Configuration Manual

MSc Research Project
MSc Cloud Computing

Mahir Ahmed Jabarullah
Student ID: X22134433

School of Computing
National College of Ireland

Supervisor: Mr. Vikas Shani

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ....Mahir Ahmed Jabarullah....................

Student ID: ....X22134433..

Programme: ....MSc Cloud Computing....................... **Year:** ....2024/2025.....

Module: ....Research Project (Configuration Manual) ....

Lecturer: ....Mr. Vikas Shani...

Submission Due Date: ....03/01/2025..

Project Title: Deep-learning and Cloud-based IoT framework for intrusion detection using video surveillance..........

Word Count: .... **Page Count:** ....25.....................................

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: ....mahir ahmed...

Date: ....03/01/25..

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

Hardware Requirements:

1. **Processor**: Multi-core CPU (Intel i7 or AMD equivalent)
2. **RAM**: Minimum 8 GB (16 GB recommended for larger datasets).
3. **Storage**: 20 GB for dataset, model files, and intermediate results.
4. **GPU Memory**: 4 GB or more for faster inference (optional).

Software Requirements:

1. **Operating System**: Windows/Linux/macOS.
2. **Programming Language**: Python 3.8 or newer.
3. **Libraries**:
    - TensorFlow/Keras: For loading and running the FaceNet model.
    - numpy: For numerical operations.
    - opencv-python: For image preprocessing (optional).
    - matplotlib: For visualization.
    - tqdm: For progress bars.
4. **Pretrained Model**: FaceNet model (facenet_keras.h5).
5. **Dataset Management**: File system or a structured folder layout for images.
6. **Environment**: Google Colab

Steps to Execute:

- **Step 1:** Have to execute the code in Google Colab. So, save the code files in google drive.
- **Step 2:** Mount the drive in the colab before run the program.
- **Step 3:** Then import following libraries which we need in the program.

- **Libraries to import:**

- `h5py`: Library for working with HDF5 files.
- `numpy`: For numerical computations.
- `matplotlib.pyplot`: For plotting.
- `%matplotlib inline`: Ensures that plots are displayed inline in Jupyter Notebook.

- **Metrics from `sklearn`:**

- Functions like `precision_recall_curve`, `accuracy_score`, `f1_score`, `precision_score`, and `recall_score` are imported to evaluate model performance.

- `warnings.filterwarnings('ignore')`: Suppresses all warnings from being displayed during the execution.

Path Setup:

```python
import os
source_dir=os.path.join('/kaggle','input','pins-face-recognition','105_classes_pins_dataset')
```

- `os`: Library to handle file and directory paths.
- `source_dir`: Defines the path to the dataset `105_classes_pins_dataset` stored in `/kaggle/input/pins-face-recognition`.

```python
class IdentityMetadata():
    def __init__(self, base, name, file):
        self.base = base
        # identity name
        self.name = name
        # image file name
        self.file = file

    def __repr__(self):
        return self.image_path()

    def image_path(self):
        return os.path.join(self.base, self.name, self.file)

def load_metadata(path):
    metadata = []
    for i in os.listdir(path):
        for f in os.listdir(os.path.join(path, i)):
            # Check file extension. Allow only jpg/jpeg' files.
            ext = os.path.splitext(f)[1]
            if ext == '.jpg' or ext == '.jpeg':
                metadata.append(IdentityMetadata(path, i, f))
    return np.array(metadata)

# metadata = load_metadata('images')
metadata = load_metadata(source_dir)
```

- **`IdentityMetadata` Class**:

- Stores metadata for each image:
  - `base`: Base directory.
  - `name`: Identity name (subfolder name).
  - `file`: Image file name.
- Provides the `image_path()` method to get the full file path.

- **`load_metadata()` Function**:

  - Iterates through the dataset directory.
  - Collects metadata for `.jpg` or `.jpeg` images.
  - Returns an array of `IdentityMetadata` objects.

