNetV2 layers
model.layers[1].trainable = False
# Compile the model
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
# Learning Rate Scheduler
def scheduler(epoch):
    if epoch <= 2:
        return 0.001
    elif epoch > 2 and epoch <= 15:
        return 0.0001
    else:
        return 0.00001
lr_callbacks = tf.keras.callbacks.LearningRateScheduler(scheduler)
# Train the model
hist = model.fit(
    train,
    epochs=20,
    callbacks=[lr_callbacks],
    validation_data=val)

```

```
# Save the model as .keras
model.save('face_detection_model.keras')
# Visualizing the accuracy and loss
epochs = 20
train_loss = hist.history['loss']
val_loss = hist.history['val_loss']
train_acc = hist.history['accuracy']
val_acc = hist.history['val_accuracy']
xc = range(epochs)
plt.figure(1, figsize=(7, 5))
plt.plot(xc, train_loss)
plt.plot(xc, val_loss)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train Loss vs Validation Loss')
plt.grid(True)
plt.legend(['Train', 'Val'])
plt.figure(2, figsize=(7, 5))
plt.plot(xc, train_acc)
plt.plot(xc, val_acc)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Train Accuracy vs Validation Accuracy')
plt.grid(True)
plt.legend(['Train', 'Val'])
plt.show()
```

## EXPLANATION

This code implements a deep learning pipeline for face detection, likely to be used for detecting real vs fake faces (e.g., in a DeepFake detection scenario). Below is a detailed explanation of the code:

### 1. Import Libraries:

- numpy as np: Library for numerical computations.
- pandas as pd: Data manipulation library, though it's not explicitly used here.
- keras.applications: This includes pre-trained models for transfer learning like MobileNet, MobileNetV2, VGG16, ResNet50, etc.

- tensorflow.keras.layers: Contains layers like Input, GlobalAveragePooling2D, Dense, BatchNormalization, and Dropout, used for building the neural network.
- tensorflow.keras.models: Used to define, compile, and manage models.
- tensorflow: Core library for defining and training machine learning models.
- ImageDataGenerator: Used for data augmentation and preprocessing.
- matplotlib.pyplot: For plotting graphs to visualize loss and accuracy during training.
- cv2: OpenCV library for handling image operations, though it's not used in this code.

## **2. Define Paths:**

- Paths to the real and fake datasets are defined for loading the image data.

```
real = "D:\\delete\\DeepFake-Detection\\archive\\real_and_fake_face\\training_real"
fake = "D:\\delete\\DeepFake-Detection\\archive\\real_and_fake_face\\training_fake"
dataset_path = "D:\\delete\\DeepFake-Detection\\archive\\real_and_fake_face"
```

## **3. Data Augmentation:**

- ImageDataGenerator: This is used to perform data augmentation. Data augmentation helps the model generalize better by providing different variations of the training images.
- horizontal\_flip=True: Randomly flips the images horizontally.
- rescale=1./255: Normalizes the pixel values to the range [0, 1].
- validation\_split=0.2: Splits 20% of the data for validation.

```
data_with_aug = ImageDataGenerator(
    horizontal_flip=True,
    vertical_flip=False,
    rescale=1./255,
    validation_split=0.2
)
```

- flow\_from\_directory: This loads images from the given directory, with the specified parameters like class mode (binary for real/fake classification), image size (224x224), and batch size.

```
train = data_with_aug.flow_from_directory(
    dataset_path,
    class_mode="binary",
    target_size=(224, 224),
    batch_size=32,
    subset="training"
)
```

```

val = data_with_aug.flow_from_directory(
    dataset_path,
    class_mode="binary",
    target_size=(224, 224),
    batch_size=32,
    subset="validation"
)

```

#### 4. Model Building:

- The model is built using **MobileNetV2**, a pre-trained convolutional neural network model suitable for image classification.
- The model starts with a Input layer specifying the input shape of the images (224, 224, 3).
- **MobileNetV2** is loaded without the top layer (to use it for transfer learning), and its weights are initialized with those trained on ImageNet.
- After passing through MobileNetV2, the output is processed with:
  - GlobalAveragePooling2D: Reduces the spatial dimensions of the feature map.
  - Dense layers with ReLU activation for the fully connected layers.
  - BatchNormalization: Normalizes the activations to speed up training.
  - Dropout: Regularization technique to prevent overfitting.
  - Finally, a Dense output layer with a softmax activation function for classification into two classes (real or fake).

