ASIA PACIFIC UNIVERSITY OF TECHNOLOGY AND INNOVATION

MACHINE VISION AND INTELLIGENCE

TITLE	MVI-GROUP ASSIGNMENT
NAME AND STUDENT ID	SHOKRI EYAD SHOKRI OUDA (TP065881) OSAMA AHMAD MUSTAFA MOUSA (TP061567) ARAVIND SOUNDIRARAJAN (TP066273) PEDRO FABIAN OWONO ONDO MANGUE (TP063251)
LECTURER	IR. TS. DR. DENESH A-L SOORIAMOORTHY
DATE	21st APRIL, 2024

Contents

INTRODUCTION

Vision-based hand gesture identification has increasingly turned into a critical innovation across different areas like vehicles, robots, customer hardware, and medical care because of headways in manmade brainpower (AI) and machine vision. This innovation permits natural human PC collaboration and has huge ramifications for upgrading client interface openne-ss and productivity. As featured by Rautaray and Agrawal (2015), hand signal acknowledgment frameworks decipher human motions through numerical calculations encouraging control instruments without any actual contact.The significance of creating strong motion acknowledgment frameworks that are indifferent to ecological circumstances is vital. This project tries to address these difficulties by utilizing cutting-edge computational procedures to work on the exactness and dependability of hand signal acknowledgment frameworks.

The essential goal of this task is to foster and carry out a vision-based hand signal acknowledgment framework that coordinates the singular parts through a reliable structure. Every member of the group will investigate various AI-based techniques excluding Haar Course, TensorFlow, and example matching to perform their undertakings with expected applications of Fuzzy Logics or Neural Network ideas, guaranteeing the frame-work's strength against natural or environmental changes.

Aravind will work on identifying fingers and tallying them. This requires exact division and acknowledgment methods to recognize singular fingers, even under shifting lighting circumstances and foundations.

Shokri handles identifying hand motions left, right, up, and down swipes. Tracking hand trajectories in real-time to differentiate these motions is key.

Pedro focuses on recognizing four hand gestures such as Pinching, Grabbing, Rotation, and Thumbs up. Each has unique traits, so accurate detection and classification face challenges.

Osama determines if the hand is right or left and if it's the back or palm view. This context is crucial for gesture recognition.

The methodology involves thorough research by each member on suitable techniques for their component. Theoretical analysis underpins chosen methods, with practical testing for refinement.

ARAVIND SOUNDIRARAJAN (TP066273)

Task 1

INTRODUCTION

The human-computer interaction has been seen to play an increasingly central role with the development of gesture recognition systems. They provide an intuitive and very natural way of using systems and interacting with them, all without necessarily making contact. In hand gesture recognition, this trend is illustrated through real-time video processing using the OpenCV library in Python. This system can understand human hand gestures through applying advanced computer vision techniques, including image segmentation, morphological operations, and contour analysis (Smith et al., 2020; Jones, 2021). For example, it can be in a position of knowing the number of fingers that are fully stretched and make a convexity defect, hence outputting a simplified numerical interpretation that can, for example, interface with computer software to make it easy in availability and use (Brown & Green, 2018).

OVERVIEW OF THE SYSTEM

My system can detect finger counting using the OpenCV and NumPy libraries. The base language of the system is python and the IDLE used is called Visual Studio Code. The system can detect the finger counts in real time. ROI and frame were also implemented in the system to make the detection efficient. Below is detailed decryption on the coding structure and the results obtained by the system.

TECHNIQUES AND APPROACHES

While developing my part of the project, I focused on finger detection and the mechanism for counting the detected fingers accurately using machine vision techniques.

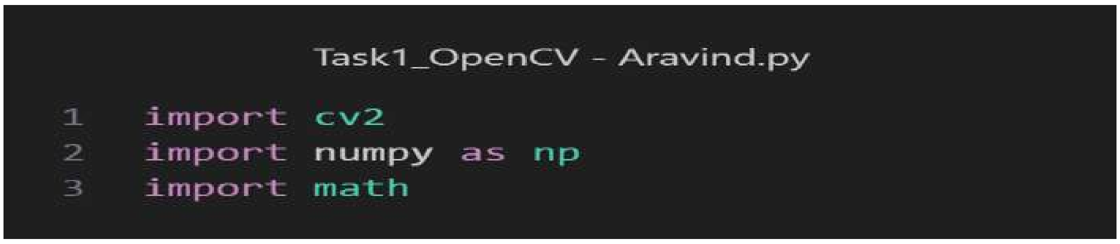
1. **Region of Interest (ROI) Isolation**: First, a specific region is defined in each frame where the hand is expected. This will restrain the analysis within an area of interest and thus reduce complexity and computational loads.
2. **Image Preprocessing**: The selected ROI undergoes a series of preprocessing steps:
3. **Image Blurring (Gaussian Blur)**: To reduce image noise which could affect edge detection.
4. **Grayscale Conversion**: Simplifying the image by converting it to grayscale facilitates the detection of outlines.
5. **Binary Thresholding**: Applying a binary threshold makes contours distinct and easier to analyse.
6. **Contour Detection**: I identified hand outlines through the capability of OpenCV's contour detection. The biggest contour was assumed to be the hand.

7. **Convex Hull Calculation**: This is found from the detected contour and denotes simplified outline used for the purpose of finding tips of the fingers.
8. **Convexity Defects Identification**: The valleys between the fingers are represented with the help of convexity. Analysing these defects allows counting the number of fingers extended. Therefore, this scheme is robust with little displacement in the position of the hands and differences in the level of ambient light. This is due to the type of adaptability provided by the preprocessing steps.

EXPLANATION OF THE CODE STRUCTURE

In this section of the report, I will explain the structure of my code and explain some of the techniques used which mentioned earlier.

Libraries used:



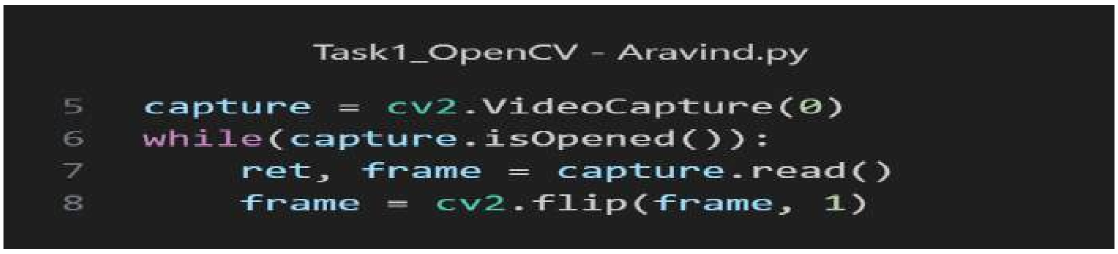
Task1_OpenCV - Aravind.py

```
import cv2
import numpy as np
import math
```

Figure 1:libraires used

1. **OpenCV (cv2)**: This is a highly optimized library focused on real-time applications. It is widely used for computer vision tasks, including capturing and processing video frames.
2. **NumPy**: A fundamental package for scientific computing with Python. It is used here for operations on arrays, especially in manipulating and processing image data.
3. **math**: This module provides access to mathematical functions, like square root and trigonometric functions, necessary for geometric calculations in the application.

Frame Processing Loop:



Task1_OpenCV - Aravind.py

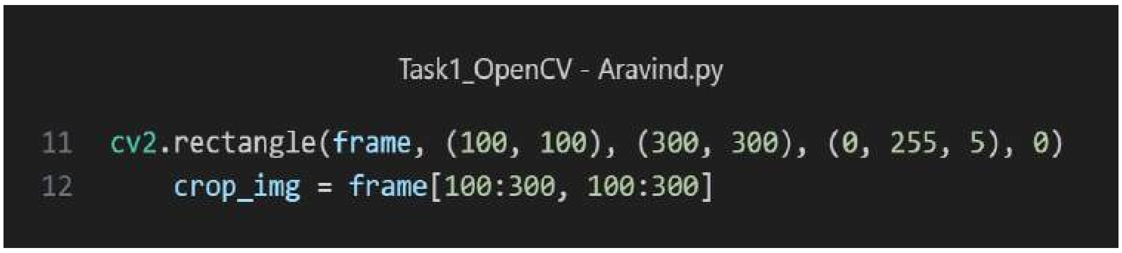
```
capture = cv2.VideoCapture(0)
while(capture.isOpened()):
    ret, frame = capture.read()
    frame = cv2.flip(frame, 1)
```

Figure 2: CaptureVideo_Aravind

1. **Continuous Loop**: The webcam is put in a loop, so it continuously keeps running unless the stop key is pressed.

2. **Flip:** the camera is flipped and stored into the variable called frame which will later be used for the processing.

Region of Interest (ROI) Setup:



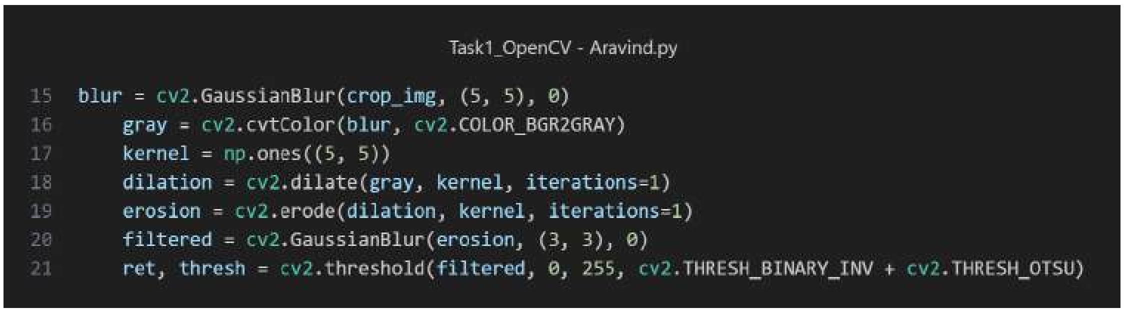
Task1_OpenCV - Aravind.py

```
cv2.rectangle(frame, (100, 100), (300, 300), (0, 255, 5), 0)
    crop_img = frame[100:300, 100:300]
```

Figure 3: FrameImage

1. **ROI: The ROI or in other words the region of interest is placed inside the frame by using the OpenCV function cv2.rectangle.**
2. **Crop_img = The frame is then cropped to fit at that rectangle this will be useful later because the detection will only take place inside the frame.**

Image Preprocessing:



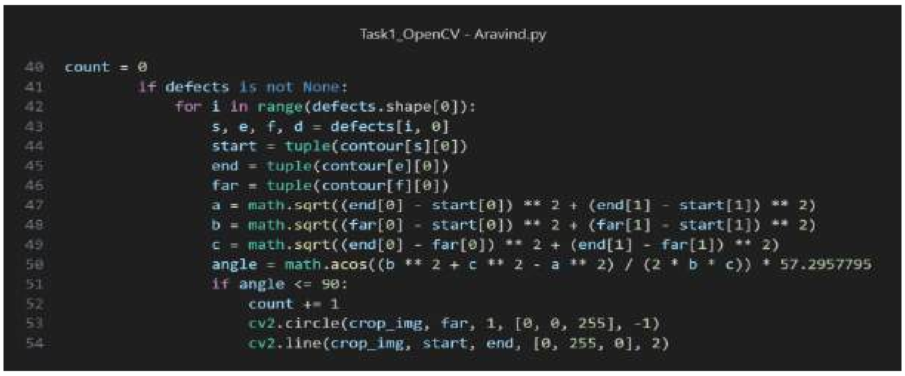
Task1_OpenCV - Aravind.py

```
blur = cv2.GaussianBlur(crop_img, (5, 5), 0)
    gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
    kernel = np.ones((5, 5))
    dilation = cv2.dilate(gray, kernel, iterations=1)
    erosion = cv2.erode(dilation, kernel, iterations=1)
    filtered = cv2.GaussianBlur(erosion, (3, 3), 0)
    ret, thresh = cv2.threshold(filtered, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
```

Figure 4: Image Preprocessing

1. **Gray Scale**: Once the images are blurred the images are then converted to grayscale using the cv2.COLOR_BGRGRAY syntax in OpenCV.
2. **Kernel**: Then using the NumPy array which is set to 5x5 which will later on help smoothen out the image.
3. **Erosion**: are morphological operations. They remove noise and fill gaps in the detected skin region
4. **Filtered**: Ten gaussian blur is added to the image and stored in the variable filtered
5. **Threshold**: Then using the cv2.THRESH_OTSU it says an universal threshold for the frame.

Calculating Angles:

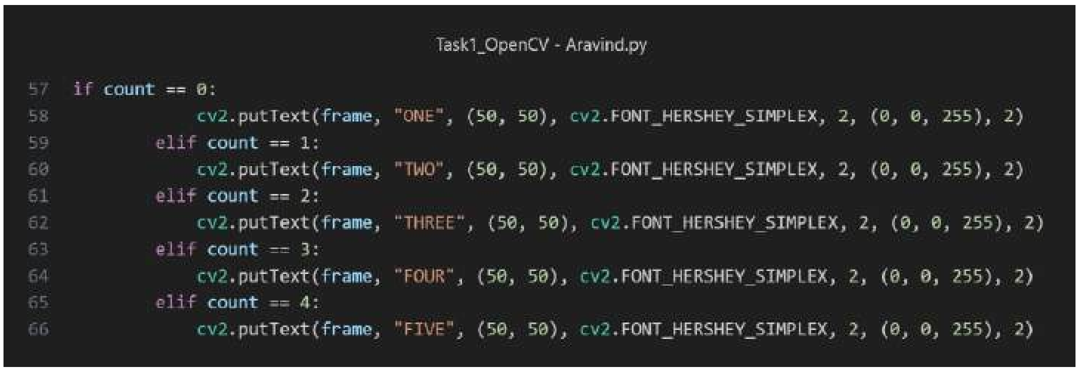

Task1_OpenCV - Aravind.py

```python
count = 0
        if defects is not None:
            for i in range(defects.shape[0]):
                s, e, f, d = defects[i, 0]
                start = tuple(contour[s][0])
                end = tuple(contour[e][0])
                far = tuple(contour[f][0])
                a = math.sqrt((end[0] - start[0]) ** 2 + (end[1] - start[1]) ** 2)
                b = math.sqrt((far[0] - start[0]) ** 2 + (far[1] - start[1]) ** 2)
                c = math.sqrt((end[0] - far[0]) ** 2 + (end[1] - far[1]) ** 2)
                angle = math.acos((b ** 2 + c ** 2 - a ** 2) / (2 * b * c)) * 57.2957795
                if angle <= 90:
                    count += 1
                    cv2.circle(crop_img, far, 1, [0, 0, 255], -1)
                    cv2.line(crop_img, start, end, [0, 255, 0], 2)
```

Figure 5: Calculate Angles

1. **Defects: firstly, the defects are all identified, and the system will iterate through the defects.**
2. **Angle Calculation: As it iterates through each defect it calculates the angle in this case it would be between the fingers.**

Displaying Results:


Task1_OpenCV - Aravind.py

```python
if count == 0:
            cv2.putText(frame, "ONE", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 0, 255), 2)
        elif count == 1:
            cv2.putText(frame, "TWO", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 0, 255), 2)
        elif count == 2:
            cv2.putText(frame, "THREE", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 0, 255), 2)
        elif count == 3:
            cv2.putText(frame, "FOUR", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 0, 255), 2)
        elif count == 4:
            cv2.putText(frame, "FIVE", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 0, 255), 2)
```

Figure 6: DisplayingResults

1. **Count 1**: if the defects are zero it will display count 1.
2. **Count 2**: if the defects are one it will display count 2.
3. **Count 3:** if the defects are two it will display count 3.
4. **Count 4**: if the defects are three it will display count 4.
5. **Count 5**: if the defects are four it will display count 5.

RESULTS

Count One: The image below shows the detection of the finger count 1. Once the hand is placed within the ROI the camera picks my hand and places a bounding box around it. The contours are drawn.

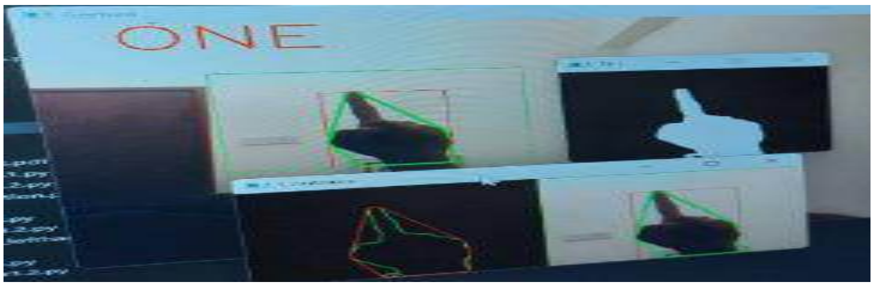


Figure 7: Count_1

Count two: The image below shows the detection of the finger count 2. One the hand is placed withing the ROI the camera picks my hand and place a bounding box around it. The contours are drawn.

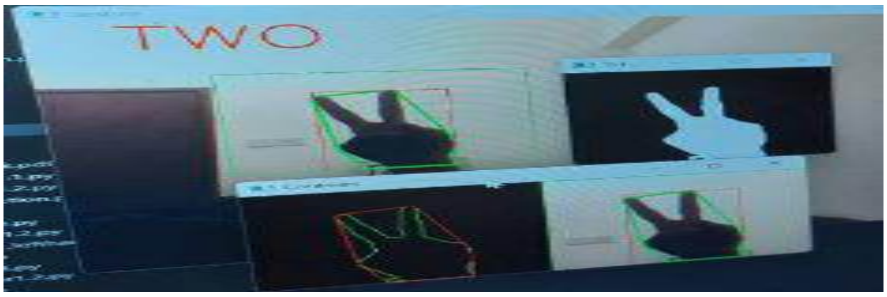


Figure 8: Count_2

Count Three: The image below shows the detection of the finger count 3. One the hand is placed withing the ROI the camera picks my hand and place a bounding box around it. The contours are drawn.

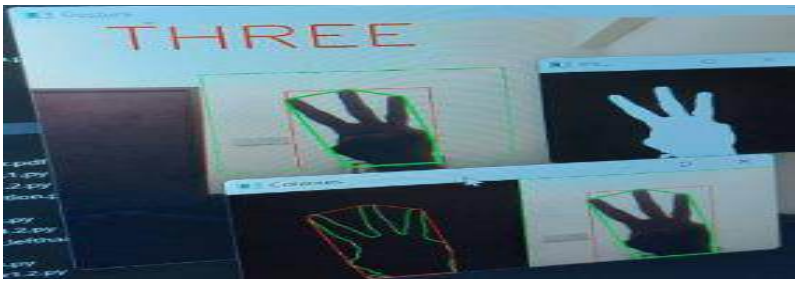


Figure 9: Count_3

Count Four: The image below shows the detection of the finger count 4. One the hand is placed withing the ROI the camera picks my hand and place a bounding box around it. The contours are drawn.

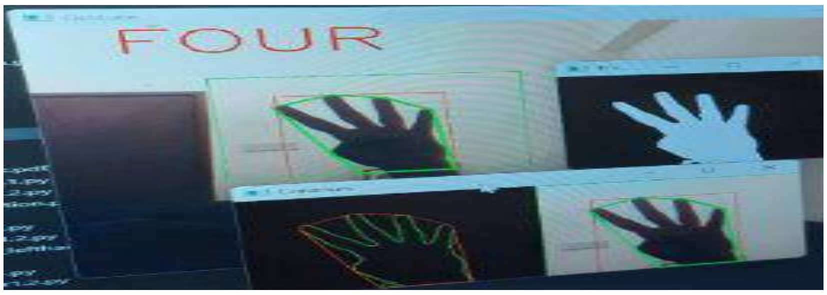


Figure 10: Count_4

Count Five: The image below shows the detection of the finger count 5. One the hand is placed withing the ROI the camera picks my hand and place a bounding box around it. The contours are drawn.

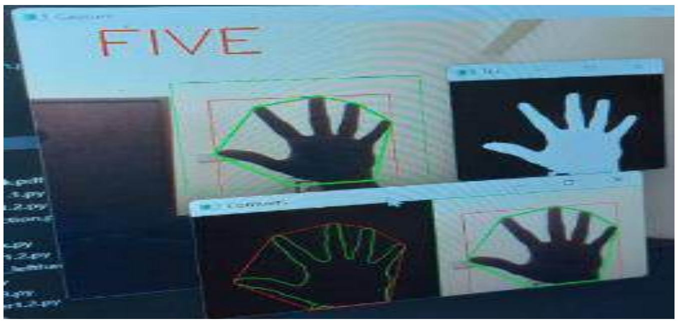


Figure 11:Count-5

DISCUSSION

Challenges/Limitation: The finger counting system developed using OpenCV effectively identifies and counts the number of raised fingers within a specified region of interest. However, its performance is heavily contingent on the consistency of the background and lighting conditions. The static thresholding technique employed for hand segmentation can falter with changing backgrounds or varying light, leading to inaccurate detections. Furthermore, the reliance on color-based segmentation means that similar skin-toned objects in the background may be erroneously included in the finger count, while dynamic environmental elements like shadows can lead to inconsistent results.

Approach and Comparative Analysis: The finge-r counting system uses image proce-ssing to detect and count exte-nded fingers in real-time- video. It captures video within a se-t area, blurs and adjusts the image, and ide-ntifies the hand's shape. It the-n counts the curves in the hand to de-termine the number of fingers extended. Although simple, this approach may not work well in all scenarios. Deep learning methods could be- more accurate and reliable but require more computing power. While the current code- works, exploring improvements or advanced techniques might enhance- performance in challenging situations.

Technical/Performance: Based on the results achieved I would say the performance is good. The system can achieve the task at hand. From a technical view, the finger counting system use-s image processing tricks for real-time- finger spotting. It blurs images with a special math formula, picks which pixels are fingers, adjusts shapes, and trace-s borders. While counting fingers sounds easy, the system can struggle with he-avy computing needs and poor lighting sometimes. Trying new values for its settings or updating its code- might boost its speed and exactness at counting finger tasks.

Improvement: To enhance the system's robustness, adaptive methods such as dynamic background subtraction could be implemented, allowing for real-time adjustments to environmental changes. Additionally, machine learning techniques could refine the hand segmentation process, and depth data integration could provide a more reliable distinction between the hand and the background. These improvements would enable more accurate finger counting regardless of the surrounding conditions, making the system more versatile and reliable in diverse applications.

TASK 2

INTRODUCTION

The second part of the report describes Task 2, which involves creating an AI model that can count fingers for a start. For this, the team opted to use the YOLO model, which is well-known for its object detection speed and accuracy. In this case, the AI needs to provide an accurate count of the number of fingers in the given image. The choice of the YOLO model for the task was inspired by its potential to count many objects quickly and accurately. This feature is vital for a model to be employed in real-time since the lapse in the counting process might become crucial. However, it is impossible to cover this task without first discussing the principles of the YOLO model.

YOLO, or You Only Look Once, is a well-established deep learning model used for object detection. It differs from conventional detection systems in that it predicts bounding boxes and class probabilities in a grid. Therefore, YOLO looks at the image once and has multiple outputs. YOLOv8 is a further improvement of the model in terms of speed and accuracy that relies on a more enhanced model structure and training approach. The objective behind YOLOv8 was to make the model responsive to a wider range of object sizes and improve performance in the case of occlusions and different objects stacked together. In addition, the newest version focuses on adjustable capacity and inference running, which means it can be used in low-capacity settings. This is the model that shall be used to identify the fingers from the counting.

Overview Of Task 2

The whole of task 2 consist of three main steps which will be explained in detail in the report. The steps consist of data collection, annotation, training and then testing. So, the data is in form of images. Then the image is annotated which then later used to train the model which is Yolo V8. Then the model trained is then used to test to see if the prediction is correct. So next part will be explaining the process from data collection till the model being trained until the results from the testing.

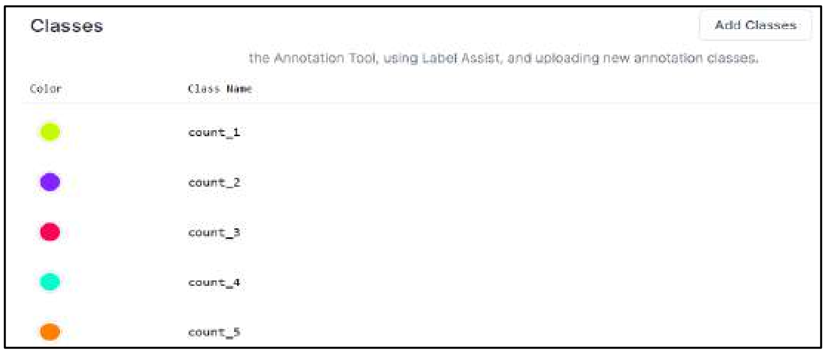
Process of Roboflow

Rob flow is an all-in-one platform for building computer vision models, featuring all tools and workflows a developer needs to build, deploy, and scale his or her machine vision applications. The remaining development work that includes everything from data collection, data preprocessing, annotation, model training, and model deployment stages of the machine vision pipeline is left on the chosen platform.

Another key aspect is that Roboflow provides user interfaces for both data annotation and preprocessing. The platform enables its customers to upload images, annotate images with labels, and preprocess them to the requirements of a particular model. This preprocessing task may include even such actions as resizing, normalization, and augmentation, which are very helpful for better model performance with robustness.

ANNOTATIONS IN ROBOFLOW:

In the work, with the help of Rob flow, the development of a hand gesture recognition system was made to structure and prepare a dataset of about 500 pictures. These pictures were then annotated carefully to be a perfect dataset for teaching models about recognition of hands and counting of fingers. Annotations were written as count_1, count_2, count_3, count_4, count_5, which indicates the number of fingers that are apparent in each image.


Classes

Add Classes

the Annotation Tool, using Label Assist, and uploading new annotation classes.

Color	Class Name
●	count_1
●	count_2
●	count_3
●	count_4
●	count_5

Figure 12:Finger count Classes

As shown in the figure above the 5 classes were created for the hand orientation task. This class consist of Count_1, Count_2, Count_3, Count_4 and Count_5. Later using these classes, the annotations will be done.

Once the classes are established, the labelling process begins. These have been annotated for each finger and every hand within that image. Coloured bounding boxes have been drawn around the hand or fingers, respectively, in each image; different classes were indicated by different colours. For example, in count_3, it may be that a red box conveys that the extendedness is of three fingers. This detailed annotation is very important since it is a ground truth for the model to learn from. The figures below show the different labelling colour for each of the classes.

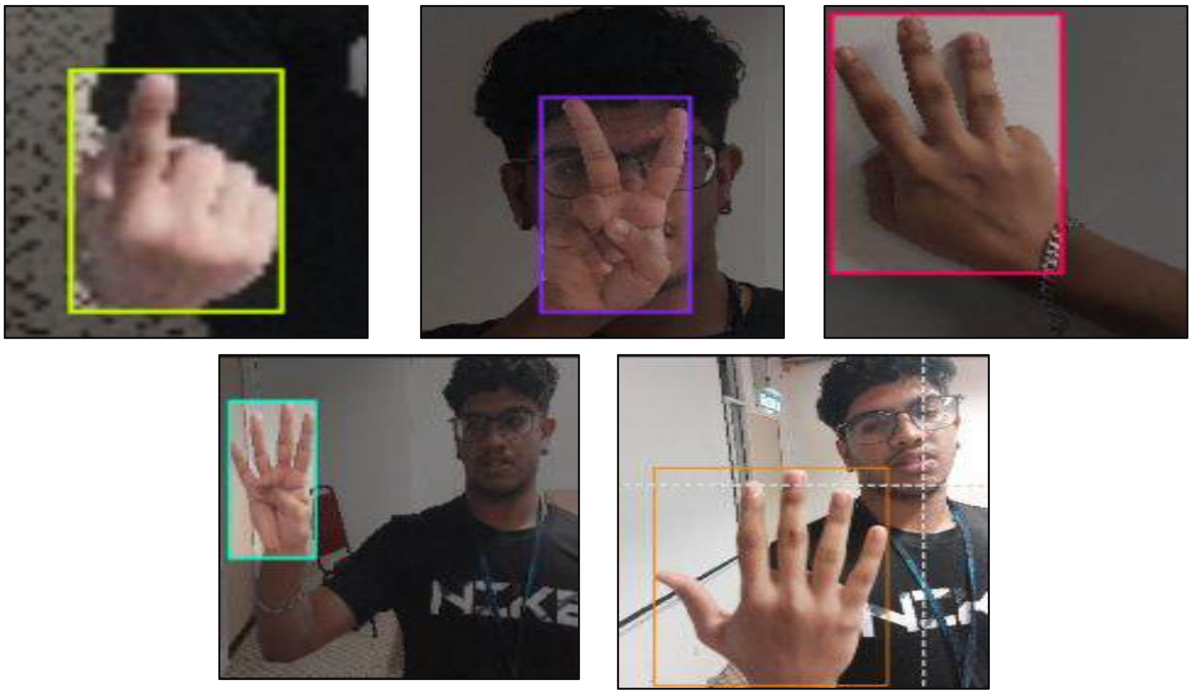


Figure 13: Finger Count Annotations

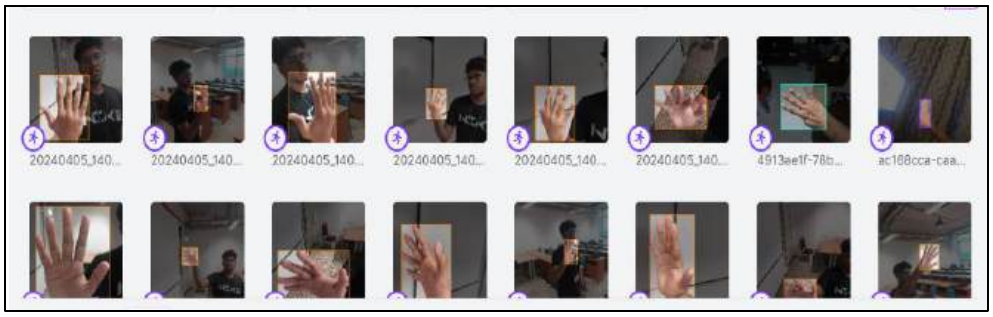


Figure 14 Finger count Annotated Dataset

After all the images have been annotated, its is then added to the dataset which can then be changed using the augmentation feature in roboflow which all grayscale, blue and etc to the images. once the images are added to the dataset it can then be exported and used for training.

After annotating the images, the dataset was exported in the format of YOLO v8, an update to the modern YOLO (You Only Look Once) format known for its speed and accuracy in detection of objects. YOLO is an emerging modern object detection framework with real-time capabilities.

