National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Sammam Sohail
Student ID: X23256800

School of Computing
National College of Ireland

Supervisor: Anu Sahni

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……. Sammam Sohail……

Student ID: ……23256800……..…..

Programme: ……MSc Data Analytics…… **Year:** …2024.……

Module: ……MSC Research Project…….……

Lecturer: ……Anu Sahni……..……

Submission Due Date: ……12/12/24……..……

Project Title: ……Innovative Study on Popular Approaches Used in Gaze Prediction

Word Count: ……1451…… **Page Count:** ……14……..…..……

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: ……Sammam Sohail……

Date: ……11/12/24……

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:	
Penalty Applied (if applicable):	

Configuration Manual

Sammam Sohail
Student ID: X23256800

1 Introduction

This configuration manual walks through the different stages of code development for the gaze prediction research project. These stages include:

- Data Collection
- Data Preprocessing
- Implementation of Gaze Prediction System
- Evaluation of Systems

2 System Configuration

2.1 Hardware Specifications

The traditional system of this study was implemented on the local machine, whereas the advanced system was implemented on Kaggle Cloud. The detailed specifications of the resources are highlighted in Figure 1.

Traditional System

Machine Name	Apple MacBook Pro M3
Processor	Apple M3 8-core CPU
RAM	8.0 GB
Graphics	10-core GPU, 16-core Neural Engine
Operating System	macOS Sequoia

Advanced System

Machine Source	Kaggle Cloud
Processor	2-core
RAM	30.0 GB
Graphics	Tesla P100

Figure 1, Hardware Specifications

2.2 Software and Libraries

The list below contains the libraries and software tools used for this research.

- Python 3.12.1
- OpenCV
- Mediapipe
- Dlib
- Sklearn
- Tensorflow
- Visual Studio Code

- Kaggle Cloud with P100 Hardware Accelerator
- Numpy
- Pandas
- 68 point facial landmarks dat file
- VGG Face model weights

Facial landmark file download link:
https://www.kaggle.com/datasets/sajikim/shape-predictor-68-face-landmarks

VGG-Face weights file link:
https://www.kaggle.com/datasets/acharyarupak391/vggfaceweights

3 Data Collection

For this research, the Columbia Gaze Dataset was utilised, which was downloaded on the local machine as a zip file. The dataset has an approximate size of 1.61 GB uncompressed.

Dataset link: https://ceal.cs.columbia.edu/columbiagaze/

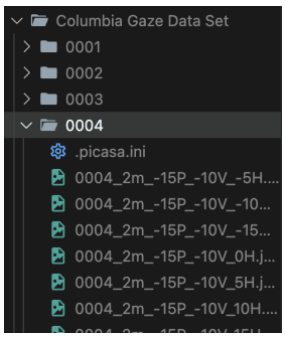


Figure 2, Dataset Structure

Figure 2 shows the file structure of the unprocessed dataset. Each folder contains images for different head-pose and gaze angles.

