1. Introduction And Discussion The Problem

Problem:

i. In given video, you should scan for the celebrity on the scene (i.e. frame).

ii. If you found the celebrity, put him/her in a bounding box.

iii. Try to register other people and scan also them on the frames.

In this problem, we are given two videos for training and testing. We need to scan the faces of the people in these videos. Afterwards, we are expected to put the found celebrity into a bounding box. We are asked to create a data set with the photographs of the celebrities we scanned and apply the following steps based on them.

Part 1

Model Phase 1

At this stage of the problem, it is necessary to first download the given videos and create a training dataset from these videos. This dataset should consist of the faces of the celebrities in the video. The sizes of face images should be variable, we can evaluate them as large face and small face. Our classes will be as follows; the small face of the famous guest, the big face of the famous guest, the big face of the presenter, the small face of the presenter and the faces of other people except these two people. Therefore, we need a total of five categories. The next part of the problem is that the model we will create must correctly classify the photographs in the data set obtained from this video. There are various classification algorithms available, I will talk about them in the following stages. Then, by giving a photo of a celebrity to the system, the algorithm we trained is expected to find out which class this celebrity belongs to. We need to demonstrate the accuracy of training and testing data at each epoch. And, we complete the first stage of the problem by giving the confusion matrix of the resulting model.

Location Finding Phase 2

When we come to the second stage of the first part of our problem, our model is expected to detect the faces of celebrities in the video and frame them. It is expected that the name of the class they belong to, that is, their location, is written in the frame.

Part 2

Registering Phase 3

When we move on to the second part of our problem, we want to use OpenCV's face detector. With this, the faces in the video will be selected, divided into classes according to our trained model, and the faces determined to not belong to any class will be collected in the created others folder. Finally, the number of dissimilar faces is expected to be determined by comparing the collected data.

After defining our problem, we can now move on to explain how these desired stages will be implemented, what methods and algorithms will be used while implementing them, and what these methods are.

2. Solving The Problem And Describing Methodologies

Part 1

Model Phase 1

There are various methods to create a dataset by capturing faces from video. Although there are various methods for face recognition and data set creation, especially OpenCV, in this problem I created the data set myself. I took screenshots of the training and test videos, the faces of the guest celebrities and the host of the program in different sizes. I made the necessary grouping for these images and I had a small data set. We can see what this data set looks like in **figures 2.1** and **2.2.**

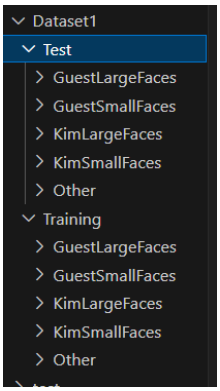


Figure 2.1

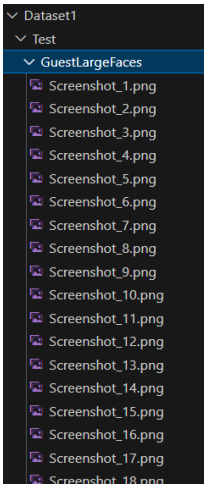
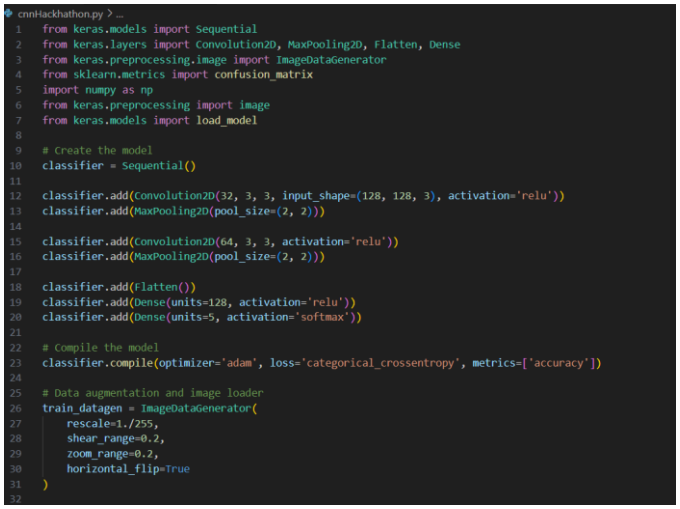