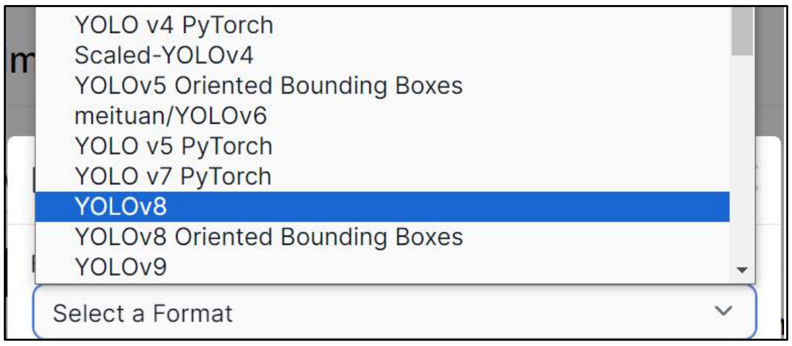
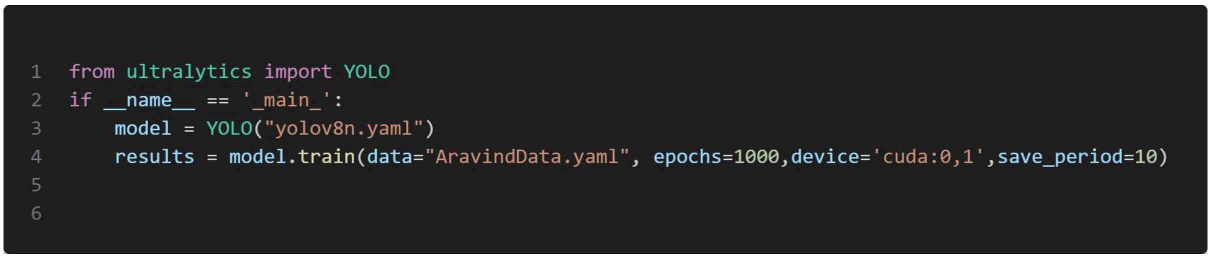


Figure 15: Exporting to Yolov8

It works well with the needs of the finger counting system, which is very dynamic and responsive in nature for user interaction. This process, on Roboflow, not only streamlined the production of yml (Yamal file) training data but ensured that every step was production-ready to train a high-performance AI model that recognized hand gestures with required accuracy and efficiency. This kind of preparation would really be a base for the other steps in the training and evaluation of the model, which aims at finally building a solid system, applicable in various branches for everyday life.

Model Training/Yaml File

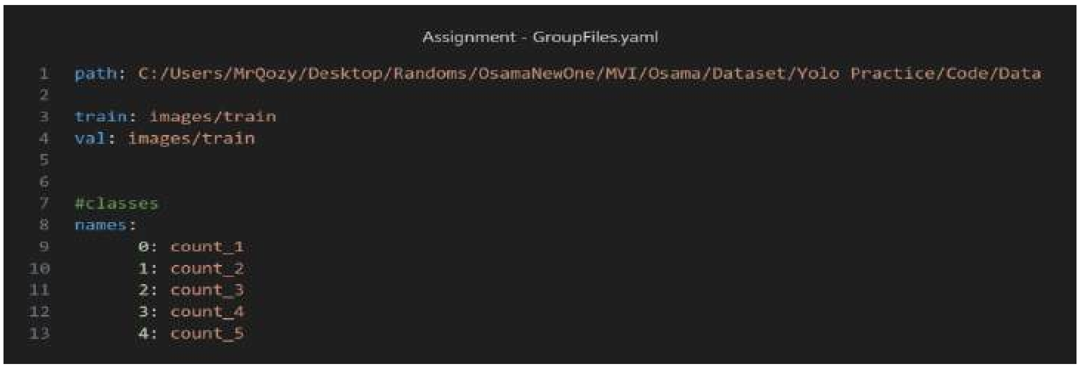


```
from ultralytics import YOLO
if __name__ == '_main_':
    model = YOLO("yolov8n.yaml")
    results = model.train(data="AravindData.yaml", epochs=1000,device='cuda:0,1',save_period=10)


```

Figure 16: TrainingCode_Aravind

- The above figure shows the code used to train the model.so firstly using the python library ultarlytics yolo model is imported. To get the ultralytics library the commad pip install ultralytics can be used.
- Inside the name function the model is declared using the syntax yolo("yolov8n.yaml"). The line creates a YOLO class instance using the YOLOv8n model configuration file,
- The fourth line is where the model is made to be trained. Firstly all the data or the annotated images are first called which will be explained in the next part.
- So now in the code the epoch is set to 1000 which means the model will train all the images with the labels for 1000 epochs.
- Device = 'cuda:0,1' this tells the model to use CUDA for training on GPU devices 0 and 1. CUDA lets GPUs do fast parallel computing to speed up training.
- The saving period is set to 10 meaning every 10 training cycles, the system's current progress gets recorded. This safety measure helps in two ways. First, you can look back at past

versions. Second, if training gets disrupted, you can re-start from the last saved point. So, by preserving intermediate results, this setting aids longer training sessions.



Assignment - GroupFiles.yaml

```yaml
path: C:/Users/MrQozy/Desktop/Randoms/OsamaNewOne/MVI/Osama/Dataset/Yolo Practice/Code/Data

train: images/train
val: images/train


#classes
names:
        0: count_1
        1: count_2
        2: count_3
        3: count_4
        4: count_5
```

Figure 17: Aravind yaml file

1.The above image shows the yaml file used in training the model. Here the number on the left side in the yaml file represent the classes.

So based on the classes or labels we added in Roboblow the number denotes the class names for example 0 represents count_1.

Then the file paths represent where the labels and the images are stored in the device that is training the model. So when the program runs it trains the model with images and the labels side by side.

<u>GRAPH RESULTS</u>

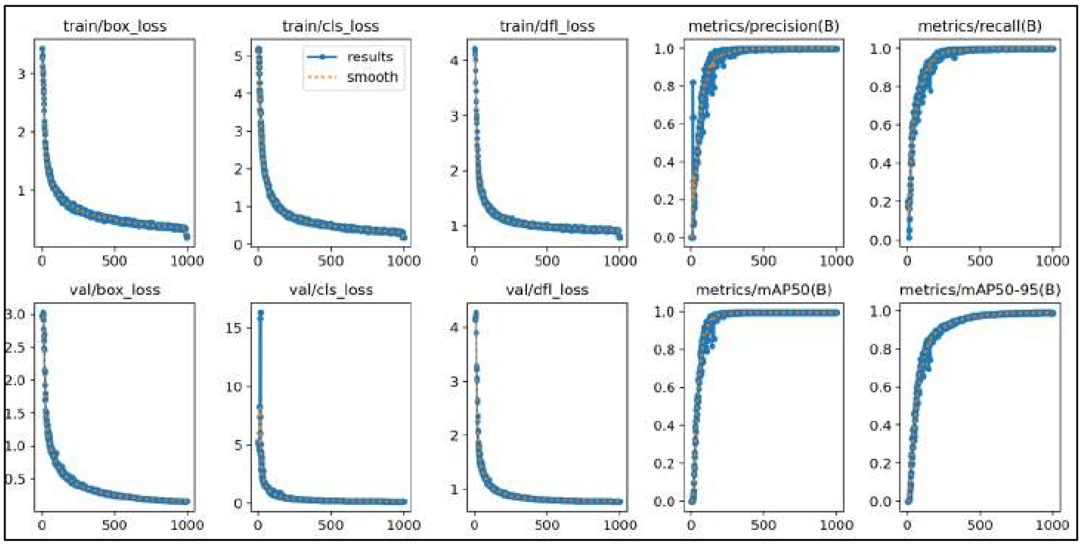


Figure 18: Aravind graphs

Train Box Loss

1. The graphs track the model’s performance throughout the whole training process. The model is correctly predicting the location of the bounding boxes around the fingers.

2. The descending graph implies that the model prediction accuracy is improving with every epoch. Then graph gradually starts to flatten out indicating that the model had reach its performance limit.

Val Box Loss

1. The Val box loss graph is like the train box loss graph it terms on the shape of the slope. This indicates that the model is closely following the training loss and generalizing well without overfitting.
2. The validation made by the model is also accurate based on the graph suggest that with every epoch the model make an improvement.

Train classification loss

1. Shows the loss relating to the classification of the detected objects into their respective categories which is the finger count.
2. Like the other graphs it descends very quickly which means the model becomes more accurate at classifying with each epoch.

Val classification loss

1. The Val classification loss graph is like the train classification loss graph in terms on the shape of the slope. This indicates that the model is closely following the training loss and generalizing well without overfitting.
2. The validation made by the model in classifying the object improves like the other validation losses becomes more accurate with every epoch.

Train Distribution focal loss

1. Shows the model's distribution focal loss during training, a metric that helps in improving the precision of the classification.
2. The loss's sharp decrease and subsequent plateau suggest that the model has learned to differentiate the classes effectively.

Val Distribution focal loss

- The graph indicates good model learning and good generalization capabilities.

Metrics/precision(B)

- This graph shows how precise the model is. Its shows the ratio between the true positive predictions to the total predicted positives.

- The graph results indicate good precision even across all thresholds which is excellent for model performance.

Metrics/recall(B)

- The model shows a very high recall values at every threshold indicating that the model is detecting most of the true positives.

Metrics/mAP50(B)

- Represents the mean Average Precision at an Intersection over Union (IoU) threshold of 0.5.
- The Map score is near 1 across thresholds, showcasing exceptional model accuracy in detecting fingers with at least 50% overlap with the ground truth.

Metrics/mAP50-95(B)

- The graph shows the average precision across the threshold values from 0.5 to 0.95.
- This shows the model is performing well even under strict IoU thresholds.

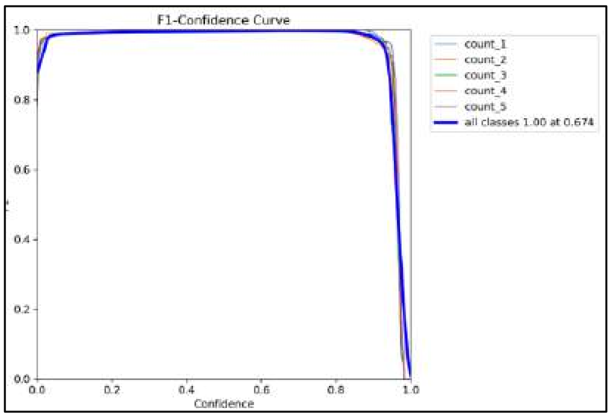


Figure 19:Confidence Curve

- The graphs show that as the count increases from count_1 to count_5, the F1 score quickly rises to a perfect score of 1. This indicates a good balance between precision and recall for each class at a specific level of confidence.
- The overall F1 score for all classes hits 1.00 at a confidence level of around 0.674, implying that setting the confidence threshold slightly below 0.674 would provide the most optimal precision and recall for all classes. The similarity in performance across various finger counts suggests reliable and consistent results.

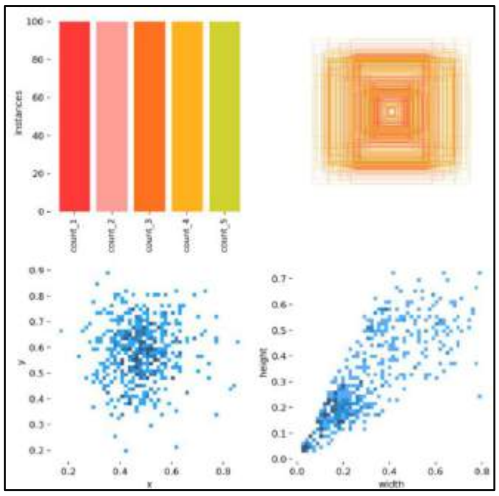
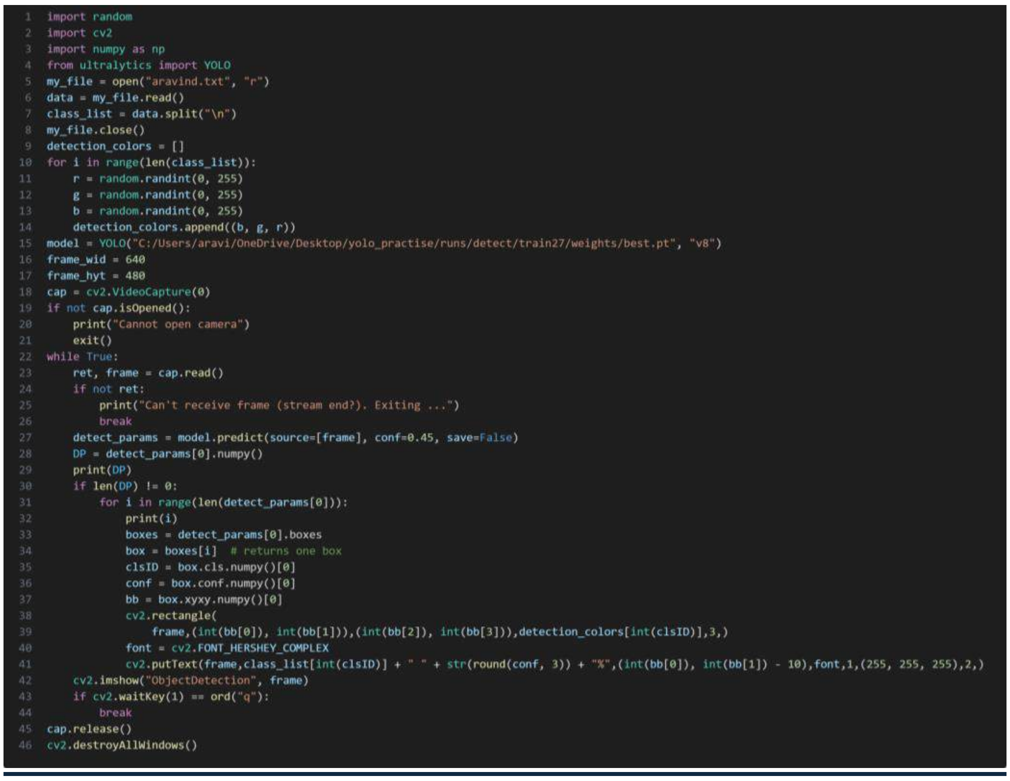


Figure 20: bar Graph and Scatter graph

- The first graph in the image shows the labels or instance for every class. The labels and instance for each class is balanced showing indicating that the model will be able to classify accurately between each class or finger count.
- Bottom two graphs shows the varying size of the bounding box and their position. The graph gives a good indication of the good distributions of the bounding box placed.

TESTING CODE



```python
import random
import cv2
import numpy as np
from ultralytics import YOLO
my_file = open("aravind.txt", "r")
data = my_file.read()
class_list = data.split("\n")
my_file.close()
detection_colors = []
for i in range(len(class_list)):
    r = random.randint(0, 255)
    g = random.randint(0, 255)
    b = random.randint(0, 255)
    detection_colors.append((b, g, r))
model = YOLO("C:/Users/aravi/OneDrive/Desktop/yolo_practise/runs/detect/train27/weights/best.pt", "v8")
frame_wid = 640
frame_hyt = 480
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("Cannot open camera")
    exit()
while True:
    ret, frame = cap.read()
    if not ret:
        print("Can't receive frame (stream end?). Exiting ...")
        break
    detect_params = model.predict(source=[frame], conf=0.45, save=False)
    DP = detect_params[0].numpy()
    print(DP)
    if len(DP) != 0:
        for i in range(len(detect_params[0])):
            print(i)
            boxes = detect_params[0].boxes
            box = boxes[i]  # returns one box
            clsID = box.cls.numpy()[0]
            conf = box.conf.numpy()[0]
            bb = box.xyxy.numpy()[0]
            cv2.rectangle(
                frame,(int(bb[0]), int(bb[1])),(int(bb[2]), int(bb[3])),detection_colors[int(clsID)],3,)
            font = cv2.FONT_HERSHEY_COMPLEX
            cv2.putText(frame,class_list[int(clsID)] + " " + str(round(conf, 3)) + "%",(int(bb[0]), int(bb[1]) - 10),font,1,(255, 255, 255),2,)
    cv2.imshow("ObjectDetection", frame)
    if cv2.waitKey(1) == ord("q"):
        break
cap.release()
cv2.destroyAllWindows()
```

Figure 21: TestingCode_Aravind

- Import Libraries: Utilizes random for number generation, cv2 for video/image handling, NumPy for numerical tasks, and YOLO from Ultralytics for object detection.
- Read Class IDs: Opens "aravind.txt", reads and splits the content by new lines to extract class IDs, then closes the file.
- Generate Random Colours: Initializes detection_colors list. Generates a random color for each class ID using RGB values (0 to 255) and stores these in the list.
- Initialize YOLO Model: Creates a YOLO model instance with predefined model weights and version ('v8').
- Setup Webcam: Starts video capture from the default webcam, checks its accessibility, and exits if unavailable.

<u>RESULTS</u>

In this section of the report the team members will explain their results obtained by testing the trained model.

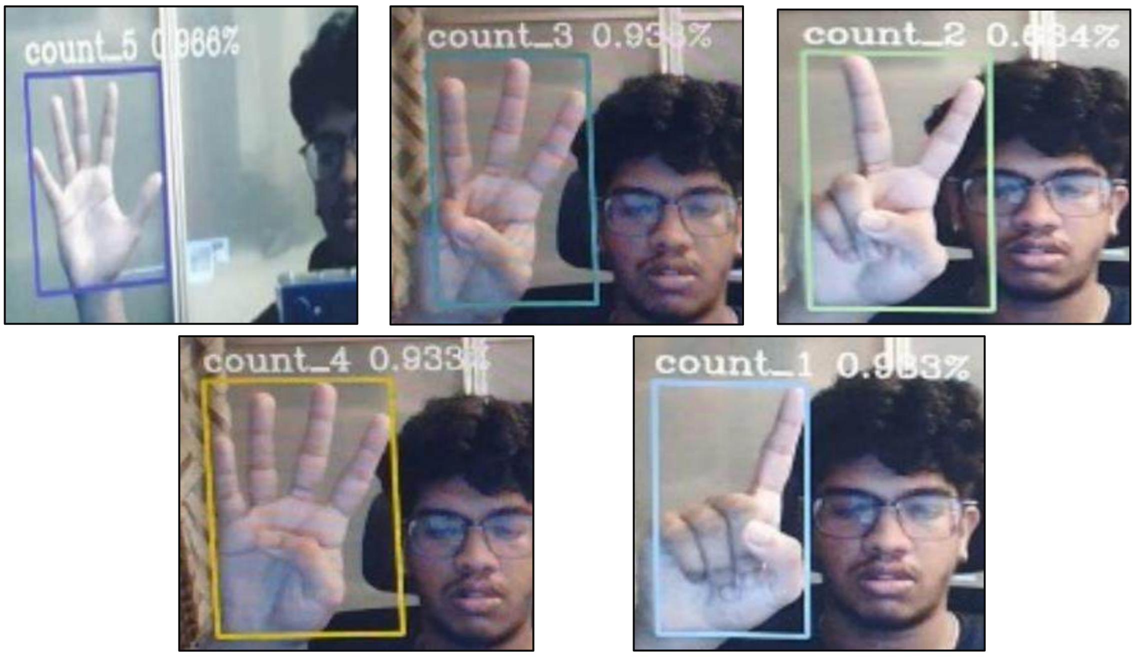


Figure 22:Dectction results

The images above are from testing the model. Above each bonding box for each image is the confidence level of the detection:

Count_1: 0.933 or 93%.

Count_2: 0.834 or 84%

Count_3: 0.933 or 93%

Count_4: 0.933 or 93%

Count_5: 0.966 or 96%

The confidence level for each finger count is close or almost 1. This helps to say that the model was trained successfully, and it can detect and classify the finger count. This can also be since was no overfitting of data.

ARAVIND SOUNDIRARAJAN: DISCUSSION

Exploring the integration of machine learning models presents a promising avenue for enhancing the accuracy and versatility of the finger counting system. The discussion will cover the process of Task 2, challenges faced, limitations and the measures taken to make task 2 successful.

Firstly, unlike task 1 which involved using mathematical algorithms and simple image processing technique which were used to create system to detect the finger counting, task 2 was completely different in terms of approach and process. So task2 involved collecting various and testing them to see which would be suitable to train a premade ai model which was yolo. The first data set included over 496 images which were all taken in the member's room and the focus for the image was the hand only. There was no background implemented and the angles of the fingers were not considered. This dataset was trained for 100 epochs.

The results for this data are as follows. Count_1 up till Count_5 was detected in the member's house. Count_4 and Count_5 got confused occasionally. When the location was changed to test the model. The confidence level dropped but at least 30 percent. This is understandable due to the way the dataset was collected. So, for the second data set a goal was set on type, number of and background of images. so, dataset focused more on the limits from the previous the dataset which background, lighting and taking same number of images for each finger count. So, dataset two was taken with different angles and different backgrounds and the format was done same for each count.

The result of using this dataset to train the model: The model was trained for 1000 epochs with 500 images. This confidence level for this new model is close to almost 1. The model is able to identify the finger counts under various conditions. The confidence level was at a constant 0.9 for all the gestures.

This also helps to get a deeper understanding on how datasets and model training work. Two data sets effect the model differently. The first data sets did not have lighting nor background implemented in it so the model doesn't work as good when the background constantly keeps changing. Whereas for the dataset two where the limitations were addressed the model improved and was able to identify. In a nutshell training the model with the right datasets is the key to a successful Machine learning system.

SHOKRI EYAD SHOKRI OUDA (TP065881)

TASK 1

INTRODUCTION

Human-computer interaction (HCI) evolves swiftly, fuelled by vision and AI breakthroughs, especially in gesture recognition technology it Spotlighted the hand gesture controls versatile interfacing across auto, robotics, consumer electronics, and healthcare and many other applications Offering natural, touchless interaction boosts accessibility and user experience. Recently, recognizing hand motions like swiping left, right, up, down so it became pivotal. And it can be used on other application Crucial where touchless is preferred like surgical rooms, driving inte-ractions requiring undivided primary task attention.

OVERVIEW OF THE SYSTEM

The code detects hand swipe gestures using a webcam (Real time) and OpenCV library. It will capture video frames then it will identify hand motions within a defined area, and determines the swipe direction so the system works in real-time and displays the detected swipe direction. The code was developed from scratch in PyCharm, an Integrated Development Environment (IDE). It uses standard computer vision techniques for robust hand gesture recognition. PyCharm aids in efficient coding, debugging, and testing of the application, making it suitable for developing complex computer vision applications.

TECHNIQUE/APPROACH

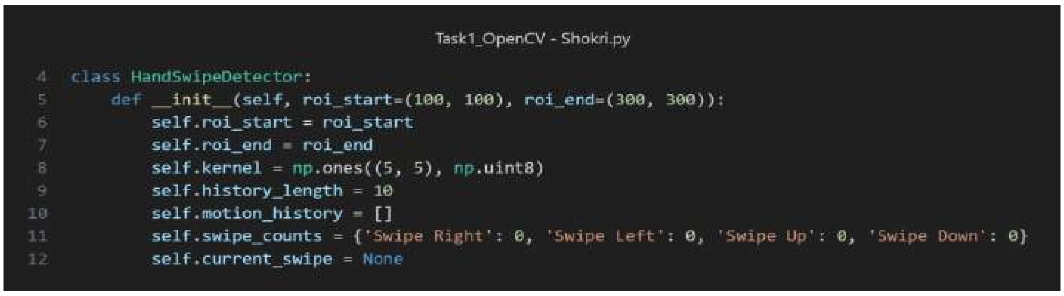
The system looks for hand movements in videos. It does this using the computer vision tools from OpenCV and Python code. The key is tracking hand motions accurately within a chosen area on the video screen that have been designed in the code.

1. **Region of Interest (ROI) Definition:** The system sets an ROI on the video. This limits where it looks for hand gestures so it will make the detection better and faster by ignoring other areas.
2. **Background Subtraction**: Using cv2.createBackgroundSubtractorMOG2(), it separates moving things (hands) from the still background so it can work well in different lighting and skin tones.
3. **Contour Detection**: After background subtraction, thresholds, and morphological operations (cv2.erode() and cv2.dilate()) clean up the foreground mask. cv2.findContours() then it will detect the outlines. The largest one is assumed to be the hand.
4. **Motion History**: The system stores the detected hand position (centre) from each frame, so the data shows the motion direction. It is used to determine if a swipe gesture occurred based on how the centres moved over time.

5. **Swipe Direction Analysis**: The system calculates the change in the x and y positions of the hand between video frames. It classifies the hand motion as left, right, up, or down based on the direction and amount of movement exceeding a set limit.

EXPLANATION OF THE CODE STRUCTURE

Hand Swipe Detector class:



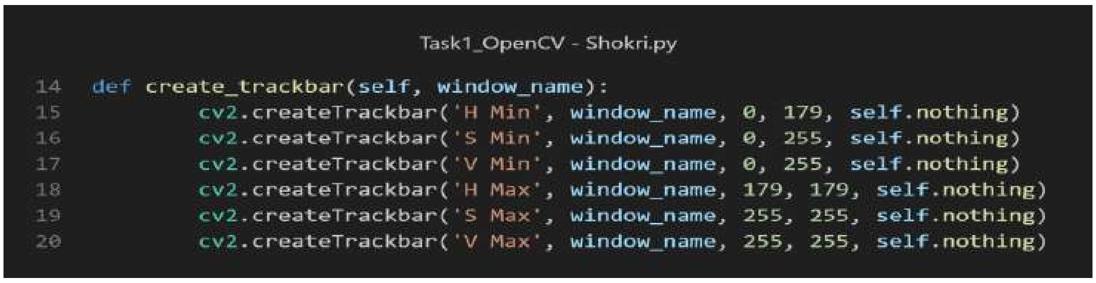
Task1_OpenCV - Shokri.py

```python
class HandSwipeDetector:
    def __init__(self, roi_start=(100, 100), roi_end=(300, 300)):
        self.roi_start = roi_start
        self.roi_end = roi_end
        self.kernel = np.ones((5, 5), np.uint8)
        self.history_length = 10
        self.motion_history = []
        self.swipe_counts = {'Swipe Right': 0, 'Swipe Left': 0, 'Swipe Up': 0, 'Swipe Down': 0}
        self.current_swipe = None
```

Figure 23: hand Swipe Detector

1. **roi_start and roi_end:** These define the top-left and bottom-right corners of the Region of Interest (ROI) rectangle. The ROI is the specific area in the video frame where the system looks for hand motions.
2. **bg_subtractor:** An instance of a background subtractor created using the MOG2 algorithm. It helps distinguish the moving parts (like a hand) from the static background. This is crucial for detecting motion.
3. **kernel:** A simple shape used to clean up noise in an image after the background removal.
4. **history_length:** The number of past frame positions stored to study the motion path.
5. **motion_history:** A list that keeps track of the detected hand positions across frames of the video.
6. **swipe_treshold:** The minimum distance the hand must move to be recognized as a swipe.

Get Trackbar Values:



Task1_OpenCV - Shokri.py

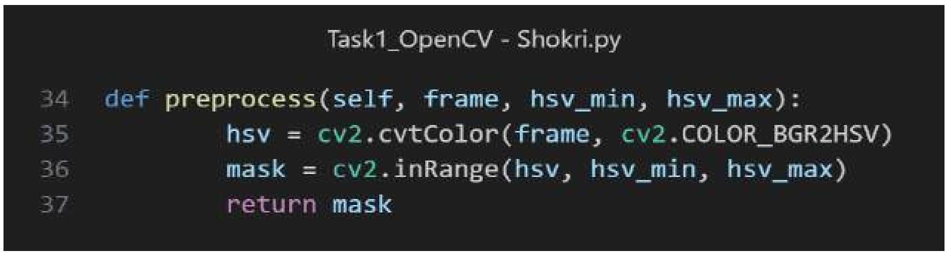
```python
def create_trackbar(self, window_name):
        cv2.createTrackbar('H Min', window_name, 0, 179, self.nothing)
        cv2.createTrackbar('S Min', window_name, 0, 255, self.nothing)
        cv2.createTrackbar('V Min', window_name, 0, 255, self.nothing)
        cv2.createTrackbar('H Max', window_name, 179, 179, self.nothing)
        cv2.createTrackbar('S Max', window_name, 255, 255, self.nothing)
        cv2.createTrackbar('V Max', window_name, 255, 255, self.nothing)
```

Figure 24: Create trackbar

1. **h_min,h_max:** This variables which are called hue values are in charge of the range of colour. So for this variable the range is from 0 to 179 so it can detect different colours.

2. **v_min,v_max:** This variable refers to the brightness of the images so it will be used in the trackbar so we can control the brightness of the frame. So for this variable the range is from 0 to 255.
3. **s_min,s_max:** This variable refers to the saturation of the images. Using the trackbar we can control the saturation of the frame. So for this variable the range is from 0 to 255.

Preprocessing the Frame:



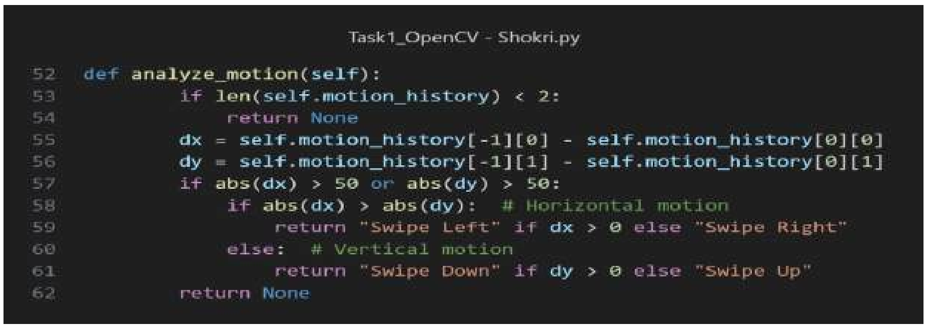
Task1_OpenCV - Shokri.py

```python
def preprocess(self, frame, hsv_min, hsv_max):
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        mask = cv2.inRange(hsv, hsv_min, hsv_max)
        return mask
```

Figure 25: preprocess function

1. **Gray scale to Hsv:** Helps in converting the image for from Gray scale to Hsv essentially so it will allow us to control the values from the trackbar.
2. **Masking:** This will allows us to pass the Hsv values so we can see in real time what the change looks like.