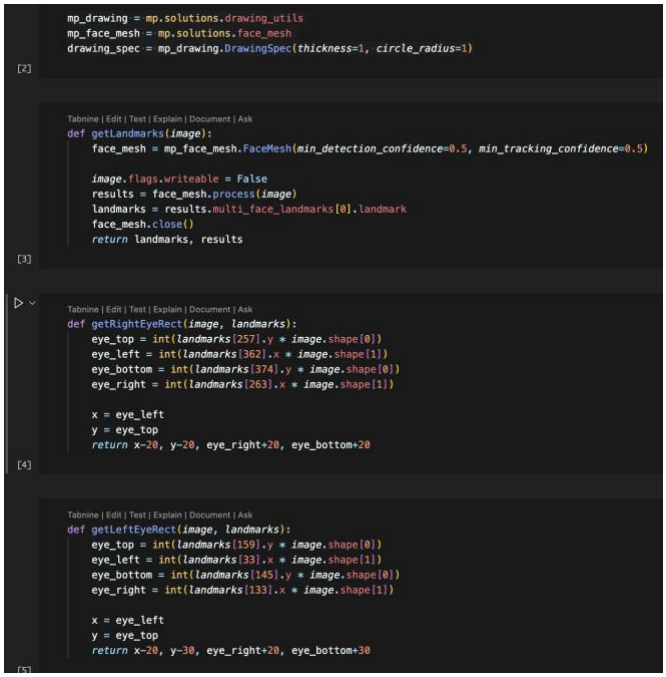
4 Dataset Preprocessing

To train the different systems of this research, various techniques were used on the dataset to extract further information. These implemented techniques are highlighted in Figure 3 and include face region extraction, left eye region, and right eye region extraction performed using the Media pipe library.

These techniques follow a stepwise execution before the final eye regions are extracted, therefore a separate function shown in Figure 4 is implemented that performs the following functions:

- Traversing the images in the directories of the dataset for processing.
- Image gaze angle extraction.
- Left eye region coordinates extraction.
- Right eye region coordinates extraction.
- Accumulating the results in an array for storing in a CSV file named "_annotations".
- Transfer of images and the CSV file to a new directory named "Columbia Dataset".

Due to the face inference limitations of the Mediapipe library, such as the inability to detect faces from images having extreme head-pose angle, the size of the dataset was reduced to 3197 images (originally 5880 images).



```
mp_drawing = mp.solutions.drawing_utils
mp_face_mesh = mp.solutions.face_mesh
drawing_spec = mp_drawing.DrawingSpec(thickness=1, circle_radius=1)
```
[2]

```
Tabnine | Edit | Test | Explain | Document | Ask
def getLandmarks(image):
    face_mesh = mp_face_mesh.FaceMesh(min_detection_confidence=0.5, min_tracking_confidence=0.5)

    image.flags.writeable = False
    results = face_mesh.process(image)
    landmarks = results.multi_face_landmarks[0].landmark
    face_mesh.close()
    return landmarks, results
```
[3]

```
Tabnine | Edit | Test | Explain | Document | Ask
def getRightEyeRect(image, landmarks):
    eye_top = int(landmarks[257].y * image.shape[0])
    eye_left = int(landmarks[362].x * image.shape[1])
    eye_bottom = int(landmarks[374].y * image.shape[0])
    eye_right = int(landmarks[263].x * image.shape[1])

    x = eye_left
    y = eye_top
    return x-20, y-20, eye_right+20, eye_bottom+20
```
[4]

```
Tabnine | Edit | Test | Explain | Document | Ask
def getLeftEyeRect(image, landmarks):
    eye_top = int(landmarks[159].y * image.shape[0])
    eye_left = int(landmarks[33].x * image.shape[1])
    eye_bottom = int(landmarks[145].y * image.shape[0])
    eye_right = int(landmarks[133].x * image.shape[1])

    x = eye_left
    y = eye_top
    return x-20, y-30, eye_right+20, eye_bottom+30
```
[5]

Figure 3, Facial feature extraction functions

```
Tabnine | Edit | Test | Explain | Document | Ask
def get_label_from_filename(filename):
    parts = filename.split('_')
    parts[-1] = parts[-1].split('.')[0]
    pose = float(parts[-3][1:-1])
    horizontal_angle = float(parts[-1][1:-1])
    return np.array([horizontal_angle, pose])

Tabnine | Edit | Test | Explain | Document | Ask
def load_images_and_labels(dataset_dir):
    with open(os.path.join(destination_dir, '_annotations.csv'), mode="a", newline="") as file:
        writer = csv.writer(file)

        for subject_dir in os.listdir(dataset_dir):

                subject_path = os.path.join(dataset_dir, subject_dir)
                if os.path.isdir(subject_path):
                    for img_name in os.listdir(subject_path):
                        img_path = os.path.join(subject_path, img_name)

                        image = cv2.imread(img_path)
                        if image is None:
                            continue

                        label = get_label_from_filename(img_path)

                        try:
                            landmark, results = getLandmarks(image)
                        except:
                            continue

                        landmarks = results.multi_face_landmarks[0].landmark
```

```
        height, width, _ = image.shape

        x_min, y_min = width, height
        x_max, y_max = 0, 0

        for landmark in landmarks:
            x, y = int(landmark.x * width), int(landmark.y * height)
            x_min, y_min = min(x_min, x), min(y_min, y)
            x_max, y_max = max(x_max, x), max(y_max, y)

        xtx = image[y_min:y_max, x_min:x_max].copy()

        try:
            landmark, results = getLandmarks(xtx)
        except:
            continue

        xe, ye, we, he = getLeftEyeRect(xtx, landmark)
        xe2, ye2, we2, he2 = getRightEyeRect(xtx, landmark)

        data = [img_name, label[0], xe, ye, we, he, xe2, ye2, we2, he2]

        writer.writerow(data)

        cv2.imwrite(os.path.join(destination_dir, img_name), xtx)

        del image, xtx
        cv2.destroyAllWindows()

load_images_and_labels(DATASET_DIR)
```