Figure 2.2

After preparing our data set, we can start preparing our model. We can see our entire model code in **figures 2.3, 2.4** and **2.5.**



```
cnnHackhathon.py > ...
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D, Flatten, Dense
from keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import confusion_matrix
import numpy as np
from keras.preprocessing import image
from keras.models import load_model

# Create the model
classifier = Sequential()

classifier.add(Convolution2D(32, 3, 3, input_shape=(128, 128, 3), activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

classifier.add(Convolution2D(64, 3, 3, activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

classifier.add(Flatten())
classifier.add(Dense(units=128, activation='relu'))
classifier.add(Dense(units=5, activation='softmax'))

# Compile the model
classifier.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Data augmentation and image loader
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)
```

Figure 2.3

```
test_datagen = ImageDataGenerator(rescale=1./255)

batch_size = 64

training_set = train_datagen.flow_from_directory(
    'Dataset1/Training',
    target_size=(128, 128),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=True,
    color_mode='rgb'
)

test_set = test_datagen.flow_from_directory(
    'Dataset1/Test',
    target_size=(128, 128),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False,
    color_mode='rgb'
)

# Train the model
classifier.fit(
    training_set,
    steps_per_epoch=len(training_set),
    epochs=10,
    validation_data=test_set,
    validation_steps=len(test_set)
)
```

Figure 2.4

```
# Save the model
classifier.save('trained_model.h5')

# Load the model
loaded_model = load_model('trained_model.h5')

# Predictions and Confusion Matrix
test_set.reset()
pred = classifier.predict(test_set, verbose=1)
pred = np.argmax(pred, axis=1)

test_labels = test_set.classes

cm = confusion_matrix(test_labels, pred)
print('Confusion Matrix: ')
print(cm)

# Example Prediction
test_image = image.load_img('Dataset1/Test/KimLargeFaces/Screenshot_1.png', target_size=(128, 128))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis=0)
result = classifier.predict(test_image)
prediction_class = np.argmax(result, axis=1)

class_indices = training_set.class_indices
class_labels = {v: k for k, v in class_indices.items()}

prediction = class_labels[prediction_class[0]]
print("Prediction:", prediction)
```

Figure 2.5

Now I will explain what is done here. First, we start by importing the necessary libraries into the project. If we briefly touch upon what these libraries are and what they do, the **Sequential** class is a class used in the **Keras** library to define and build a model sequentially. This class allows you to build a neural network model by adding layers sequentially. So, after adding one layer, you can add the next layer and this way you can build the model step by step.

Convolution2D, MaxPooling2D, Flatten, and Dense layers are different types of layers used to create neural network models and are often used in image processing problems.

Convolution2D Layer:

This layer is used to perform convolution-based feature extraction. It applies the convolution process to identify features in images and highlight important features. For example, it can learn edges, patterns, and other visual features.

MaxPooling2D Layer:

Max pooling is used to reduce the size of the feature map obtained from the convolution process. This helps reduce size while highlighting important features in the feature map.

Flatten Layer:

The Flatten layer converts the feature map into a flat layer by flattening it (converting it to a vector). This turns the features from the convolution and pooling layers into a flat vector.

Dense Layer:

The Dense layer is a fully connected layer and contains connections between all input neurons and output neurons. This layer usually represents one or more fully connected layers of the neural network.

These layers are often used sequentially within a Sequential model. For example, in a typical Convolutional Neural Network (CNN) model, the Convolutional (Conv2D) and Pooling (MaxPooling2D) layers perform feature extraction, while the Flatten layer performs the flattening process and the Dense layers perform the classification task.

The keras.preprocessing.image module is part of the Keras library used to process image data. This module includes auxiliary functions for data augmentation and image data loading, especially used during neural network training.

The confusion matrix function is used to evaluate the performance of the model in classification problems. This function creates a confusion matrix based on actual and predicted class labels. The rows of the matrix represent the actual classes, and the columns represent the classes predicted by the model. This matrix is used to visualize correct classifications, incorrect classifications, and performance by class.

The numpy is a library for working on large, multidimensional arrays and matrices in Python. The alias np is often used as an abbreviation for numpy. It is frequently used in neural network applications, especially for data manipulation and mathematical operations.

Keras' image module is used for image processing and matching to neural network models. The **load_img** function is used to load an image with the specified size, while the img_to_array function is used to convert a PIL (Python Imaging Library) image into a NumPy array.