- **Example**:

  - Loads metadata from `source_dir`, organizing image details for later use.

```
print('metadata shape :', metadata.shape)
```

  - prints the shape of the `metadata` array. Since `metadata` is a NumPy array of `IdentityMetadata` objects (created by `load_metadata()`), its shape will indicate the total number of images processed from the directory.

```
type(metadata[1500]), metadata[1500].image_path()
```

`type(metadata[1500]), metadata[1500].image_path()`:

- `type(metadata[1500])` checks the type of the 1501st metadata entry (likely `<class '__main__.IdentityMetadata'>`).
- `metadata[1500].image_path()` returns the full path of the image corresponding to that metadata entry.

```
import cv2
def load_image(path):
    img = cv2.imread(path, 1)
    # OpenCV loads images with color channels
    # in BGR order. So we need to reverse them
    return img[...,::-1]
```

`load_image(path)` Function:

- Uses OpenCV (`cv2`) to load an image from the given `path`.
- The `cv2.imread(path, 1)` function reads the image in color mode.
- The slicing `[..., ::-1]` reverses the order of color channels (BGR → RGB) to make it compatible with libraries like Matplotlib.

```
load_image('/kaggle/input/pins-face-recognition/105_classes_pins_dataset/pins_Emilia Clarke/Emilia Clarke247_998.jpg')
```

- Load the image.
- Print its type (`numpy.ndarray`) and shape (e.g., `(height, width, 3)`).

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import ZeroPadding2D, Convolution2D, MaxPooling2D, Dropout, Flatten, Activation

def vgg_face():
    model = Sequential()
    model.add(ZeroPadding2D((1,1),input_shape=(224,224, 3)))
    model.add(Convolution2D(64, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(Convolution2D(4096, (7, 7), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Convolution2D(4096, (1, 1), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Convolution2D(2622, (1, 1)))
    model.add(Flatten())
    model.add(Activation('softmax'))
    return model
```

VGG Face Model Definition (`vgg_face()` function):

- Builds a custom VGG-style convolutional neural network (CNN) architecture using `Sequential` from Keras.
- Contains layers like `ZeroPadding2D`, `Convolution2D`, `MaxPooling2D`, `Dropout`, `Flatten`, and `Activation`.
- Includes 16 convolutional layers with increasing filters (64, 128, 256, 512) followed by pooling layers and dropout.
- Ends with fully connected layers for classification (`4096` and `2622` units), dropout, and a softmax activation for multi-class classification.
- The network is designed for face recognition, based on the VGG architecture but with specific weight configurations.

```python
model = vgg_face()

model.load_weights('../input/vgg-face-weights/vgg_face_weights.h5')
```

- The model's weights are loaded from `vgg_face_weights.h5` using `model.load_weights()` method. The weights should correspond to the architecture defined in `vgg_face()`.

```python
from tensorflow.keras.models import Model
vgg_face_descriptor = Model(inputs=model.layers[0].input, outputs=model.layers[-2].output)
```

• Creates a new Keras `Model` (`vgg_face_descriptor`) that takes the same input as the original model but outputs the second-to-last layer (`model.layers[-2]`), which is the feature descriptor for the face.
• This new model is often used for extracting embeddings (feature vectors) for each image, which can then be used for tasks like face verification or identification.

```python
type(vgg_face_descriptor)
```

```
tensorflow.python.keras.engine.functional.Functional
```

```python
vgg_face_descriptor.inputs, vgg_face_descriptor.outputs
```

- **`type(vgg_face_descriptor)`:**
  - This returns the type of the model object (likely <class 'tensorflow.python.keras.engine.training.Model'>), indicating it's a Keras Model object.

- `vgg_face_descriptor.inputs` and `vgg_face_descriptor.outputs`:
  - These give information about the model's input and output layers. For instance, the input is typically an image of shape `(224, 224, 3)` and the output will be a 4096-dimensional feature vector.

```python
# Get embedding vector for first image in the metadata using the pre-trained model
img_path = metadata[0].image_path()
img = load_image(img_path)

# Normalising pixel values from [0-255] to [0-1]: scale RGB values to interval [0,1]
img = (img / 255.).astype(np.float32)
img = cv2.resize(img, dsize = (224,224))
print(img.shape)

# Obtain embedding vector for an image
# Get the embedding vector for the above image using vgg_face_descriptor model and print the shape
embedding_vector = vgg_face_descriptor.predict(np.expand_dims(img, axis=0))[0]
print(embedding_vector.shape)
```

Image Preprocessing:

- **Load Image**: `img = load_image(metadata[0].image_path())` loads the image and converts it from BGR to RGB.
- **Normalization**: `img = (img / 255.).astype(np.float32)` scales pixel values to [0, 1].
- **Resize**: `img = cv2.resize(img, (224, 224))` resizes the image to 224x224.

Generate Embedding:

- `embedding_vector = vgg_face_descriptor.predict(np.expand_dims(img, axis=0))[0]`:
  - Adds batch dimension, predicts embedding vector, and extracts the vector (shape `(4096,)`).

```
embedding_vector[0], type(embedding_vector), type(embedding_vector[0])
```

```
0.018536601, numpy.ndarray, numpy.float32)
```

```
embedding_vector[2], embedding_vector[98], embedding_vector[-2]
```

- Fetches the first element of the `embedding_vector` array and displays its type alongside the type of the entire array.
- Accesses elements at index 2, 98, and the second-to-last index of the `embedding_vector`.