Untitled design - Whiteboard

Figure 4, Preprocessing function

5 System Implementation and Results

5.1 Traditional System Implementation

Figure 5 shows the code snippet for loading the processed dataset. The load_data function stores the images and labels from the directory into two separate arrays. This function also performs image resizing, conversion to grayscale, and conversion of gaze angles to 3 gaze classes.

```
import pandas as pd
import numpy as np
import os
import cv2 as cv


annotations1 = pd.read_csv('./Columbia DataSet/_annotations.csv')
img_size = 224

Tabnine | Edit | Test | Explain | Document | Ask
def load_data(IMAGES_PATH, annotations):
    images = []
    gaze_labels = []


    for index, row in annotations.iterrows():

            img_path = os.path.join(IMAGES_PATH, row[0])
            image = cv.imread(img_path)

            image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)

            image_resized = cv.resize(image, (img_size, img_size))

            images.append(image_resized)

            label = row[1]

            if label < -1:
                gaze_labels.append(0)
            elif label > 1:
                gaze_labels.append(2)
            else:
                gaze_labels.append(1)


    return np.array(images), np.array(gaze_labels)

X, Y = load_data('./Columbia DataSet', annotations1)
```

Figure 5, Loading dataset for traditional system

The implementation for predicting the gaze direction is shown in Figure 6, containing the stepwise working of the system