The load_model function is used to load a model trained in Keras from disk. The weights and architecture of the model are included, making it possible to reuse or continue a previously trained model.

After defining the necessary modules, we can come to the part where we create the model. This section is about creating a Convolutional Neural Network (CNN) model and adding its layers. A model is created from the Sequential class. This class is used to define simple and sequential models by adding layers sequentially. Convolutional layer is added. This layer contains 32 filters, each 3x3 in size. The input shape (input_shape) is an image with a size of 128x128 pixels and 3 color channels (RGB). ReLU (Rectified Linear Unit) is used as the activation function. Afterwards, the MaxPooling layer is added. This layer takes the output of the previous convolution layer and reduces the size by taking the maximum value within each 2x2 region. This highlights important features in the feature map and reduces the computational cost. Then the second Convolutional layer is added. The number of filters of this layer is 64 and it also has filters of 3x3 size. We do not need to specify the input form, because the output of the previous layer is automatically received. And the second MaxPooling layer is added and reduces the size even further by taking the output of the previous Convolutional layer. The Flatten layer flattens (vectorizes) the output of the previous layer into a flat layer. This is a necessary step to transition to fully connected layers. Two fully connected layers are then added. The first layer contains 128 neurons and the ReLU activation function. The second layer has 5 neurons and a softmax activation function.

Softmax is used for multi-class classification problems (assumed to be 5 classes) and produces probabilities for each class.

This block of code creates a CNN model and provides an architecture that can be used specifically for image classification tasks.

ReLU (Rectified Linear Unit) and **Softmax** are activation functions used in neural networks.

Activation functions are mathematical functions that determine the output of each neuron in neural networks. These functions are used to increase the learning abilities of neural networks and achieve better performance in various tasks. If the problem is a classification problem and a neural network is established accordingly, the sigmoid activation function can be used on all neurons. Recently, the use of relu has been preferred in neurons in other layers other than the last neuron. I used softmax instead of sigmoid in the output neuron. Because it gives better performance in problems involving more than two classes. Sigmoid activation function generally provides better performance in binary classification. Next, there is a block of code that shows how data loaders are created and configured for the model's training and testing datasets. During the compilation phase, the optimizer, loss function and evaluation metrics of the model are determined.

optimizer='adam': Adam optimizer is an optimization algorithm with adaptive learning rate. It usually works effectively on various types of problems.

loss='categorical_crossentropy': It is the cross-entropy loss commonly used for multi-class classification problems. It measures the difference between the model's predictions and the actual labels.

metrics=['accuracy']: The metrics used to evaluate the performance of the model are determined. In this case, classification accuracy is measured. Data sets are loaded, purified and normalized.

After getting our data ready and training our model, we save and load the model as it will be needed in other stages. And now we create our confusion matrix. We want our model to classify the image according to the image given with the last part.

Output Discussion

In **Figure 2.6** we can see the output of the model I trained.

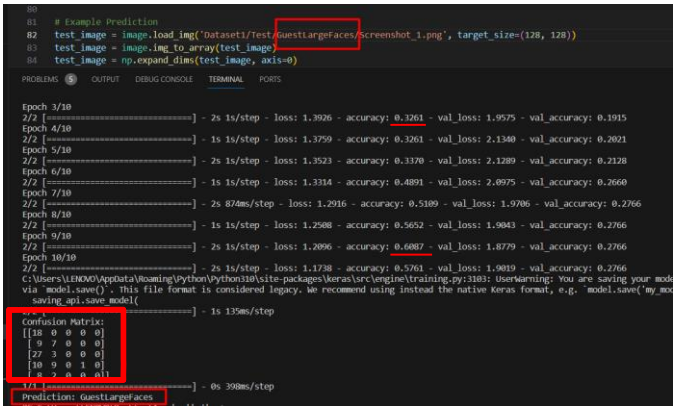


Figure 2.6

We can also see the change in accuracy by following the epochs. The model increased its accuracy rate as it learned, but we cannot always say that the highest accuracy rate is the best because this may also be overfitting. It is important to pay attention to the optimal range. At the same time, we see our confusion matrix in the output, since our data set is not large enough, the accuracy values are a little bad, but it still gives us information about our model. And finally, we can see in the output that it correctly classified a test image we gave to our model. In Fig**ure 2.7,** we can visually see our **accuracy** and **loss graphs.**