```

input_layer = Input(shape=(224, 224, 3))
mnet = MobileNetV2(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
x = mnet(input_layer)
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
x = Dense(128, activation="relu")(x)
x = Dropout(0.1)(x)
output = Dense(2, activation="softmax")(x)
model = Model(inputs=input_layer, outputs=output)

```

- **Freeze the MobileNetV2 layers:** Only the newly added layers will be trained, which allows for faster training.

```

model.layers[1].trainable = False

```

## 5. Model Compilation:

- The model is compiled with the adam optimizer and sparse\_categorical\_crossentropy loss function (since it's a binary classification problem with two classes).

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",  
metrics=["accuracy"])
```

## 6. Learning Rate Scheduler:

- A custom learning rate scheduler function adjusts the learning rate during training:
  - Initial learning rate of 0.001 for the first two epochs.
  - Reduced to 0.0001 after 2 epochs.
  - Further reduced to 0.00001 after 15 epochs.

```
def scheduler(epoch):
```

```
    if epoch <= 2:
```

```
        return 0.001
```

```
    elif epoch > 2 and epoch <= 15:
```

```
        return 0.0001
```

```
    else:
```

```
        return 0.00001
```

```
lr_callbacks = tf.keras.callbacks.LearningRateScheduler(scheduler)
```

## 7. Model Training:

- The model is trained for 20 epochs with the training and validation data. The learning rate scheduler is passed as a callback.

```
hist = model.fit(  
    train,  
    epochs=20,  
    callbacks=[lr_callbacks],  
    validation_data=val  
)
```

## 8. Save the Model:

- After training, the model is saved in .keras format for future use.  
model.save('face\_detection\_model.keras')

## 9. Visualizing the Training Process:

- The loss and accuracy curves for both training and validation datasets are plotted using matplotlib.

```
train_loss = hist.history['loss']
```

```
val_loss = hist.history['val_loss']
```

```
train_acc = hist.history['accuracy']
```

```
val_acc = hist.history['val_accuracy']
plt.plot(xc, train_loss)
plt.plot(xc, val_loss)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train Loss vs Validation Loss')
plt.grid(True)
plt.legend(['Train', 'Val'])
plt.plot(xc, train_acc)
plt.plot(xc, val_acc)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Train Accuracy vs Validation Accuracy')
plt.grid(True)
plt.legend(['Train', 'Val'])
plt.show()
```

# APP.PY

```
import os
from flask import Flask, render_template, request
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
from werkzeug.utils import secure_filename

# Initialize Flask application
app = Flask(__name__)

# Load the model
model = load_model('face_detection_model.keras')

# Set the image upload folder and allowed extensions
UPLOAD_FOLDER = 'static/uploads'
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg'}
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
# Check if the file is an allowed image type
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in
ALLOWED_EXTENSIONS
# Index route to display the HTML page
@app.route('/')
def index():
    return render_template('index.html')
# Route to handle file upload and prediction
@app.route('/predict', methods=['POST'])
def predict():
    if 'file' not in request.files:
        return render_template('index.html', error="No file part")
```



```

file = request.files['file']
if file.filename == "":
    return render_template('index.html', error="No selected file")
if file and allowed_file(file.filename):
    filename = secure_filename(file.filename)
    filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
    file.save(filepath)
    # Preprocess the image and make a prediction
    img = image.load_img(filepath, target_size=(224, 224))
    img_array = image.img_to_array(img) / 255.0 # Normalize the image
    img_array = np.expand_dims(img_array, axis=0)
    # Predict using the model
    prediction = model.predict(img_array)
    # Check the prediction result
    result = 'Real' if np.argmax(prediction) == 0 else 'Fake'
    confidence = round(np.max(prediction) * 100, 2)
    return render_template('index.html', prediction=result,
confidence=confidence, filename=filename) return
render_template('index.html', error="Invalid file format")
# Run the app
if __name__ == "__main__":
    if not os.path.exists(UPLOAD_FOLDER):
        os.makedirs(UPLOAD_FOLDER)
    app.run(debug=True)

```

## EXPLANATION

This code sets up a Flask web application for face detection using a pre-trained model. The app accepts an image file, processes it, and predicts whether the face in the image is "real" or "fake." Below is an explanation of each part of the code:

### 1. Import Libraries:

os: Used to interact with the operating system (e.g., to handle file paths). Flask, render\_template, request: Flask is a web framework used to create web

applications. `render_template` is used to render HTML templates, and `request` handles incoming requests.

