Task Overview

The objective of this task is to classify images of players into separate classes from a badminton game data, consisting of two folders and an image of the court.

Ideas and Thought Process

1. **Implementation 1- CNN based approach:** The task at hand is a standard image classification problem, and based on the structure of the data (image-based, with distinct visual features separating the players), a deep learning approach using Convolutional Neural Networks (CNNs) was the first approach I thought of.
   Since, CNNs are well-suited for image classification because they efficiently capture spatial hierarchies and features (edges, textures, patterns) using convolutional layers. As the network goes deeper, it captures more complex features, which helps the model differentiate between similar classes (players in our case).

2. **Implementation 2- Using OpenCV techniques:** The main idea was to isolate the players from the background (court) and then classify them into different groups based on visual features. The first step involves removing the court's color, which acts as a background, so we can focus only on the players. Once the players are isolated, the next step was to classify these segmented players into four clusters.
   a. Segmentation: To isolate the players from the background, we used color segmentation. Since players are generally distinguishable from the court by their color, we set specific color ranges (in HSV format) to identify the court and remove it from the image.
   b. Clustering: After segmenting the players from the court, clustering was used to assign each player to one of four groups. Instead of using deep learning models for classification here, I opted for a KMeans clustering algorithm, which allows grouping based on visual similarity.

3. **Implementation 3- YOLO based approach:** This approach was thought of and sort of implemented (not completely or successfully :‹) just out of curiosity. Since the effectiveness of YOLO lies in detection tasks, which can be modified to perform classification tasks by concatenating classification scores with the detection output. But this is usually done for objects belonging to entirely different classes like car, dog, person, or tv. I wanted to see its performance/effectiveness

in classifying different people especially on low quality images pertaining to our task. Since we only needed four classes, I decided to fine-tune YOLOv5 on a custom dataset. The model will identify these four distinct players based on annotated images.

Explanation of the Implementations

1. **Implementation 1- CNN based approach:**
    a. Dataset Creation- I started by creating a training set for our proposed CNN based model. I used 20 images per class (i.e 80/360) images to train the model. These images are present inside the ‘Implementation_1’ directory. The images were selected such that they capture variety in a particular class.
    b. Data Preprocessing- The data was processed using ImageDataGenerator with several transformations to ensure better generalization of the model. The images were resized to a target size of 150x150, normalized (rescaled by 1/255). The augmentation included random rotations, shifts, shearing, zooming, and flips. This prevents the model from overfitting by making it more resilient to variations in the data.
    c. Model Architecture- The model was constructed using four convolutional layers, each followed by a max-pooling layer to reduce the dimensions of the feature maps. This allows the model to focus on important features while reducing computational load. I started with 32 filters in the first convolutional layer and doubled the filters with each successive layer to progressively capture more complex features from the images.
    A fully connected dense layer with 512 units was added after flattening the convolutional layers, followed by a softmax activation function for multi-class classification.

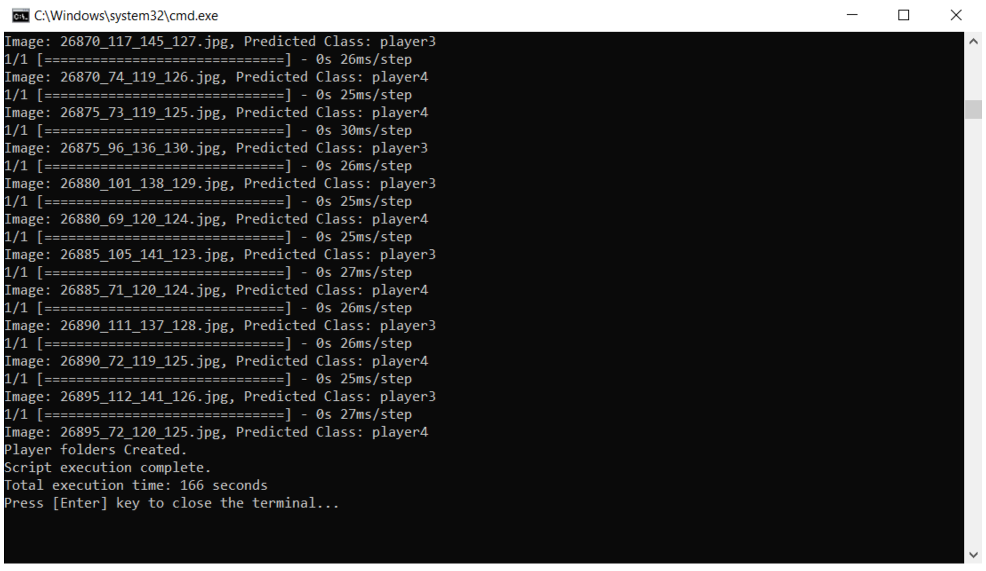
    Since this is a multi-class classification problem, I chose categorical cross-entropy as the loss function and Adam optimizer, which is computationally efficient and popularly used for deep learning tasks.