```
total_images = len(metadata)

print('total_images :', total_images)
```

- Counts the number of entries in the `metadata` object and prints the total number of images.

```
embeddings = np.zeros((metadata.shape[0], 2622))
for i, m in enumerate(metadata):
    img_path = metadata[i].image_path()
    img = load_image(img_path)
    img = (img / 255.).astype(np.float32)
    img = cv2.resize(img, dsize = (224,224))
    embedding_vector = vgg_face_descriptor.predict(np.expand_dims(img, axis=0))[0]
    embeddings[i]=embedding_vector
```

- Initializes an array `embeddings` to store feature vectors of size `2622` for each image in `metadata`.
- Loads, normalizes, resizes, and passes each image through a model (`vgg_face_descriptor`) to generate its embedding.

- Stores the embedding in the `embeddings` array at the corresponding index.

```
embeddings[0], embeddings[988], embeddings[988].shape
```

- Fetches the embedding of the first image (`embeddings[0]`).
- Fetches the embedding of the 988th image (`embeddings[988]`).
- Retrieves the shape of the embedding for the 988th image.

```python
def distance(emb1, emb2):
    return np.sum(np.square(emb1 - emb2))
```

- Calculates the squared Euclidean distance between two embeddings `emb1` and `emb2`.

```python
def show_pair(idx1, idx2):
    plt.figure(figsize=(8,3))
    plt.suptitle(f'Distance between {idx1} & {idx2}= {distance(embeddings[idx1], embeddings[idx2]):.2f}')
    plt.subplot(121)
    plt.imshow(load_image(metadata[idx1].image_path()))
    plt.subplot(122)
    plt.imshow(load_image(metadata[idx2].image_path()));

show_pair(900, 901)
show_pair(900, 1001)
```

- Visualizes two images side-by-side, calculates, and displays their distance (using embeddings).

- **Usage**: Calls the `show_pair` function for different pairs of images.

```python
show_pair(1100, 1101)
show_pair(1100, 1300)
```

- Same function as above, used here for different pairs (images at indices 1100 and 1101, and 1100 and 1300).

```python
show_pair(1407, 1408)
show_pair(1408, 1602)
```

- Again, compares two pairs of images using their embeddings, one close in indices (likely similar) and another farther apart (likely dissimilar).

```
train_idx = np.arange(metadata.shape[0]) % 9 != 0     #eve
test_idx = np.arange(metadata.shape[0]) % 9 == 0

# one half as train examples of 10 identities
X_train = embeddings[train_idx]

# another half as test examples of 10 identities
X_test = embeddings[test_idx]
targets = np.array([m.name for m in metadata])

#train labels
y_train = targets[train_idx]

#test labels
y_test = targets[test_idx]
```

- Splits data into training and testing sets using indices divisible by 9.
- Stores embeddings (`x_train`, `x_test`) and corresponding labels (`y_train`, `y_test`).

```
print('X_train shape : ({0},{1})'.format(X_train.shape[0], X_train.shape[1]))
print('y_train shape : ({0},)'.format(y_train.shape[0]))
print('X_test shape : ({0},{1})'.format(X_test.shape[0], X_test.shape[1]))
print('y_test shape : ({0},)'.format(y_test.shape[0]))

X_train shape : (15585 2622)
```

- Prints the shapes of training and test datasets for both features (`X_train`, `X_test`) and labels (`y_train`, `y_test`).

```
y_test[0], y_train[988]

('pins_Alex Lawther', 'pins_Mark Zuckerberg')

len(np.unique(y_test)), len(np.unique(y_train))
```

- Accesses specific entries in `y_test` and `y_train` arrays to verify the labels (e.g., names of individuals).
- Counts the number of unique classes (labels) in `y_test` and `y_train`.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)


print(le.classes_)
y_test_encoded = le.transform(y_test)
```

- Encodes the string labels in `y_train` and `y_test` into numerical values using `LabelEncoder`.
- `le.classes_` stores the mapping of class names to numbers.

```
print('y_train_encoded : ', y_train_encoded)
print('y_test_encoded : ', y_test_encoded)
```

- Prints the encoded training and testing labels to confirm the transformation was successful.

```
# Standarize features
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
```

