Class note

FACEnet

Importing Libraries

```
import tensorflow as tf
from keras_facenet import FaceNet
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
import os
```

- **TensorFlow** and **Keras** are imported for building and training the deep learning model.
- **FaceNet** is imported from the `keras_facenet` package for generating facial embeddings.
- Various layers and model components (`Dense`, `Dropout`, `Flatten`, `Model`, etc.) are imported from Keras.
- **ImageDataGenerator** is used for data augmentation and preprocessing.
- **os** is imported for file handling.

Setting Constants

```
BATCH_SIZE = 32
IMAGE_SIZE = (160, 160)  # FaceNet input size
TRAIN_DIR = '/content/drive/MyDrive/test_vggface'  # Training dataset directory
NUM_CLASSES = 10  # Adjust to your number of face classes
```

- `BATCH_SIZE`: Number of samples processed before the model is updated.
- `IMAGE_SIZE`: Dimensions to which input images will be resized (FaceNet requires images of size 160x160).
- `TRAIN_DIR`: Directory path where the training dataset is stored.
- `NUM_CLASSES`: Number of different classes (faces) in the dataset.

Data Augmentation

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,
    width_shift_range=0.2,
```

```
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest',
    validation_split=0.2  # Use 20% of the data for validation
)
```

- **ImageDataGenerator** creates batches of tensor image data with real-time data augmentation.
- **rescale** normalizes pixel values to the range [0, 1].
- **rotation_range, width_shift_range, height_shift_range, shear_range, zoom_range, horizontal_flip** are various augmentations applied to increase diversity in the training dataset.
- **validation_split** specifies that 20% of the data will be set aside for validation.

Creating Data Generators

```
train_generator = train_datagen.flow_from_directory(
    TRAIN_DIR,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='training'  # Use this for the training set
)

val_generator = train_datagen.flow_from_directory(
    TRAIN_DIR,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='validation'  # Use this for the validation set
)
```

- **flow_from_directory** generates batches of augmented data from the training directory.
- **target_size** resizes images to the specified size.
- **class_mode='categorical'** indicates that the labels are one-hot encoded (for multi-class classification).
- Separate generators are created for training and validation datasets.

Loading FaceNet Model

```
facenet_model = FaceNet()
```

- Initializes the FaceNet model for generating facial embeddings.

Building the Classification Model

```
inputs = tf.keras.layers.Input(shape=(160, 160, 3))
embeddings = facenet_model.model(inputs)

# Add classification layers
x = Flatten()(embeddings)  # Flatten the embeddings
x = Dense(128, activation='relu')(x)  # Add a fully connected layer
x = Dropout(0.5)(x)  # Dropout to reduce overfitting
output = Dense(NUM_CLASSES, activation='softmax')(x)  # Output layer for
classification
```

- **Input Layer**: Defines the input shape for the model.
- **embeddings**: The FaceNet model processes the input to produce embeddings.
- **Flatten**: Converts the 3D embeddings to 1D.
- **Dense**: Fully connected layer with 128 neurons and ReLU activation function.
- **Dropout**: Regularization technique to prevent overfitting by randomly setting 50% of the inputs to zero during training.
- **Output Layer**: Final layer with softmax activation for multi-class classification.

Defining the Model

```
model = Model(inputs=inputs, outputs=output)
```

- Combines the input and output into a Keras model.

Compiling the Model

```
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- The model is compiled with the Adam optimizer, categorical cross-entropy loss function, and accuracy metric.

Training the Model

```
EPOCHS = 20  # Adjust this based on your hardware and needs

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // BATCH_SIZE,
    validation_data=val_generator,
    validation_steps=val_generator.samples // BATCH_SIZE,
    epochs=EPOCHS,
```

```
    verbose=1
)
```

- **EPOCHS**: Defines the number of times the model will see the entire training dataset.
- **fit**: Trains the model using the training and validation generators.
    - **steps_per_epoch** and **validation_steps** are calculated based on the number of samples and batch size.
    - **verbose** controls the amount of output during training.

Evaluating the Model

```
# Evaluate the model on training data
loss, accuracy = model.evaluate(train_generator)
print(f'Training Accuracy: {accuracy * 100:.2f}%')

# Evaluate the model on validation data
val_loss, val_accuracy = model.evaluate(val_generator)
print(f'Validation Accuracy: {val_accuracy * 100:.2f}%')
```

- Evaluates the trained model on both training and validation datasets, printing the accuracy for each.