    d. Training parameters- I trained the model for 50 epochs, monitoring the training accuracy using a callback (ModelCheckpoint) to save the best model. I also used a batch size of 8 due to the limited data (train_generator). The ModelCheckpoint in question helps us in saving the best model weights (as 'best_model.keras', present in the directory ‘Implementation_1’ directory) by monitoring the epoch where the training accuracy was maximum. This is the model that we are actually using to predict the classes for the rest of the images.

e. Prediction and Image organization- After training, I used the saved model to predict the class of each image. For this, I implemented functions to create directories, load each image, predict its class (from class_labels), and move it into the corresponding directory. The 'predict_and_move_images' function is responsible for predicting the class and then moving the image to the correct folder.

f. Output- Finally, the images were moved to their respective player folders (player1, player2, player3, player4) based on the predictions.

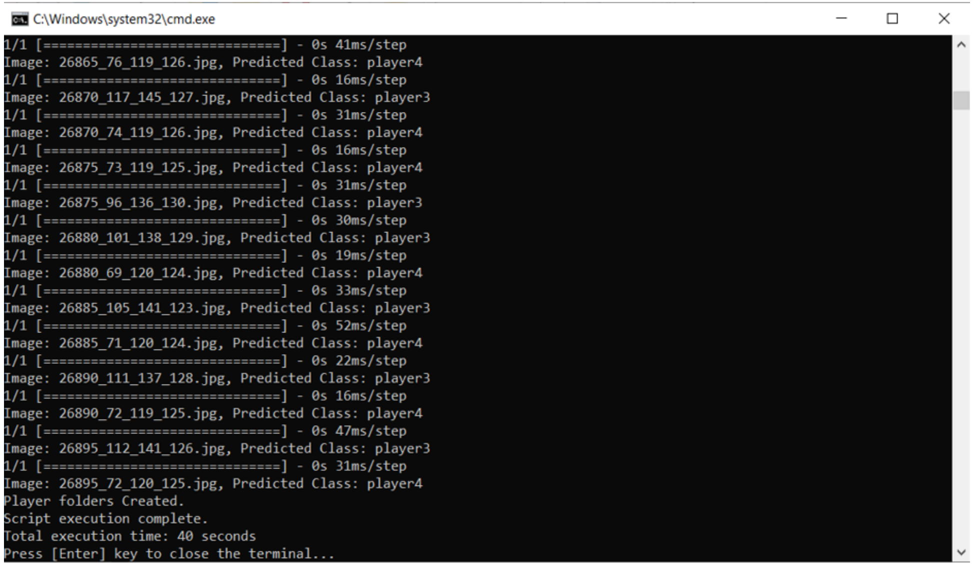
Note: There are two execute.sh files for this implementation, namely- ***'execute.sh'*** and ***'execute_.sh'.*** The former runs the main ***'player_segregation.py'*** script in the ***'Implementation_1'*** directory that defines and also trains our model on the ***'train'*** files present in the same and then runs the prediction function for the 360 images ('two_players_top' and 'two_players'bot' directories). All of this took 166 seconds to execute on my local system but is significantly faster on the google colab environment. (I could've reduced the number of epochs to bring this value to <120 seconds for my local system, but the training accuracy was at an impressive ***98.5%*** which I didn't want to compromise on, since this is my best implementation. Though I believe we can reduce the number of epochs to 40 without any loss in accuracy by going through the training logs. But haven't tried doing this).



```
Image: 26870_117_145_127.jpg, Predicted Class: player3
1/1 [==============================] - 0s 26ms/step
Image: 26870_74_119_126.jpg, Predicted Class: player4
1/1 [==============================] - 0s 25ms/step
Image: 26875_73_119_125.jpg, Predicted Class: player4
1/1 [==============================] - 0s 30ms/step
Image: 26875_96_136_130.jpg, Predicted Class: player3
1/1 [==============================] - 0s 26ms/step
Image: 26880_101_138_129.jpg, Predicted Class: player3
1/1 [==============================] - 0s 25ms/step
Image: 26880_69_120_124.jpg, Predicted Class: player4
1/1 [==============================] - 0s 25ms/step
Image: 26885_105_141_123.jpg, Predicted Class: player3
1/1 [==============================] - 0s 27ms/step
Image: 26885_71_120_124.jpg, Predicted Class: player4
1/1 [==============================] - 0s 26ms/step
Image: 26890_111_137_128.jpg, Predicted Class: player3
1/1 [==============================] - 0s 26ms/step
Image: 26890_72_119_125.jpg, Predicted Class: player4
1/1 [==============================] - 0s 25ms/step
Image: 26895_112_141_126.jpg, Predicted Class: player3
1/1 [==============================] - 0s 27ms/step
Image: 26895_72_120_125.jpg, Predicted Class: player4
Player folders Created.
Script execution complete.
Total execution time: 166 seconds
Press [Enter] key to close the terminal...
```