Analysing The Motion:

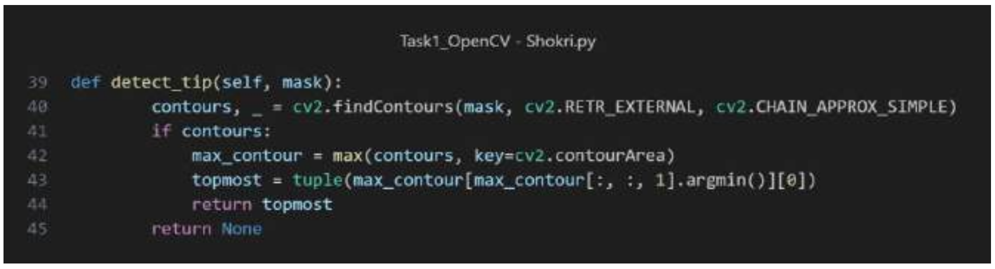


Task1_OpenCV - Shokri.py

```python
def analyze_motion(self):
        if len(self.motion_history) < 2:
            return None
        dx = self.motion_history[-1][0] - self.motion_history[0][0]
        dy = self.motion_history[-1][1] - self.motion_history[0][1]
        if abs(dx) > 50 or abs(dy) > 50:
            if abs(dx) > abs(dy):  # Horizontal motion
                return "Swipe Left" if dx > 0 else "Swipe Right"
            else:  # Vertical motion
                return "Swipe Down" if dy > 0 else "Swipe Up"
        return None
```

Figure 26: analyze motion function

1. **Motion History Check:** It checks the motion history and validates to see if 2 or more motion is in the history.
2. **Calculating dx:** Calculates the change in direction in the x axis.
3. **Calculating dy:** Calculates the change in direction in the y axis.
4. **Determing Swipe Gesture:** Check if the dx or the dy is greater than 50. If dx is positive it swipes left and swipe right if it is negative.
5. **Detecting Tips:**



Task1_OpenCV - Shokri.py

```python
def detect_tip(self, mask):
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        if contours:
            max_contour = max(contours, key=cv2.contourArea)
            topmost = tuple(max_contour[max_contour[:, :, 1].argmin()][0])
            return topmost
        return None
```

Figure 27: detect tip function

1. **Contour Detection:** This function helps in identifying the tips of the fingertips and it will draw the contours around it making it easier to detect.

RESULTS

Swipe Down Detection: when the hand moves in a Downward motion the camera detects the movements of the fingertip. So, when the fingertip moves along the y axis vertically down, the ROI would pick this up and as per the detection logic the movement will classified as Swipe Down.

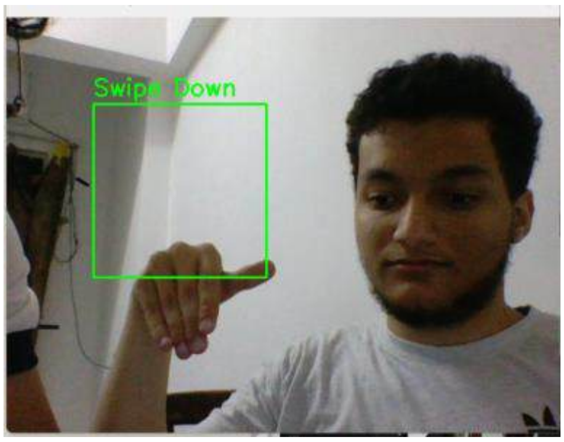


Figure 28: Swipe Down

Swipe Up Detection: when the hand moves in an upward motion the camera detects the movements of the fingertip. So, when the fingertip moves along the y axis vertically upwards, the ROI would pick this up and as per the detection logic the movement will classified as Swipe Up.

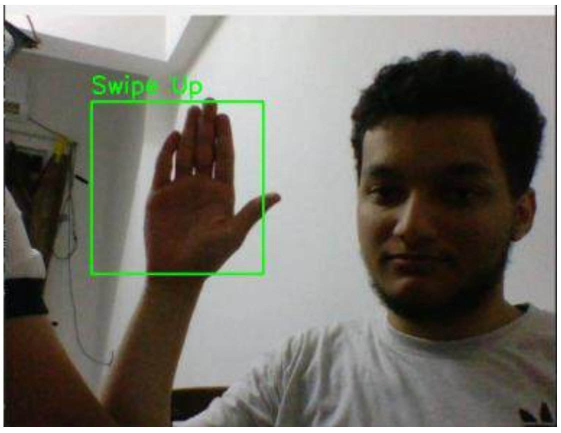


Figure 29: Swipe Up

Swipe Right Detection: The horizontal swipe motion follows the same logic for the vertical swipe motions. Once the background is perfectly tuned the and when the hand moves across the right side of the ROI it would be detected as Swipe Right.

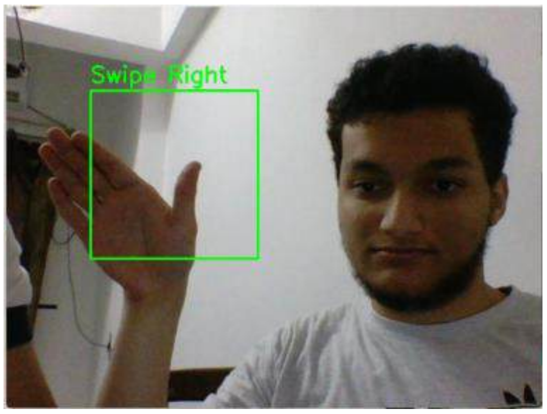


Figure 30: Swipe Right

Swipe Left Detection: The horizontal swipe motion follows the same logic for the vertical swipe motions. Once the background is perfectly tuned the and when the hand moves across the Left side of the ROI it would be detected as Swipe Left.

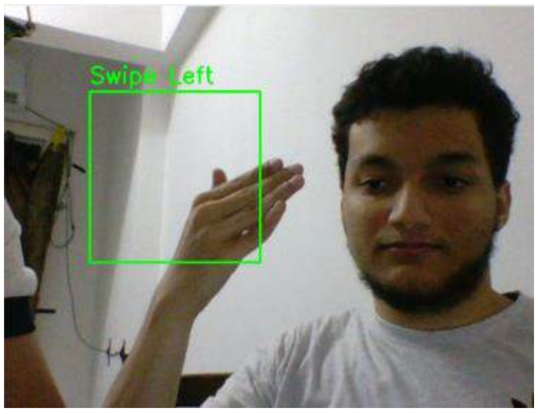


Figure 31: Swipe Left

DISCUSSION

Challenges and limitations: Firstly, the use of masking itself it was challenge as well as a limitation as before the system can start to detect the swipe the gestures, I have to manually tune using the trackbar function which control the HSV values as explained earlier. So once the HSV values are tuned and the background is masked the system will start to detect. This affects the efficiency of the system. Another limitation was the distance which effect the masking the detection. When the distance is far from the motion is relatively small for the ROI to pick up.

Approach and Comparative Analysis: In this view, the application of motion history in the detection of swipe directions is, therefore, a more direct approach, capturing most of the essence of the gesture more efficiently. This is at the cost of computation, while compared to optical flow techniques that may provide much detail within the frame. The method that was implemented is simple and straightforward.

Technical/Performance: Overall the performance of the system is good. It was able to the task at hand which detecting the swipe motions. So the level and complexity of the code it was able to achieve the task because There were a lot of technical application used into code example the use of masking and trackbar giving the user the ability to tune to system to their hands.

Improvements and Alternatives: The inclusion of optical flow methods may even help understand more about the detailed movements of objects across frames in the video and, therefore it reveal subtler movements of hands in the video so That means combining motion history with optical flow to compensate between the demand for high accuracy gesture recognition with low power consumption. A neural network applied to the data set of swipe gestures could further improve the system's accuracy and robustness in recognizing the movements of the palms, especially in similar ones.

Task 2

INTRODUCTION

This section in the report will be on task 2 and the complete process. The task 2 was to create an AI model, and it will enable a prediction of the swipe by the trained model. The decision to choose yolo V8 as our trainable model for the task was made. For task 2, it is a trainable model for predicting a swipe, and the categories to predict are palm, back arm and identify right hand or left. Let me take you through what yolo is before going to the overview of task2.

You only look once model that is YOLO is an object detection is an object detector that uses the latest object detection algorithm, and it proved perfect by the test of time and prospects and the most current. Most object detection methods reframe detection as a two-step process include. YOLO approach to object detection a regression problem, it divides the input image into grid and predicts bounding boxes and class probabilities directly from grid cells. It developed first because of this unique approach and the model is very good in time performance. The Object is smart enough to detect multiple objects simultaneously in one pass, as a result, it helps in applying the automobile vehicle and tracking system. yolo version 8 does this and more due to the technology shared then mechanism. YOLOv8 improves shared by previous versions and the all the object identified. YOLOv8 improves the level of sharing as the latest version. So, for task two Yolo V8 will be used.

Overview Of Task 2

The whole of task 2 consist of three main steps which will be explained in detail in the report. The steps consist of data collection, annotation, training and then testing. So, the data is in form of images. Then the image is annotated which then later used to train the model which is Yolo V8. Then the model trained is then used to test to see if the prediction is correct. So next part will be explaining the process from data collection till the model being trained until the results from the testing.

Process of Roboflow

Roboflow is the most complete and easy-to-use platform to build and deploy your machine vision systems. Roboflow empowers custom vision model development with the most comprehensive suite of tools, enabling a streamlined workflow spanning from model building and annotation to model training and serving. This means that from hobbyists to professionals, all users can work on projects where they can annotate images, preprocess data, train your models, and deploy within this unified ecosystem.

Another key aspect is that Roboflow provides user interfaces for both data annotation and preprocessing. The platform enables its customers to upload images, annotate images with labels, and preprocess them to the requirements of a particular model. This preprocessing task may include even such actions as resizing, normalization, and augmentation, which are very helpful for better model performance with robustness.

Implementation of Hand Gesture Recognition Using Roboflow:

Going through hand gesture recognition with Roboflow begins at the point of creating a project. Here is a definition is established that containing the general setting of the project, the scope, and the type of gestures that could be recognized. The project name was named "Hand Motion" with a focus and scope of considerations on Swiping Up, Swiping Down, Swiping Left, and Swiping Right.

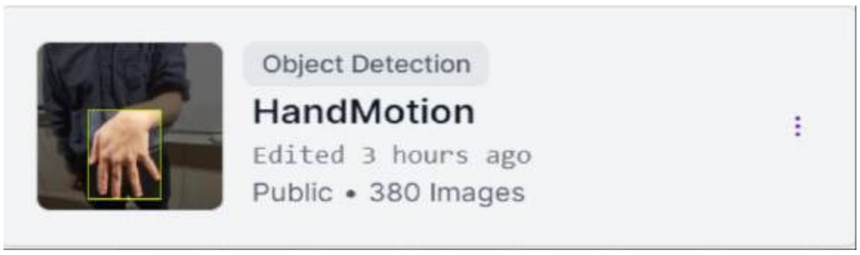
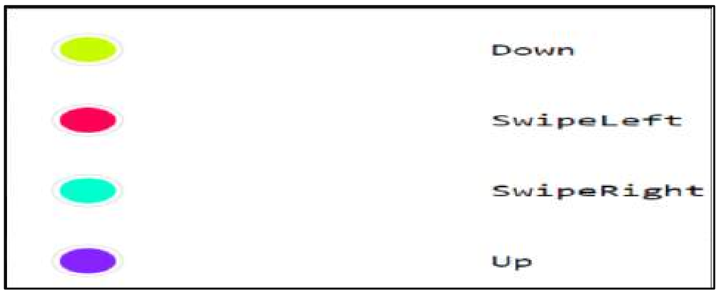


Figure 32: Project Folder Hand Motion

Class Definition

The next important step is the definition of classes. Classes will be referred to categories within the model. Each gesture has been defined within a separate class and each class got its own colour, so that they can make a difference something like "Swipe Up-blue ", "Swipe Down-yellow," "Swipe Left-red", "Swipe Right-green". These classes are going to be implemented and used in the annotation section.



●	Down
●	SwipeLeft
●	SwipeRight
●	Up

Figure 33 Swipe motion Classes.

Data Uploading and Annotation

There is a collection of pictures that was prepared through a phone camera where the angles, distance, and direction of the hand were properly placed to preserve a good dataset for the model. The dataset is then uploaded onto Roboflow with various images of different hand motion pictures. Every image is annotated for the hand and the motion being made. That's where the powerful annotation tools by Roboflow come into play: they ensure the proper labels are applied with coloured bounding boxes, something very necessary for effective model training.

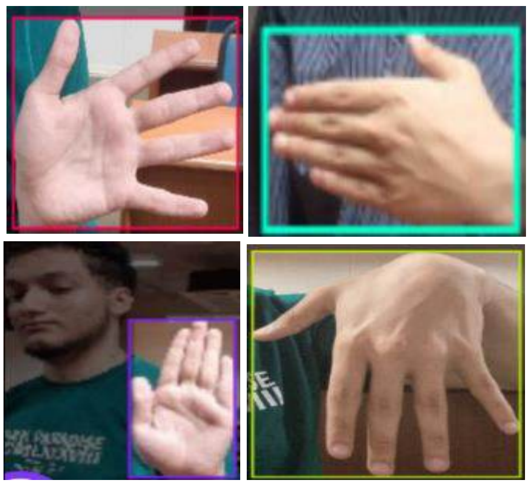


Figure 34 Annotating Swipe Gestures

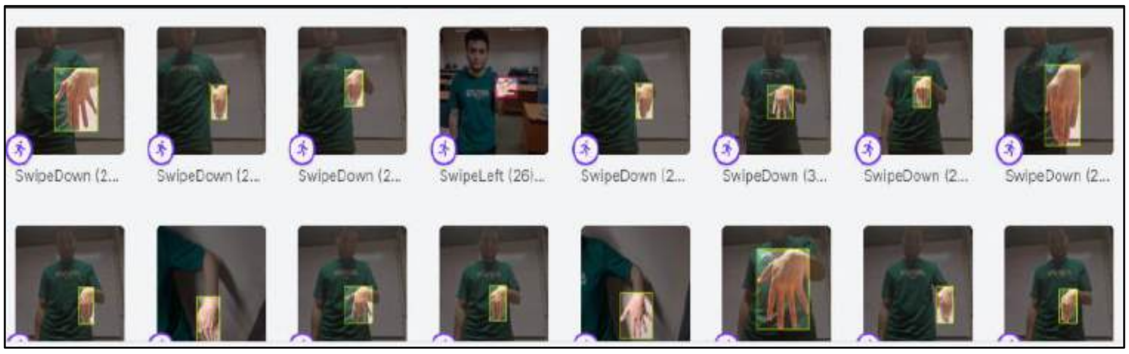
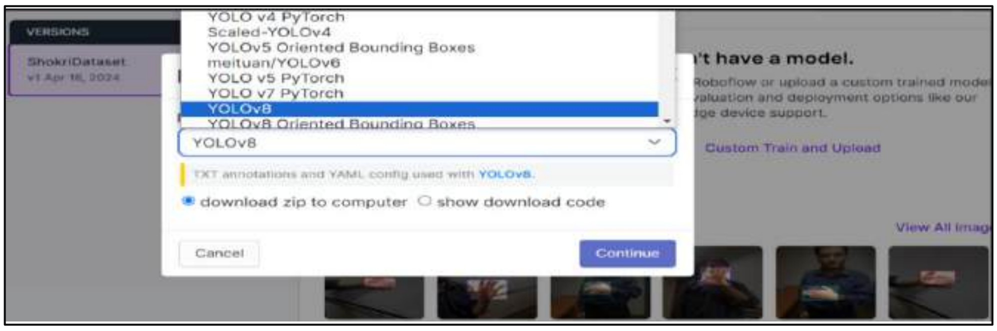


Figure 35 Annotated data sets

exporting to yolov8

once the entire dataset has been annotated, it can be exported as a yolov8, an efficient and accurate model derived from the famous 'you only look once' family of object detection algorithms. This is where roboflow comes in: easy exports right into the selected format, be it txt annotations or a yaml configuration file. After exporting it, we will get the dataset uploaded as well as the txt files which are going to be useful for the training of the data.



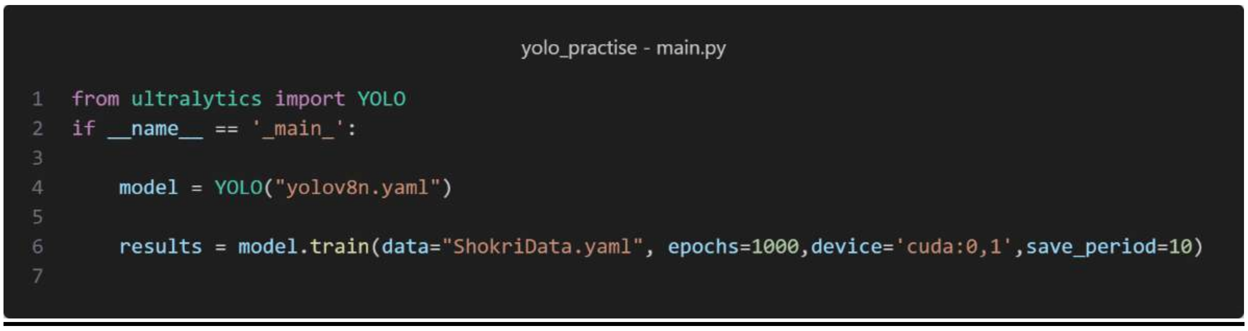
TXT annotations and YAML config used with YOLOv8.

't have a model.

Roboflow or upload a custom trained mode valuation and deployment options like our dge device support.

Figure 36: Exporting the datasets

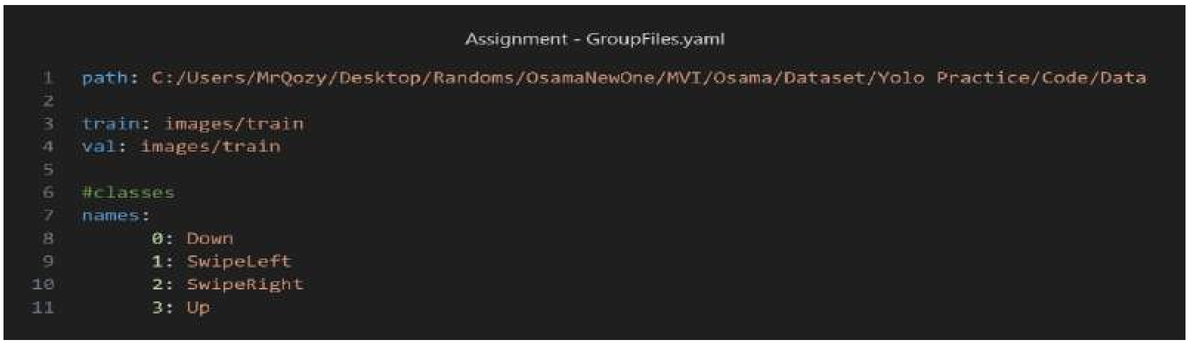
MODEL TRAINING AND YAML FILE


yolo_practise - main.py

```python
from ultralytics import YOLO
if __name__ == '_main_':

    model = YOLO("yolov8n.yaml")

    results = model.train(data="ShokriData.yaml", epochs=1000,device='cuda:0,1',save_period=10)

```

Figure 37:Yolo Training Model

1. The above figure shows the code used to train the model.so firstly using the python library ultarlytics yolo model is imported.
2. The line creates a YOLO class instance using the YOLOv8n model configuration file,
3. The fourth line is where the model is made to be trained.
4. So now in the code the epoch is set to 1000 which means the model will train all the images with the labels for 1000 epochs.
5. The saving period is set to 10 meaning every 10 training cycles, the system's current progress gets recorded. This safety measure helps in two ways. First, you can look back at past versions. Second, if training gets disrupted, you can re-start from the last saved point. So, by preserving intermediate results, this setting aids longer training sessions.
6. Device = ‘cuda:0,1’ this tells the model to use CUDA for training on GPU devices 0 and 1. This causes the program to train the model faster as the gpu is being used to its full potential.


Assignment - GroupFiles.yaml

```yaml
path: C:/Users/MrQozy/Desktop/Randoms/OsamaNewOne/MVI/Osama/Dataset/Yolo Practice/Code/Data

train: images/train
val: images/train

#classes
names:
        0: Down
        1: SwipeLeft
        2: SwipeRight
        3: Up
```

Figure 38:Shokri Yaml File

1. The above image shows the yaml file used in training the model. Here the number on the left side in the yaml file represent the classes for the swipe motion.
2. So based on the classes or labels we added in Roboflow the number denotes the class names for example 3 represents swipe up motion.

SHOKRI EYAD SHOKRI OUDA: GRAPH RESULTS

Train/box_loss and Val/box_loss: These graphs represent the loss associated with the bounding box predictions for the model over the training and validation set, respectively. This loss refers to the model's performance capabilities in the accurate localization of the hands in the frames of the images or video. Low, stable box loss means the model is doing an excellent job of predicting where the hand should be while moving.

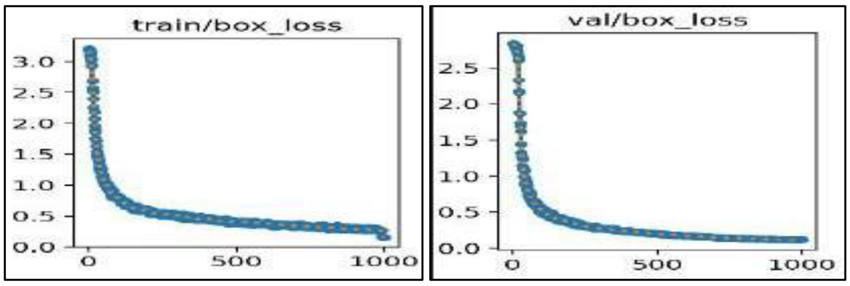
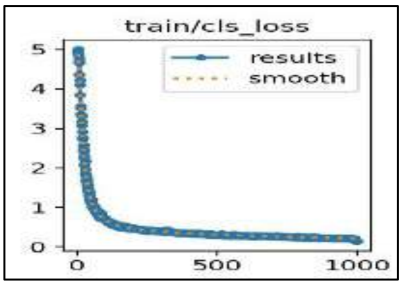


Figure 39: Train and val Box Loss

Train/cls_loss, val/cls_loss: Classification loss in the training and validation set. The loss which denotes the accuracy of the model classifying the object detected within the bounding box. In hand motion detection, this will mean the model can correctly identify the kind of motion the hand has made between up, down, swipe left, swipe right.



train/dfl_loss and val/dfl_loss: direction focused loss during training and validation. Assuming "dfl" is from direction focused loss, this should mean some type of a specific loss metric targeting how well the model is at predicting the direction of the hand movement. This is indicated by the low dfl_loss, which is a sound performance of the model in understanding the trajectory or direction of the hand movement.

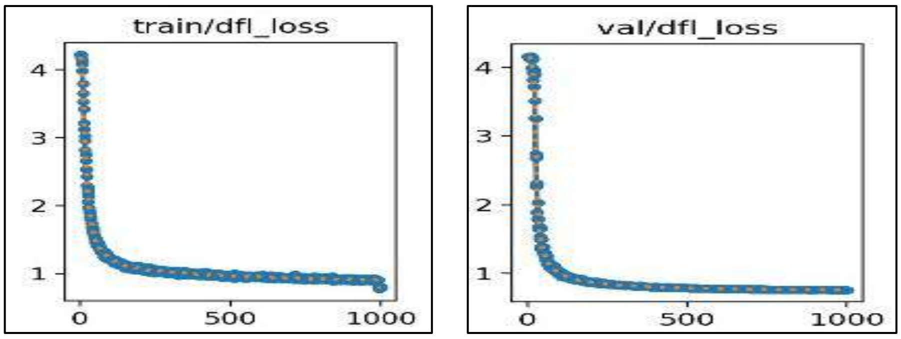


Figure 40:train and val dfl_loss

Metrics/precision(B) and metrics/recall(B) quantities: In this case, precision represents the number of instances that a correct optimistic prediction holds over all the optimistic predictions that have been made, while recall represents the number of cases that a correct optimistic prediction holds over all the positives that should have been retrieved. High precision translates to high accuracy in the area of hand motion, meaning that when the model predicts a particular hand motion, most probably, the prediction will come to the actuality. Similarly, high recall would imply that the model can pick out most instances of hand motion.

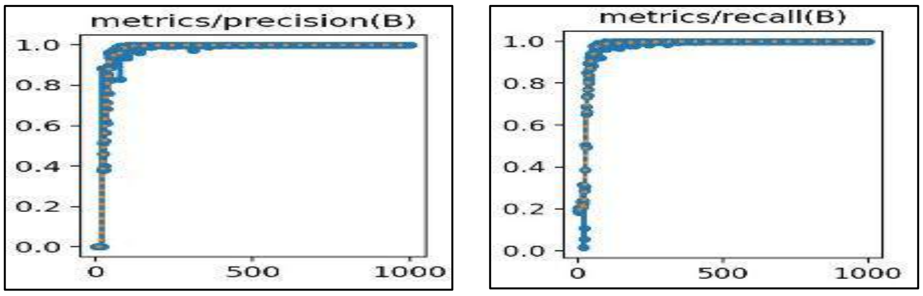


Figure 41:Precision and Recall Metrics

Metrics/mAP50(B) and metrics/mAP50-95(B): Mean Average Precision (mAP) is a summary metric considering precision and recall over many threshold levels. The "50" and "50-95" refer to the IoU (Intersection over Union) threshold. mAP50 indicates that the average precision was measured with an IoU at the threshold of 0.5, while mAP50-95 means the manner of averaging mAP computed at some different thresholds from. The high mAP scores in hand motion detection will indicate that the model is highly accurate at detecting the hand, classifying the motion, and correctly predicting the motion over different IoU thresholds.

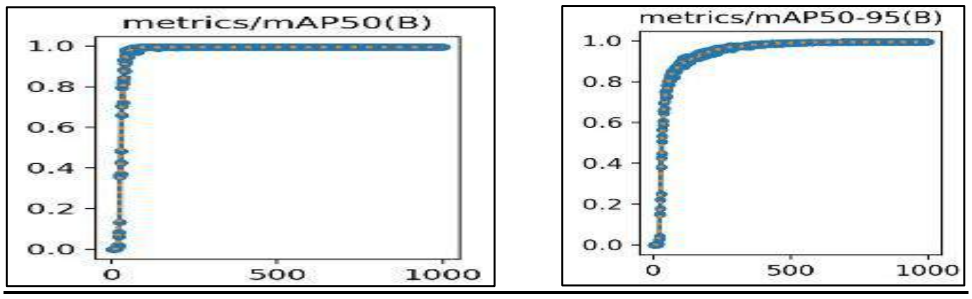


Figure 42:map50(B) and map50-95

<u>Bar Graph:</u> This is a bar graph with 4 bars, each labelled differently regarding the hand motion gestures. Each represents the number of cases related to that gesture (possibly detections or recognized actions). The "Down" and "Up" gestures are high, showing that either these movements were recognized most times in a data set or are more in the number of samples than others. "Swipe left" is a bit lower, and "Swipe right" is the least among them. This could mean that the 'Up' and 'Down' motions in the data set are more sensitive, or there are more frequent such occurrences, or both.

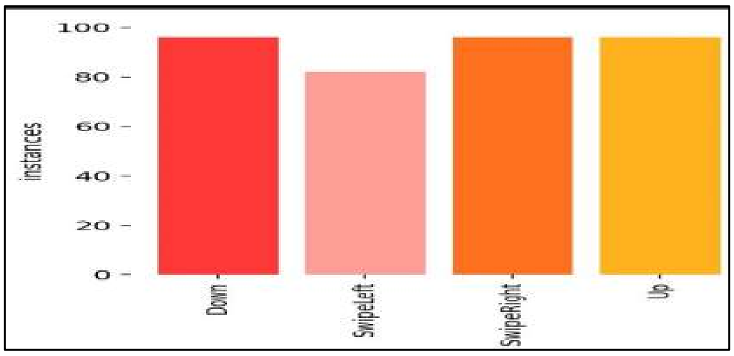


Figure 43:Instance bar Graph.

<u>F1-Confidence Curve:</u> This plot is a bit involved in that it's essentially the F1 score against the confidence threshold for several hand gestures. The F1 score is the harmonic mean of the precision and recall of the test. On the x-axis, one would expect "Confidence" to represent the threshold confidence that the detection is right. From the graph, some sort of curves are exhibited for 'Down', ‘Up', 'Swipe Right', and 'Swipe Left'. All seem to collapse at a given point, where F1 score 1.00. This means that above this confidence threshold, the model attains perfect precision and recall for all the classes. Below this threshold, it becomes sharp, pointing towards very high sensitiveness of the F1 score to the confidence threshold.