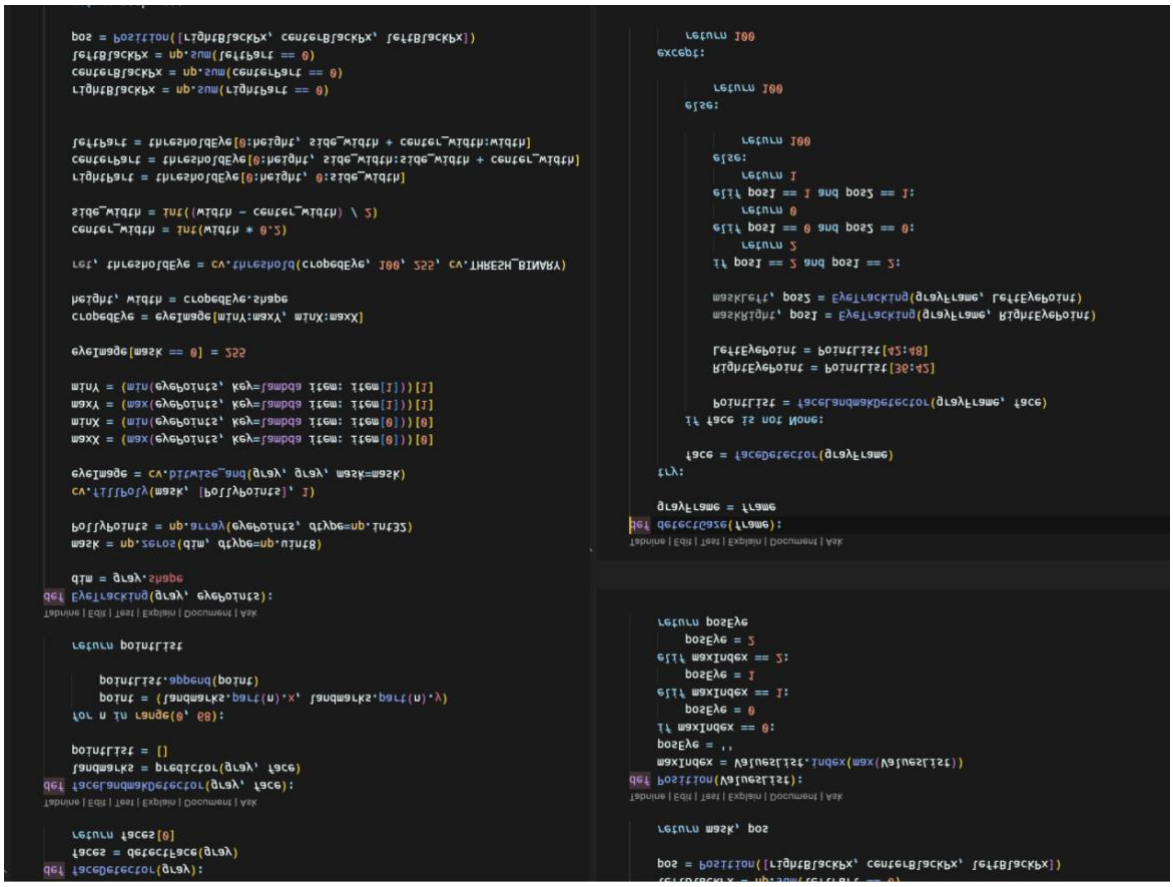


Figure 6, Traditional Gaze Detection system code snippet

To produce the predictions from the system, the code in Figure 7 is used. The test data is input to the system and the consecutive predictions are stored in an array.

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, shuffle=True, random_state=42)



    y_pred = []

    for x in X_test:
            y_pred.append(detectGaze(x))


    accuracy = np.mean(y_pred == Y_test)
    print("Accuracy:", accuracy)

Accuracy: 0.5338894681960376
```

Figure 7, Code snippet for producing predictions

Lastly, the code in Figure 8 is used to evaluate the system, which produces metrics such as precision, recall, accuracy, f1-score, and confusion matrix.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix


accuracy = accuracy_score(Y_test, y_pred)
precision = precision_score(Y_test, y_pred, average='weighted', zero_division=0)
recall = recall_score(Y_test, y_pred, average='weighted', zero_division=0)
f1 = f1_score(Y_test, y_pred, average='weighted', zero_division=0)
conf_matrix = confusion_matrix(Y_test, y_pred)


print(f"- Accuracy: {accuracy:.2f}")
print(f"- Precision: {precision:.2f}")
print(f"- Recall: {recall:.2f}")
print(f"- F1-Score: {f1:.2f}")
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Left", "Center", "Right", "None"],
            yticklabels=["Left", "Center", "Right", 'None'])
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```

Figure 8, Evaluation of the Traditional System

5.2 Advanced System Implementation

5.2.1 Direct Approach CNN Model

Figure 9 shows the code for loading, normalizing, and resizing the images for the direct approach CNN model.

```
import pandas as pd
import numpy as np
import os
from tensorflow.keras.utils import Sequence
import cv2
import matplotlib.pyplot as plt

img_size = 224

def load_data(IMAGES_PATH, annotations):
    images = []
    gaze_labels = []

    for index, row in annotations.iterrows():

            img_path = os.path.join(IMAGES_PATH, row[0])
            image = cv2.imread(img_path)

            image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            image = image / 255.0

            image_resized = cv2.resize(image, (img_size, img_size))

            images.append(image_resized)

            label = row[1]

            if label < -1:
                gaze_labels.append('left')
            elif label > 1:
                gaze_labels.append('right')
            else:
                gaze_labels.append('center')


    return np.array(images), np.array(gaze_labels)

annotations1 = pd.read_csv('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet/_annotations.csv')
X, Y = load_data('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet', annotations1)
```

Figure 9, Dataset loading for Direct CNN Model

The code snippet in Figure 10 shows the code for expanding the dimensions of the images, applying a label encoder to the gaze directions, and splitting the dataset for training and testing.