While the ***'execute_.sh'*** file runs the ***'player_segregation_final.py'*** script in the ***'Implementation_1'*** that only contains the prediction function by directly loading the

‘best_model.keras’ model. This took 40 seconds to run on my local system but is again significantly faster on the google colab environment.



C:\Windows\system32\cmd.exe

```
1/1 [==============================] - 0s 41ms/step
Image: 26865_76_119_126.jpg, Predicted Class: player4
1/1 [==============================] - 0s 16ms/step
Image: 26870_117_145_127.jpg, Predicted Class: player3
1/1 [==============================] - 0s 31ms/step
Image: 26870_74_119_126.jpg, Predicted Class: player4
1/1 [==============================] - 0s 16ms/step
Image: 26875_73_119_125.jpg, Predicted Class: player4
1/1 [==============================] - 0s 31ms/step
Image: 26875_96_136_130.jpg, Predicted Class: player3
1/1 [==============================] - 0s 30ms/step
Image: 26880_101_138_129.jpg, Predicted Class: player3
1/1 [==============================] - 0s 19ms/step
Image: 26880_69_120_124.jpg, Predicted Class: player4
1/1 [==============================] - 0s 33ms/step
Image: 26885_105_141_123.jpg, Predicted Class: player3
1/1 [==============================] - 0s 52ms/step
Image: 26885_71_120_124.jpg, Predicted Class: player4
1/1 [==============================] - 0s 22ms/step
Image: 26890_111_137_128.jpg, Predicted Class: player3
1/1 [==============================] - 0s 16ms/step
Image: 26890_72_119_125.jpg, Predicted Class: player4
1/1 [==============================] - 0s 47ms/step
Image: 26895_112_141_126.jpg, Predicted Class: player3
1/1 [==============================] - 0s 31ms/step
Image: 26895_72_120_125.jpg, Predicted Class: player4
Player folders Created.
Script execution complete.
Total execution time: 40 seconds
Press [Enter] key to close the terminal...
```

2. Implementation 2- Using OpenCV techniques:

a. Data Preprocessing- I directly read the images from the two directories, ‘two_players_top’ and ‘two_players_bot’, which contain the top and bottom player images respectively.

b. Color Segmentation- I used the ‘hsv_range.py’ script present in the ‘Implementations_2’ directory to get the HSV value of any pixel I click on, which in our case happens to be our badminton court (as in getting the pixel value in GR first and then converting it to HSV using ‘cv2.COLOR_BGR2HSV’) and then for getting a HSV range I’m just subtracting and adding (10,40,40) to the pixel value.
So, I used the HSV color space to filter out the background (court) by specifying the lower and upper bound for the court’s color. Once the court was removed, I performed morphological operations (closing) to clean up the mask and improve segmentation accuracy.

c. Feature Extraction- Now Instead of using deep learning or predefined model features, I focused on Histogram of Oriented Gradients (HOG) features to extract essential details about the players. HOG is efficient in capturing texture and shape features, which are crucial for differentiating the players.

d. Clustering and Image organization- After extracting HOG features, I used KMeans clustering to group the images into four clusters, each corresponding to one player. KMeans works by minimizing the distance between images in the same cluster, making it suitable for tasks where players can have distinct appearances.

e. Output- Based on the clustering results, I saved the images into four folders (player1, player2, player3, player4), ensuring the correct players were assigned to their respective output directories.