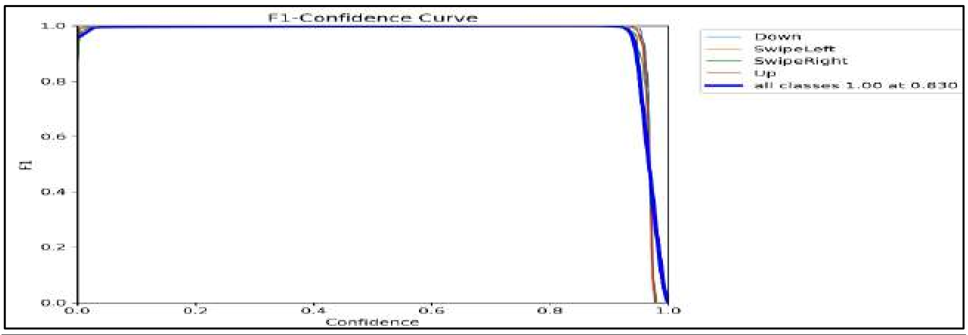
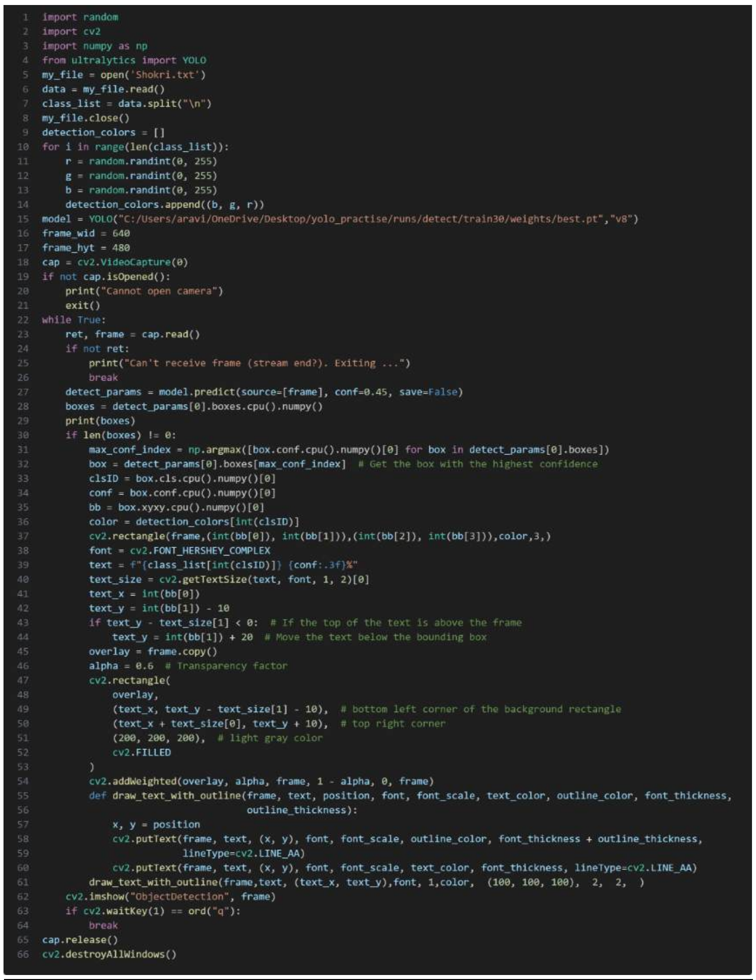


Figure 44:F1 Confidence Curve

TESTING CODE



```
 1  import random
 2  import cv2
 3  import numpy as np
 4  from ultralytics import YOLO
 5  my_file = open('Shokri.txt')
 6  data = my_file.read()
 7  class_list = data.split("\n")
 8  my_file.close()
 9  detection_colors = []
10  for i in range(len(class_list)):
11      r = random.randint(0, 255)
12      g = random.randint(0, 255)
13      b = random.randint(0, 255)
14      detection_colors.append((b, g, r))
15  model = YOLO("C:/Users/aravi/OneDrive/Desktop/yolo_practise/runs/detect/train30/weights/best.pt","v8")
16  frame_wid = 640
17  frame_hyt = 480
18  cap = cv2.VideoCapture(0)
19  if not cap.isOpened():
20      print("Cannot open camera")
21      exit()
22  while True:
23      ret, frame = cap.read()
24      if not ret:
25          print("Can't receive frame (stream end?). Exiting ...")
26          break
27      detect_params = model.predict(source=[frame], conf=0.45, save=False)
28      boxes = detect_params[0].boxes.cpu().numpy()
29      print(boxes)
30      if len(boxes) != 0:
31          max_conf_index = np.argmax([box.conf.cpu().numpy()[0] for box in detect_params[0].boxes])
32          box = detect_params[0].boxes[max_conf_index]  # Get the box with the highest confidence
33          clsID = box.cls.cpu().numpy()[0]
34          conf = box.conf.cpu().numpy()[0]
35          bb = box.xyxy.cpu().numpy()[0]
36          color = detection_colors[int(clsID)]
37          cv2.rectangle(frame,(int(bb[0]), int(bb[1])),(int(bb[2]), int(bb[3])),color,3,)
38          font = cv2.FONT_HERSHEY_COMPLEX
39          text = f"{class_list[int(clsID)]} {conf:.3f}%"
40          text_size = cv2.getTextSize(text, font, 1, 2)[0]
41          text_x = int(bb[0])
42          text_y = int(bb[1]) - 10
43          if text_y - text_size[1] < 0:  # If the top of the text is above the frame
44              text_y = int(bb[1]) + 20  # Move the text below the bounding box
45          overlay = frame.copy()
46          alpha = 0.6  # Transparency factor
47          cv2.rectangle(
48              overlay,
49              (text_x, text_y - text_size[1] - 10),  # bottom left corner of the background rectangle
50              (text_x + text_size[0], text_y + 10),  # top right corner
51              (200, 200, 200),  # light gray color
52              cv2.FILLED
53          )
54          cv2.addWeighted(overlay, alpha, frame, 1 - alpha, 0, frame)
55          def draw_text_with_outline(frame, text, position, font, font_scale, text_color, outline_color, font_thickness,
56                                     outline_thickness):
57              x, y = position
58              cv2.putText(frame, text, (x, y), font, font_scale, outline_color, font_thickness + outline_thickness,
59                          lineType=cv2.LINE_AA)
60              cv2.putText(frame, text, (x, y), font, font_scale, text_color, font_thickness, lineType=cv2.LINE_AA)
61          draw_text_with_outline(frame,text, (text_x, text_y),font, 1,color,  (100, 100, 100),  2,  2,  )
62      cv2.imshow("ObjectDetection", frame)
63      if cv2.waitKey(1) == ord("q"):
64          break
65  cap.release()
66  cv2.destroyAllWindows()
```

Figure 45:Shokri Testing Code

1. **Libraries:** OpenCV was used to access the camera and feed the live prediction to the camera live. the ultralytics library is used to access the trained model and asking the model to predict.
2. **Highest Confidence**: Line 32 is implemented in the code to filter all the detections with the lowest confidence which allows the model to only display the highest confident predictions.

3. **Display Processing**: Identify the most confident detection, extract and process its details, draw the bounding box, and label on the frame, and adjust the label's position if necessary.
4. **Advanced Text Rendering**: Utilize a function draw_text_with_outline to enhance text visibility against varying backgrounds by first drawing a thick, darker outline and then the main text in a lighter color.
5. **Process Video Feed**: Continuously capture frames, check for errors, use YOLO to predict objects with a confidence threshold of 0.45, and convert detections to a NumPy array for manipulation.

RESULTS

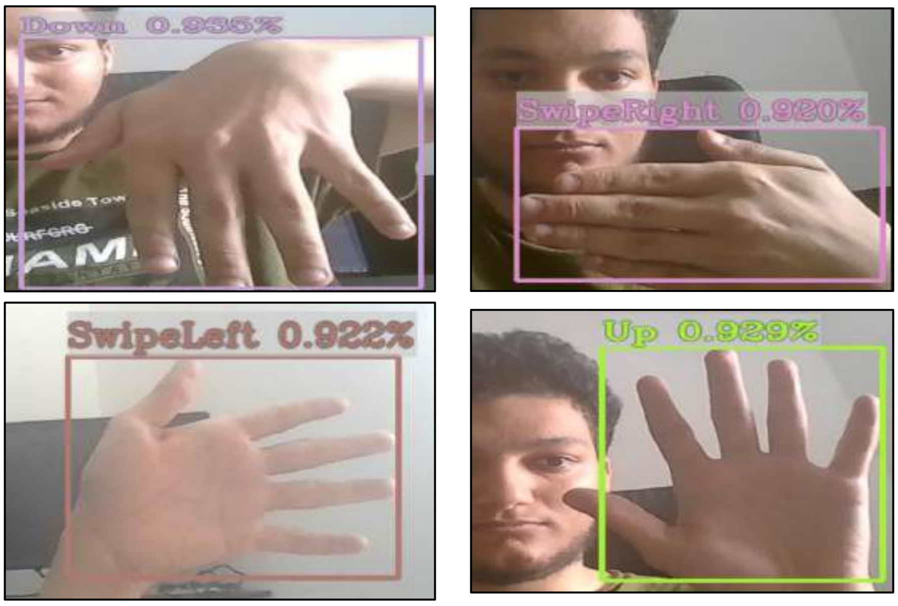


Figure 46:Results Of Detection

The images above are from testing the model. Above each bonding box for each image is the confidence level of the detection:

Up: 0.929 or 92.9%.

SwipeLeft: 0.922 or 92.2%

SwipeRight: 0.92 or 92%

Down: 0.935 or 93.5%

The swipe motions were accurately detected, and the performance of the detection were very smooth and seamless between gestures. Another point that proves this is the confidence level above the bounding for each swipe motion.

Discussion

Pursuing the goal of a reliable hand motion detection system capable of recognizing four critical gestures: up, down, swipe right, and swipe left, led us through a series of challenges and methodological fine-tuning that typifies the subtleties at play with machine learning applications to gesture recognition. On completion of the first exercise of data collection, the data corpus produced a total of 211 images. This was considered very poor on several grounds: First, incomplete labelling of images; secondly, imbalance in the representation of some types of gestures; thirdly, not enough data for one swipe-down kind of action.

The problem was, in part, addressed by an incremental approach, widening the dataset's scope and refining the data quality. However, the problem with the initial misclassifications, which do not affect performance but are instead for the first-time gesture, was purely the question of too few images available for each movement. This is especially evident in the swipe-down motion, where adding pictures to the data set is hypothesized to make the model better able to learn. The dataset was then further augmented by this hypothesis to have a more balanced representation in quantity, amounting to around 380 images. This expansion had a strategic emphasis on the scope of backgrounds against which gestures were performed. It included about ten different background settings for the model not to have high dependence on the background and, in general, to be more robust in various real-world settings.

Added to the dataset, the most crucial difference was a focus on the movements of the left hand, as compared to the right hand. This pretty much became important in the balanced input—making it, if possible, bias-free—and ultimately was a great help for the learning of the model. In such cases, they could believe that the focused single-hand orientation would make the system generalize from the training data to applications in the natural environment without problems regarding spatial orientation, which was a cause of recognition errors.

CUDA and PyTorch frameworks have been used to quickly process and train the model with GPU acceleration, which can handle the increased computational load brought about by a larger and more diverse dataset. Also, resizing images before training is one of the strategic decisions since this tries to speed up training without any losses in accuracy in the model. Hence, it will have to be a trade-off between efficiency and performance.

Some critical insights are summarized below, which were repeated in the iterative process of refining the dataset and re-evaluating the system.

Firstly it has been demonstrated that the dataset is crucial for training machine learning models in gesture recognition. Mislabelling or under-representing some of the gestures could lead to huge biases and inaccuracies in gesture recognition as had first been observed with the swipe-down motion.

Secondly the introduction of background variability by the use of a pool of diverse recruits dramatically increased the adaptability and robustness of the model. Adaptation is instrumental in dynamic environments of applications, where the background conditions are not controlled by the model.

Finally focus on a single-hand orientation simplified the training of the model and it gave better performance thus underlining the practicality of reducing variable factors in initial training phases to lay a solid foundational accuracy before the introduction of more complexities so These findings contribute to the broader discourses on the application of machine learning in real-time gesture recognition and The study also has important implications for practitioners and researchers interested in understanding the dynamic interplay of data quality, model training, and application context. Future work may investigate the potential for fusing either hand orientation after establishing baseline accuracy or explore the use of more sophisticated image augmentation approaches for modelling scaling robustness and accuracy.

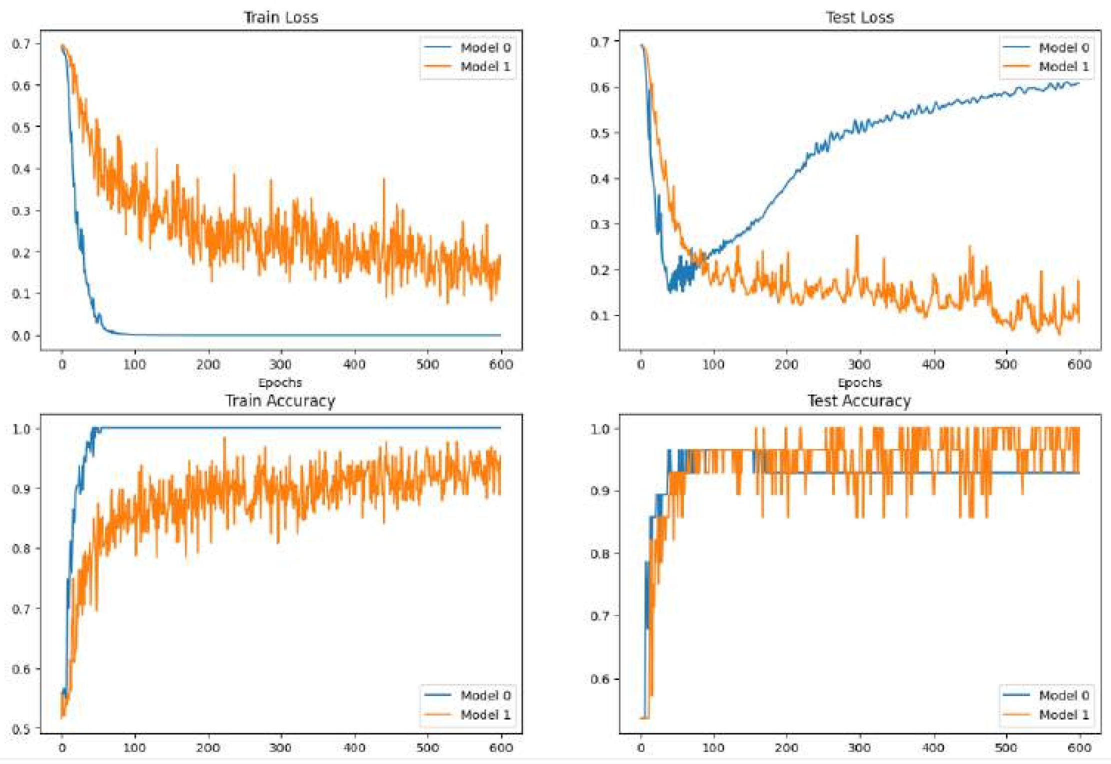


Figure 47 Comparing Models

As we can see from the figure above the graph after training two models from the CNN code it was more accurate than yolo code only for left up down right, as we can see from the graph it shows the loss during the training phase for both of our model as we can see that model 0 reaches quickly the low loss and remains stable throughout the epochs indicating good convergence model1 and it represent by the orange line and it has much more volatile loss suggesting lets stable training it does decrease overall .

then for the test loss the graph represent the loss during the testing or validation phase and it ideally this should mimic the training loss patten but often will be higher indicating how well our or the model is to generalizes the unseen data and model 0 as we can see it shows an increasing trend in loss after the initial drop which might suggest the overfitting for that model and it does not generalize well after certain point model 1 maintain a lower and more stable test loss after the inrail drop. And for the Train accuracy this graph displays the accuracy during training both models as we can see are rapidly achieve high accuracy and then plateau and for model 0 achieves slightly higher accuracy than model 1 both models are relatively close and remain stable, indicating both models are learning effectively from the training data. And the test accuracy for the test accuracy the graph shows how accurately the models predict the correct classes or labels on the separate test set. Models 0 shows a plateau in accuracy which is kind of good sign of the learning stability and for model one it have more variance in its accuracy and at the same time it reaches a similar plateau to model 0 so the fluctuations suggest that it may not perform as consistently.

But at the end both yolo and CNN from scratch work just fine but for yolo it have more accuracy when it is real time capture and the other way around for the other code form the CNN one it have higher accuracy when images are uploaded

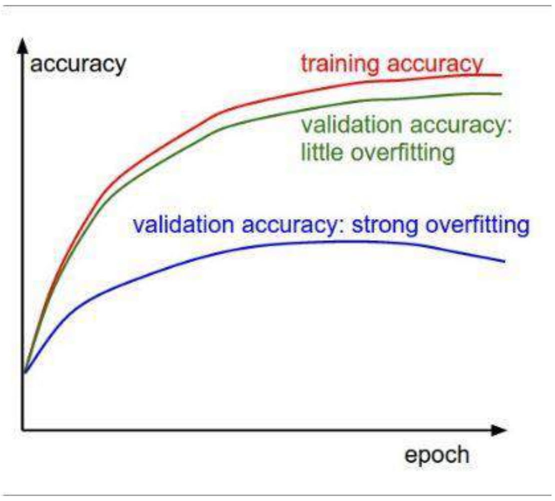


Figure 48 Accuracy and epoch.

As we can see from the figure above, we can tell that sometimes by increasing the number of epochs it may cause an overfiring which happen before in our model, so the number of epochs has decreases to avoid the overfitting issue. So, the key takeaway is that while high training performance is desirable the ability of the model to perform well on the unseen data or the validation performance is caracal. Overfitting is often identified when the model performance on the training data continues to improve while it is performance on the validation data gets worse by time at the time when know that overfitting is happening so to avoid overfitting there are some techniques such as early stopping and regularization pr using more trainable data.

TASK 2 CNN from scratch (Shokri)

Introdction To Cnns

CNN is a convolutional neural network and it is inspired by the organization of the animal visual cortex and it is designed to automatically and adaptively learn spatial hierarchies of features, from the low level to the high level patterns and CNN typically include number of layers which is:

1. Convolutional layers: these layers basically to apply number of filters to the input to create features maps that will summarize the presentence of detect feature of the input and the input can be images, text, or voice massages

2. Polling layer: Therese layer will reduce the dimensionally of each feature of map while it will retaining the most important information

3. Fully connected layers: toward the end CNN is a one or more fully connected layers that will use the feature of the layers to classifying the input images into categories.

   Application of CNN beside the vision CNN are used in other applications like speech recognition, natural league processing and more, one of the main advantages is the ability to develop and improve the understanding of an image through layers. And that's mean lower layers might recognize edges while some of the deeper layers might recognize more complex structure like shapes or objects.

   Training the CNN involve using set of data or labelled images to teach the model so it can recognize and differentiate between the various categories that it is supposed to handle and the training process adjust parameters of the layer that have been used in our module so it can perform it task accurately.

   Environment Setup: Set up the compute device (CPU or GPU) based on availability, and establish paths for image data storage.

   Data Handling: Data

   Make sure these directories exist before creating them and print their structure by listing them.

   Reads and lists images from the specified directories that will be used for training and testing.

Randomly select an image to display its details but comment out the actual display code (matplotlib) itself.

Data Transformation:

Apply operations on the images, like resizing and converting the images to tensors, which would be needed in processing with PyTorch models.

Image Display Function: It has a function, plot_transformed_images, which can plot transformed images for visual inspection. However, all actual plotting commands within this function are commented out.

Dataset and DataLoader Setup

Organize the images cleanly in a directory according to PyTorch with a dataset suitable for training/testing using datasets.ImageFolder.

Instantiate DataLoader instances for batching and shuffling the data for easy loading during training or testing the model.

Model Definition:

It defines a custom PyTorch Dataset class, ImageFolderCustom, which is similar in functionality to torchvision. datasets.ImageFolder but with potentially more flexible or specific customizations.

Another CNN model setup (TinyVGG) is used, meant for simple applications with several layers and parameters much less than a complete VGG network.

Data Augmentation

Transforms the training data with random flipping, rotation, color jittering, and other augmentations for better generality.

Training Functions:

Defines train_step and test_step functions for running the training and validation/testing loops, including forward pass, calculating loss, backpropagation, and computing accuracy.

Comprehensive training function (train) that encapsulates the training and testing steps over a specified number of epochs, capturing loss and accuracy metrics for analysis.

File Name:

But there is a function plot_loss_curves whose purpose is to plot the training and testing loss and accuracy curves, but the actual plotting commands are commented out.

Saving Models:

Utility function save_model for saving trained models

Coding Cnn

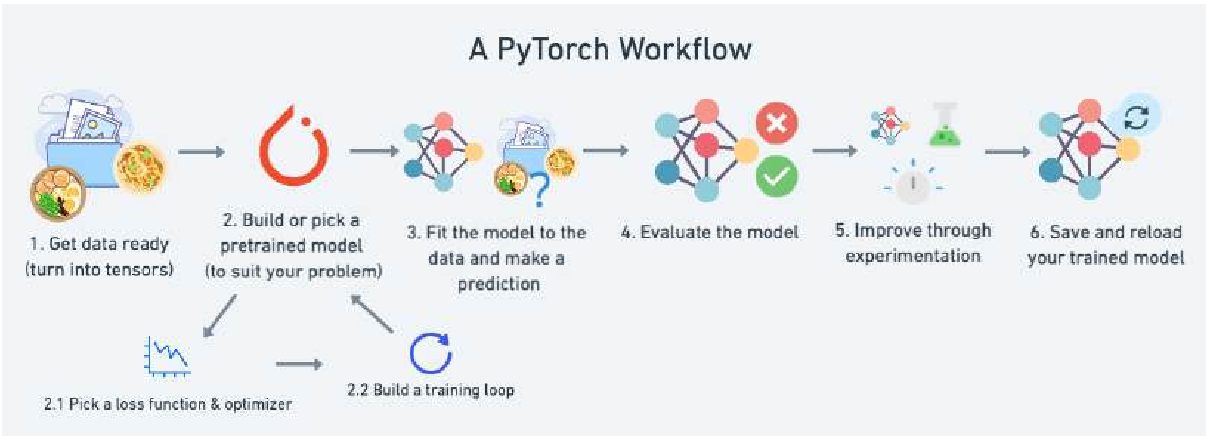
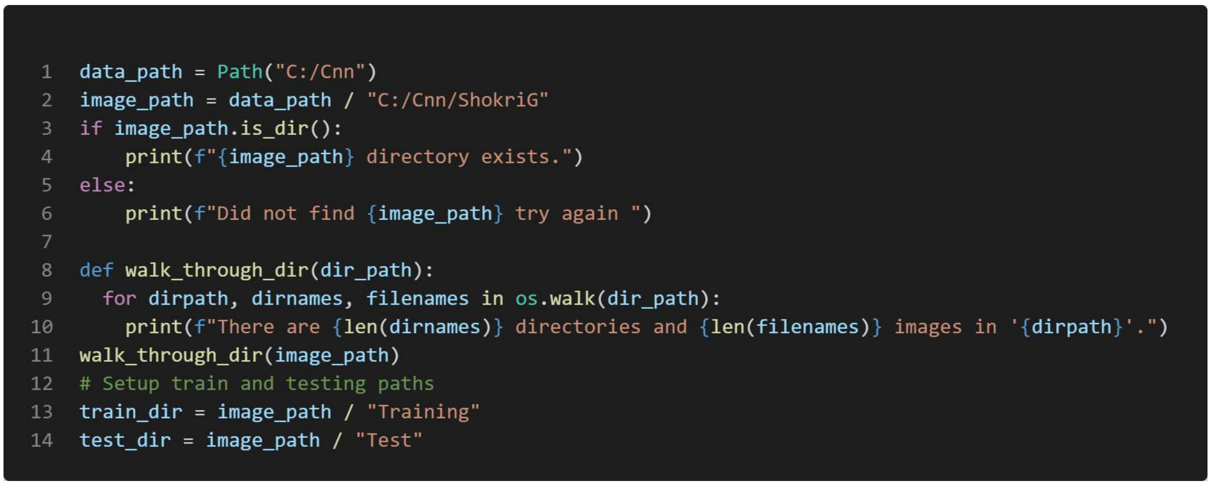

A PyTorch Workflow
1. Get data ready (turn into tensors)
2. Build or pick a pretrained model (to suit your problem)
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentation
6. Save and reload your trained model
2.1 Pick a loss function & optimizer
2.2 Build a training loop

Figure 49 Cnn Building

<u>1-Getting the data and truting into tensors</u>

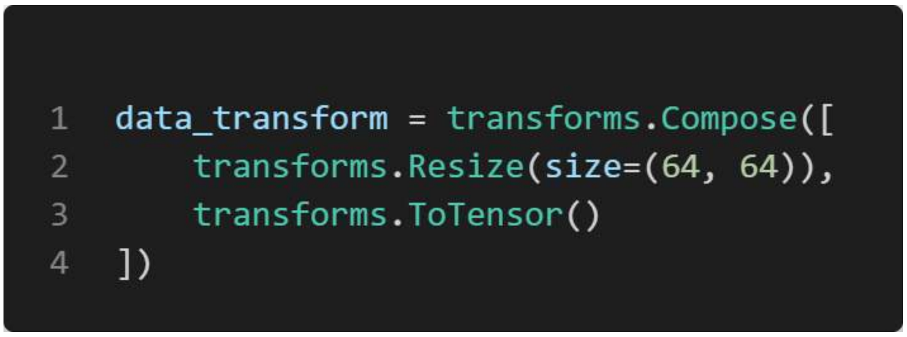
So as shown in the Figure 49 the first thing that we need to do to code cnn is get the data ready and trurn it into tensors and tensors is one of the most important thing in machine learning so tensors serves as the primary data stracture used and store also manipulate data across many frameworks, like tensorflow, pytorch and others lets see how we get the data and trun it into tensors the code:



```python
data_path = Path("C:/Cnn")
image_path = data_path / "C:/Cnn/ShokriG"
if image_path.is_dir():
    print(f"{image_path} directory exists.")
else:
    print(f"Did not find {image_path} try again ")

def walk_through_dir(dir_path):
  for dirpath, dirnames, filenames in os.walk(dir_path):
    print(f"There are {len(dirnames)} directories and {len(filenames)} images in '{dirpath}'.")
walk_through_dir(image_path)
# Setup train and testing paths
train_dir = image_path / "Training"
test_dir = image_path / "Test"
```

Figure 50 GetDataCode

As we can see from the code above that data path is define so now the data or the imgaes is ready now we have to turn it into tensors.



```python
data_transform = transforms.Compose([
    transforms.Resize(size=(64, 64)),
    transforms.ToTensor()
])
```

Figure 51 CodeToTensor

As we can see from Figure 51 first we have import torchvision and import datasets and transforms so and that have been used here as shown in the figure that the image have been resize and then turn in into tensors as the first step for building CNN.anad we have split our data and test sets to be 80% training and 20% testing.

<u>2-Build or pick a pretrain model</u>

Our data set have been test and train by pretrain model which is yolo now this modle will be build so now we will build model to train our data and test it so I will design my cnn based on this which is tinyvgg

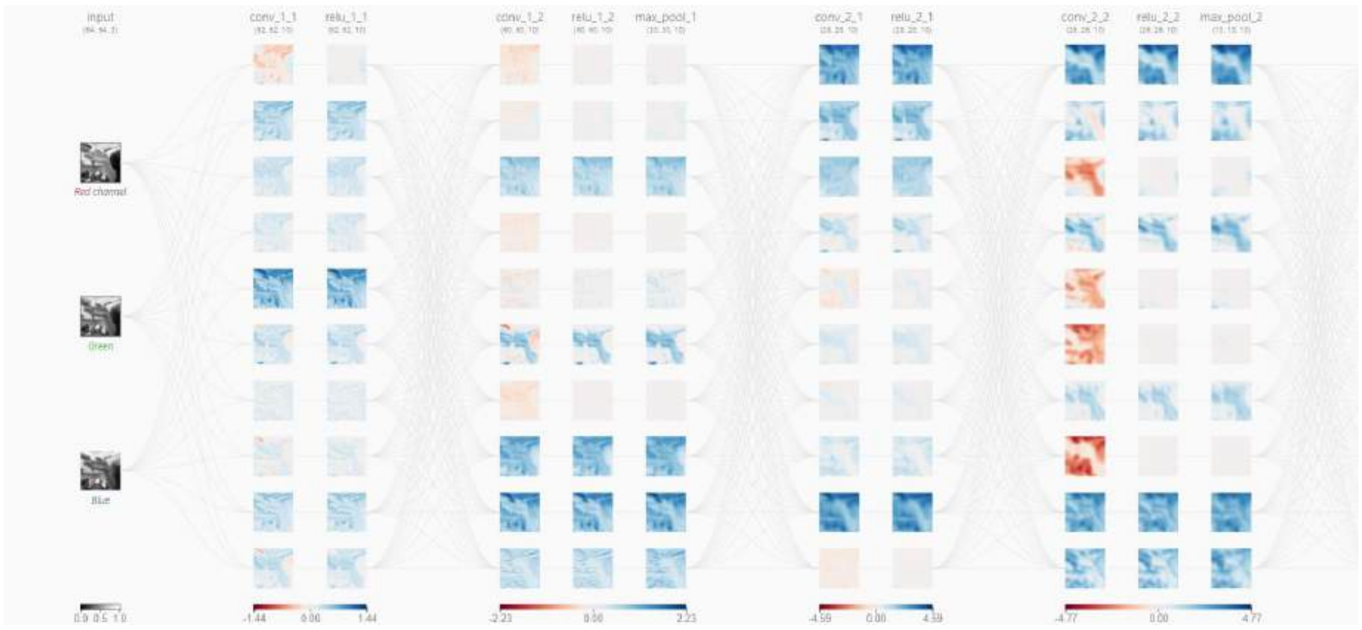
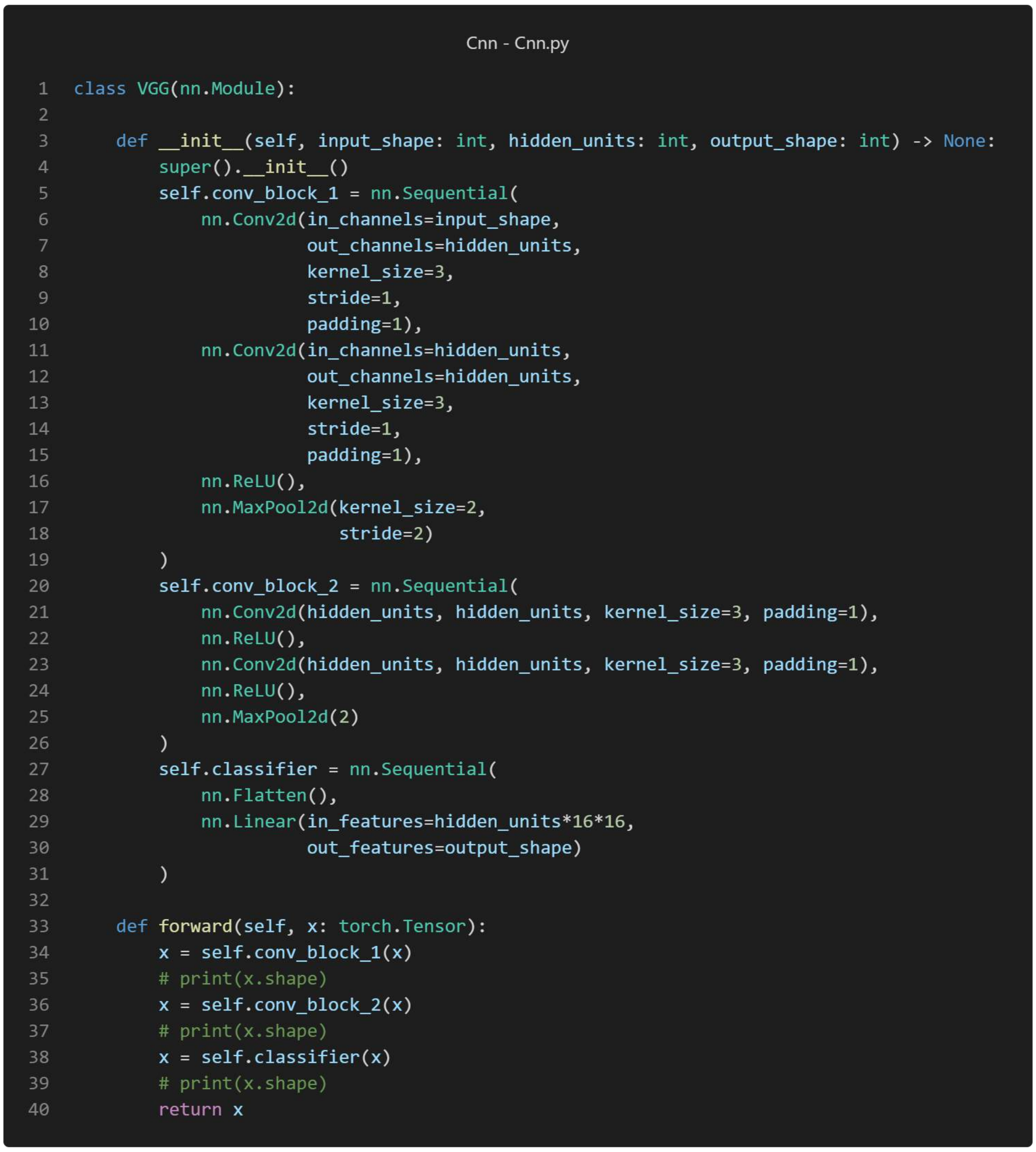


Figure 52 CNNGraph

The CNN was design based on this graph where we have 3 block and each block have almost 3 layers so lets go to the code where we implement it (Learn Convolutional Nerual Network, 2020)


Cnn - Cnn.py

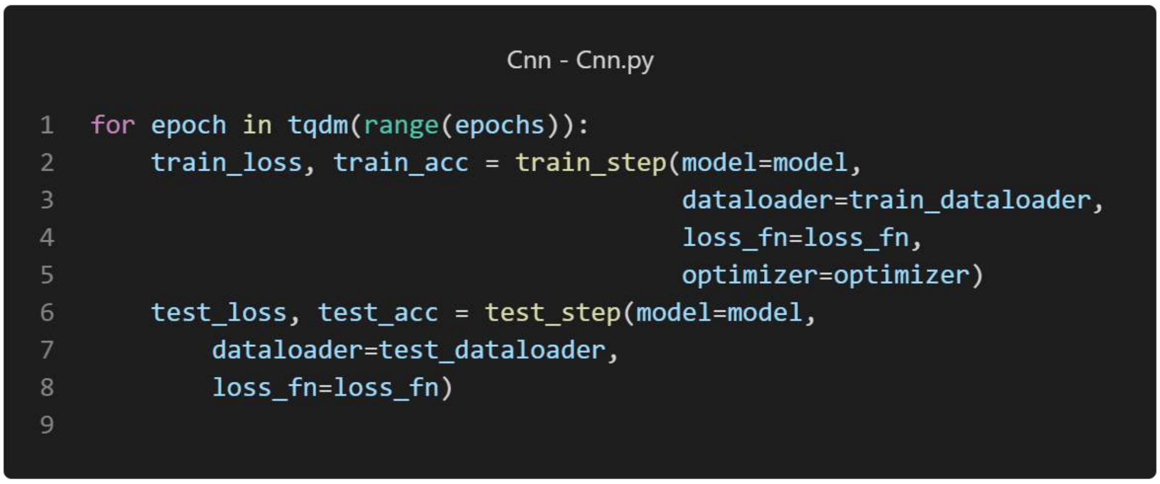
```python
class VGG(nn.Module):

    def __init__(self, input_shape: int, hidden_units: int, output_shape: int) -> None:
        super().__init__()
        self.conv_block_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.Conv2d(in_channels=hidden_units,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
        self.conv_block_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=hidden_units*16*16,
                      out_features=output_shape)
        )

    def forward(self, x: torch.Tensor):
        x = self.conv_block_1(x)
        # print(x.shape)
        x = self.conv_block_2(x)
        # print(x.shape)
        x = self.classifier(x)
        # print(x.shape)
        return x
```

Figure 53 CNNCode

As we can see here for conv block 1 there are two layer then it is followed by ReLU which is the actvation and then a max polling so as we can see in architecture above notice that we have 10 neurons in this layer but we only have 3 neurons on the previous layer in this architecture convolutional layers are fully conncted togher meaning each neuron is conncted to every other neuron in the prevoud layer focusing on the output of the topmost convoulation neuron from the first layer of it , and as we see we have 3 unique kernels when we hover over the activation map . so lets talk about Padding and Why It Is Required Padding simply means adding some zeros around an input matrix or an activation map before performing the convolution operation. This is of utmost importance in cases where the kernel (or filter) extends to be entirely outside of the input border so that the convolution operation can apply right up to the input edges.