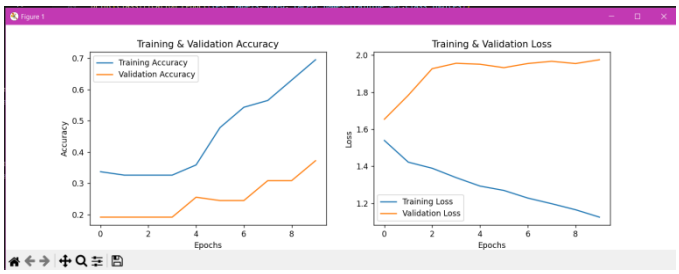


Figure 2.7

In summary, at this stage, I have fulfilled the requirements of the first stage by using the convolution neural network model and training our model on our data set.

Location Finding Phase 2

At this stage, I enabled our model to recognize the faces in the video, show them in the frame and show their locations. We can see the code of the solution I brought at this stage of the problem in **figures 2.8** and **2.9.**

```python
import cv2
import numpy as np
from tensorflow.keras.models import load_model

# Read the video
cap = cv2.VideoCapture("Test_Video_Jimmy Kimmel-Jodie Foster.mp4")
model = load_model("trained_model.h5")
img_size = (128, 128)
class_names = ["Guest Large Face", "Guest Small Face", "Kim Large Face", "Kim Small Face", "Other"]

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Face detection
    face_cascade = cv2.CascadeClassifier(
        cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(
        gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

    for (x, y, w, h) in faces:
        # Crop the face
        face_image = frame[y:y+h, x:x+w]

        # Resize and normalize the face
        face_image = cv2.resize(face_image, img_size)
        face_image = face_image / 255.0  # Normalization

        # Make predictions using the model
        predictions = model.predict(np.expand_dims(face_image, axis=0))
        predicted_class_index = np.argmax(predictions)
        predicted_class = class_names[predicted_class_index]
```

Figure 2.8

```python
        # Display the results on the frame
        text = f"{predicted_class} ({x}, {y})"
        cv2.putText(frame, text, (x, y-10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)

    # Show the results
    cv2.imshow("Video", frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Figure 2.9

This code detects faces by reading each frame in a video file and performs classification using a trained deep learning model for each detected face. It renders facial details using the content of **haarcascade_frontalface_default.xml** with a Haar Cascade classifier available in the OpenCV library. Each detected face is resized and normalized to be fed as input to the model. The model outputs colors for its faces and determines the predicted class name. A rectangle of the entirety of one of its faces is drawn and displayed on the screen along with its location. The reason why we recorded and loaded our model in the first stage was to be able to detect and classify faces in the video with the model we recorded.

Output Discussion

We can see a few examples of the outputs of our code in **figures 2.10** and **2.11**, where we capture the faces of the people in the video with the model we trained and show their locations.

Figure 2.10

Figure 2.11

Part 2

Registering Phase 3

When we come to the last part, the task expected from us here is to detect the faces in the video and enable our model to classify them. If our model detects faces that are contrary to the classes it has been trained in, it should create an others folder for them and tell us the total unique face as a result. We can see the code I prepared for this in **figures 2.12** and **2.13.**

```python
import cv2
import os
import hashlib
from keras.models import load_model
from keras.preprocessing import image
import numpy as np

# Open the video file
video_path = 'Test_Video_Jimmy Kimmel-Jodie Foster.mp4'
cap = cv2.VideoCapture(video_path)

# Load your custom face detection model
model_path = 'trained_model.h5'
face_detection_model = load_model(model_path)

# Create a folder for the "others" dataset
others_folder = 'others'
os.makedirs(others_folder, exist_ok=True)

# Create a list to keep track of previously seen faces and their hashes
seen_faces = []

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Perform face detection
    resized_frame = cv2.resize(frame, (128, 128))
    img_array = image.img_to_array(resized_frame)
    img_array = np.expand_dims(img_array, axis=0) / 255.0
```

Figure 2.12

```python
    # Face detection
    predictions = face_detection_model.predict(img_array)
    if predictions[0][0] > 0.5:  # Example threshold value, you can adjust it according to your needs
        face_image = resized_frame

        # Convert image data to a bytes object
        face_bytes = cv2.imencode('.jpg', face_image)[1].tobytes()

        # Create the hash of the face
        face_hash = hashlib.sha256(face_bytes).hexdigest()

        # If this feature (hash) has not been seen before, add the face to the "others" dataset
        if face_hash not in seen_faces:
            seen_faces.append(face_hash)
            face_number = len(seen_faces)
            cv2.imwrite(os.path.join(others_folder, f"face_{face_number}.jpg"), face_image)