- Standardizes the feature matrix `X_train` (mean = 0, std = 1).
- Stores the standardized features in `X_train_std`.

```
print('X_train_std shape : ({0},{1})'.format(X_train_std.shape[0], X_train_std.shape[1]))
print('y_train_encoded shape : ({0},)'.format(y_train_encoded.shape[0]))
print('X_test_std shape : ({0},{1})'.format(X_test_std.shape[0], X_test_std.shape[1]))
print('y_test_encoded shape : ({0},)'.format(y_test_encoded.shape[0]))
```

- Prints the shapes of standardized feature matrices and encoded label arrays for both training and testing datasets.

```
from sklearn.decomposition import PCA

pca = PCA(n_component = 128)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
```

- Reduces the dimensionality of standardized features using PCA to `128` components.
- `fit_transform` is applied on the training set, and `transform` is applied on the test set using the same PCA model

```
from sklearn.svm import SVC

clf = SVC(C=5., gamma=0.001)
clf.fit(X_train_pca, y_train_encoded)
```

- Initializes an SVM classifier with specified parameters (`C` for regularization, `gamma` for the kernel coefficient).
- Trains the classifier on the PCA-transformed training data.

```
print('y_predict : ',y_predict)
print('y_test_encoded : ',y_test_encoded)
```

- Displays the predicted labels (`y_predict`) from the model.
- Shows the actual encoded labels (`y_test_encoded`) for comparison.

```
y_test_encoded[32:49]
```

- Accesses and displays a slice of the actual encoded test labels (from index 32 to 48) for a detailed examination of predictions vs. ground truth.

```
# Find the classification accuracy
accuracy_score(y_test_encoded, y_predict)
```

- Computes the classification accuracy by comparing the predicted labels with the actual test labels.

```
pca = PCA(n_component = 256)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
```

- Uses PCA to reduce dimensionality of standardized feature data to `256` components.
- `fit_transform` is applied on the training data; the test data is transformed using the same PCA model.

```
from sklearn.svm import SVC

clf = SVC(C=5., gamma=0.001)
clf.fit(X_train_pca, y_train_encoded)
y_predict = clf.predict(X_test_pca)
y_predict_encoded = le.inverse_transform(y_predict)
```

- Trains an SVM classifier with PCA-reduced training data and labels.
- Predicts test labels and decodes them back to their original categorical form (`y_predict_encoded`).

```
example_idx = 401

example_image = load_image(metadata[test_idx][example_idx].image_path())
example_prediction = y_predict[example_idx]
example_identity =  y_predict_encoded[example_idx]

plt.imshow(example_image)
plt.title(f'Identified as {example_identity}');
```

- Loads an example image from the test set, predicts its label, and displays the image with the predicted label.
- `example_idx = 401` specifies the index of the example in the test set.

```
example_idx = 900

example_image = load_image(metadata[test_idx][example_idx].image_path())
example_prediction = y_predict[example_idx]
example_identity =  y_predict_encoded[example_idx]

plt.imshow(example_image)
plt.title(f'Identified as {example_identity}');
```

- Processes and visualizes a different test image (at index `900`) for classification.

```
example_idx = 317

example_image = load_image(metadata[test_idx][example_idx].image_path())
example_prediction = y_predict[example_idx]
example_identity =  y_predict_encoded[example_idx]

plt.imshow(example_image)
plt.title(f'Identified as {example_identity}');
```

- Allows inspection of another test image's classification result.

```
example_idx = -27

example_image = load_image(metadata[test_idx][example_idx].image_path())
example_prediction = y_predict[example_idx]
example_identity =  y_predict_encoded[example_idx]

plt.imshow(example_image)
plt.title(f'Identified as {example_identity}');
```

- Demonstrates flexibility in index selection (using negative indexing) for visualization and validation.

Celebs_face_recognition

```
# Common
import os
import cv2 as cv
import numpy as np
from IPython.display import clear_output as cls

# Data
from tqdm import tqdm
from glob import glob

# Data Visuaalization
import plotly.express as px
import matplotlib.pyplot as plt

# Model
from tensorflow.keras.models import load_model
```

- Imports commonly used libraries for file handling (`os`, `glob`), numerical computations (`numpy`), image processing (`cv2`), and visualization (`plotly`, `matplotlib`).
- Imports tools for progress tracking (`tqdm`) and model loading (`load_model`).

```
# Setting a random
np.random.seed(42)

# Define the image dimensions
IMG_W, IMG_H, IMG_C = (160, 160, 3)
```

- Sets a random seed for reproducibility in any random operations.
- Defines image dimensions (`160x160x3`, where 3 represents RGB channels).

```
# Specify the root directory path
root_path = '/kaggle/input/pins-face-recognition/105_classes_pins_dataset/'

# Collect all the person names
dir_names = os.listdir(root_path)
person_names = [name.split("_")[-1].title() for name in dir_names]
n_individuals = len(person_names)

print(f"Total number of individuals: {n_individuals}\n")
print(f"Name of the individuals : \n\t{person_names}")
```