`load_model`: Loads a previously saved Keras model.

`image`: Provides functions for loading and preprocessing images.

`numpy`: A fundamental library for numerical computations, used here to handle the image data.

`secure_filename`: Ensures that file names are safe to use for storing files on the server.

## **2. Initialize Flask Application:**

`app = Flask(__name__)`: Creates a new instance of the Flask application.

## **3. Load the Pre-trained Model:**

`model = load_model('face_detection_model.keras')`: Loads the face detection model that was trained earlier and saved as `face_detection_model.keras`.

## **4. Set Image Upload Folder and Allowed Extensions:**

`UPLOAD_FOLDER = 'static/uploads'`: Specifies the folder where uploaded images will be stored.

`ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg'}`: Specifies the allowed image file formats (PNG, JPG, JPEG).

`app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER`: Configures the Flask app to store files in the specified folder.

## **5. Allowed File Check Function:**

`allowed_file(filename)`: A helper function that checks if the uploaded file is of an allowed type (i.e., PNG, JPG, or JPEG).

## **6. Index Route (/):**

The root route (`/`) handles the request to display the main HTML page (`index.html`). This page allows the user to upload an image.

```
python
```

```
@app.route('/')
```

```
def index():
```

```
    return render_template('index.html')
```

## **7. Prediction Route (/predict):**

This route handles the image file upload and prediction:

Check for a file: The app checks if a file was uploaded.

File validation: It ensures that the file is of an allowed type using the `allowed_file()` function.

Save the file: If the file is valid, it is saved to the server in the `static/uploads` folder.

```
@app.route('/predict', methods=['POST'])
def predict():
    if 'file' not in request.files:
        return render_template('index.html', error="No file part")
    file = request.files['file']
    if file.filename == '':
        return render_template('index.html', error="No selected file")
    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(filepath)
```

### **Image Preprocessing:**

The image is loaded and resized to 224x224 pixels using `image.load_img()`. It is then converted to a NumPy array and normalized to have pixel values between 0 and 1. `np.expand_dims(img_array, axis=0)` reshapes the array to add a batch dimension, as the model expects input in batches.

```
img = image.load_img(filepath, target_size=(224, 224))
img_array = image.img_to_array(img) / 255.0 # Normalize the image
img_array = np.expand_dims(img_array, axis=0)
```

### **Prediction:**

The preprocessed image is passed to the model for prediction using `model.predict()`.

The prediction output is a probability distribution across classes (real or fake). `np.argmax(prediction)` returns the index of the class with the highest probability (0 for "Real", 1 for "Fake").

The confidence is calculated by multiplying the maximum probability by 100.

```
prediction = model.predict(img_array)
# Check the prediction result
result = 'Real' if np.argmax(prediction) == 0 else 'Fake'
confidence = round(np.max(prediction) * 100, 2)
```

### **Return the Result:**

The result (Real or Fake), the confidence, and the filename of the uploaded image are passed to the HTML template `index.html` for display.

```
return render_template('index.html', prediction=result,
confidence=confidence, filename=filename)
```

### **Invalid File Format:**

If the uploaded file is not of an allowed type, an error message is displayed

```
return render_template('index.html', error="Invalid file format")
```

## 8. Run the Application:

The application checks if the UPLOAD\_FOLDER exists. If it doesn't, it creates it. The Flask app is run in debug mode, which provides helpful error messages during development.

```
if __name__ == "__main__":  
    if not os.path.exists(UPLOAD_FOLDER):  
        os.makedirs(UPLOAD_FOLDER)  
    app.run(debug=True)
```

### Key Flow:

The user visits the index page (/), which renders a form to upload an image.

The user uploads an image, which is sent to the /predict route.