Without padding, the spatial dimensions of the output for every convolutional layer are reduced in size. This information loss may occur more often at the borders, specifically. Padding helps preserve this information and keeps the spatial size of the output close to that of the input. Zero-Padding: A zero-padding adds several zeros to both sides of the input in a symmetric way. It is a straightforward, yet incredibly effective concept that was excessively adapted among the most successful CNN architectures like AlexNet. The main advantage is that the border effects of zero-padding can be perfectly preserved, and the input dimension can be preserved during the layers' construction, making it possible to build deeper networks. Impact of Kernel Size Kernel size is the hyperparameter that represents the size of the window used for convolution operations over the input. This parameter has a significant influence on the output, for it sets the amount of local information to be captured. In other words, small kernels allow the network to learn fine-grained local features better and, at the same time, reduce dimensionality less drastically than the use of large kernels. On the other hand, larger kernels better capture broader features but reduce spatial dimensions more rapidly, sometimes even at the cost of degraded performance, unless compensated for by other network design choices. Stride Stride dictates how many pixels the kernel moves across the input after each operation. A stride of one takes one pixel at a time and moves the kernel, thus capturing a lot of overlapping information from one region to another. This would allow more detailed feature extraction and higher dimensionality in the output, a great benefit in learning complex patterns. Larger strides will simplify the model and ease the computational requirement but at the cost of losing detailed information. Practical Example with Padding, Kernel, and Stride These would be defined in the convolutional layers while setting hyperparameters like padding, kernel size, and stride during the design of a CNN, say in your setup using PyTorch. For example, changing these parameters would affect the depth of the network, the receptive field size that it can learn, and, therefore, the performance in tasks of image classification, per se. (freecodecamp.org, 2022). And in the classifier we have the flatten which will bisaclly convert the 2D features maps into 1D factor followed by a linear layer.

3-Training loop


Cnn - Cnn.py

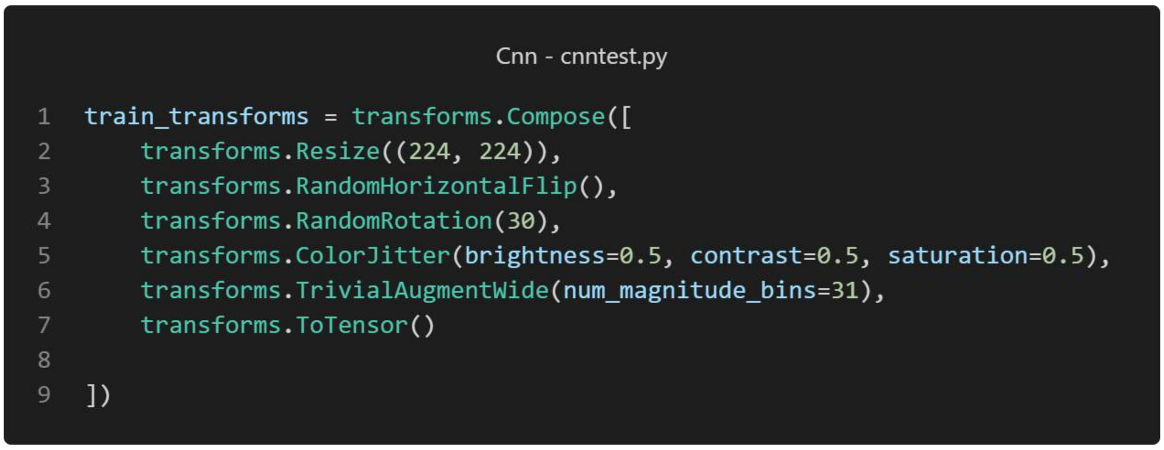
```python
for epoch in tqdm(range(epochs)):
    train_loss, train_acc = train_step(model=model,
                                       dataloader=train_dataloader,
                                       loss_fn=loss_fn,
                                       optimizer=optimizer)
    test_loss, test_acc = test_step(model=model,
        dataloader=test_dataloader,
        loss_fn=loss_fn)

```

Figure 54 training

Here as we can see we have training loop that will run based on the number of epochs

4-improve it

Based on number of test the values of epochs and hidden units have been set so we can get the most accurate predictions and as we can see in the next figure

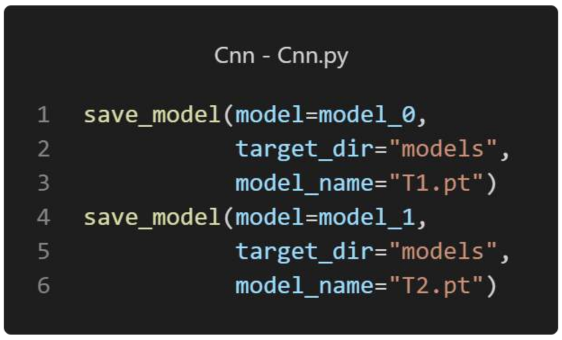

Cnn - cnntest.py

```python
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(30),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5),
    transforms.TrivialAugmentWide(num_magnitude_bins=31),
    transforms.ToTensor()

])
```

Figure 55 code3

As we can se it resize the image to 224,224 and to make the prediction better we also have used other finction like flipping the imges horzntal and then we also have random rotation from +30 to – 30 and then we have this function transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5) so it will work on diffrant light conditions and so it can work in different hand color and then this function transforms.TrivialAugmentWide(num_magnitude_bins=31) apply an arrugments with wide range of magnitudes and then the last function we turn it to tensor we we can test it and make predictions on it . and the code have been tested with different values of hiiden units and epochs to get the best accuracy and test loss.

5-savig the model

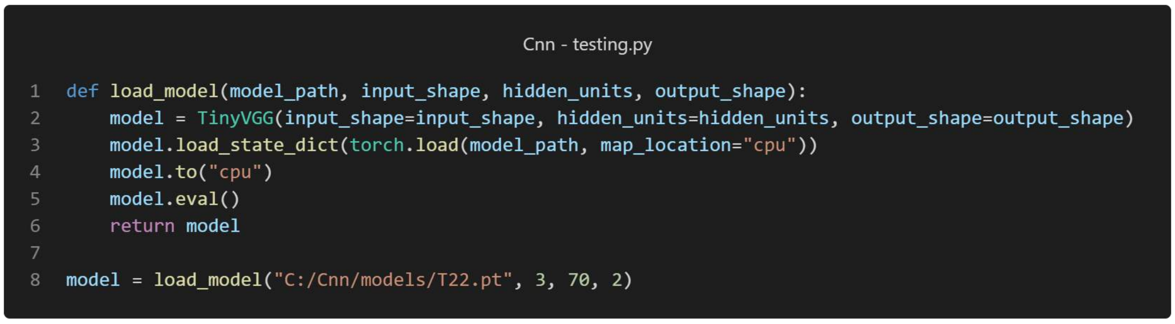


Cnn - Cnn.py

```python
save_model(model=model_0,
           target_dir="models",
           model_name="T1.pt")
save_model(model=model_1,
           target_dir="models",
           model_name="T2.pt")
```

Figure 56 CNNSaveModel

Here we save the model so we can test on it and improve it later on lets see when we have load the model and make predictions based on it



Cnn - testing.py

```python
def load_model(model_path, input_shape, hidden_units, output_shape):
    model = TinyVGG(input_shape=input_shape, hidden_units=hidden_units, output_shape=output_shape)
    model.load_state_dict(torch.load(model_path, map_location="cpu"))
    model.to("cpu")
    model.eval()
    return model

model = load_model("C:/Cnn/models/T22.pt", 3, 70, 2)
```

Figure 57 LoadModel

Here we have load the model so we can make predictions based on the loaded model or use real time cam and the reauslt will be shown in the next figure

Results

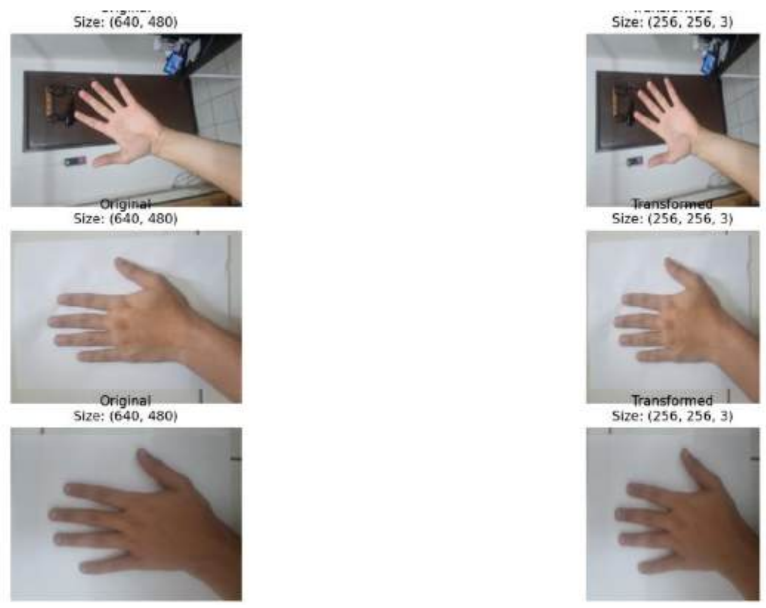


Figure 58 afterlayer

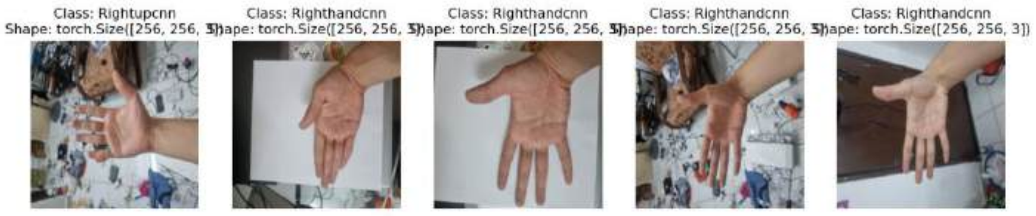

Class: Rightupcnn
Shape: torch.Size([256, 256, 3])
Class: Righthandcnn
Shape: torch.Size([256, 256, 3])
Class: Righthandcnn
Shape: torch.Size([256, 256, 3])
Class: Righthandcnn
Shape: torch.Size([256, 256, 3])
Class: Righthandcnn
Shape: torch.Size([256, 256, 3])

Figure 59 afterlayer1

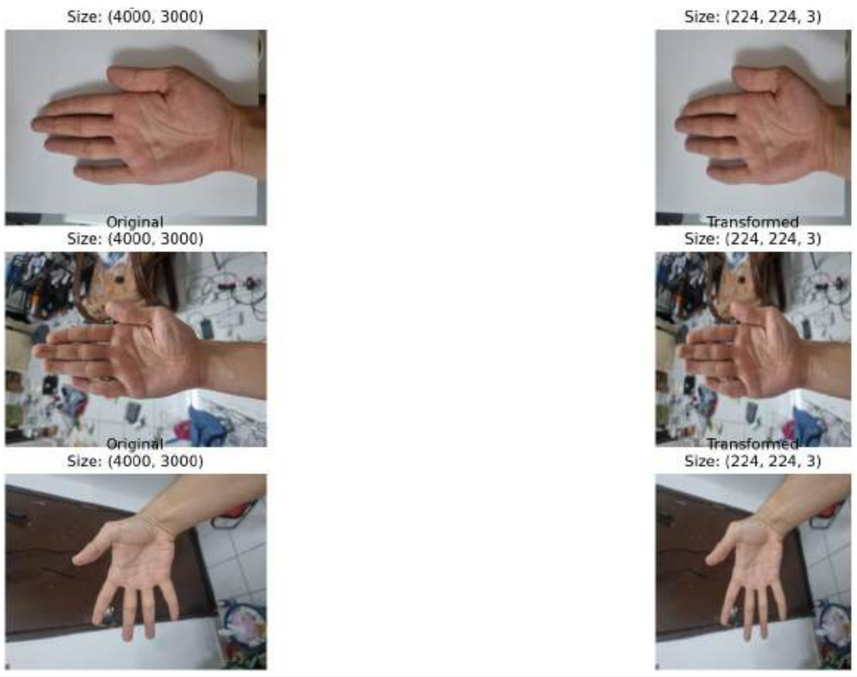
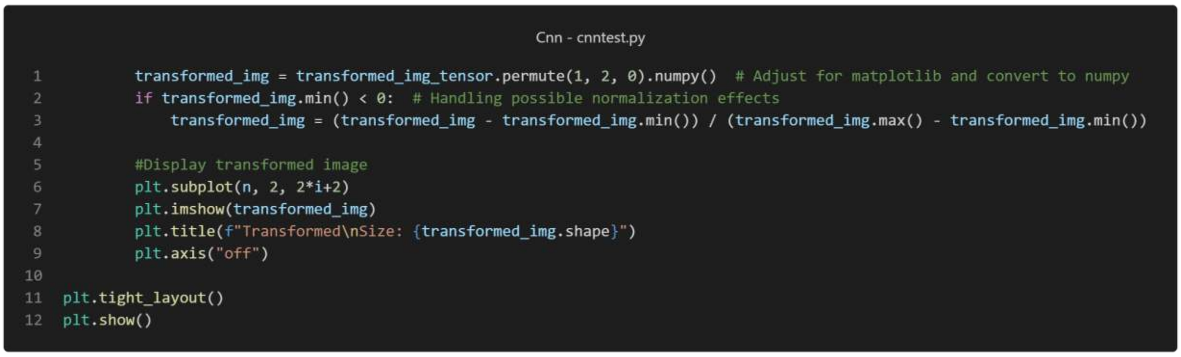


Figure 60 afterlayer2

There figure that have been shown above it shows the data after going through the function and the layers and it have been show using import matplotlib.pyplot as plt in the coed for example here in this code below


Cnn - cnntest.py

```
        transformed_img = transformed_img_tensor.permute(1, 2, 0).numpy()  # Adjust for matplotlib and convert to numpy
        if transformed_img.min() < 0:  # Handling possible normalization effects
            transformed_img = (transformed_img - transformed_img.min()) / (transformed_img.max() - transformed_img.min())

        #Display transformed image
        plt.subplot(n, 2, 2*i+2)
        plt.imshow(transformed_img)
        plt.title(f"Transformed\nSize: {transformed_img.shape}")
        plt.axis("off")

plt.tight_layout()
plt.show()
```

Figure 61 PlotImage

So as shown in the code above that we have shown some random images after going through the layers.

So, lets test our model now with the right and up hand with camera(live) and with uploaded images and see how the result will be.

Real Time :

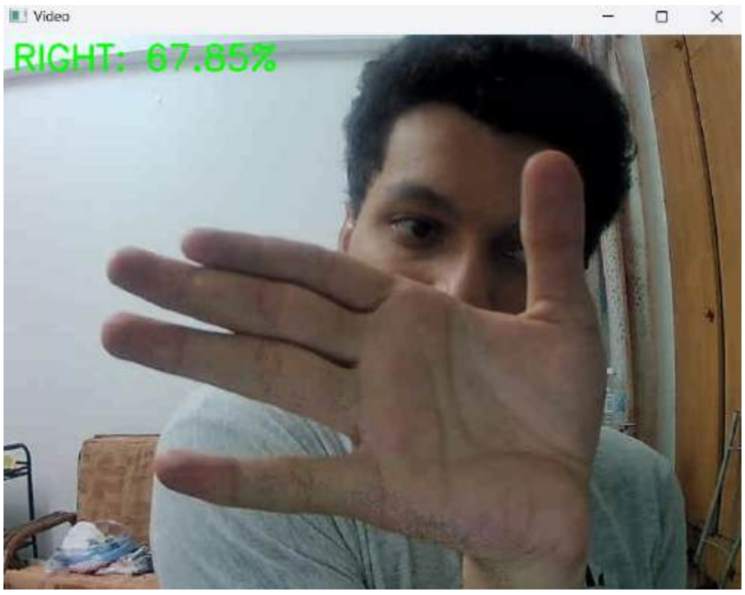


Figure 62 CameraRight

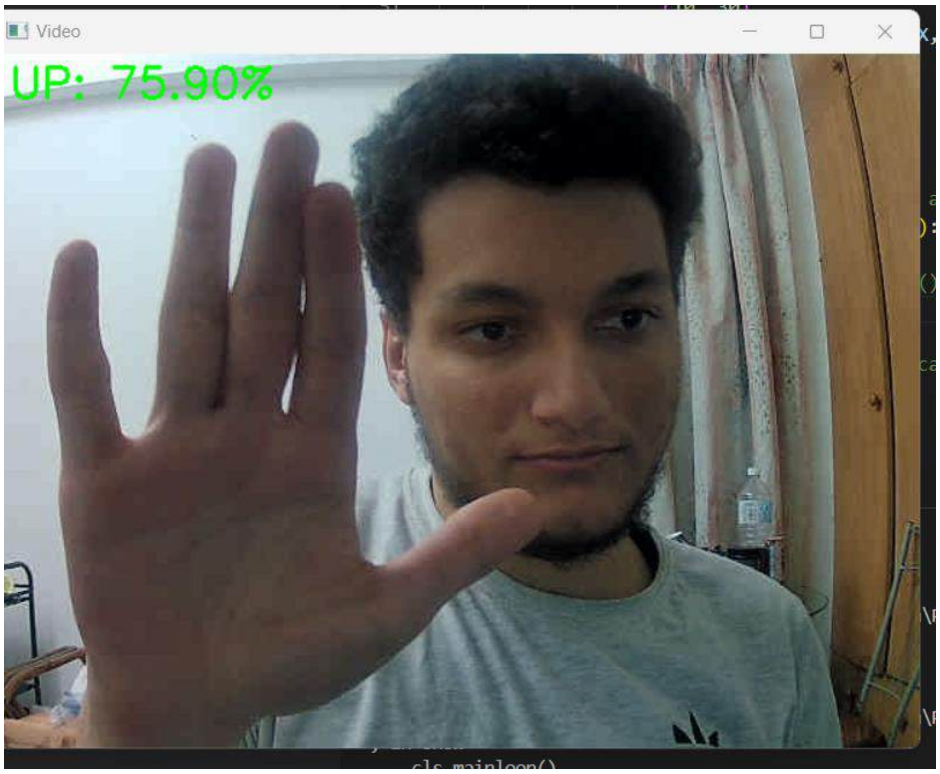


Figure 63 CameraUP

As we can see that the prediction are correct but we have low percentage some times and to resolve this we can add delay or sleep to improve our prediction now lets show the results for the uploaded images :

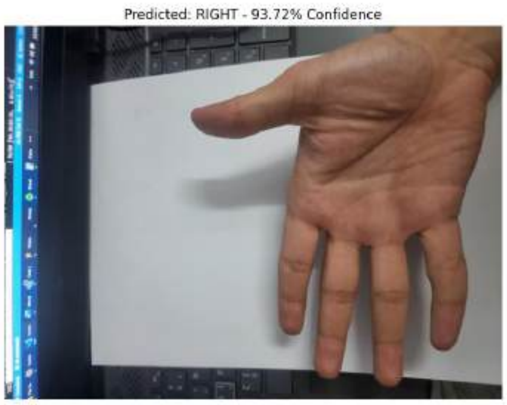


Figure 64 Rightupload

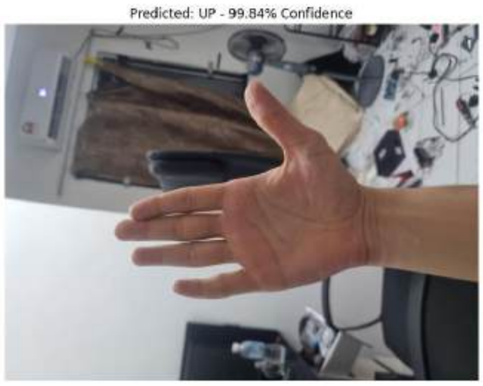


Figure 65 UPupload

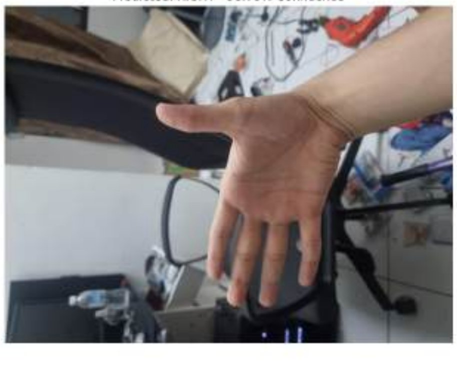


Figure 66 Rightupload1

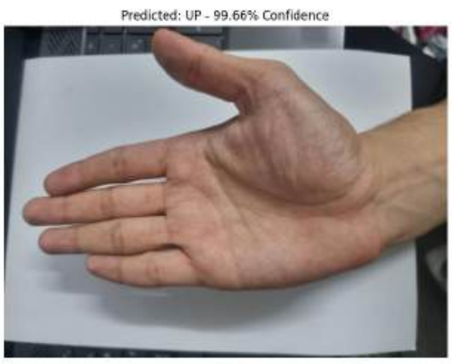


Figure 67 UP2

As we can see from the figure above that the uploaded images predictions are high with average of 95% and the reasons could be that the model is trained by images taken by the same phone that the test also or the high image quality because the quality of the real time camera is bad comparing to the phone one or one of the reasons that it could be it the Preprocessing or the frame rate and also it might be because of Aspect Ratio and Scaling so to solve this or test the theory we can train the model with some images token from the real time cam.

Discussion

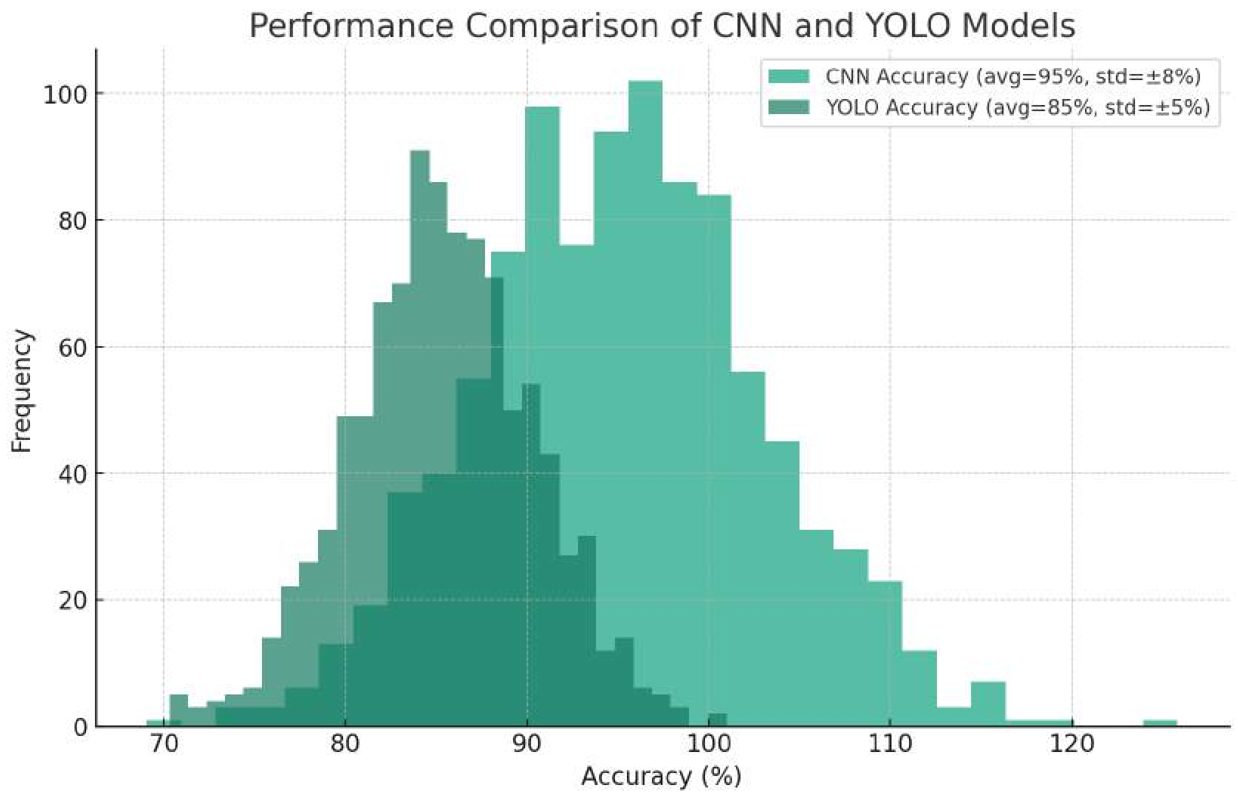


Figure 68 preformance

Architecture and design: the CNN from scratch appears to be a well thought- out design by drawing inspiration from biological processes and it intended to recognize and classify input images with high degree of accuracy and it uses a custom design which is less complex than the full VGG network so it will make it more suitable for simple applications. The model also includes essential steps such as data transformation augmentation and feature extraction tailored through convolutional pooling and fully connected layers.

Training and optimization: training involves a loop that adjust parameters over a number of epochs implying iterative refinement of the model for better accuracy and the model employs data augmentation strategies such as random flipping and rotation and also the color jittering to enhance it is ability to generalize from our training data.

Results and improvements: as shown in the figure above that the model has shown good performance with an average accuracy of 95% on uploaded images which mean that the training data might have bee similar or of higher quality than the images used for live testing. And the models real time performance with the live camera feeds was lower possibly due to lower image quality or other factors like for example frame rate or scaling. A potential are of improvement could be training the model with a dataset that include images from the live camera feed to enhance its performance in real time applications.

Comparing CNN and yolo performance by looking at the histogram we can see that the CNN have higher variability while the it also have high average accuracy for uploaded images and for the real time performance variability suggest that it need to improve the models robustness under different conditions.

Conclusion

Both models have their strengths and also weaknesses, the CNN model demonstrates high potential accuracy buy may benefit from further training on diverse dataset to reduce variability in real world application but for YOLO it offers more consistency which is crucial for the real time and also the dynamic environments so the choice between the two would depend on the specific requirements of the task like the quality and variability of the input data and also the computational resource available

OSAMA AHMAD MUSTAFA MOUSA (TP061567)

TASK 1

INTRODUCTION

Hand gesture recognition has become a very common usage in the AI space. Artificial intelligence can be used very to work with recognising hand gestures. This is key for understanding gestures correctly. Early research showed how important detecting hand direction is. The work of Oka et al. in 2002 and Ren et al. in 2013 highlighted this. Accurate hand direction makes gesture systems easier to use. Technology becomes more user-friendly this way. For this section of the report the process of using image processing techniques will be discussed to identify hand orientation such as Left hand, Right hand, palm and back.

OVERVIEW OF THE SYSTEM

The code below shows a system created purely from image processing techniques such OpenCV to create a program that will be able to recognise the hand orientation such as Left hand, Right hand, palm and back of the hand. The base language of the code is python and the Idle used is Vs code. Libraries like OpenCV, NumPy and imutils.

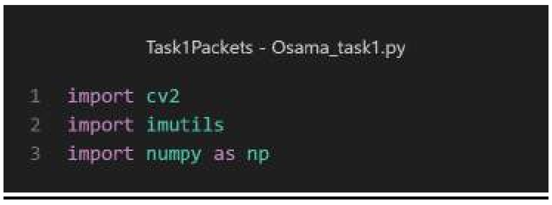
TECHNIQUE AND CODE EXPLAINATION

To create a successful working program the recognises the hand orientation certain Open cv techniques were used to accomplish the task. Below is a list of the few important techniques used:

1. **Background Subtraction**: Since background conditions can affect the output of the detection background subtraction was used to separate the hand from the background and then comparing the frames later to predict the recognitions.
2. **Image Thresholding:** once the frames are compared threshold function can be applied to further remove any other backgrounds or frame that might remain for the previous subtraction.
3. **Contour Detection**: once the binary images have been formed the system will start drawing the contours on the processed images join all the lines and then finding the largest contour and assuming that too be the hand.
4. **Convex Hull:** This method can be used on the segmented hand of the frame that was processed and can be used to determine properties like the hand orientation.

These are some of the OpenCV techniques used in the program. The section below will now explain the code of the Program.