- Sets the root path to the dataset.
- Extracts and formats the names of individuals (from directory names).
- Counts the number of unique individuals and prints their names.

```
# Number of images available per person
n_images_per_person = [len(os.listdir(root_path + name)) for name in dir_names]
n_images = sum(n_images_per_person)

# Show
print(f"Total Number of Images : {n_images}.")
```

- Counts the number of images available per person by traversing subdirectories.
- Calculates and displays the total number of images in the dataset.

```
# Plot the Distribution of number of images per person.
fig = px.bar(x=person_names, y=n_images_per_person, color=person_names)
fig.update_layout({'title':{'text':"Distribution of number of images per person"}})
fig.show()
```

- Creates a bar chart using Plotly to visualize the distribution of the number of images per individual.
- Adds a title to the chart and displays it.

```python
# Select all the file paths : 50 images per person.
filepaths = [path  for name in dir_names for path in glob(root_path + name + '/*')[:50]]
np.random.shuffle(filepaths)
print(f"Total number of images to be loaded : {len(filepaths)}")

# Create space for the images
all_images = np.empty(shape=(len(filepaths), IMG_W, IMG_H, IMG_C), dtype = np.float32)
all_labels = np.empty(shape=(len(filepaths), 1), dtype = np.int32)

# For each path, load the image and apply some preprocessing.
for index, path in tqdm(enumerate(filepaths), desc="Loading Data"):

    # Extract label
    label = [name[5:] for name in dir_names if name in path][0]
    label = person_names.index(label.title())

    # Load the Image
    image = plt.imread(path)

    # Resize the image
    image = cv.resize(image, dsize = (IMG_W, IMG_H))

    # Convert image stype
    image = image.astype(np.float32)/255.0

    # Store the image and the label
    all_images[index] = image
    all_labels[index] = label
```

- Selects 50 image paths per person, shuffles them, and initializes arrays for image and label storage.
- Processes each image: reads, resizes, normalizes, and stores it along with its label.

```python
def show_data(
    images: np.ndarray,
    labels: np.ndarray,
    GRID: tuple=(15,6),
    FIGSIZE: tuple=(25,50),
    recog_fn = None,
    database = None
) -> None:

    """
    Function to plot a grid of images with their corresponding labels.

    Args:
        images (numpy.ndarray): Array of images to plot.
        labels (numpy.ndarray): Array of corresponding labels for each image.
        GRID (tuple, optional): Tuple with the number of rows and columns of the plot grid. Defaults to (15,6).
        FIGSIZE (tuple, optional): Tuple with the size of the plot figure. Defaults to (30,50).
        recog_fn (function, optional): Function to perform face recognition. Defaults to None.
        database (dictionary, optional): Dictionary with the encoding of the images for face recognition. Defaults to None.

Returns:
        None
    """

    # Plotting Configuration
    plt.figure(figsize=FIGSIZE)
    n_rows, n_cols = GRID
    n_images = n_rows * n_cols

    # loop over the images and labels
    for index in range(n_images):

        # Select image in the corresponding label randomly
        image_index = np.random.randint(len(images))
        image, label = images[image_index], person_names[int(labels[image_index])]

        # Create a Subplot
        plt.subplot(n_rows, n_cols, index+1)

        # Plot Image
        plt.imshow(image)
        plt.axis('off')

        if recog_fn is None:
            # Plot title
            plt.title(label)
        else:
            recognized = recog_fn(image, database)
            plt.title(f"True:{label}\nPred:{recognized}")

    # Show final Plot
    plt.tight_layout()
    plt.show()
```