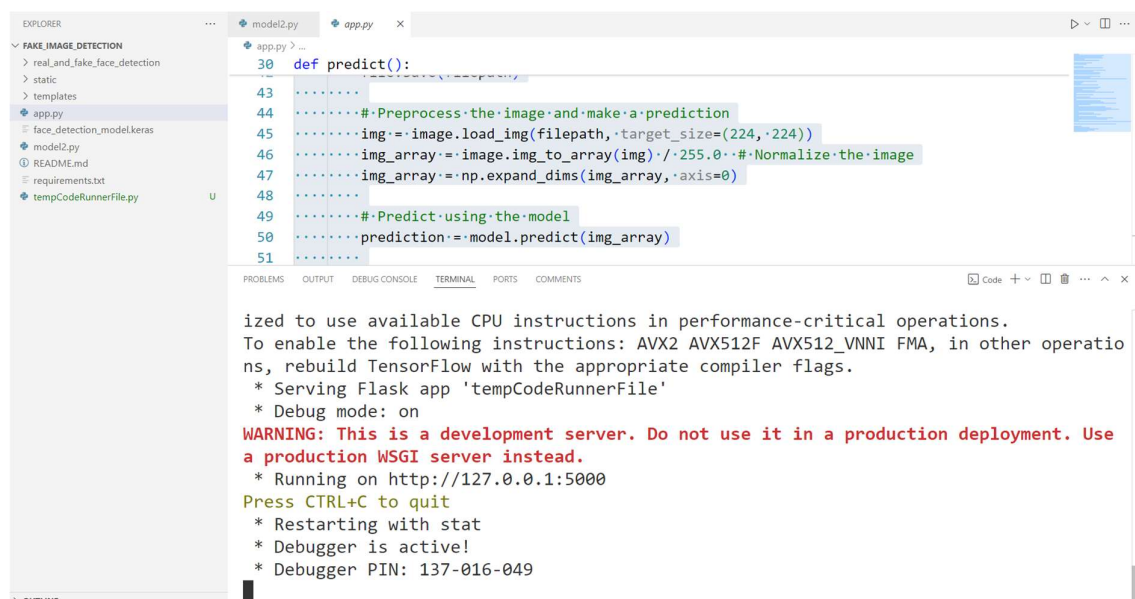
The image is processed, passed to the trained model, and a prediction (either "Real" or "Fake") is made.

The prediction and confidence level are displayed to the user on the same page (index.html).

### How it works:

The app allows users to upload an image for face detection using the trained model.

The result (whether the face in the image is real or fake) and the confidence level are displayed to the user.