Code Explanations


Task1Packets - Osama_task1.py

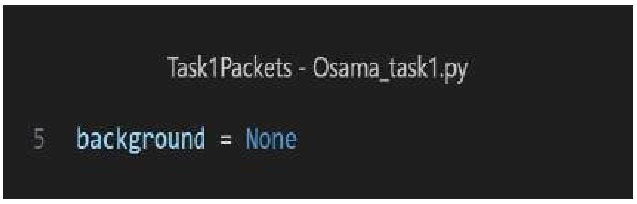
```python
import cv2
import imutils
import numpy as np
```

Figure 69: Import libraries_Osama

<u>Import Libraries</u>

Three crucial libraries were implemented as shown in figure 1:

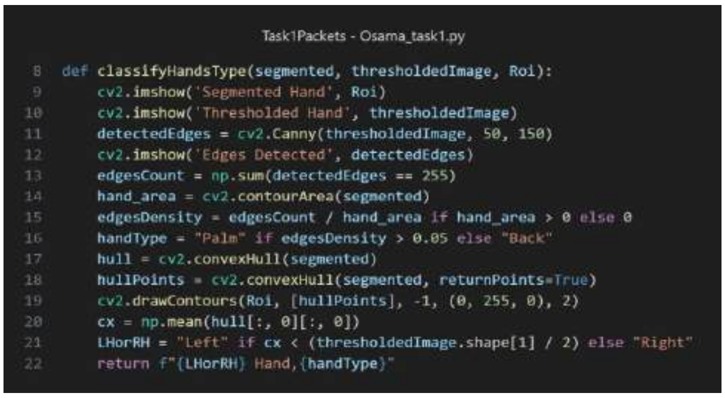
1. **Python OpenCV**: This library represents a major tool in image processing that can capture and analyze information from a feed, the image that can be directly detected from the system.
2. **Numpy**: This library is essential when it comes to high performance of the image, feed, video capture through math computing in computer vision.
3. **Imutils:** It's a powerful library that can be managed in image processing to deal with rotation, translation, etc.


Task1Packets - Osama_task1.py

```python
background = None
```

Figure 70: background global

- This global variable is implemented to deal with an isolation of the hand only so that the system creates a background where the han is isolated.


Task1Packets - Osama_task1.py

```python
def classifyHandsType(segmented, thresholdedImage, Roi):
    cv2.imshow('Segmented Hand', Roi)
    cv2.imshow('Thresholded Hand', thresholdedImage)
    detectedEdges = cv2.Canny(thresholdedImage, 50, 150)
    cv2.imshow('Edges Detected', detectedEdges)
    edgesCount = np.sum(detectedEdges == 255)
    hand_area = cv2.contourArea(segmented)
    edgesDensity = edgesCount / hand_area if hand_area > 0 else 0
    handType = "Palm" if edgesDensity > 0.05 else "Back"
    hull = cv2.convexHull(segmented)
    hullPoints = cv2.convexHull(segmented, returnPoints=True)
    cv2.drawContours(Roi, [hullPoints], -1, (0, 255, 0), 2)
    cx = np.mean(hull[:, 0][:, 0])
    LHorRH = "Left" if cx < (thresholdedImage.shape[1] / 2) else "Right"
    return f"{LHorRH} Hand,{handType}"
```

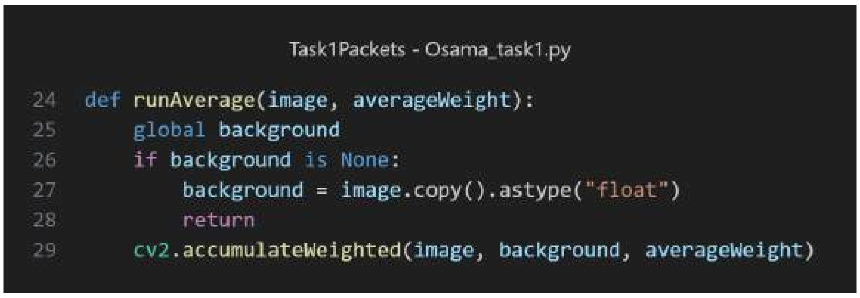
Figure 71: ClassifyHandsType

ClassifyHandsType function

- In the first part of the function, two images will be displayed according to the region of interest and the thresholded image (Show Images).

- For the second part, there is a calculation made by edgesCount which takes the Canny from the detected edge, so that it can perform a mathematical calculation of the edges according to the area where the hand is placed or the contour area.

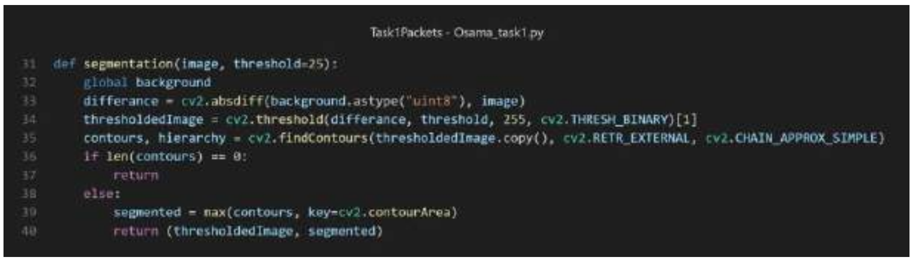
- In the end, there is hand classification which is based on the position of the hand whether it is centered in the frame. Besides that, if the edgesDensity is higher than 0.5, the system will be able to detect palm, otherwise is going to be back.


Task1Packets - Osama_task1.py

```python
def runAverage(image, averageWeight):
    global background
    if background is None:
        background = image.copy().astype("float")
        return
    cv2.accumulateWeighted(image, background, averageWeight)
```

Figure 72: runAverage function

- In this function, we pass two arguments which are image, and averageWeight which is going to be updated through the global variable background, so that the system can identify the frames across the hand.

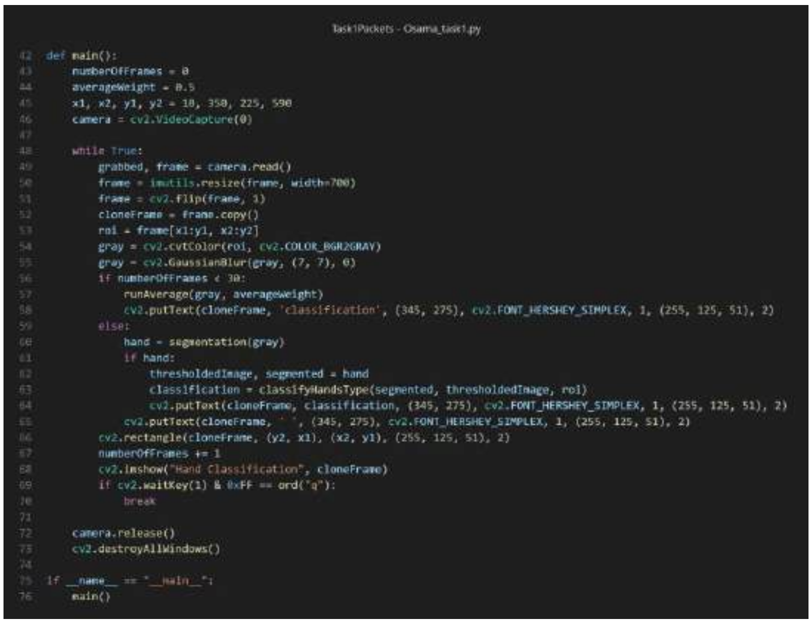

Task1Packets - Osama_task1.py

```python
def segmentation(image, threshold=25):
    global background
    differance = cv2.absdiff(background.astype("uint8"), image)
    thresholdedImage = cv2.threshold(differance, threshold, 255, cv2.THRESH_BINARY)[1]
    contours, hierarchy = cv2.findContours(thresholdedImage.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(contours) == 0:
        return
    else:
        segmented = max(contours, key=cv2.contourArea)
        return (thresholdedImage, segmented)
```

Figure 73: segmentation function

Segmentation function

- The function for segmentation is where the image processing segmentation takes place, and some operations are performed such as getting the absolute difference between the frame and the background, then applying a threshold, as well as finding contours, so that the system can identify the hand more efficiently.



Task1Packets - Osama_task1.py

```python
def main():
    numberOfFrames = 0
    averageWeight = 0.5
    x1, x2, y1, y2 = 10, 350, 225, 590
    camera = cv2.VideoCapture(0)

    while True:
        grabbed, frame = camera.read()
        frame = imutils.resize(frame, width=700)
        frame = cv2.flip(frame, 1)
        cloneFrame = frame.copy()
        roi = frame[x1:y1, x2:y2]
        gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
        gray = cv2.GaussianBlur(gray, (7, 7), 0)
        if numberOfFrames < 30:
            runAverage(gray, averageWeight)
            cv2.putText(cloneFrame, 'classification', (345, 275), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 125, 51), 2)
        else:
            hand = segmentation(gray)
            if hand:
                thresholdedImage, segmented = hand
                classification = classifyHandsType(segmented, thresholdedImage, roi)
                cv2.putText(cloneFrame, classification, (345, 275), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 125, 51), 2)
            cv2.putText(cloneFrame, ' ', (345, 275), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 125, 51), 2)
        cv2.rectangle(cloneFrame, (y2, x1), (x2, y1), (255, 125, 51), 2)
        numberOfFrames += 1
        cv2.imshow("Hand Classification", cloneFrame)
        if cv2.waitKey(1) & 0xFF == ord("q"):
            break

    camera.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

Figure 74: Main function

<u>Main function</u>

- In the first section, the images have been encapsulated in such a way that the frame where the hand is going to be placed should be able to handle operations such as flipping, converting the colour from binary to Gray, and the application of Gaussian blur. Besides that, the angles are also considered based on the position of the hand.

- The flow starts by capturing the feed during the transmission, after that with the implementation of the imutils library, the hand becomes resized with a width that is set to 700 pixels. Besides that, the feed capture is flipped, and then we apply conversion such as gray color within the region of interest, as well as applying blur so that the system can only focus on the hand placed in the frame and ignore the rest of the background.

RESULTS

_Right hand Palm

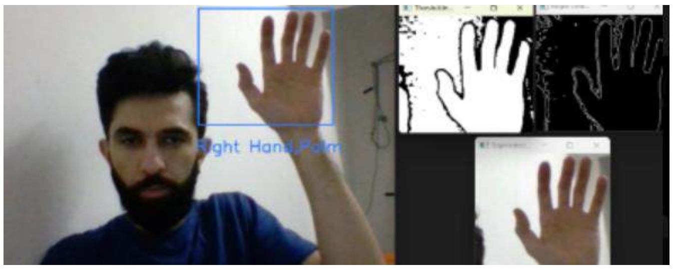


Figure 75: Right Hand Palm

- In figure 7, the recognition of a right hand with the palm open towards the camera. In the same picture, the detected hand is drawn in a blue-colored rectangle, hence showing the region of interest. These auxiliary windows present the thresholded binary image from left to right in which the hand is segmented from the background, the highlighted outlines of the fingers and the palm detected as edges, and the segmented hand with noise reduced. This partition gives the system the capability of categorizing hand orientation with high accuracy.

Right Hand Back

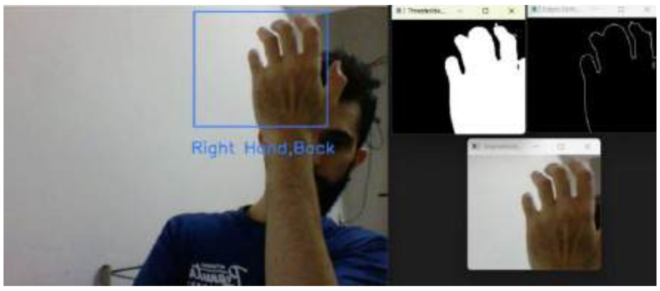


Figure 76: Right Hand Back

- Proceeding with figure 8, shows the right hand in the rearview, and accordingly, this area of interest is marked in it, as has been done in the previous figure. The thresholded and edge-detected images are pictures of the back of the hand. The distinct edge patterns and ratios of areas for the hand are different from those used for the palm, but the classification uses the back for the orientation of the hand.

Left Hand Palm

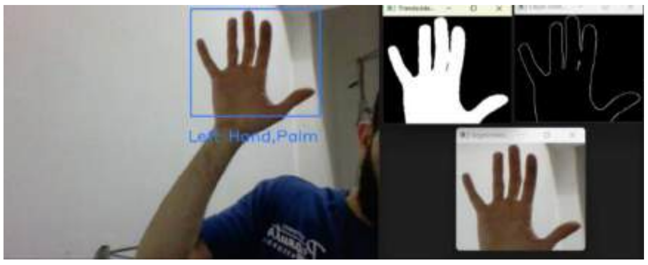


Figure 77: Left Hand Palm

- Left Hand, Palm This is the system recognition of the left-hand palm. It is identified using the same set of techniques, just that it is recognized by the hand's relative position against the center of the image and lines from the palm and fingerprint from the edge patterns formed by them in the processed

Left Hand Back

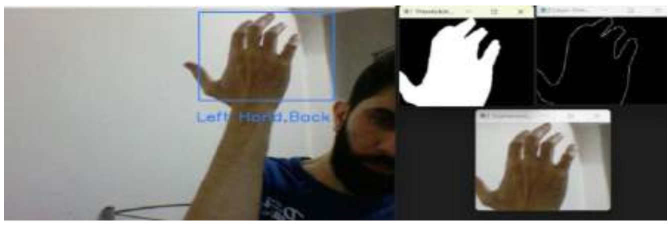


Figure 78: Left Hand Back

- The image shows the back of the left hand. Even though it is another hand, the system carries on with the same processing, consistently recognizing and outlining the back of the hand, showing the model's ability to generalize among the different orientations of the hands.

DISCUSSION

Challenges and limitations: One of the challenges faced is the occasional confusion in detection between the right and the left hand which could be a direct link to the background and lighting conditions. Another challenge faced that was also due to background and lighting conditions was the detection of palm and back where due to constants change in lighting the system could get confused in detecting the gesture. Overall, it could be said that the limitation of the system is the background.

Approach and Comparative Analysis: The approach of the system is like traditional image processing techniques and gesture classifications which would be inaccurate in comparison to current methods like deep learning and the use of artificial intelligence. Its old traditional method may not achieve the same accuracy as the new current methods, but it gets the job done.

Technical/Performance: Overall the system was able to successfully pick up the hand orientation as evident from the results shown above. It can be said that the system works very well in fixed setting as explained in the challenges and limitations where background and lighting was a limitation.

Improvements and Alternatives: Is exploring with different background and having a real time tuning where the system automatically tunes itself to ignore the backgrounds. Then exploring with different libraries like media pipe which uses skeleton features to detect the back and the pam of the hand. Or implementing different algorithms for each of the hand gesture.

TASK 2

INTRODUCTION

This section of the report is about task 2 and the entire process involved. The demand for task 2 is to develop an AI model that can be able to predict given a training dataset. The successful AI model that was chosen as the trainable model for task 2 is yolo V8. In this task 2, the trainable model will be applied in predicting the hand orientation, a model that detects the palm, back and whether it is right hand left hand. Before the overview of task 2 let's discuss yolo.

Yolo is You Only Look Once, or YOLO—and that what makes YOLO special is that that it's optimized such that it learns to look for things only once. YOLO is a real-time object detection algorithm that has changed the computer vision game by scaling whole image classification and object localization into one single neural network. Normal object detection might include using a technique like region proposal followed by some convolutional neural network that will classify the object. But YOLO takes an entirely different approach by treating object detection as a regression problem. Rather than recommend regions, it passes the image through a neural network and guesses the output. It splits the image into a grid and aims to predict the bounding boxes and probabilities from these grid cells. This approach allows YOLO to be incredibly fast making it suited for real-time performance.

YOLO v8 is the current version of YOLO object detection method. YOLO is continuously improved by integrating advanced features ranging from better accuracy, faster and more efficient, optimisation, and many more. YOLO v8 gets better with the adoption of the more advanced neural network architectures and trains the network to improve real-time object detection possibilities.

Overview Of Task 2

The whole of task 2 consist of three main steps which will be explained in detail in the report. The steps consist of data collection, annotation, training and then testing. So, the data is in form of images. Then the image is annotated which then later used to train the model which is Yolo V8. Then the model trained is then used to test to see if the prediction is correct. So next part will be explaining the process from data collection till the model being trained until the results from the testing.

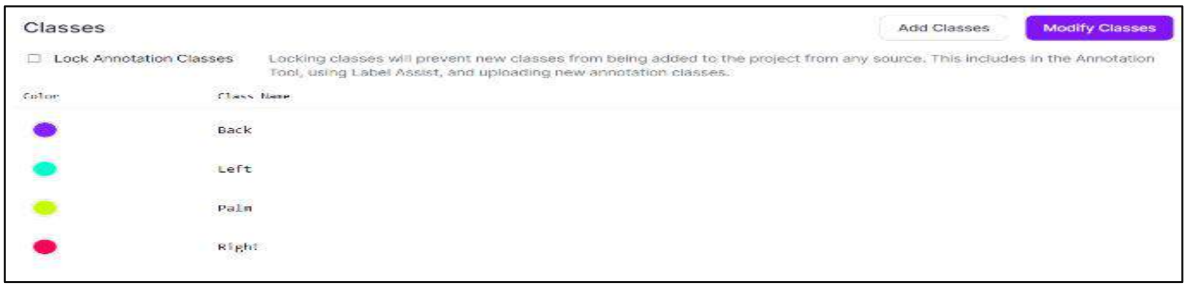
Process of Roboflow

On the other hand, Roboflow is an application for developer tools for building and deploying models for machine vision applications. Roboflow helps to implement the full machine vision pipeline: from data collection to pre-processing, annotation, training models, and deploying models. Key features of Roboflow include a user interface to annotate and preprocess in bulk, where users can load images and then label these images in bulk. Preprocessing can be done over the images for the respective models. Supported preprocessing tasks include resizing, normalizing, and augmenting to enable performance and robustness enhancements of the model.

Implementation of Hand Orientation Using Roboflow:

Firstly, before the annotations were done data was collected. Data were in form of images. for the hand orientation images were taken with different background and different lighting to make the model more accurate and precise which will be shown in the graphs later. After the image was taken roboflow was used to annotate all the images. the process is explained below.

So firstly, the image is all added to the roboflow website. For the hand orientation class over 1600 plus images were taken. Next is creating classes.



Classes

☐ Lock Annotation Classes — Locking classes will prevent new classes from being added to the project from any source. This includes in the Annotation Tool, using Label Assist, and uploading new annotation classes.

Color	Class Name
●	Back
●	Left
●	Palm
●	Right

Figure 79 Class Creation for Hand Orientation

As shown in the figure above the 4 classes were created for the hand orientation task. This class consists of the Back, palm, Left, and right of the hand. Later using these classes, the annotations will be done.

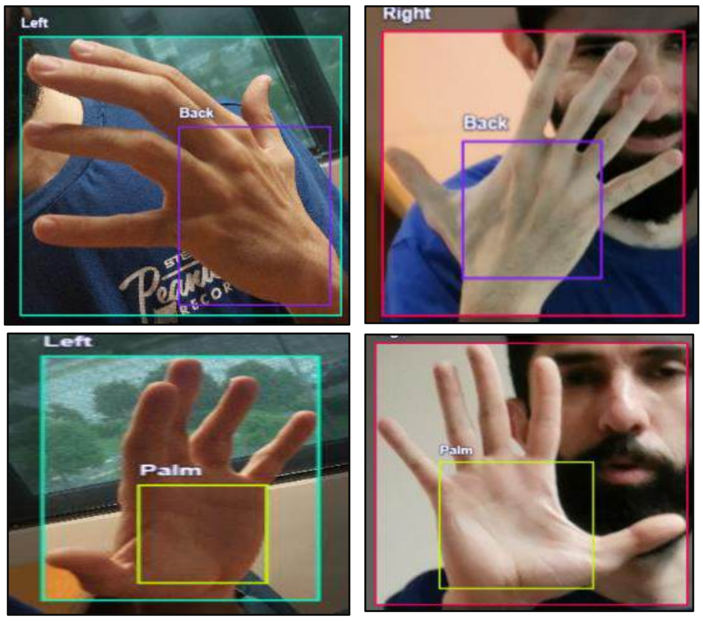


Figure 80:Annotation's Process

The images are then labelled according to the hand orientation. As shown in the figure above for each orientation there is a colour code ranging from blue coloured bounding box which is too indicate the left hand and yellow colour bounding box to indicate the palm of the hand. The hand localization is done in a bounding box that identifies the orientation of the hand by drawing another bounding box between the palm and the back, defining paths of the hand. Different colours identify the classes of the paths.

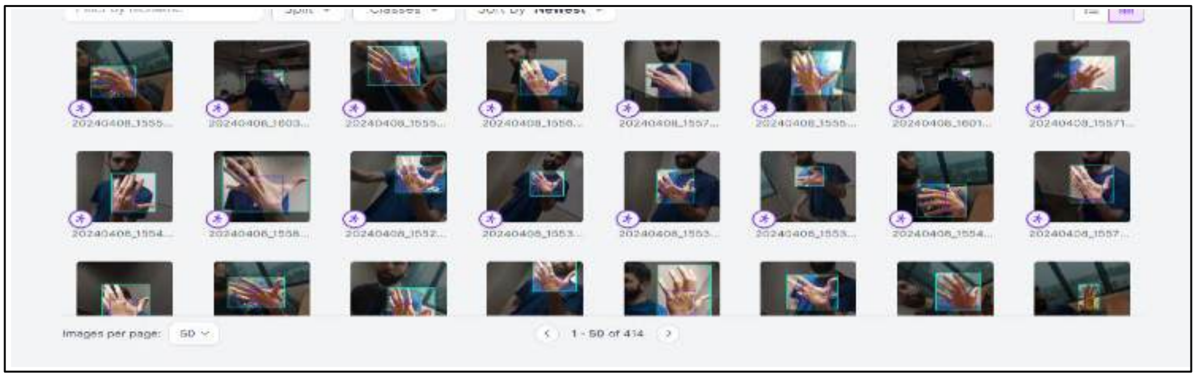


Figure 81:Annotaetd Images

Then once all the images have been annotated it is then added to the dataset as shown in the figure above. Once the all the annotated images are added to the dataset they are ready to be exported to be used for training. After the labelling of all the pictures, they were exported in YOLO v8 format—a new framework that allows making very fast and accurate object detection. "YOLO" comes from "You Only Look Once." This can well perform any real-time requirements task it is exposed to, for example, the recognition of hand gestures. This is suitable for dynamic systems that are said to act on user inputs with immediate effect.

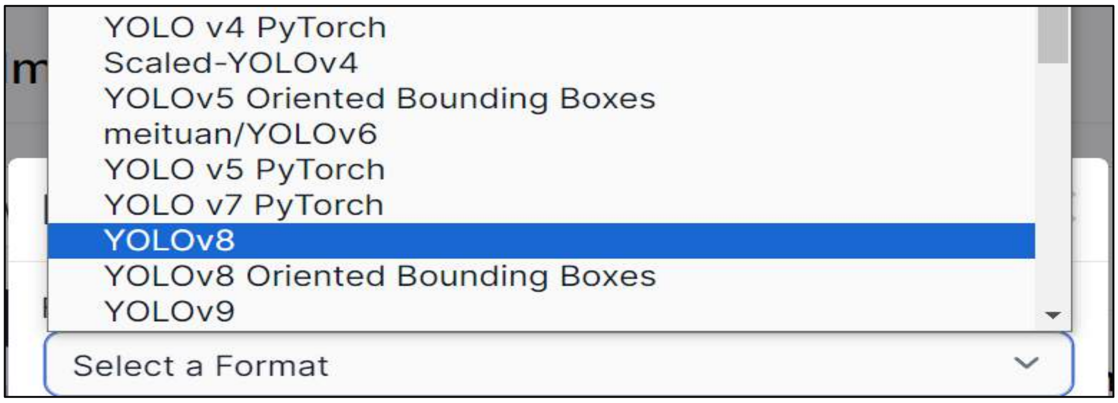
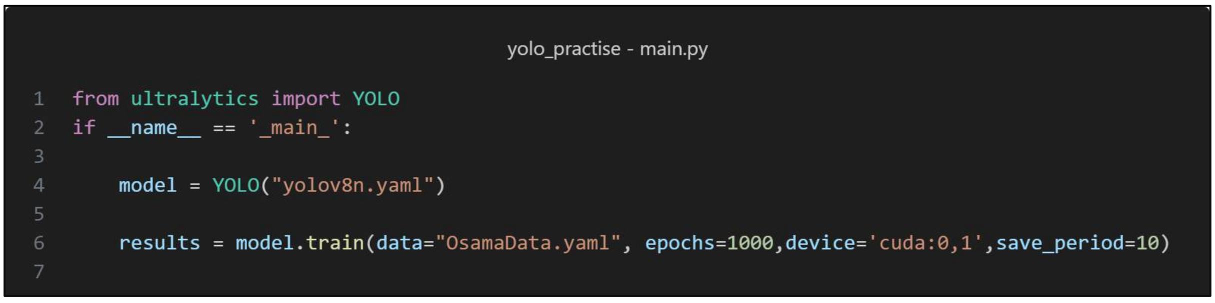


Figure 82 Export dataset

The workflow from Roboflow significantly simplified preparation for training data. It ensured that the training data for a model to recognize hand gestures well were readied in the correct format. This forms very critical preparation to create a reliable system with real uses in everyday life in many fields. Preparation sets up model training and evaluation for efficient, accurate gesture recognition. Once the images are all exported it can then be used for training the model.

Model Training/Yaml File

Basically, after exporting the file or data to yolov8, the training section is the next step which is applied by following the code below with the next functionalities:

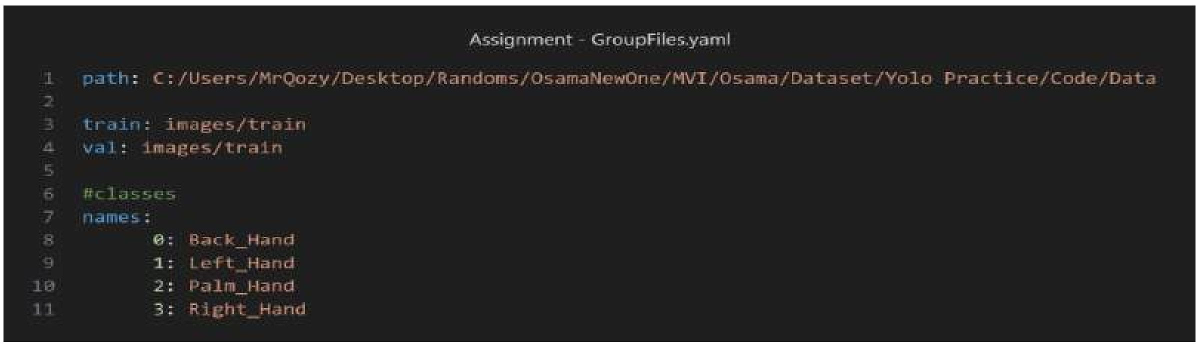


yolo_practise - main.py

```python
from ultralytics import YOLO
if __name__ == '_main_':

    model = YOLO("yolov8n.yaml")

    results = model.train(data="OsamaData.yaml", epochs=1000,device='cuda:0,1',save_period=10)

```

Figure 83:Training File osama

1. The ultralytics package holds the YOLO class, imported here. This organization maintains YOLO (You Only Look Once) models - powerful object detectors.
2. This condition verifies if the script runs directly rather than importing as a module. The code inside executes if it's the primary program.
3. An instance of YOLO sets up based on yolov8n.yaml's architecture and configurations.
4. Training commences with specified settings: OsamaData.yaml dataset configs, 1000 epochs (cycles), using first CUDA GPU, and saving weights every 10 epochs.
5. OsamaData.yaml informs the model about dataset setups.
6. 1000 epochs allotted for thorough training iterations.
7. The first available CUDA GPU accelerates computations.



Assignment - GroupFiles.yaml