```
import numpy as np
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical

X = np.stack((X,) * 3, axis=-1)
label_encoder = LabelEncoder()
y_train_class_flat = label_encoder.fit_transform(Y.flatten())
y_train_class_encoded = to_categorical(y_train_class_flat)
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, y_train_class_encoded, test_size=0.3, shuffle=True, random_state=42)
```

Figure 10, Code for stacking and splitting the dataset

After the dataset processing, the model is set up with the VGG-Face as the base model shown in Figure 11.

```
from tensorflow.keras.applications import VGG16

vgg_face = VGG16(include_top=False, input_shape=(224, 224, 3))

vgg_face.load_weights('/kaggle/input/vgg-face-weights/tensorflow2/default/1/vgg_face_weights.h5', by_name=True)

for layer in vgg_face.layers[0:4]:
    layer.trainable = False
```

Figure 11, Model Setup for direct CNN

The rest of the structure of the model is shown in Figure 12.

```
input_tensor = layers.Input(shape=(224, 224, 3))

x = vgg_face(input_tensor)

x = layers.Conv2D(64, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.Conv2D(64, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.MaxPooling2D((2, 2), padding='same',strides = (2,2))(x)

x = layers.Conv2D(128, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.Conv2D(128, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.MaxPooling2D((2, 2), padding='same', strides = (2,2))(x)

x = layers.Conv2D(256, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.Conv2D(256, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.MaxPooling2D((2, 2), padding='same', strides = (2,2))(x)

x = layers.Conv2D(256, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.Conv2D(256, (20, 20), activation='relu', padding='same', strides = (1,1))(x)
x = layers.MaxPooling2D((2, 2), padding='same', strides = (2,2))(x)

x = layers.Flatten()(x)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dropout(0.5)(x)
x = layers.Dense(128, activation='relu')(x)
x = layers.Dropout(0.5)(x)

classification_output = layers.Dense(3, activation='softmax', name='classification_output')(x)

final_model = models.Model(inputs=input_tensor, outputs=[classification_output])
```

+ Code + Markdown

```
final_model.compile(
    optimizer='adam',
    loss={'classification_output': 'categorical_crossentropy'},
    metrics={'classification_output': 'accuracy'}
)

final_model.summary()
```

Figure 12, Direct CNN Model Architecture

The code for training the model is highlighted in Figure 13.

```
history = final_model.fit(
    X_train,
    {'classification_output': Y_train},
    batch_size=16,
    epochs=10,
)
```

Figure 13, Direct CNN Model Training

Finally, for evaluating the model, the code snippet in Figure 14 is used to produce the accuracy and loss plots for the training phase.

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.title('Accuracy vs. Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.title('Loss vs. Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Figure 14, Direct CNN evaluation

5.2.2 Pipelined CNN Model

For this model, two separate CNN modules are developed, which are then pipelined together to produce the gaze predictions. In these modules, some steps, such as data splitting and dimension expansion, are the same as they were in the direct CNN model.

5.2.2.1 Eye Localization Module

The code shown in Figure 15 is used to load the images and bounding box coordinates for the eye regions from the pre-processed dataset directory.

```
img_size = 224

def load_data(IMAGES_PATH, annotations):
    images = []
    bboxes = []

    for index, row in annotations.iterrows():

            img_path = os.path.join(IMAGES_PATH, row[0])
            image = cv2.imread(img_path)
            image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            image_resized = cv2.resize(image, (img_size, img_size))

            image_resized = image_resized / 255.0
            images.append(image_resized)

            x1, y1, x2, y2 = row[2], row[3], row[4], row[5]
            orig_h = image.shape[0]
            orig_w = image.shape[1]

            x1_scaled = (x1 / orig_w) * img_size
            y1_scaled = (y1 / orig_h) * img_size
            x2_scaled = (x2 / orig_w) * img_size
            y2_scaled = (y2 / orig_h) * img_size