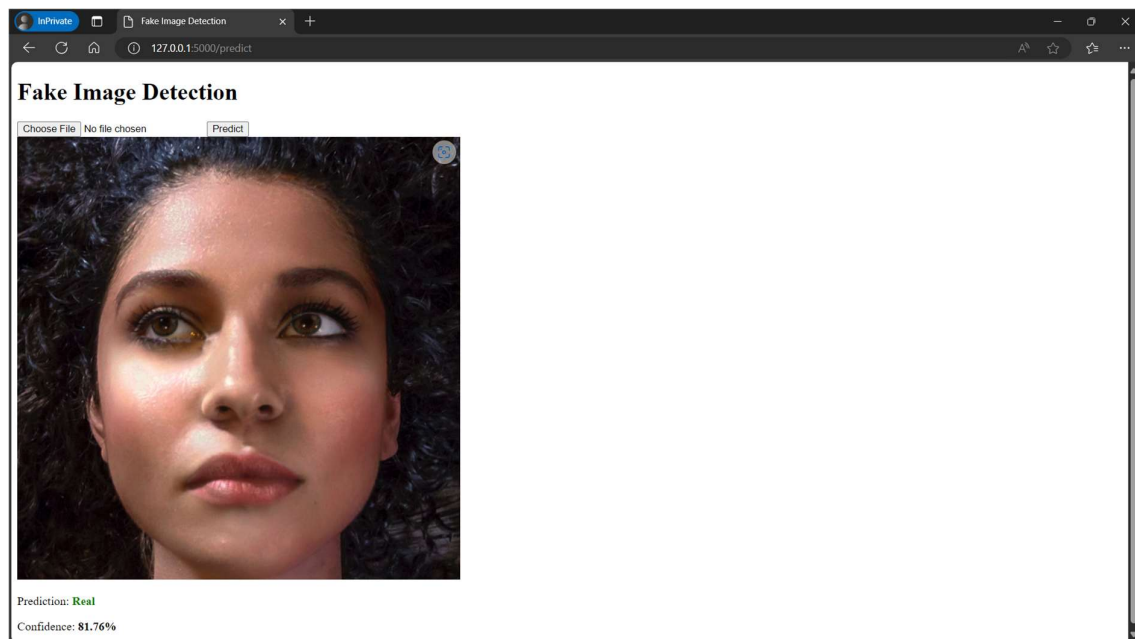
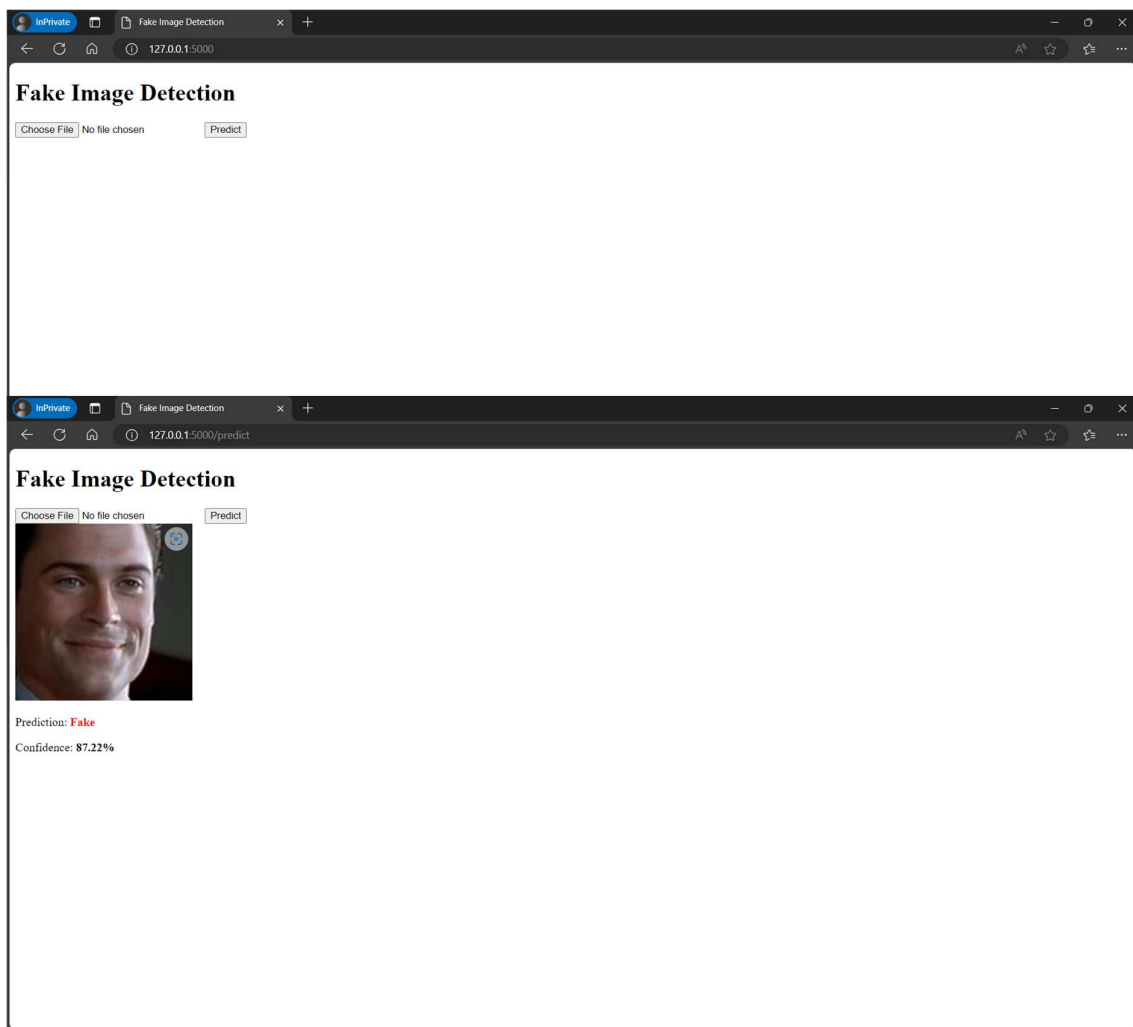