```yaml
path: C:/Users/MrQozy/Desktop/Randoms/OsamaNewOne/MVI/Osama/Dataset/Yolo Practice/Code/Data

train: images/train
val: images/train

#classes
names:
      0: Back_Hand
      1: Left_Hand
      2: Palm_Hand
      3: Right_Hand
```

Figure 84 yaml File

1. path: The directory path where training and validation data reside is specified.
2. train: images/train: This relative path defines where the training images are located from the root - inside a train folder within an images directory.

3. val: images/train: The validation images share the same path as training images, an atypical setup that risks overfitting the machine learning model to the dataset.
4. names: Each class the model should detect is listed with an index under names. These classes represent different hand positions or gestures the model will identify. Example (0: Back_Hand: Identifying the back of a hand)

OSAMA MOUSA MOUSTAPHA: GRAPH RESULTS

The newly set graphs provided about the performance of the hand orientation model give a bit more insight into how much the model can effectively discriminate between the classes (right hand back/palm, left hand back/palm) concerning the confidence it has in those predictions.

Train/box_loss & val/box_loss:

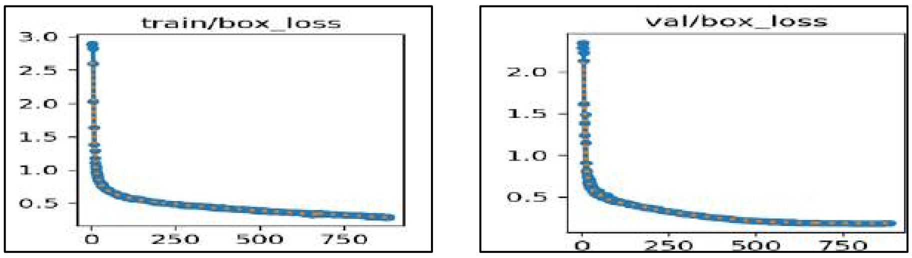


Figure 85:train and box val

1. These two graphs visualize the bounding box regression loss for the training and validation set, respectively.
2. The box loss is a measure of how well the model can predict the box boundary positions around the hands.
3. The quick drop indicates rapid learning, and as the epochs increase, the rapid drop in the value suggests the model is localizing the hands better in the images.

Train/cls_loss & val/cls_loss:

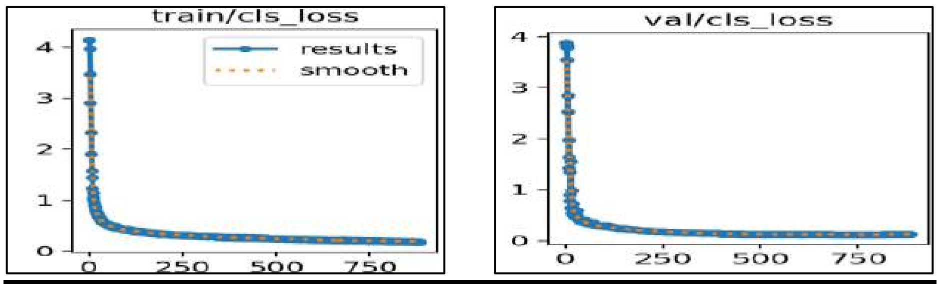


Figure 86 Train and Val cls loss

1. Classification loss in the training and validation set. A measure of how accurate our model is at finding out if a given bounding box contains the right hand (back and palm) or the left hand (back and palm).
2. Steep decreases and low values for loss at the end of training suggest high accuracy in classification.

Train/dfl_loss and val/dfl_loss:

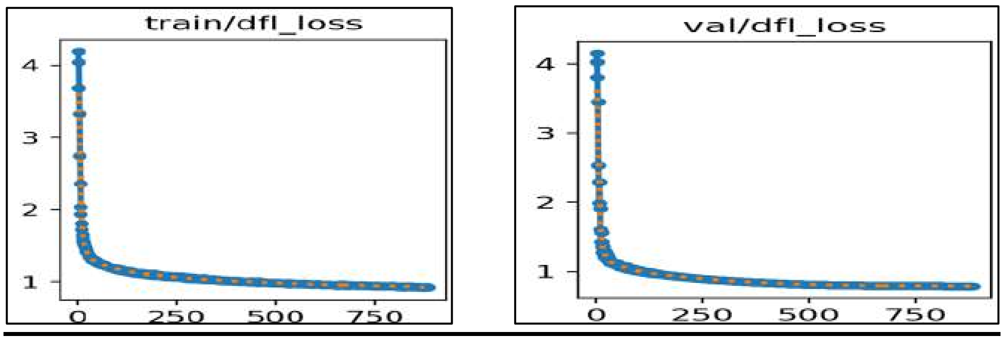


Figure 87train and val dfl loss

1. These would represent the Direction Focused Losses and could be facing the hand's orientation (back or palm).
2. It shows how well the model predicts the orientation of the hands. The loss is decreasing very sharply in both cases of training and validation, which strongly indicates that the model is performing the very best on this task.

Metrics/precision(B) and Metrics/mAP50(B):

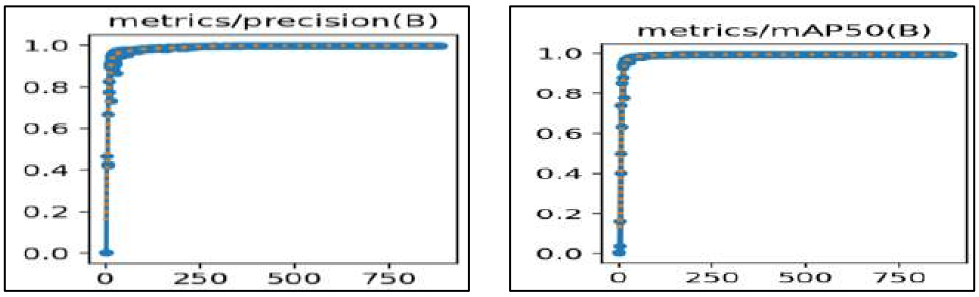


Figure 88:Precision metrics

1. Metrics/precision(B): Precision is the ratio of true positive to the sum of true positive and false positive. High precision relates to low false positive rate. The graph gives a precision close to 1,which is very good because it means that the model is very accurate in its predictions regarding the right-hand orientation when it says it has detected one.
2. Metrics/mAP50(B): It represents graph mean Average Precision (mAP50) at 50% IoU threshold. Average precision calculates the average precision value for recall value over 0 to 1. It's one of the most popular metrics for evaluating object detection models.
3. Around 1 is a good value, indicating that with high accuracy the model can detect the orientations of the hand when the IoU threshold is at 50%. The graph resembles the mAP50, considering the mean Average Precision but throughout several IoU thresholds, specifically from 0.5 to 0.95 (increment by 0.05).

Metrics/recall(B) and Metrics/mAP50-95(B):

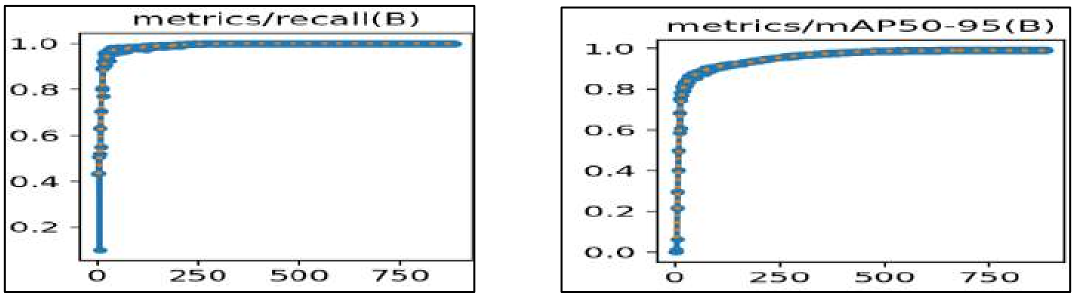


Figure 89:recall metrics

1. metrics/recall(B): The graph illustrates a sharp rise in the recall right from the outset, indicating that most relevant hand gestures are identified, and the false negatives are very low. After the first jump, the recall value plateaus to around 1.0 from where more data do not significantly change the recall. High recall near 1.0 across the graph indicates excellent performance in detecting the right-hand back/palm and left-hand back/palm gestures.
2. metrics/mAP50-95(B): It also shows that there is a quick slope towards the high mAP values, showing that the model is very accurate in the classification of the hand gestures at different IoU thresholds. The mAP near 1.0 over many IoU thresholds means that the predicted bounding boxes of the model are quasi-exact with high precision for ground truth.

F1 Confidence Curve:

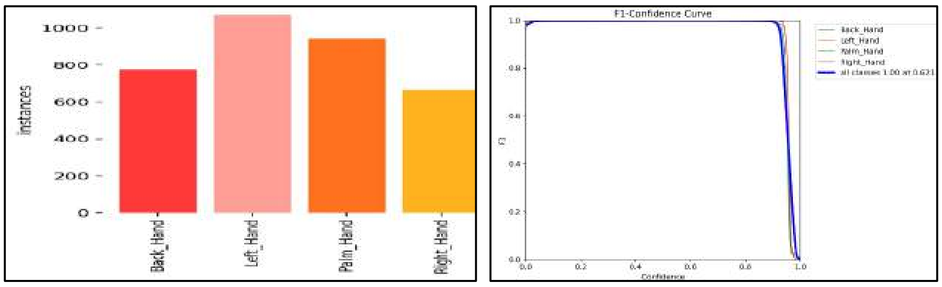
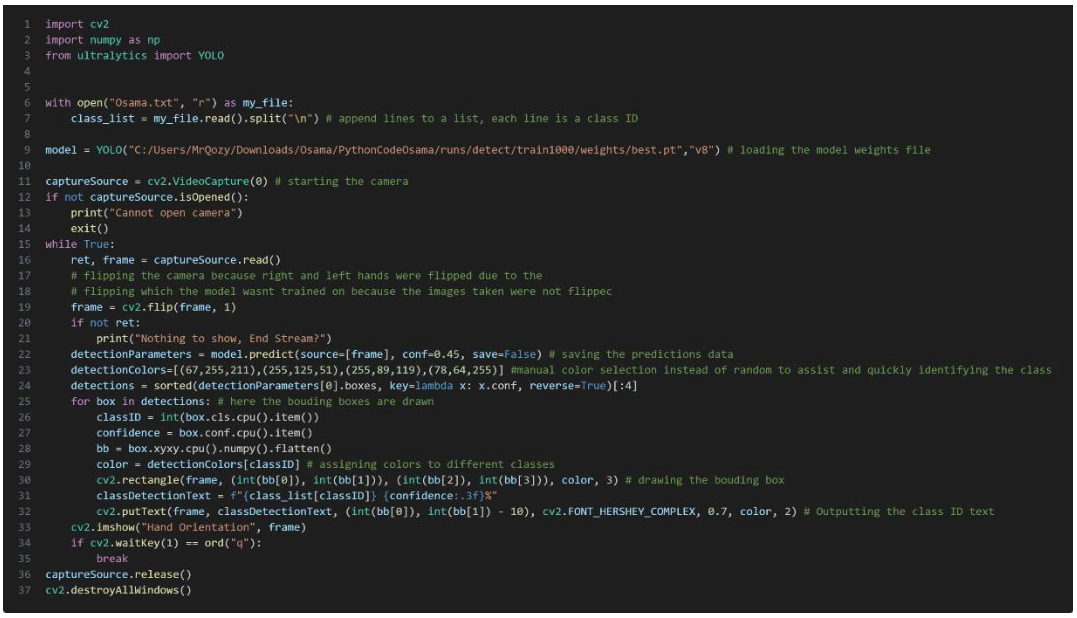


Figure 90:fdsfds

1. The F1-Confidence curve shows a sharp rise with very high classifier accuracy, displaying high F1 scores. This is seen from the steep F1 curve and high F1 scores that commence even with small confidence thresholds.
2. The graph of the Instances Distribution shows that the dataset is in a different number of instances belonging to each category, being dominant Left_Hand.
3. Such an imbalance could give rise to classifier performance problems requiring oversampling, under sampling, or synthetic data augmentation techniques to neutralize bias.
4. The high performance on the part of the classifier in the F1-Confidence Curve would suggest that the classifier might have done well in handling these dataset imbalances; however, further scrutiny of the training methodology is required to understand in detail.

TESTING CODE



```python
import cv2
import numpy as np
from ultralytics import YOLO


with open("Osama.txt", "r") as my_file:
    class_list = my_file.read().split("\n") # append lines to a list, each line is a class ID

model = YOLO("C:/Users/MrQozy/Downloads/Osama/PythonCodeOsama/runs/detect/train1000/weights/best.pt","v8") # loading the model weights file

captureSource = cv2.VideoCapture(0) # starting the camera
if not captureSource.isOpened():
    print("Cannot open camera")
    exit()
while True:
    ret, frame = captureSource.read()
    # flipping the camera because right and left hands were flipped due to the
    # flipping which the model wasnt trained on because the images taken were not flippec
    frame = cv2.flip(frame, 1)
    if not ret:
        print("Nothing to show, End Stream?")
    detectionParameters = model.predict(source=[frame], conf=0.45, save=False) # saving the predictions data
    detectionColors=[(67,255,211),(255,125,51),(255,89,119),(78,64,255)] #manual color selection instead of random to assist and quickly identifying the class
    detections = sorted(detectionParameters[0].boxes, key=lambda x: x.conf, reverse=True)[:4]
    for box in detections: # here the bouding boxes are drawn
        classID = int(box.cls.cpu().item())
        confidence = box.conf.cpu().item()
        bb = box.xyxy.cpu().numpy().flatten()
        color = detectionColors[classID] # assigning colors to different classes
        cv2.rectangle(frame, (int(bb[0]), int(bb[1])), (int(bb[2]), int(bb[3])), color, 3) # drawing the bouding box
        classDetectionText = f"{class_list[classID]} {confidence:.3f}%"
        cv2.putText(frame, classDetectionText, (int(bb[0]), int(bb[1]) - 10), cv2.FONT_HERSHEY_COMPLEX, 0.7, color, 2) # Outputting the class ID text
    cv2.imshow("Hand Orientation", frame)
    if cv2.waitKey(1) == ord("q"):
        break
captureSource.release()
cv2.destroyAllWindows()
```

Figure 91Testing code

1. **Libraries: OpenCV was used to access the camera and feed the live prediction to the camera live. the ultralytics library is used to access the trained model and asking the model to predict.**
2. **Class_List: This appends all the class id for each gesture into the list. The class id is the first values in the textile. This value determines the gesture. For example: If it 0 it will be Right_hand or if it is 2 it is palm of the hand.**
3. **Cv2.flip: This was implemented to prevent confusion when testing. Since the images were flipped the right hand becomes the left hand and vice versa. So, this was use to flip the images back to original.**
4. **Bounding Box colour: instead of putting multiple colours bounding boxes that are generated at random each time the program runs, setting the bounding box to a fixed colour. This can also be used as an identification for the classes. So, if it a yellow bounding box the object detected is the palm.**
5. **Draw Bounding Boxes and Labels: For selected detections, retrieves class ID and confidence level, converts bounding box coordinates, and draws them with predefined colours.**

RESULTS

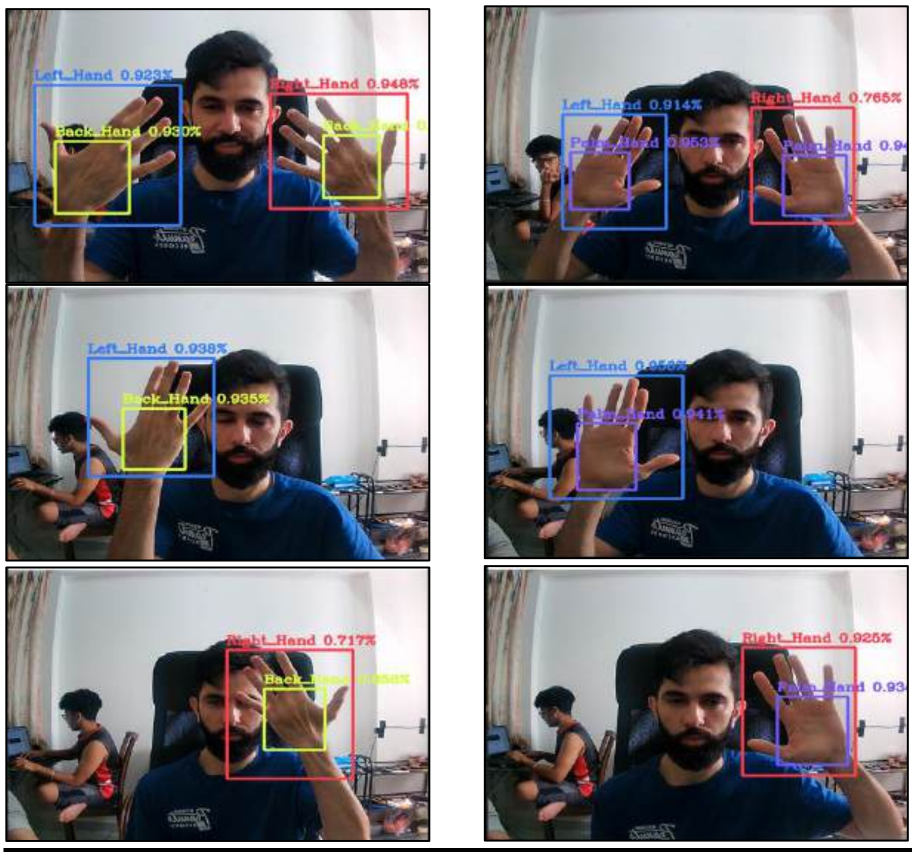


Figure 92:Results

The images above are from testing the model. Above each bonding box for each image is the confidence level of the detection:

Left Hand: 0.914 to 0.958 or 91.4% to 95.8%

Right Hand: 0.925 to 0.948 or 92.5% or 94.8%

Palm: 0.93 to 0.953 or 93% to 95%

Back: 0.935 or 93.5%

The model was successfully trained. This can be said due to the confidence levels for each hand orientation. For example: the confidence level for the palm is from 0.93 to 0.953. This indicates that the Trained model was able to detect and successfully predict the object in front of it 93 to 95 percent of the time.

DISCUSSION

Discussion From the dataset preparation, model training, and evaluation, several challenges come up while developing and optimizing a hand detection system using deep learning approaches. This discussion becomes very complex on the problems faced during the project and offers solutions to how these were solved; hence, it exposes the readers to a knowledge base of both the issues and how the solution strategies are underway.

Variability and Preprocessing Challenges in the Dataset One of the significant challenges in this project was the management of the variability in the datasets. The original dataset had a type of background variety, distance variation, and lointrocts of hand angles.

Such diversity is helpful during the stage of building a robust model, but unfortunately, it introduces big noise and confusion through the process of training. Therefore, the system could not generally learn from this data set, as it had a very high variance in input data, which most often resulted in low accuracy in detection. To minimize these problems, the second dataset was done in a more controlled manner; only the hand without changing the distance or angle was used. On the other side, this approach had also defects. Because it did not consider enough variability, the resulted model poorly performed when introduced to new, unseen conditions. Therefore, the ideal dataset should balance the need to incorporate enough variability so that the model trained generalizes and, at the same time, maintains enough consistency not to overwhelm the model with noise.

Image PreSub-processing and the image preprocessing played a critical importance in solving some of the challenges noted. Grayscale application to ignore hand colour failed to come up with the desired result and may suggest that colour carries important discriminative features for hand detection. The front end resizes the augmented images to speed up training and augments them diversely to increase the robustness of the model possibly. The augmentation includes much horizontal flipping, especially to balance the number of representations between the left and right hands in the data set. This, however, posed a complication to the detection system since now mirrored images had to be brought in to confuse between the left and right-hand labels.

To clear any doubt that could arise from the fact that the images may be mirrored, during annotation, explicit labels were used to denote each orientation of hands and each part (back or palm). Differentiating between the back and palm of the hands and even between left and right in the labels helped me guide the model better during training and, therefore, enhanced the detection. Model Training and Hardware Utilization The project employed PyTorch in building and training the model, leveraging CUDA-enabled GPU computational resources. This, therefore, called for handy use of dual GPU cards in considerably reducing the training time and smooth training of even more complex models by handling those training processes that are of high computational intensity.

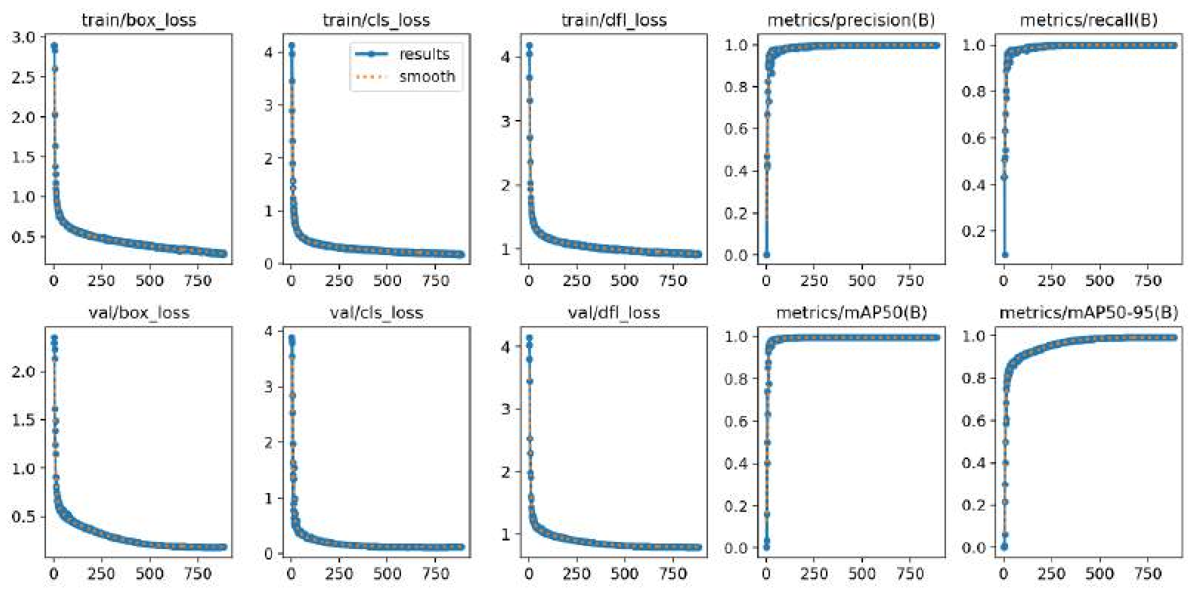


Figure 93 Tranloss,BoxLoss

As shown in the figure above that the box loss during the training and the validation box loss typically pertain to object detection of the tasks where our model must predict the bounding boxes which is around the objects. A decreasing loss as shown in the figure mean the models perfections are getting closer to the ground truth. Classification lost graph it reports how well the model is preforming in the terms of classifying the objects within the bounding boxes. A declining loss here it indicates an improvement in the model's class accuracy. Then we have the dimension loss graph it usually related to the size and shape accuracy of the predicted bounding boxes again as we can see there is a decreasing loss and it indicates that the predicted box dimension are aligning more accurately with the actual dimension.

Metrics (Map50-95): mean average precision Map is a comprehensive matric that is used to evaluate the object detection models and the AP stand for the average precision for the single class or prediction at different thresholds then the Map50-95 would be the mean AP calculated over different intersection over the union so from 0.5 the loses criteria for the correct detection to the 0.95 which is the strict criteria so a higher mAP indicate more accurate model.

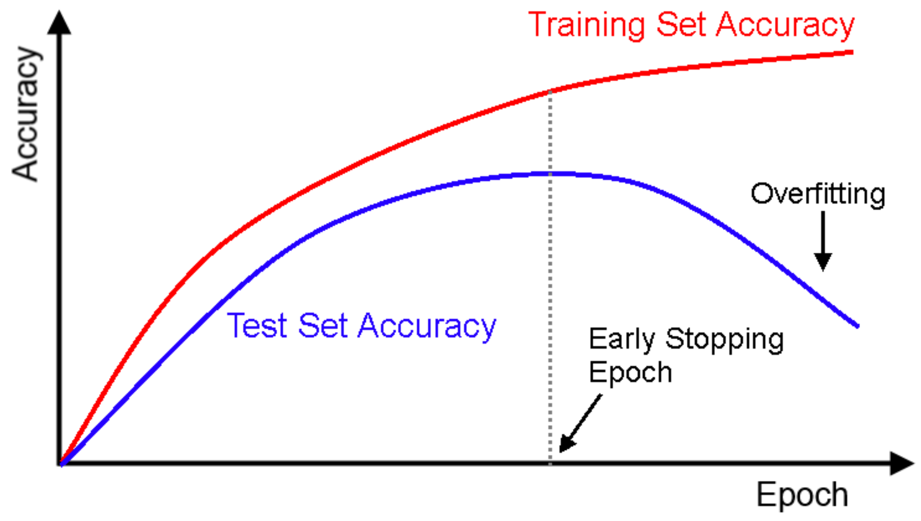


Figure 94 accuracy curve

The accuracy curve shown above is also known as the training accurate cure and it shows us how good the model is at making the correct perfections based on the training data as it will go through the training process. Accuracy is measure in percentage so it will tell us the proportion of instance the model correctly classified out the total number of instance so the accuracy cure is given us an idea how our model firs the training data and improve it ability to make accurate predictions and we can see that from our data that the training loss is decreasing which it mean that the accuracy is increasing and training have been tested with difference value of epochs to avoid the overfitting.

TASK 2 CNN (Osama)

Introduction

Convolutional Neural Networks (CNNs) Convolutional Neural Networks (CNN) is a type of deep neural network, basically used in the analysis of visual imagery. More specifically, the network has more powerful abilities to conduct tasks like image recognition, object detection, and segmentation. The standard layers in CNN architecture include (freecodecamp.org, 2022):

Convolutional layers: In this layer, a set of filters is convolved with the input. Each filter captures some specific features at some particular locations of the input.

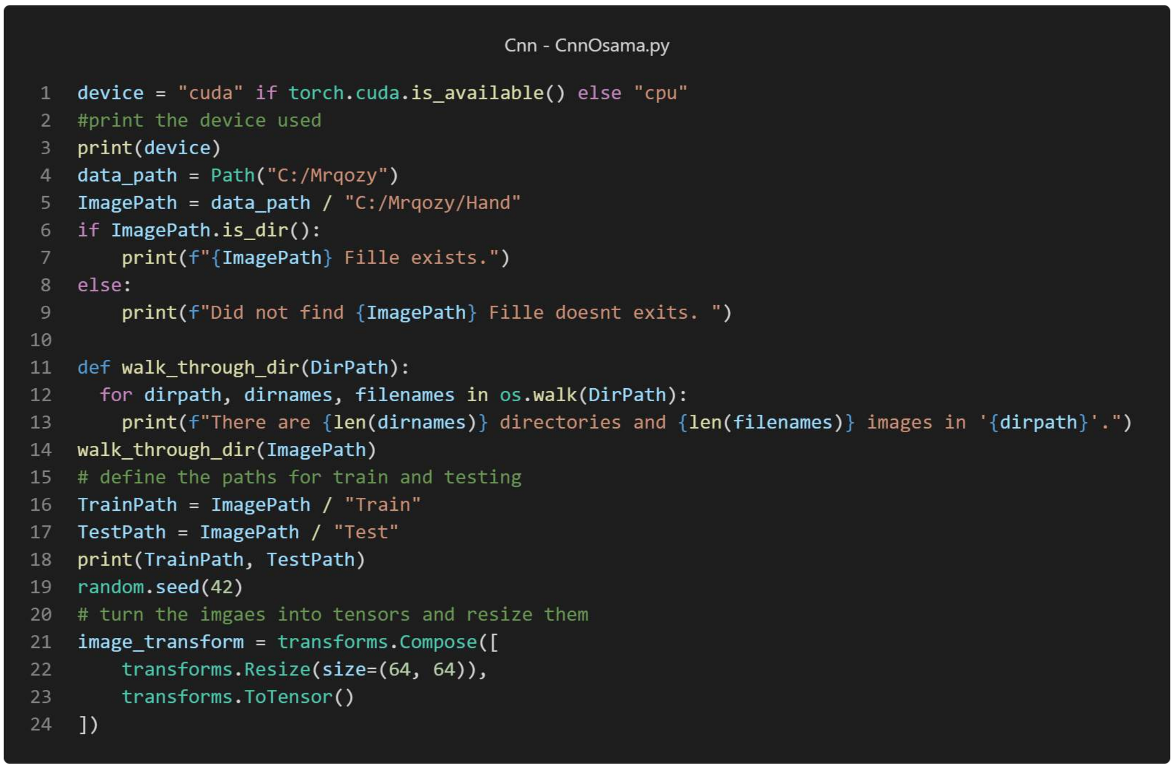
An activation function, for instance, ReLU (Rectified Linear Unit), adds non-linearity to the model so that the complex pattern could be learned in the model.

For example, max pooling is a pooling layer that reduces the spatial size of the representation, thus making the model easier to manage and reducing the number of parameters.

Dense layers, i.e., fully connected layers, classify the image based on the features extracted and down-sampled by convolutional and pooling layers.

CNNs use spatial her son data and have been particularly successful in computer vision tasks, given that they can detect patterns with high variability.

Code

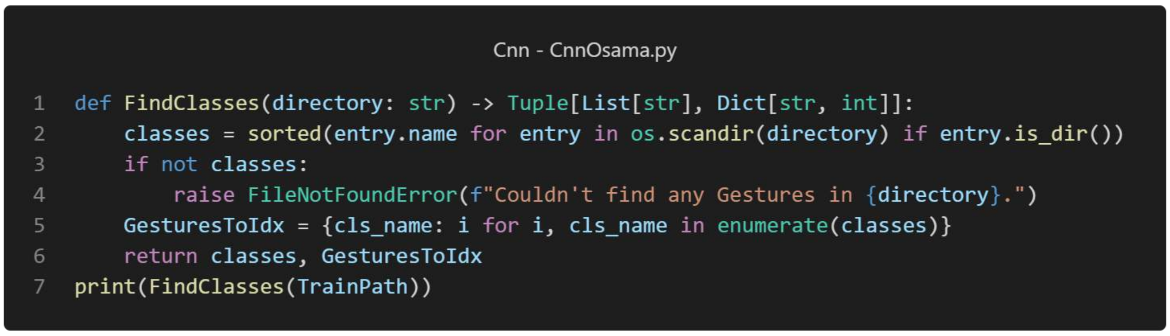


Cnn - CnnOsama.py

```python
device = "cuda" if torch.cuda.is_available() else "cpu"
#print the device used
print(device)
data_path = Path("C:/Mrqozy")
ImagePath = data_path / "C:/Mrqozy/Hand"
if ImagePath.is_dir():
    print(f"{ImagePath} Fille exists.")
else:
    print(f"Did not find {ImagePath} Fille doesnt exits. ")

def walk_through_dir(DirPath):
  for dirpath, dirnames, filenames in os.walk(DirPath):
    print(f"There are {len(dirnames)} directories and {len(filenames)} images in '{dirpath}'.")
walk_through_dir(ImagePath)
# define the paths for train and testing
TrainPath = ImagePath / "Train"
TestPath = ImagePath / "Test"
print(TrainPath, TestPath)
random.seed(42)
# turn the imgaes into tensors and resize them
image_transform = transforms.Compose([
    transforms.Resize(size=(64, 64)),
    transforms.ToTensor()
])
```

Figure 95 Device

In the figure above as we can see that the program will check if the can use the gpu or not and if not cpu will be used. Then in the cdoe the paths have been defined and check if there exists or not so if it doesn’t exits it will print that the fille doesn’t exits and it will take all the images in the file and resize them and trun the imgaes into tensor so it can be trained and tested.

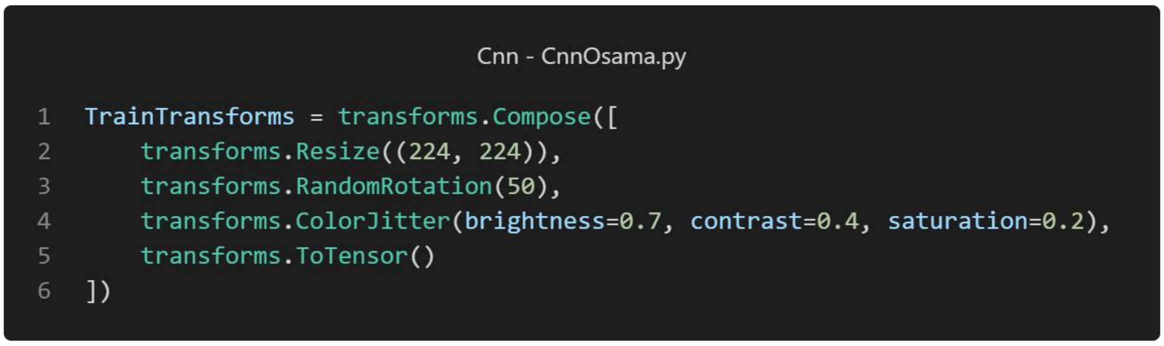


Cnn - CnnOsama.py