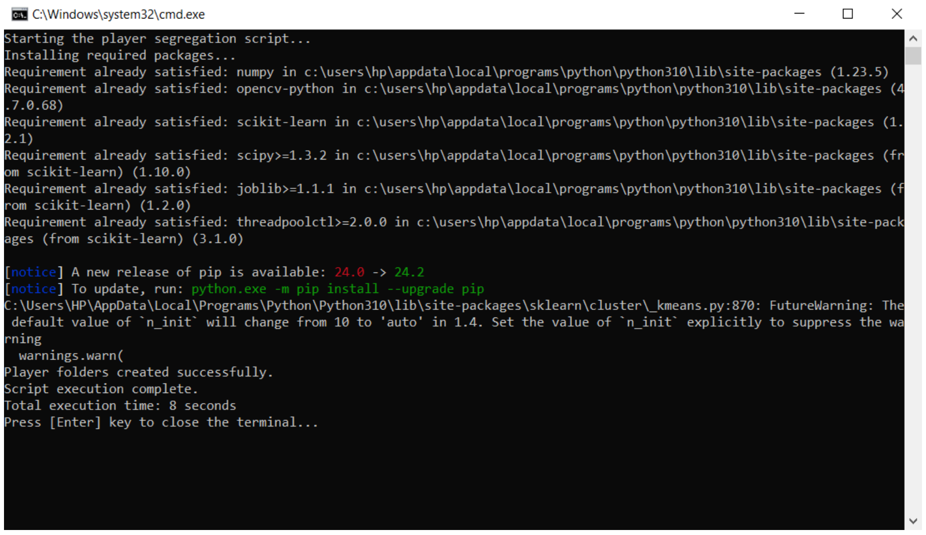
Challenges and a Probable Solution:

a. I hypothesized that color based clustering would perform better than HOG based clustering but this was not the case. ***The color of the players' clothes are similar and might be a reason***, but more underlying reasons might be present. Hence, I experimented with HOG features as they capture shape and texture well, but even this didn't perform very well. ***This might be due to the poor quality and low res images in consideration***.

   Color information is not included in HOG (Histogram of Oriented Gradients) features. HOG features focus on the shape and texture of an image by analyzing the distribution of gradient directions (edges and contours) in localized regions of the image. HOG operates on grayscale images, as it computes gradients in both horizontal and vertical directions.

b. A possible solution to this might be using a (HOG+C) descriptor. I couldn't find a read-made cv2 hog+c descriptor. But we can maybe combine HOG with color histogram and then extract and cluster the combined features.

Note: ***'execute2.sh'*** is the shell file for this implementation i.e ***Implementation_2'***, all the corresponding files present in the same.
The execution time for this implementation on my local system was 8 seconds.



```
C:\Windows\system32\cmd.exe
Starting the player segregation script...
Installing required packages...
Requirement already satisfied: numpy in c:\users\hp\appdata\local\programs\python\python310\lib\site-packages (1.23.5)
Requirement already satisfied: opencv-python in c:\users\hp\appdata\local\programs\python\python310\lib\site-packages (4
.7.0.68)
Requirement already satisfied: scikit-learn in c:\users\hp\appdata\local\programs\python\python310\lib\site-packages (1.
2.1)
Requirement already satisfied: scipy>=1.3.2 in c:\users\hp\appdata\local\programs\python\python310\lib\site-packages (fr
om scikit-learn) (1.10.0)
Requirement already satisfied: joblib>=1.1.1 in c:\users\hp\appdata\local\programs\python\python310\lib\site-packages (f
rom scikit-learn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\hp\appdata\local\programs\python\python310\lib\site-pack
ages (from scikit-learn) (3.1.0)

[notice] A new release of pip is available: 24.0 -> 24.2
[notice] To update, run: python.exe -m pip install --upgrade pip
C:\Users\HP\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The
 default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the wa
rning
  warnings.warn(
Player folders created successfully.
Script execution complete.
Total execution time: 8 seconds
Press [Enter] key to close the terminal...
```