            x1, y1, x2, y2 = row[6], row[7], row[8], row[9]

            xx1_scaled = (x1 / orig_w) * img_size
            yy1_scaled = (y1 / orig_h) * img_size
            xx2_scaled = (x2 / orig_w) * img_size
            yy2_scaled = (y2 / orig_h) * img_size

            bboxes.append([x1_scaled, y1_scaled, x2_scaled, y2_scaled,
                           xx1_scaled, yy1_scaled, xx2_scaled, yy2_scaled])

    return np.array(images), np.array(bboxes)

annotations1 = pd.read_csv('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet/_annotations.csv')
X, Y = load_data('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet', annotations1)
```

Figure 15, Data loading for Eye Localisation Module

Figure 16 shows the custom function for IoU evaluation metric developed for this model.

```python
import tensorflow as tf

def iou_metric(y_true, y_pred):
    y_true_x1 = y_true[:, 0]
    y_true_y1 = y_true[:, 1]
    y_true_x2 = y_true[:, 2]
    y_true_y2 = y_true[:, 3]

    y_pred_x1 = y_pred[:, 0]
    y_pred_y1 = y_pred[:, 1]
    y_pred_x2 = y_pred[:, 2]
    y_pred_y2 = y_pred[:, 3]

    x1 = tf.maximum(y_true_x1, y_pred_x1)
    y1 = tf.maximum(y_true_y1, y_pred_y1)
    x2 = tf.minimum(y_true_x2, y_pred_x2)
    y2 = tf.minimum(y_true_y2, y_pred_y2)

    intersection = tf.maximum(0.0, x2 - x1) * tf.maximum(0.0, y2 - y1)
    true_area = (y_true_x2 - y_true_x1) * (y_true_y2 - y_true_y1)
    pred_area = (y_pred_x2 - y_pred_x1) * (y_pred_y2 - y_pred_y1)

    union = true_area + pred_area - intersection
    iou = intersection / tf.maximum(union, 1e-10)

    return tf.reduce_mean(iou)
```

Figure 16, Eye localization module IoU function

The code for building the architecture for this module is shown in Figure 17.

```python
img_size = 224
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(img_size, img_size,3))

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

input_layer = base_model.output

x = layers.Conv2D(64, (7,7), activation='relu', padding='same')(input_layer)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(128, (7,7), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.2)(x)  # Add dropout for regularization

x = layers.Conv2D(256, (7,7), activation='relu', padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.2)(x)

x = layers.Conv2D(128, (7,7), activation='relu', padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.2)(x)

x = layers.Conv2D(128, (7,7), activation='relu', padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.3)(x)

x = layers.GlobalAveragePooling2D()(x)

x = layers.Dense(256, activation='relu')(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.3)(x)

output_layer = layers.Dense(8)(x)

model = models.Model(inputs=base_model.input, outputs=output_layer)
model.compile(optimizer='RMSprop', loss='mean_squared_error', metrics=[iou_metric])
```

Figure 17, Eye localisation module architecture

Figure 18 shows the code used for training the eye localisation module.

```
history = model.fit(
    X_train, Y_train,
    epochs=15,
    validation_data=(X_test, Y_test), verbose=1,
    batch_size=16)
```

Figure 18, Eye localisation model training

To evaluate and save the module file, the code snippet shown in Figure 19 is used.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['iou_metric'], label='Training IoU')
plt.xlabel('Epochs')
plt.ylabel('IoU')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

model.save('modelx9.h5')
```