- Displays a grid of images with their labels or predictions.
- Optionally performs face recognition and shows predicted labels alongside true labels.

```python
def load_image(image_path: str, IMG_W: int = IMG_W, IMG_H: int = IMG_H) -> np.ndarray:
    """Load and preprocess image.

    Args:
        image_path (str): Path to image file.
        IMG_W (int, optional): Width of image. Defaults to 160.
        IMG_H (int, optional): Height of image. Defaults to 160.

    Returns:
        np.ndarray: Preprocessed image.
    """

    # Load the image
    image = plt.imread(image_path)

    # Resize the image
    image = cv.resize(image, dsize=(IMG_W, IMG_H))

    # Convert image type and normalize pixel values
    image = image.astype(np.float32) / 255.0

    return image

def image_to_embedding(image: np.ndarray, model) -> np.ndarray:
    """Generate face embedding for image.

    Args:
        image (np.ndarray): Image to generate encoding for.
        model : Pretrained face recognition model.

    Returns:
        np.ndarray: Face embedding for image.
    """

    # Obtain image encoding
    embedding = model.predict(image[np.newaxis,...])

    # Normalize bedding using L2 norm.
    embedding /= np.linalg.norm(embedding, ord=2)

    # Return embedding
    return embedding

def generate_avg_embedding(image_paths: list, model) -> np.ndarray:
    """Generate average face embedding for list of images.

    Args:
        image_paths (list): List of paths to image files.
        model : Pretrained face recognition model.

    Returns:
        np.ndarray: Average face embedding for images.
    """
```

```python
# Collect embeddings
embeddings = np.empty(shape=(len(image_paths), 128))

# Loop over images
for index, image_path in enumerate(image_paths):

    # Load the image
    image = load_image(image_path)

    # Generate the embedding
    embedding = image_to_embedding(image, model)

    # Store the embedding
    embeddings[index] = embedding

# Compute average embedding
avg_embedding = np.mean(embeddings, axis=0)

# Clear Output
cls()

# Return average embedding
return avg_embedding
```

This code processes images to generate face embeddings using a pretrained model:

1. **`load_image`**: Loads, resizes, and normalizes an image for model input.
2. **`image_to_embedding`**: Generates and normalizes a face embedding for the image.
3. **`generate_avg_embedding`**: Processes multiple images to compute their average face embedding, representing them as a single embedding.

```python
# Load model
model = load_model('/kaggle/input/facenet-keras/facenet_keras.h5')
```

- Loads a pretrained FaceNet model for generating 128-dimensional face embeddings.

```python
# Select all the file paths : 50 images per person.
filepaths = [np.random.choice(glob(root_path + name + '/*'), size=10) for name in dir_names]

# Create data base
database = {name:generate_avg_embedding(paths, model=model) for paths, name in tqdm(zip(filepaths, person_names), desc="Generating Embeddings")}
```

- Randomly selects 10 images per person from directories (`root_path`).
- `dir_names` contains subdirectory names corresponding to persons.

- Creates a dictionary (`database`) where:
    - **Keys**: Person names (`person_names`).
    - **Values**: Average embeddings generated for their images using `generate_avg_embedding`.

```python
def compare_embeddings(embedding_1: np.ndarray, embedding_2: np.ndarray, threshold: float = 0.8) -> int:
    """
    Compares two embeddings and returns 1 if the distance between them is less than the threshold, else 0.

    Args:
    - embedding_1: A 128-dimensional embedding vector.
    - embedding_2: A 128-dimensional embedding vector.
    - threshold: A float value representing the maximum allowed distance between embeddings for them to be considered a match.

    Returns:
    - 1 if the distance between the embeddings is less than the threshold, else 0.
    """

    # Calculate the distance between the embeddings
    embedding_distance = embedding_1 - embedding_2

    # Calculate the L2 norm of the distance vector
    embedding_distance_norm = np.linalg.norm(embedding_distance)

    # Return 1 if the distance is less than the threshold, else 0
    return embedding_distance_norm if embedding_distance_norm < threshold else 0
```

Compares two 128-dimensional embeddings:

- Computes the Euclidean distance (`L2 norm`) between embeddings.
- Returns the distance if it's below the threshold (match), else returns `0` (no match).

```python
def recognize_face(image: np.ndarray, database: dict, threshold: float = 1.0, model = model) -> str:
    """
    Given an image, recognize the person in the image using a pre-trained model and a database of known faces.

    Args:
        image (np.ndarray): The input image as a numpy array.
        database (dict): A dictionary containing the embeddings of known faces.
        threshold (float): The distance threshold below which two embeddings are considered a match.
        model (keras.Model): A pre-trained Keras model for extracting image embeddings.

    Returns:
        str: The name of the recognized person, or "No Match Found" if no match is found.
    """

    # Generate embedding for the new image
    image_emb = image_to_embedding(image, model)

    # Clear output
    cls()

    # Store distances
    distances = []
    names = []

    # Loop over database
    for name, embed in database.items():

        # Compare the embeddings
        dist = compare_embeddings(embed, image_emb, threshold=threshold)

        if dist > 0:
            # Append the score
            distances.append(dist)
            names.append(name)

    # Select the min distance
    if distances:
        min_dist = min(distances)

        return names[distances.index(min_dist)].title().strip()

    return "No Match Found"
```

Function: `recognize_face`

This function recognizes a face in a given image by comparing its embedding to a database of known face embeddings.