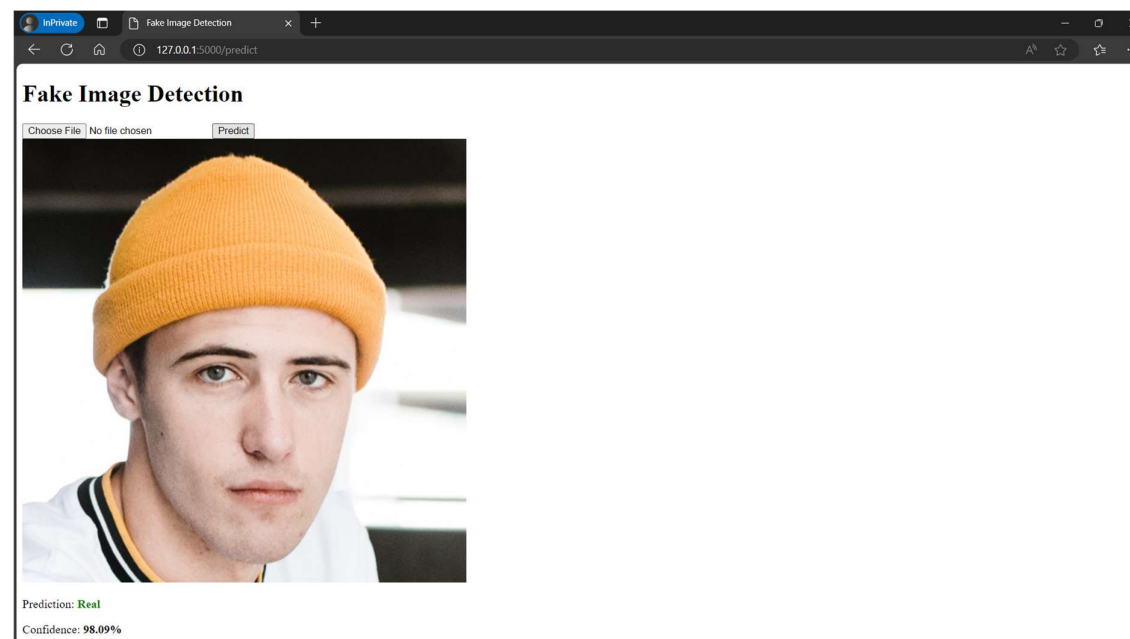
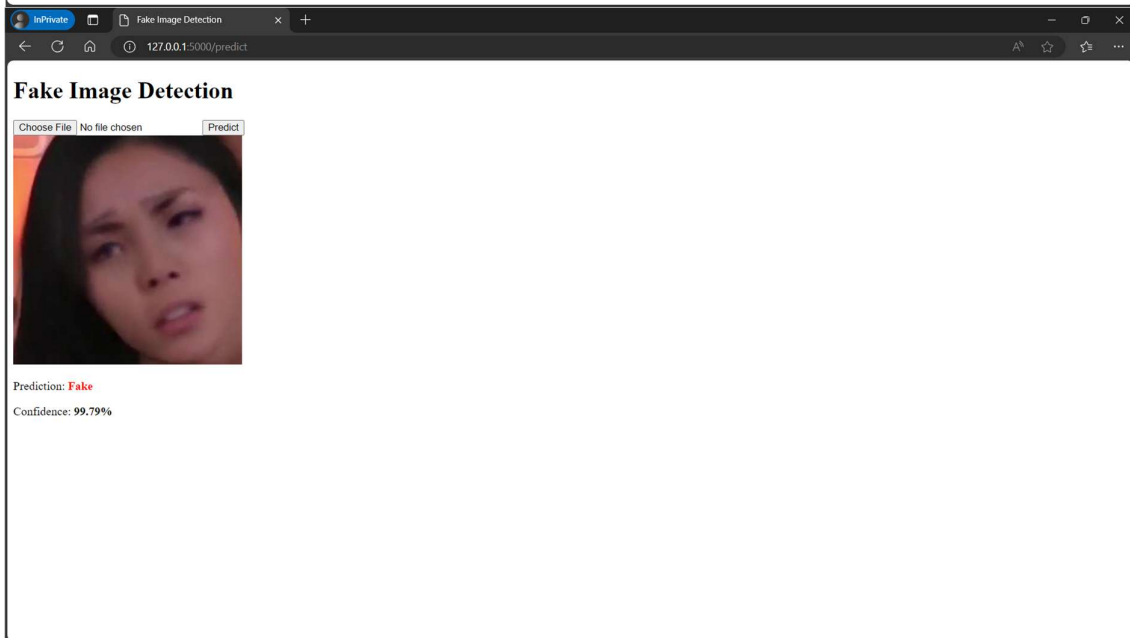
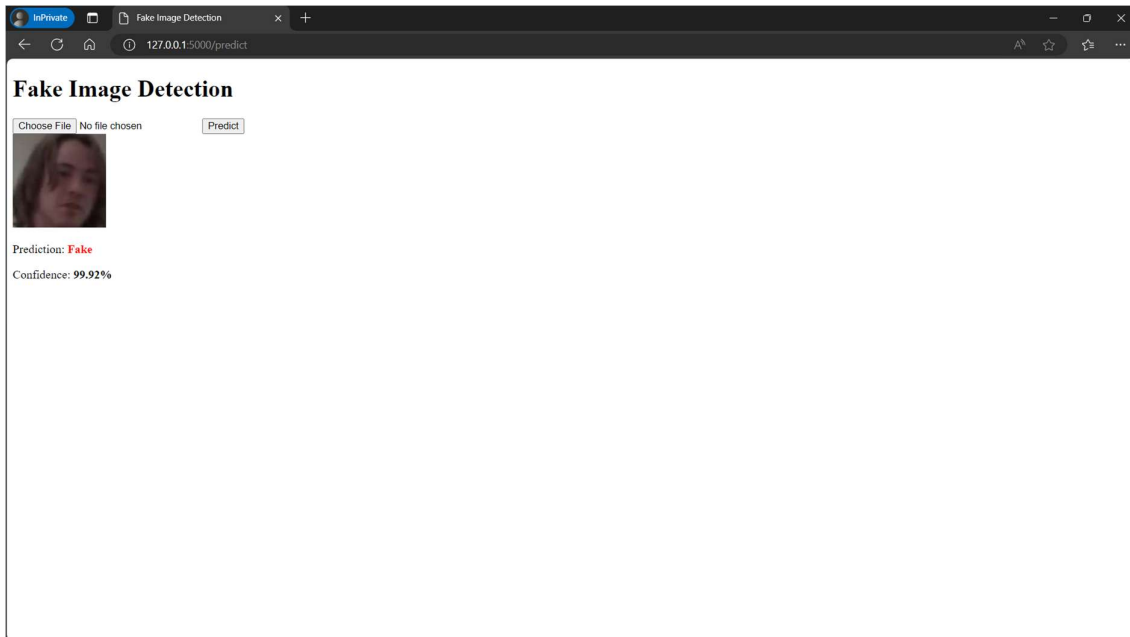
The screenshot shows a VS Code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'FAKE\_IMAGE\_DETECTION' with subfolders 'real\_and\_fake\_face\_detection', 'static', and 'templates'. The file 'app.py' is selected. The code editor shows the following Python code:

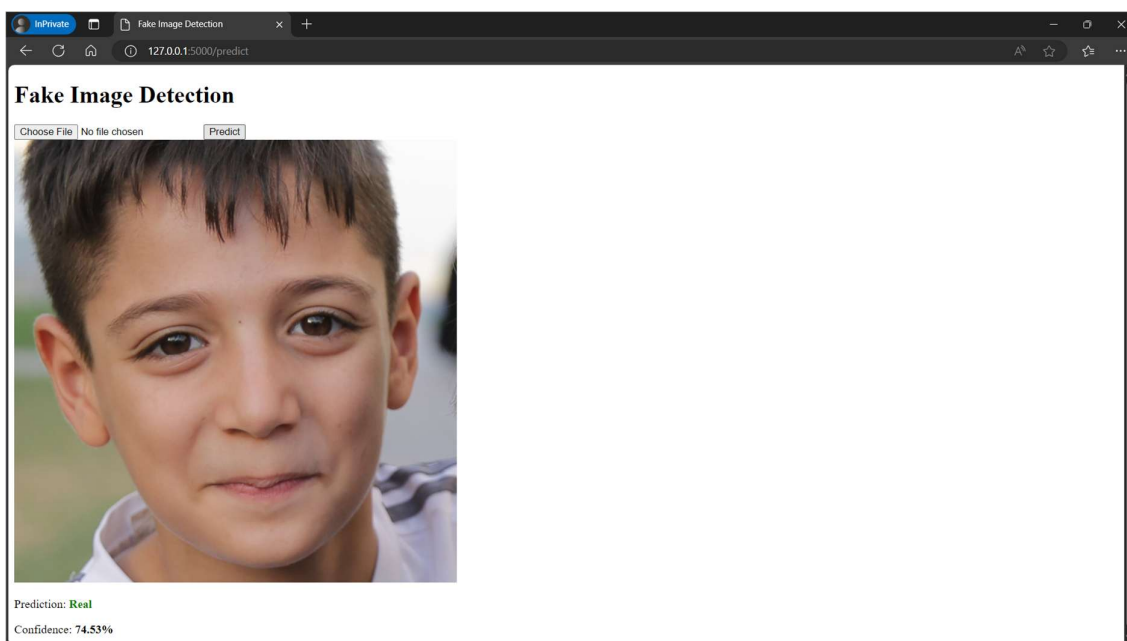
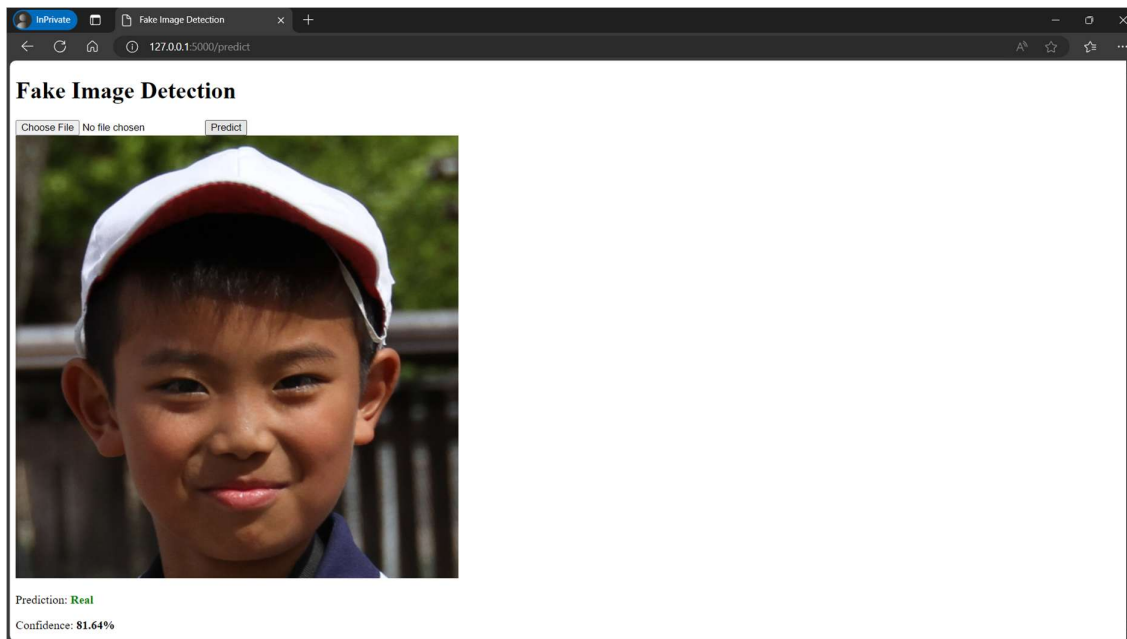
```
30 def predict():  
43 .....  
44 .....#Preprocess the image and make a prediction  
45 .....img=image.load_img(filepath,target_size=(224,224))  
46 .....img_array=image.img_to_array(img)/.255.0..#Normalize the image  
47 .....img_array=np.expand_dims(img_array,axis=0)  
48 .....  
49 .....#Predict using the model  
50 .....prediction=model.predict(img_array)  
51 .....
```

The terminal window at the bottom shows the following output:

```
ized to use available CPU instructions in performance-critical operations.  
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operatio  
ns, rebuild TensorFlow with the appropriate compiler flags.  
* Serving Flask app 'tempCodeRunnerFile'  
* Debug mode: on  
WARNING: This is a development server. Do not use it in a production deployment. Use  
a production WSGI server instead.  
* Running on http://127.0.0.1:5000  
Press CTRL+C to quit  
* Restarting with stat  
* Debugger is active!  
* Debugger PIN: 137-016-049
```







Thank  
you

[This Photo](#) by Unknown Author is licensed under [CC BY-NC-ND](#)