Figure 19, Evaluating and saving the eye localisation module

5.2.2.2 Gaze Detection Module

The code shown in Figure 20 is used to load the eye images and gaze directions from the pre-processed dataset directory.

```
img_size = 224

def load_data(IMAGES_PATH, annotations):
    images = []
    gaze_labels = []

    for index, row in annotations.iterrows():

            img_path = os.path.join(IMAGES_PATH, row[0])
            image = cv2.imread(img_path)

            image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            image = image / 255.0

            x1, y1, x2, y2 = row[2], row[3], row[4], row[5]
            xx1, yy1, xx2, yy2 = row[6], row[7], row[8], row[9]

            image_left = image[y1:y2, x1:x2]
            image_right = image[yy1:yy2, xx1:xx2]

            imageLeft_resized = cv2.resize(image_left, (img_size, img_size))
            imageRight_resized = cv2.resize(image_right, (img_size, img_size))

            images.append([imageLeft_resized, imageRight_resized])

            label = row[1]

            if label < -1:
                gaze_labels.append('left')
            elif label > 1:
                gaze_labels.append('right')
            else:
                gaze_labels.append('center')

    return np.array(images), np.array(gaze_labels)

annotations1 = pd.read_csv('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet/_annotations.csv')
X, Y = load_data('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet', annotations1)
```

Figure 20, Loading data for the gaze detection module

Figure 21 shows the code containing the architecture of the gaze detection module.

```
img_size = 224

base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(img_size, img_size, 3))

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


input_1 = layers.Input(shape=(img_size, img_size, 3))
input_2 = layers.Input(shape=(img_size, img_size, 3))

features_1 = base_model(input_1)
features_2 = base_model(input_2)

combined_features = layers.Concatenate()([features_1, features_2])

x = layers.Conv2D(64, (8,8), activation='relu', padding='same')(combined_features)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(128, (8,8), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.3)(x)

x = layers.Conv2D(256, (8,8), activation='relu', padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.3)(x)
                                                    Notebook editor cells
x = layers.GlobalAveragePooling2D()(x)

x = layers.Dense(256, activation='relu')(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.4)(x)

output_layer = layers.Dense(3, activation='softmax', name='output')(x)

model = models.Model(inputs=[input_1, input_2], outputs=output_layer)

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 21, Architecture for the gaze detection module

To train the model, the separate eye images were concatenated into a single vector and passed to the model for training, as shown in Figure 22.

```
X_train_1 = X_train[:, 0, :, :, :]
X_train_2 = X_train[:, 1, :, :, :]

X_test_1 = X_test[:, 0, :, :, :]
X_test_2 = X_test[:, 1, :, :, :]
```

+ Code + Markdown

```
history = model.fit([X_train_1, X_train_2], Y_train, epochs=10,
                    validation_data=([X_test_1, X_test_2], Y_test), verbose=1, batch_size=16)
```

Figure 22, Gaze detection module training

The code used for creating the accuracy plot, loss plot, and saving the model is shown in Figure 23.

```
model.save('model_gaze_3.h5')
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 23, Gaze detection module evaluation and saving

5.2.2.3 Pipelined Gaze Detection Model

This is the stage where the previous two CNN models are combined to form a pipeline for gaze prediction. To start with the implementation, Figure 24 shows the code to extract the gaze directions and images into separate arrays.

```
import pandas as pd
import numpy as np
import os
from tensorflow.keras.utils import Sequence
import cv2
import matplotlib.pyplot as plt


img_size = 224

def load_data(IMAGES_PATH, annotations):
    images = []
    gaze_labels = []


    for index, row in annotations.iterrows():

            img_path = os.path.join(IMAGES_PATH, row[0])
            image = cv2.imread(img_path)

            image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            image = image / 255.0

            image_resized = cv2.resize(image, (img_size, img_size))

            images.append(image_resized)

            label = row[1]

            if label < -1:
                gaze_labels.append('left')
            elif label > 1:
                gaze_labels.append('right')
            else:
                gaze_labels.append('center')


    return np.array(images), np.array(gaze_labels)

annotations1 = pd.read_csv('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet/_annotations.csv')
X, Y = load_data('/kaggle/input/columbia-face-and-bb-dataset/Columbia DataSet', annotations1)
```

Figure 24, Data preparation for the pipeline

Figure 25 shows the code that imports the individual model files created in the previous steps.

```
import tensorflow as tf

bbox_model = tf.keras.models.load_model('/kaggle/input/eye-detection-model/tensorflow1/default/1/modelx9.h5')
gaze_model = tf.keras.models.load_model('/kaggle/input/gaze-detection-model-v3/tensorflow2/default/1/model_gaze_3.h5')
```