Steps:

1. **Generate Embedding**:
   - The function calls `image_to_embedding` to extract the embedding of the input image using the pretrained model.
2. **Compare Embeddings**:
   - Loops through the `database` dictionary (which contains known embeddings).
   - For each person in the database, compares their embedding with the input image embedding using `compare_embeddings`.
3. **Find Closest Match**:
   - Stores distances for all matches below the threshold.
   - Returns the name of the person with the smallest distance.
   - If no match is found (all distances exceed the threshold), returns "No Match Found."

```
# Randomly select an index
index = np.random.randint(len(all_images))

# Obtain an image and its corresponding label
image_ = all_images[index]
label_ = person_names[int(all_labels[index])]

# Recognize the face in the image
title = recognize_face(image_, database)

# Plot the image along with its true and predicted labels
plt.imshow(image_)
plt.title(f"True:{label_}\nPred:{title}")
plt.axis('off')
plt.show()
```

```
show_data(all_images, all_labels, recog_fn = recognize_face, database = database)
```

```
show_data(all_images, all_labels, recog_fn = recognize_face, database = database)
```

- Selects an image and its true label from the dataset.
- Predicts the label for the selected image using the `recognize_face` function.
- Displays the image along with the true label and predicted label.

```python
# Count the number of images
n_images = 50

# Initialize the number of correct predictions
n_correct = 0

# Randomly Select images
indicies = np.random.permutation(n_images)
temp_images = all_images[indicies]
temp_labels = all_labels[indicies]

# Iterate over each image and its corresponding label
for (image, label) in zip(temp_images, temp_labels):

    # Extract the true label of the person in the image
    true_label = person_names[int(label)]

    # Use the recognize_face function to predict the label of the person in the image
    pred_label = recognize_face(image, database)

    # If the true label and the predicted label match, increment the number of correct predictions
    if true_label == pred_label:
        n_correct += 1

# Calculate the accuracy of the model
acc = (n_correct / n_images) * 100.0

# Print the accuracy of the model
print(f"Model Accuracy: {acc}%!!!")
```

This code evaluates the accuracy of a face recognition model:

1. **Setup**: Selects 50 random images and their labels.
2. **Prediction**: Uses `recognize_face` to predict the label for each image.
3. **Comparison**: Compares predicted labels to true labels and counts correct matches.
4. **Accuracy Calculation**: Computes accuracy as the percentage of correct predictions.
5. **Result**: Prints the model's accuracy.

```python
# Select all the file paths : 50 images per person.
filepaths = [np.random.choice(glob(root_path + name + '/*'), size=50) for name in dir_names]

# Create data base
large_database = {name:generate_avg_embedding(paths, model=model) for paths, name in tqdm(zip(filepaths, person_names), desc="Generating Embeddings")}
```

- For each person, randomly selects 50 image file paths.
- Creates a database (`large_database`) with average embeddings for each person using `generate_avg_embedding`.

```python
show_data(all_images, all_labels, recog_fn = recognize_face, database = large_database)
```

  - Displays data (images and labels) for verification using the `recognize_face` function and the `large_database`.

```python
# Count the number of images
n_images = 100

# Initialize the number of correct predictions
n_correct = 0

# Randomly Select images
indicies = np.random.permutation(n_images)
temp_images = all_images[indicies]
temp_labels = all_labels[indicies]

# Iterate over each image and its corresponding label
for (image, label) in zip(temp_images, temp_labels):

    # Extract the true label of the person in the image
    true_label = person_names[int(label)]

    # Use the recognize_face function to predict the label of the person in the image
    pred_label = recognize_face(image, large_database)

    # If the true label and the predicted label match, increment the number of correct predictions
    if true_label == pred_label:
        n_correct += 1

# Calculate the accuracy of the model
acc = (n_correct / n_images) * 100.0

# Print the accuracy of the model
print(f"Model Accuracy: {acc}%!!!")
```

- Randomly selects 100 images from `all_images` and corresponding labels from `all_labels`.
- Iterates through the selected images and predicts labels using `recognize_face`.
- Compares true and predicted labels, updating the count of correct predictions.
- Computes accuracy as the percentage of correctly classified images.
- Prints the result.

```python
# Select all the file paths : 25 images per person.
filepaths = [np.random.choice(glob(root_path + name + '/*'), size=25) for name in dir_names]

# Create data base
med_database = {name:generate_avg_embedding(paths, model=model) for paths, name in tqdm(zip(filepaths, person_names), desc="Generating Embeddings")}
```