    # Show the video
    cv2.imshow(frame)
    if cv2.waitKey(25) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()

# Print the number of unique faces seen
print(f"Number of Unique Faces: {len(seen_faces)}")
```

Figure 2.13

This code determines faces using the threshold value based on the face detection results of each frame and does not re-register faces it has seen before. In this way, it stores new faces that were not detected in previous frames in the folder named "others". I also used hashlib library here. **Hashing** is used to create a unique representation of data, as used in this example.

The hashlib library is a Python standard library that provides various hash algorithms. The resulting facial image data (byte sequence) is summarized using the SHA-256 hash algorithm. The generated face hash is checked against the hash list of previously seen faces (seen_faces).

If this hash has not been seen before, the face is added to the "others" dataset and added to the hash list of seen faces. This usage is used to prevent the same face from being registered more than once. That is, even if the same face appears in multiple frames, it is recorded only once. Hashing is a method that ensures uniqueness and allows for quick comparison.

Output Discussion

My code classified the faces with the help of the model I trained, as in figure **2.14**, and detected different faces and collected them in the others folder. And if it came across the same face in the others folder, it did not put it, so unique faces were collected in the others folder. When we query our Others folder, we expect the result that tells us how many unique faces there are.

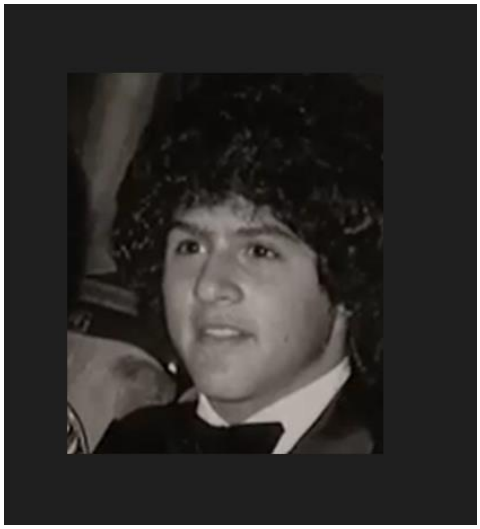


Figure 2.14

Alternatives And Corrections

Alternatively, we could solve our problem by using different artificial neural network models. For example, support vector machine (SVM) can be used. SVM is particularly effective in classification tasks and is used in many application areas. However, generally deep learning models, especially Convolutional Neural Networks (CNN), perform better on image classification tasks. If your data set is small, it may be useful to use SVM or other traditional machine learning models. Model selection may vary depending on the available data set, complexity of the task, and other factors.

Possible corrections to solve the problem are as follows: The data set may need to be edited first. A larger data set will increase training performance. It would also be useful to carry out studies to prevent the danger of overfitting in the model. Adjusting the optimizer and learning rate can have a big impact on performance. Different optimization algorithms (SGD, Adam, RMSprop) and learning rates can also be tried.

Conclusion And Future Advises

As a result, I can list the steps I took throughout the problem as follows: Data collection, Model Training and model evolution. The Convolutional Neural Network (CNN or ConvNet) model was created by processing artificial neural network methods. The model was trained with the obtained data set and the desired steps in the problem were carried out. Convolutional Neural Networks are particularly successful in image processing applications, especially by reducing the number of parameters and learning features within the data hierarchically. That's why I chose this model. This type of architecture is also suitable for transfer learning, that is, it enables the reusability of a previously trained model for a new task.

If I had to give advice, I would suggest paying attention to the following issues.

It may be a matter of debate how sufficient our data set is or whether it is suitable for the desired problem. Because a poor data set cannot train our model sufficiently and may cause various performance problems. For this, the data set we will use must be good and sufficient. At the same time, factors such as the architecture of the model, the number of layers used, their dimensions, and activation functions also affect its performance. The architecture of the model must be suitable for a specific task. An inadequate or overly complex model can have a negative impact on accuracy. Additionally, hyperparameters such as learning rate, number of epochs, and batch size are parameters that need to be adjusted during training of the model. Incorrect selection of these values can affect the training process and results. Finally, following developing technologies closely but also trying to understand the logic of the system with old technologies will take us much further.