Figure 25, Importing individual modules

Figure 26 shows the core of the pipeline in which a Lambda layer is used to concatenate the two models. The lambda layer uses the crop_eyes_single function to crop the eye region using the eye region coordinates.

```
face_input = Input(shape=(224, 224, 3))
bbox_output = bbox_model(face_input)

def crop_eyes_single(face_img, bbox_coords):
    x1, y1, x2, y2, xx1, yy1, xx2, yy2 = tf.split(bbox_coords,
                                                  num_or_size_splits=8, axis=-1)

    x1, y1, x2, y2 = [tf.squeeze(tf.cast(coord, tf.int32)) for coord in [x1, y1, x2, y2]]
    xx1, yy1, xx2, yy2 = [tf.squeeze(tf.cast(coord, tf.int32)) for coord in [xx1, yy1, xx2, yy2]]

    left_eye = face_img[y1:y2, x1:x2, :]
    right_eye = face_img[yy1:yy2, xx1:xx2, :]

    left_eye_resized = tf.image.resize(left_eye, [224, 224])
    right_eye_resized = tf.image.resize(right_eye, [224, 224])

    return tf.stack([left_eye_resized, right_eye_resized], axis=0)

def crop_eyes(face_imgs, bbox_coords):
    return tf.map_fn(
        lambda x: crop_eyes_single(x[0], x[1]),
        (face_imgs, bbox_coords),
        dtype=tf.float32
    )
```

```
cropped_eyes = Lambda(
    lambda x: crop_eyes(x[0], x[1]),
    output_shape=(2, 224, 224, 3),
    name="Lambda_Layer"
)([face_input, bbox_output])

left_eyes = cropped_eyes[:, 0, :, :, :]
right_eyes = cropped_eyes[:, 1, :, :, :]

gaze_direction = gaze_model([left_eyes, right_eyes])
combined_model = Model(inputs=face_input, outputs=gaze_direction)

for layer in bbox_model.layers:
    layer.trainable = False

for layer in gaze_model.layers:
    layer.trainable = False


combined_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 26, Structure of the pipeline model

These coordinates are provided the first model and are used to strip down the original image into two separate eye images. These eye images are then resized to match the input dimensions of the next module.

```
accuracy = accuracy_score(true_labels, predicted_labels)
precision = precision_score(true_labels, predicted_labels, average='weighted', zero_division=0)
recall = recall_score(true_labels, predicted_labels, average='weighted', zero_division=0)
f1 = f1_score(true_labels, predicted_labels, average='weighted', zero_division=0)
conf_matrix = confusion_matrix(true_labels, predicted_labels)


print(f"- Accuracy: {accuracy:.2f}")
print(f"- Precision: {precision:.2f}")
print(f"- Recall: {recall:.2f}")
print(f"- F1-Score: {f1:.2f}")
```

Figure 27, Pipelined Model Evaluation Metrics

Figure 27 shows the metrics that are used to evaluate the model, whereas Figures 28 and 29 highlight the code used for plotting the confusion matric and ROC respectively.

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Left", "Center", "Right", "None"],
            yticklabels=["Left", "Center", "Right", 'None'])
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```

Figure 28, Code for plotting the Confusion Matrix

```
def plot_roc_auc(y_test, y_pred_probs, n_classes):

    fpr = {}
    tpr = {}
    roc_auc = {}

    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred_probs[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    plt.figure()
    colors = ['blue', 'green', 'red', 'purple', 'orange']
    classes = ['left', 'center', 'right']
    for i, color in zip(range(n_classes), colors[:n_classes]):
        plt.plot(
            fpr[i],
            tpr[i],
            color=color,
            lw=2,
            label=f'Gaze {classes[i]} prediction ROC curve (area = {roc_auc[i]:.2f})'
        )

    plt.plot([0, 1], [0, 1], 'k--', lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC For Pipelined Gaze prediction')
    plt.legend(loc='lower right')
    plt.show()

plot_roc_auc(Y_test, results, 3)
```

Figure 29, Code for plotting the ROC