- Creates a medium-sized database with 25 randomly selected images per person.

```python
show_data(all_images, all_labels, recog_fn = recognize_face, database = med_database)
```

- Displays images and corresponding labels using the `recognize_face` function and the medium/large database.

```python
# Count the number of images
n_images = 100

# Initialize the number of correct predictions
n_correct = 0

# Randomly Select images
indicies = np.random.permutation(n_images)
temp_images = all_images[indicies]
temp_labels = all_labels[indicies]

# Iterate over each image and its corresponding label
for (image, label) in zip(temp_images, temp_labels):

    # Extract the true label of the person in the image
    true_label = person_names[int(label)]

    # Use the recognize_face function to predict the label of the person in the image
    pred_label = recognize_face(image, med_database)

    # If the true label and the predicted label match, increment the number of correct predictions
    if true_label == pred_label:
        n_correct += 1

# Calculate the accuracy of the model
acc = (n_correct / n_images) * 100.0

# Print the accuracy of the model
print(f"Model Accuracy: {acc}%!!!")
```

- Randomly picks 100 images and corresponding labels.
- Uses `recognize_face` to predict the label for each image and compares it with the true label.
- Calculates and prints the model accuracy as a percentage.

```python
# Select all the file paths : 65 images per person.
filepaths = [np.random.choice(glob(root_path + name + '/*'), size=65) for name in dir_names]

# Create data base
med_database = {name:generate_avg_embedding(paths, model=model) for paths, name in tqdm(zip(filepaths, person_names), desc="Generating Embeddings")}
```

```python
show_data(all_images, all_labels, recog_fn = recognize_face, database = med_database)
```

- Creates a larger database with 65 images per person for more comprehensive recognition.

```python
# Count the number of images
n_images = 100

# Initialize the number of correct predictions
n_correct = 0

# Randomly Select images
indicies = np.random.permutation(n_images)
temp_images = all_images[indicies]
temp_labels = all_labels[indicies]

# Iterate over each image and its corresponding label
for (image, label) in zip(temp_images, temp_labels):

    # Extract the true label of the person in the image
    true_label = person_names[int(label)]

    # Use the recognize_face function to predict the label of the person in the image
    pred_label = recognize_face(image, med_database)

    # If the true label and the predicted label match, increment the number of correct predictions
    if true_label == pred_label:
        n_correct += 1

# Calculate the accuracy of the model
acc = (n_correct / n_images) * 100.0

# Print the accuracy of the model
print(f"Model Accuracy: {acc}%!!!")
```

This code evaluates face recognition accuracy:

1. **Randomly selects 100 images** from the dataset.
2. **Predicts labels** for each image using `recognize_face` with the medium-sized database (`med_database`).
3. **Compares predictions** to true labels and counts correct matches.
4. **Calculates accuracy** as `(correct predictions / 100) * 100`.
5. **Prints the model accuracy** as a percentage.

References:

1. Kandhro, I.A., Alanazi, S.M., Ali, F., Kehar, A., Fatima, K., Uddin, M. and Karuppayah, S., 2023. Detection of real-time malicious intrusions and attacks in IoT empowered cybersecurity infrastructures. *IEEE Access*, *11*, pp.9136-9148.Fathy, C., & Saleh, S. N. (2022). Integrating Deep Learning-Based IoT and Fog Computing with Software-Defined Networking for Detecting Weapons in Video Surveillance Systems. Sensors. https://doi.org/10.3390/s22145075

2. Nizamudeen, S.M.T., 2023. Intelligent intrusion detection framework for multi-clouds–IoT environment using swarm-based deep learning classifier. *Journal of Cloud Computing*, *12*(1), p.134.Norov, S. K. (2023). Intelligent Intrusion Detection Framework for Multi-Clouds – Iot Environment Using Swarm-Based Deep Learning Classifier. https://doi.org/10.21203/rs.3.rs-2409418/v1

3. Wang, G., Li, J., Wu, Z., Xu, J., Shen, J. and Yang, W., 2023. EfficientFace: an efficient deep network with feature enhancement for accurate face detection. *Multimedia Systems*, *29*(5), pp.2825-2839.

4. Mamieva, D., Abdusalomov, A.B., Mukhiddinov, M. and Whangbo, T.K., 2023. Improved face detection method via learning small faces on hard images based on a deep learning approach. *Sensors*, *23*(1), p.502.

5. Singh, S., Ahuja, U., Kumar, M., Kumar, K. and Sachdeva, M., 2021. Face mask detection using YOLOv3 and faster R-CNN models: COVID-19 environment. *Multimedia Tools and Applications*, *80*, pp.19753-19768.