3. Implementation 3- YOLO based approach:

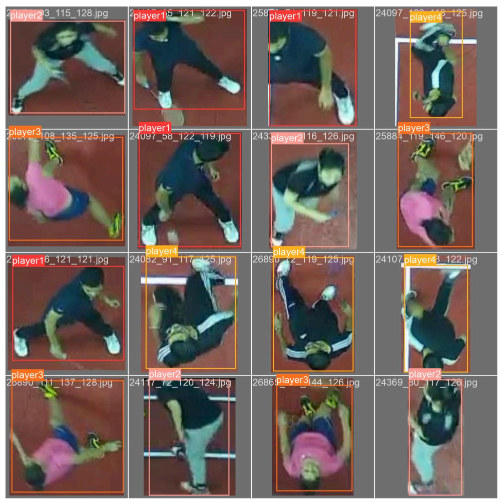
a. Dataset Creation- I created a custom dataset comprising images of four players. The images were manually annotated using ***LabelImg***, an open-source graphical image annotation tool. Each player was assigned one of four classes (player1, player2, player3, player4), and the bounding boxes were drawn around the players in the images.
   Once annotated, the labels were saved in the YOLO format, which contains the class ID and bounding box coordinates relative to the image size. These are present in the 'Implementation_3' directory.
   I. Training Set: 64 images (16 images per class)
   II. Validation Set: 16 images (4 images per class)
   (These are the same 80 images I used to train our DL model in Implementation 1).

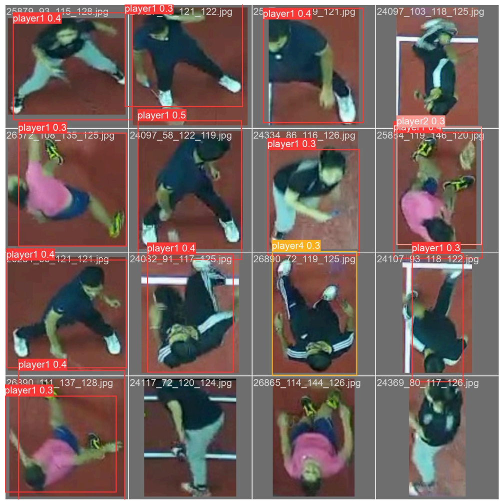
b. Cloning YOLOv5 Repository- I cloned the official YOLOv5 repository, which provides a reliable and customizable framework for training and using YOLO models.
c. Creating the dataset YAML config file- For YOLOv5 training, I created a dataset.yaml file, which specifies the paths to the training and validation images and the number of classes.
d. Label Mapping for our task- To ensure that only the player classes (player1, player2, player3, player4) were detected, I modified the label files. The dataset had additional labels (e.g., dog, car, tv), which I mapped to the player classes by adjusting the label numbers. The labels 15, 16, 17, 18 were remapped to 0, 1, 2, 3, respectively.
e. Training YOLOv5- I used a pre-trained YOLOv5 model (yolov5s.pt) and fine-tuned it on my dataset, which contained the four player classes. The training was done for 30 epochs with a batch size of 16.
f. Running the Inference job (Predicting)- After training, I used the trained model to detect players in two sets of images (two_players_top and two_players_bot). The detection results were saved as text files, each containing the class ID and bounding box coordinates of the detected players. These are saved into 2 directories named, player_segregation_top and player_segregation_bot
g. Output- To organize the detection results, I created a script that reads the class IDs from the detection files and moves the corresponding images into player-specific folders (player1, player2, player3, player4).

Challenges and Issues:

a. Predictions have low confidence: In the val_batch0_pred.jpg image, many of the predicted bounding boxes have relatively low confidence scores (e.g., 0.3 or 0.4). This indicates that the model is unsure about its predictions, which might suggest insufficient learning during training.

b. Incorrect player assignments: Some player predictions seem incorrect, with players being labeled wrongly (e.g., most of the players are misclassified as player1). This suggests that the model is struggling to differentiate between the four classes. Also a single image (with a single player) is being labeled twice as different classes.


val_batch0_labels


val_batch0_pred

c. The most probable reason for such poor results is our reliance on such a small training set. We are using 16 images per class for training and 4 images per class for validation. But the recommended number of images per class is >=1500 as per Glenn Jocher (Founder of Ultralytics) https://github.com/ultralytics/yolov5/discussions/9335
Thus yolo might not be the most practical solution for such a task (where the total number of images is 360 and of low quality).

Note: I have not created a execute.sh file for this implementation, due to such poor performance in my base run. The code has been implemented and can be seen in the ***'yolo_train.ipynb'*** file present in the ***'Implementation_3'*** directory.