Predicting Random Images

```
import numpy as np
import matplotlib.pyplot as plt
import random

# Function to predict random images from the validation generator
def predict_random_images(generator, model, num_images=9):
    # Get class indices and names
    class_indices = generator.class_indices
    class_names = list(class_indices.keys())  # Class names in the order of their
indices
    total_batches = len(generator)  # Total number of batches
    num_batches = (num_images + generator.batch_size - 1) // generator.batch_size
# Number of batches needed

    # Randomly select batch indices
    random_batches = random.sample(range(total_batches), num_batches)

    images = []
    actual_labels = []
    predicted_classes = []
    confidences = []

    for batch_idx in random_batches:
        # Get the batch of images and labels
```

```
        batch_images, batch_labels = generator[batch_idx]

        # Make predictions for the batch
        predictions = model.predict(batch_images)

        # Loop through each image in the batch
        for i in range(len(batch_images)):
            if len(images) >= num_images:
                break  # Stop if we have enough images
            images.append(batch_images[i])
            actual_labels.append(batch_labels[i])
            predicted_class = np.argmax(predictions[i])
            predicted_classes.append(predicted_class)
            confidences.append(np.max(predictions[i]) * 100)  # Get confidence
score

    # Plot the images with their actual and predicted labels
    plt.figure(figsize=(12, 12))
    for i in range(num_images):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i])  # Display the image
        plt.title(f"Actual: {class_names[np.argmax(actual_labels[i])]} \n"  #
Getting the actual class name
                  f"Predicted: {class_names[predicted_classes[i]]}\n"
                  f"Confidence: {confidences[i]:.2f}%")
        plt.axis("off")
    plt.show()

# Call the function to predict 9 random images from the validation generator
predict_random_images(val_generator, model, num_images=9)
```

- Imports **NumPy** and **Matplotlib** for data handling and visualization.
- Defines a function `predict_random_images` to randomly select images from the validation set and predict their classes using the trained model.
- The function retrieves actual and predicted classes, confidence scores, and displays the results in a grid.

Summary

This code sets up a complete pipeline for training a facial recognition model using pre-trained embeddings from FaceNet, including data augmentation, model creation, training, evaluation, and visualization of predictions. The model is capable of classifying images into multiple face classes based on the provided dataset.

FaceNet is a deep learning model developed by Google that performs face recognition and verification. It is based on a convolutional neural network (CNN) architecture and uses a technique called triplet loss to learn a compact embedding of faces. Here's an overview of FaceNet and how it is utilized in the provided project.

Overview of FaceNet

1. **Purpose**:
    - FaceNet aims to map facial images into a compact Euclidean space where the distance between points corresponds to the similarity between faces. The closer two points are in this space, the more similar the corresponding faces are.
2. **Embedding**:
    - Instead of classifying faces directly, FaceNet generates **embeddings** (vector representations) of faces. Each embedding is a 128-dimensional vector that captures the unique features of a face.
    - The embeddings allow for efficient face recognition and verification. In practice, two faces can be compared by calculating the distance between their embeddings (e.g., using Euclidean distance or cosine similarity).
3. **Triplet Loss**:
    - FaceNet uses a **triplet loss function** during training. A triplet consists of three images: an anchor image (a face to identify), a positive image (a different image of the same person), and a negative image (an image of a different person). The goal is to ensure that the anchor is closer to the positive than to the negative in the embedding space.
    - This training approach allows FaceNet to learn the critical features of faces that differentiate one person from another effectively.
4. **Applications**:
    - Face recognition (identifying or verifying a person from an image).
    - Face clustering (grouping images of the same person).
    - Face search (finding a particular face from a large dataset).

How FaceNet is Used in Your Project

In your project, FaceNet is employed for the following purposes:

1. **Embedding Generation**:
    - FaceNet is loaded as a pre-trained model (using `facenet_model = FaceNet()`). The model processes the input images to generate embeddings, which are used for classification.

In your code, the embeddings are obtained using the line:

```
embeddings = facenet_model.model(inputs)
```

    -
2. **Model Architecture**:

    - The embeddings produced by FaceNet are fed into a custom neural network model for classification.
    - After extracting the embeddings, the model adds classification layers, including a `Flatten` layer, a fully connected `Dense` layer, a `Dropout` layer for regularization, and an output `Dense` layer with softmax activation for multi-class classification. This means your model learns to distinguish between different faces based on their embeddings.
3. **Training and Evaluation**:
    - The model is trained on a dataset of images where each image corresponds to a different face class (e.g., different individuals).
    - By using the FaceNet embeddings, the model can generalize better and achieve higher accuracy in classifying faces, even with variations in lighting, expression, or orientation.
4. **Image Classification**:
    - The project aims to classify images into `NUM_CLASSES`, representing different individuals' faces. The embeddings serve as features that the model uses to predict the class of each input image.

Summary

Overall, FaceNet provides a powerful embedding mechanism that enhances the performance of the facial recognition system in your project. By using a pre-trained model, you leverage the rich feature representation learned from a vast amount of data, allowing your classification model to focus on distinguishing between classes effectively. The combination of FaceNet's embeddings and your custom model architecture allows for robust and accurate facial recognition capabilities.