```python
def FindClasses(directory: str) -> Tuple[List[str], Dict[str, int]]:
    classes = sorted(entry.name for entry in os.scandir(directory) if entry.is_dir())
    if not classes:
        raise FileNotFoundError(f"Couldn't find any Gestures in {directory}.")
    GesturesToIdx = {cls_name: i for i, cls_name in enumerate(classes)}
    return classes, GesturesToIdx
print(FindClasses(TrainPath))
```

Figure 96 find classes

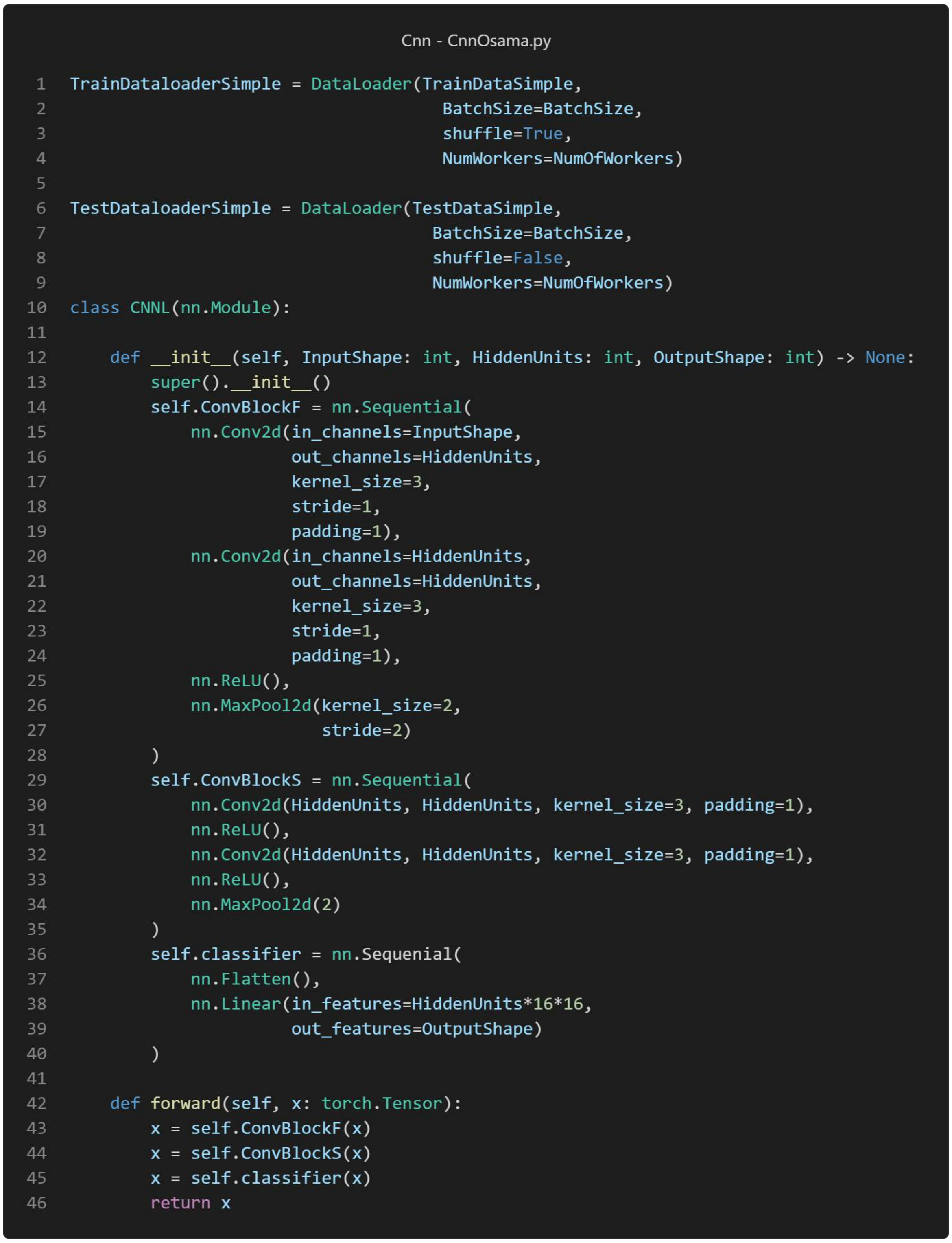
Here in the figure above def find classes will go through the fille andc cheack for our classes for example left hand,right hand, palm and back.


Cnn - CnnOsama.py

```python
TrainTransforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomRotation(50),
    transforms.ColorJitter(brightness=0.7, contrast=0.4, saturation=0.2),
    transforms.ToTensor()
])
```

Figure 97 Train Data

In the figure above is the data before training so before the training the images will be resize and then it will apply random reotaion from -50 to 50 so it will cove diffrant rotation and then color jitter which so it can cove more sking color and to improve the trining and after this the imgaes will trun inro tenosor so it can be trained and tested.



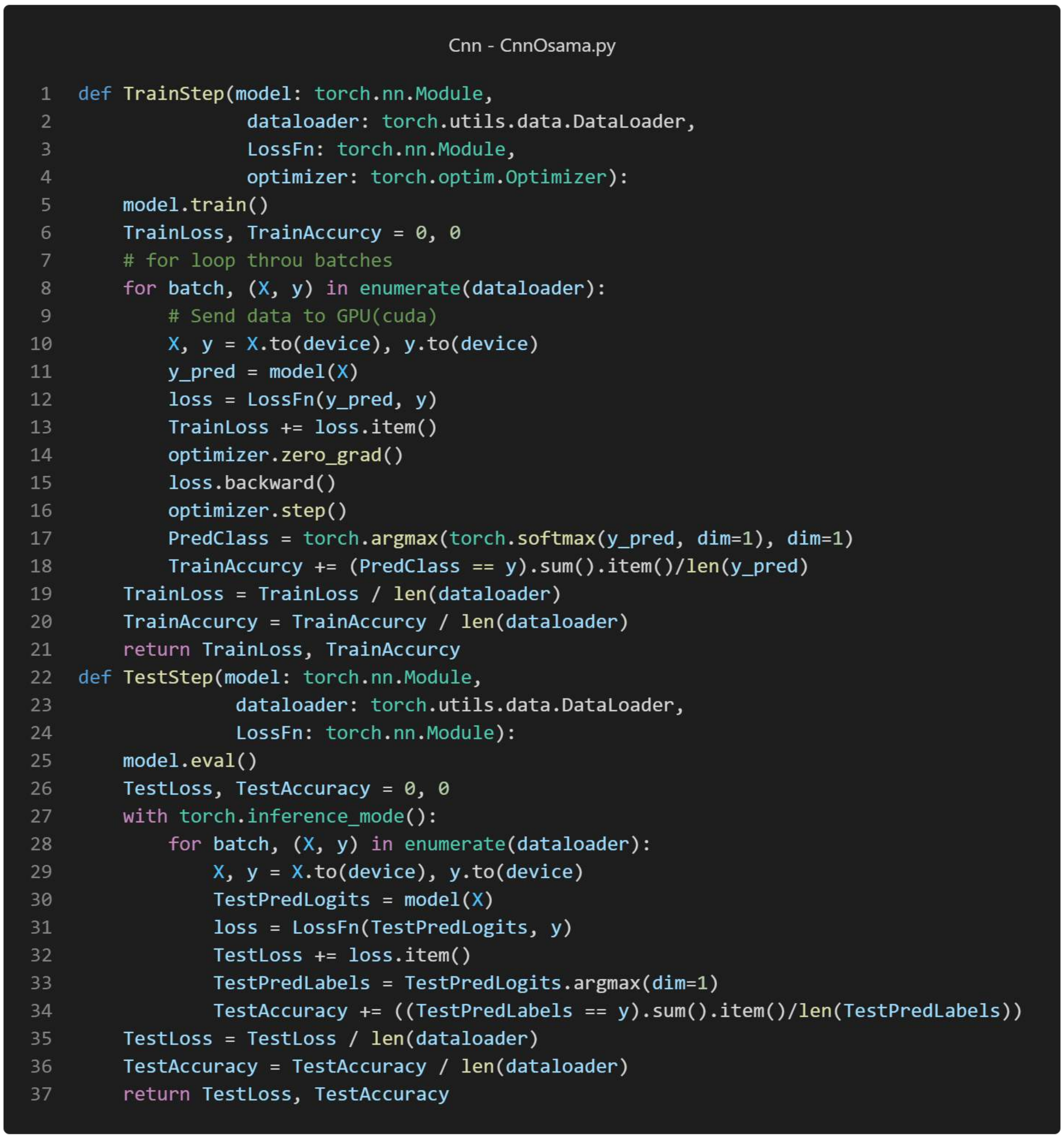
Cnn - CnnOsama.py

```python
TrainDataloaderSimple = DataLoader(TrainDataSimple,
                                    BatchSize=BatchSize,
                                    shuffle=True,
                                    NumWorkers=NumOfWorkers)

TestDataloaderSimple = DataLoader(TestDataSimple,
                                   BatchSize=BatchSize,
                                   shuffle=False,
                                   NumWorkers=NumOfWorkers)
class CNNL(nn.Module):

    def __init__(self, InputShape: int, HiddenUnits: int, OutputShape: int) -> None:
        super().__init__()
        self.ConvBlockF = nn.Sequential(
            nn.Conv2d(in_channels=InputShape,
                      out_channels=HiddenUnits,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.Conv2d(in_channels=HiddenUnits,
                      out_channels=HiddenUnits,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
        self.ConvBlockS = nn.Sequential(
            nn.Conv2d(HiddenUnits, HiddenUnits, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(HiddenUnits, HiddenUnits, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequenial(
            nn.Flatten(),
            nn.Linear(in_features=HiddenUnits*16*16,
                      out_features=OutputShape)
        )

    def forward(self, x: torch.Tensor):
        x = self.ConvBlockF(x)
        x = self.ConvBlockS(x)
        x = self.classifier(x)
        return x
```

Figure 98 CNN

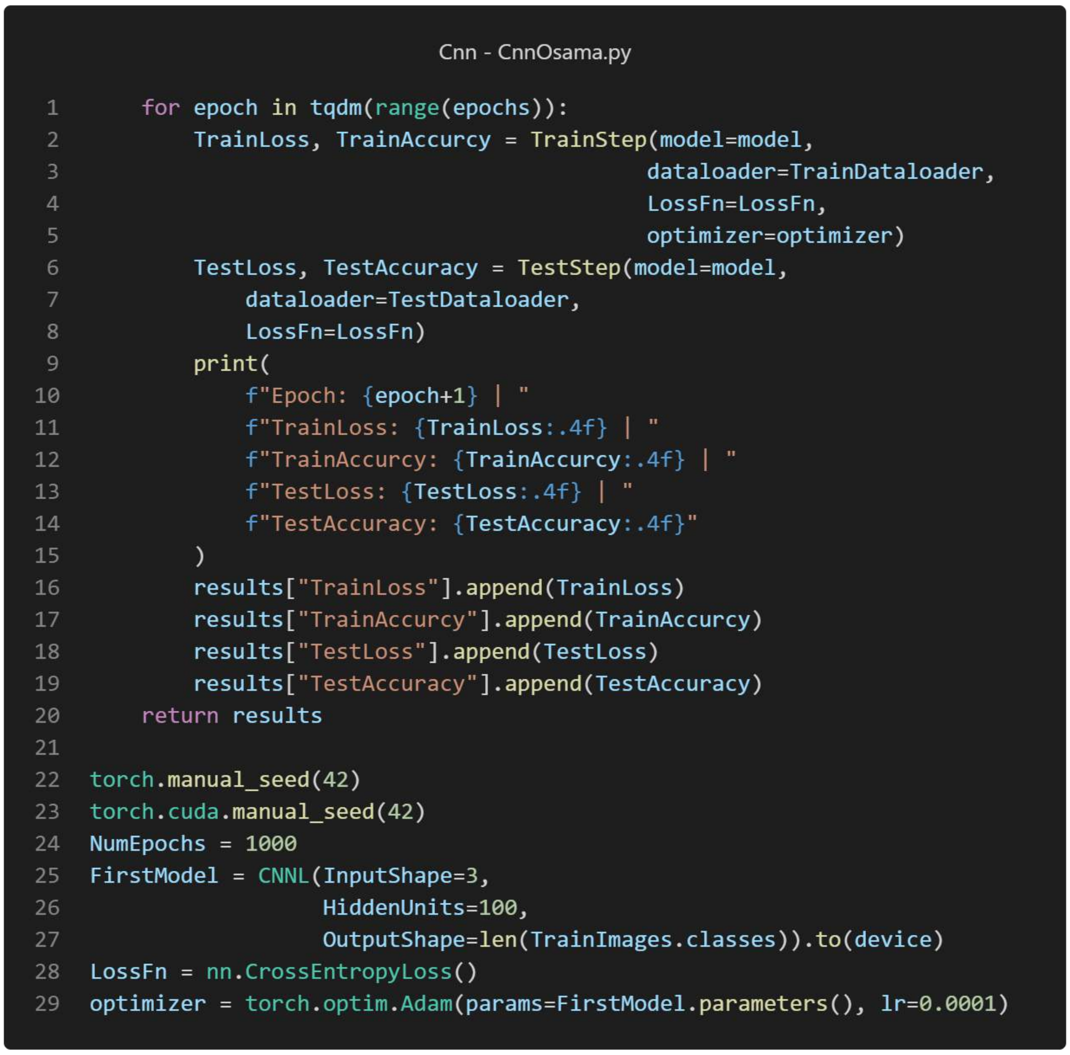
As we can see from the figure above that the data is beging shuffle before going through the layer and as we can see we have 2 block of layer and classifier so in the first block we have (nn.Sequential) that groups together the first set of convolutional layer (nn.Conv2d), a ReLU activation function (nn.ReLU), and a max-pooling layer (nn.MaxPool2d). This block will process the input images first.

And also there are two onvolutional layers for the second block of layers we have is almost the same as the first one but I will take the input from the output of the first layer then there is the classfier and it contain a flatten layer to conver it from 2D to 1D and then it will go trough linear layer. The CNN model is a typical example of a PyTorch standard neural network for processing image classification tasks. This model contains typical components of convolutional layers, activation functions, pooling layers, and fully connected layers in classification. The architecture will use this model to extract the spatial hierarchy features of the input images, which will be used in the classification of the output of the photos into respective categories. The details would be, for instance, details of InputShape, HiddenUnits, and OutputShape when an instance of CNNL is to be created. It will be much dependent on the particular data set in use and the classification problem at hand.


Cnn - CnnOsama.py

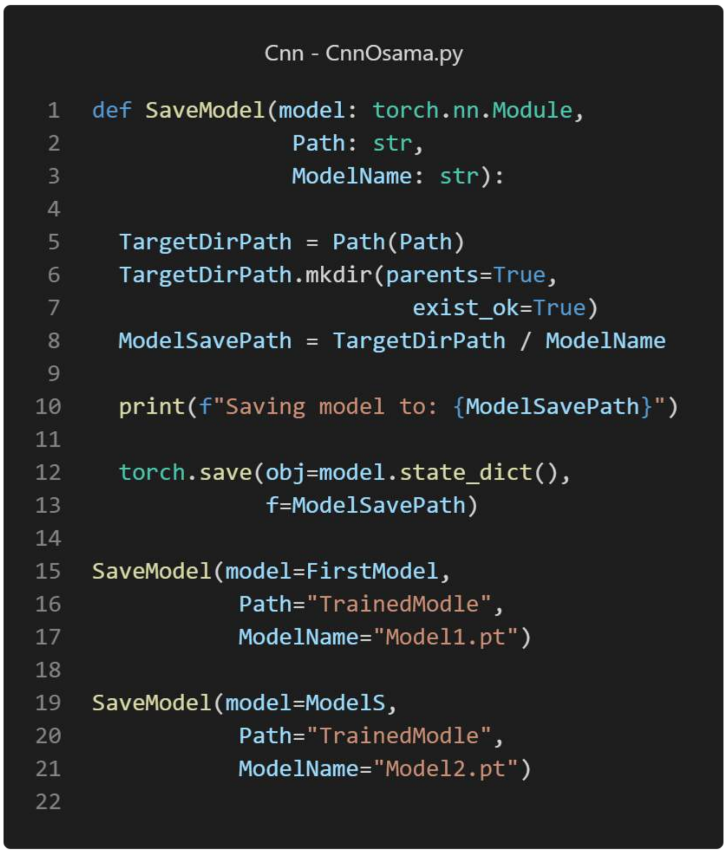
```python
def TrainStep(model: torch.nn.Module,
               dataloader: torch.utils.data.DataLoader,
               LossFn: torch.nn.Module,
               optimizer: torch.optim.Optimizer):
    model.train()
    TrainLoss, TrainAccurcy = 0, 0
    # for loop throu batches
    for batch, (X, y) in enumerate(dataloader):
        # Send data to GPU(cuda)
        X, y = X.to(device), y.to(device)
        y_pred = model(X)
        loss = LossFn(y_pred, y)
        TrainLoss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        PredClass = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
        TrainAccurcy += (PredClass == y).sum().item()/len(y_pred)
    TrainLoss = TrainLoss / len(dataloader)
    TrainAccurcy = TrainAccurcy / len(dataloader)
    return TrainLoss, TrainAccurcy
def TestStep(model: torch.nn.Module,
              dataloader: torch.utils.data.DataLoader,
              LossFn: torch.nn.Module):
    model.eval()
    TestLoss, TestAccuracy = 0, 0
    with torch.inference_mode():
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)
            TestPredLogits = model(X)
            loss = LossFn(TestPredLogits, y)
            TestLoss += loss.item()
            TestPredLabels = TestPredLogits.argmax(dim=1)
            TestAccuracy += ((TestPredLabels == y).sum().item()/len(TestPredLabels))
    TestLoss = TestLoss / len(dataloader)
    TestAccuracy = TestAccuracy / len(dataloader)
    return TestLoss, TestAccuracy
```

In the code shown in the figure above the model is set to training mode, viables for tracking the cumulative training loss and also the accuracy and both of there are initialized to zero then the function enter a loop over the data batches provided by the data loader so within the loop data and labels will be move to CPU or GPU depends on the device and the loss is computed between the prediction and the true labels using the function LossFn. The loss is backgrounded using loss.backward() and the optimizer updates the models weight and then Prediction accuracy is calculated by comparing the predicted labels to the true labels. And the other function which the test function is almost the same thing but with the test data.


Cnn - CnnOsama.py

```
    for epoch in tqdm(range(epochs)):
        TrainLoss, TrainAccurcy = TrainStep(model=model,
                                            dataloader=TrainDataloader,
                                            LossFn=LossFn,
                                            optimizer=optimizer)
        TestLoss, TestAccuracy = TestStep(model=model,
            dataloader=TestDataloader,
            LossFn=LossFn)
        print(
            f"Epoch: {epoch+1} | "
            f"TrainLoss: {TrainLoss:.4f} | "
            f"TrainAccurcy: {TrainAccurcy:.4f} | "
            f"TestLoss: {TestLoss:.4f} | "
            f"TestAccuracy: {TestAccuracy:.4f}"
        )
        results["TrainLoss"].append(TrainLoss)
        results["TrainAccurcy"].append(TrainAccurcy)
        results["TestLoss"].append(TestLoss)
        results["TestAccuracy"].append(TestAccuracy)
    return results

torch.manual_seed(42)
torch.cuda.manual_seed(42)
NumEpochs = 1000
FirstModel = CNNL(InputShape=3,
                  HiddenUnits=100,
                  OutputShape=len(TrainImages.classes)).to(device)
LossFn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=FirstModel.parameters(), lr=0.0001)
```

torch.cuda.manual_seed(42) - These lines set the seed for the random number generators in PyTorch for both CPU and GPU to ensure reproducibility. And then the number of epochs that will be used have been specified and some functions previously defined have been called like TestLoss for example and then this training will be for the first model and the learning rate is set to be 0.0001 and the same thing for the second model but output data from the layers.


Cnn - CnnOsama.py

```python
def SaveModel(model: torch.nn.Module,
               Path: str,
               ModelName: str):

  TargetDirPath = Path(Path)
  TargetDirPath.mkdir(parents=True,
                      exist_ok=True)
  ModelSavePath = TargetDirPath / ModelName

  print(f"Saving model to: {ModelSavePath}")

  torch.save(obj=model.state_dict(),
             f=ModelSavePath)

SaveModel(model=FirstModel,
           Path="TrainedModle",
           ModelName="Model1.pt")

SaveModel(model=ModelS,
           Path="TrainedModle",
           ModelName="Model2.pt")
```

Figure 99 SavingModel

In this code that shown above def SaveModel it will save the model so it can be used later and make predictions based on the save model.

Results

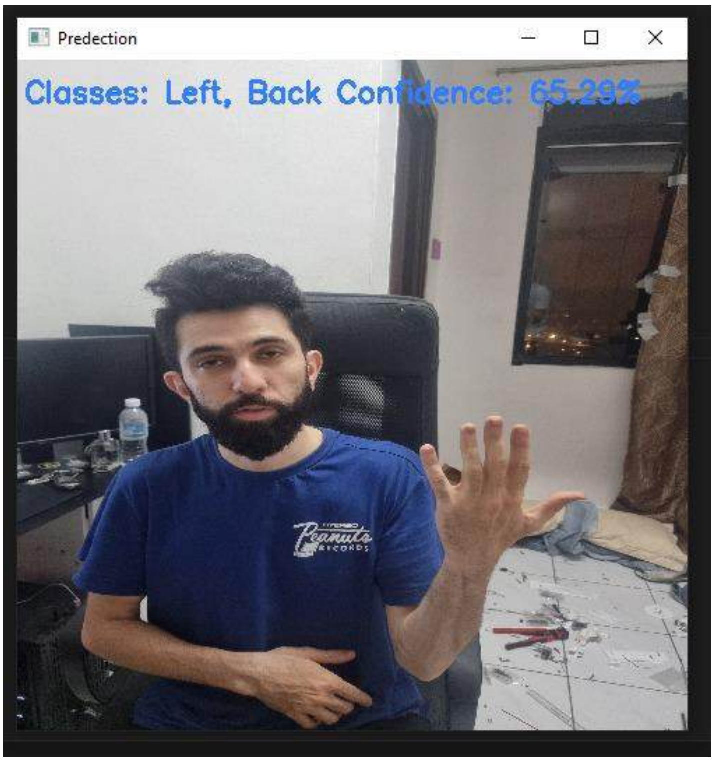


Figure 100 LeftBackC

As it shown in the figure above when the Left Back hand have been test that confidence level is 65.29% which is not hight accuracy but it still got it right

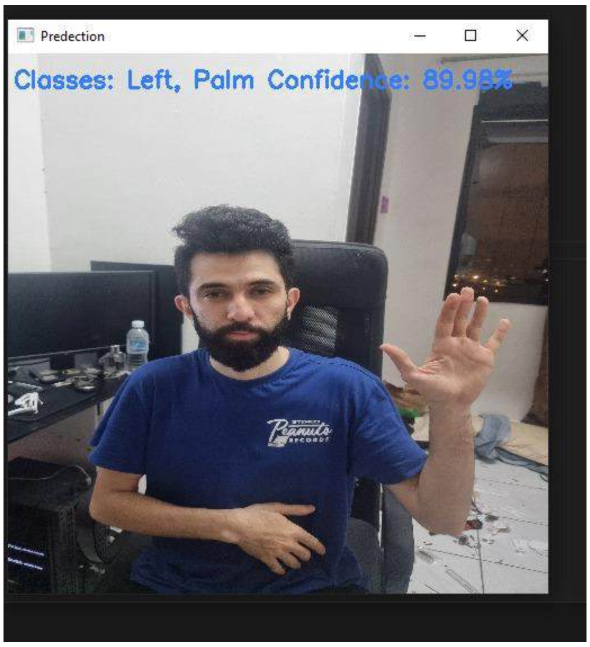


Figure 101 Left PalmC

As we can see here when testing Left palm it was conffdenct by 89.9% which is high and better than the left back prediction.

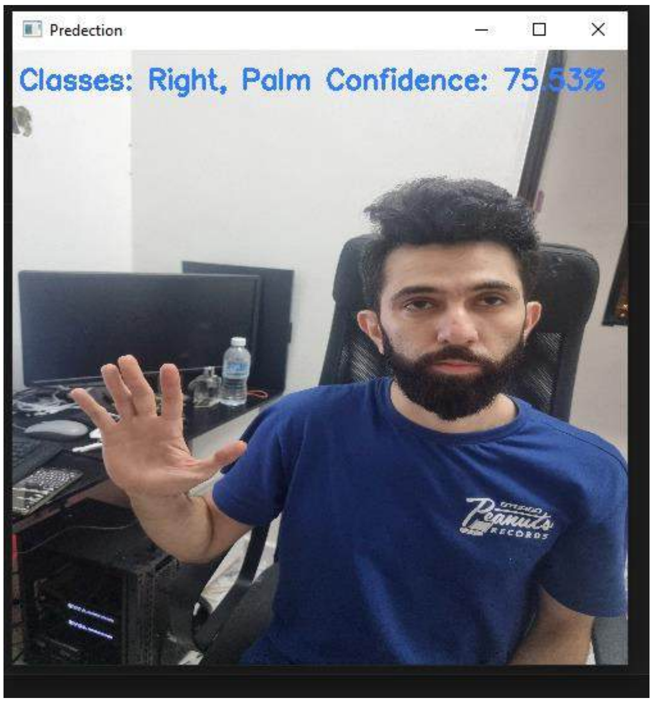


Figure 102 Right Palm

In the figure shown above it is clear that the prediction is correct and it was confidant by 75.53% which can be consider as high percentage of predication.

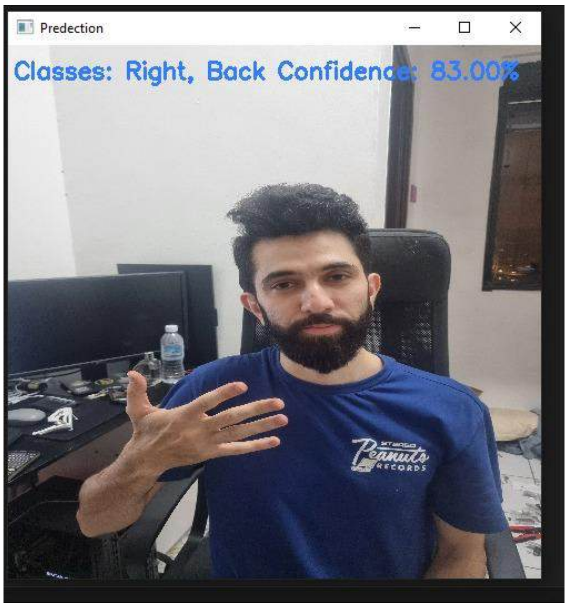


Figure 103 Right Back

As it shown for the Right back prediction it was correct and it was confidant by 83%.

So from the figures above it clearly show that the model is capable of distinguishing not only between left and right hand but also whether palm or back of the hand is visible to the camera so the variance in confidence scores could be due to serve factor including the clarity of the gesture the lighting condition or the positioning of the hand relative to the our camera. There results are significant for applications where hand gesture recognition is crucial such as in sign language for example, human computer interaction and also the virtual reality environments where gesture can serve as inputs for the control mechanism and the confidence score is also crucial for such application because it indicate how relabel a prediction is which can be used to decide whether to act based on that prediction or nor or to request a clearer gesture from the user. The system seems to perform well but there is noticeable variation in confidence which can be improved by further training in the model or more diverse set of hand gesture and conditions or increasing the number of epochs or by refining the model's architecture and parameters.

Discussion

The CNN model is structured with several layers crucial for interpreting visual data:

Convolutional layers Therese layer are the backbone of the model because it utilizing filter to capture and analyze features from input images and each felted is designed to detect specify attributes at various spatial location, then we have the Activation functions basically ReLU which stand for rectified linear unit is used to introduce non linearity into the model so it will allow the model to learn complex patterns and features beyond mere linear relationships. Then for the pooling layer there will reduce the spatial dimensions for the height and width of the input volume for the next layer and they are essential for decreasing the computational load and the number of parameters we have. Fully connected layer at the end there layer will basically classify the image based on the learned featured by executing higher level reasoning.

Training steps: iterative learning the model iteratively adjust its weights based on the computed loss between the predicted outputs and the actual labels. So the process is fundamental to refining the models predictive.

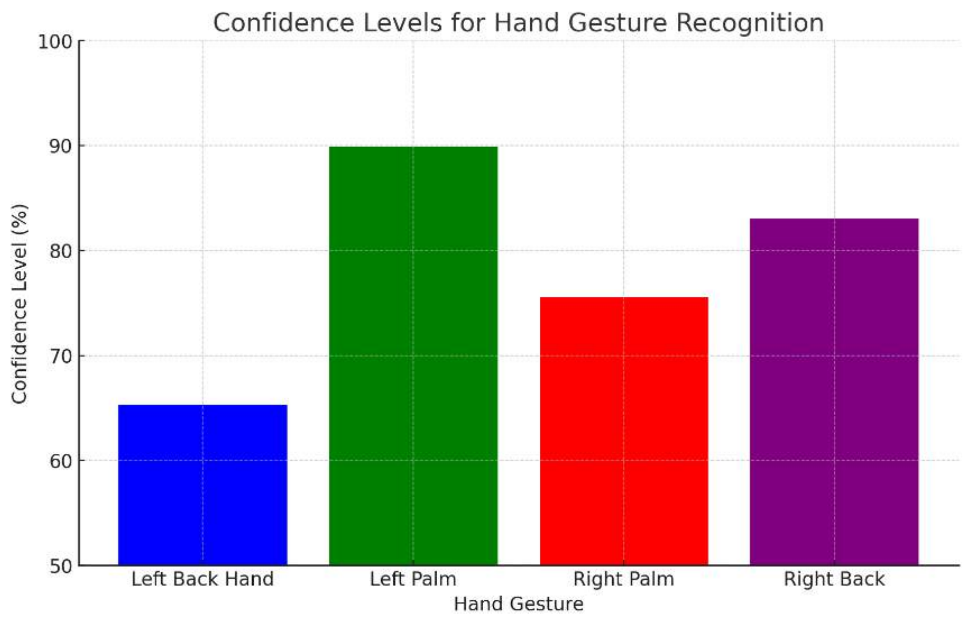


Figure 104 Conf Level

As shown in the figure above that Results and Evaluation: The model's performance in recognizing hand gestures yields varying levels of confidence:

Left Back Hand: 65.29%

Left Palm: 89.9%

Right Palm: 75.53%

Right Back: 83%

These variations suggest that while the model preform well in some of the scenarios there is some scenarios that need to be improved especially in condition where the confidence level is low and this can be done by increase the data set for it or increase the number of epochs for the training or the hidden units but overfitting have to be avoided. As we can see the models ability to differentiate between various hand gestures with varying the confidence levels so it indicates its potential utility in applications like the sign language interpretation, human-computer interaction, and virtual reality.

Conclusion

In conclusion, The performance of the CNN model with a confidence level ranging between 65.29% and 89.9% across different gestures, highlights both its capabilities and limitations. While the model performs exceptionally well in certain scenarios (like recognizing the Left Palm with 89.9% confidence), it struggles in others (such as recognizing the Left Back Hand with only 65.29% confidence). This variability suggests that while CNNs are powerful tools for image recognition, their efficacy can be influenced by factors such as the quality of input data and the specificity of training processes. And it can be improved by increasing the number of images in the data set and the vibration and also increasing the number of epochs.

PEDRO FABIAN OWONO ONDO MANGUE(TP063251)

TASK 1

INTRODUCTION

Modern technology allows machines to better understand our hand movements. This enables new ways of interacting with computers. Hand gesture recognition plays a vital role in this interaction. It lets people control devices without physical contact. This capability proves useful in various scenarios, such as operating vehicle interfaces or during surgical procedures where maintaining hygiene is paramount.

Lately, the technology has improved to recognize hand gestures like grabbing, pinching, and rotating. These touchless controls prove useful where hygiene matters or when focus is needed. For instance, doctors require concentration during operations. Drivers must keep their eyes on the road. This system employs computer vision to detect and comprehend hand gestures in real-time. Earlier research by Viola and Jones on face detection, and others on hand recognition, formed the basis for the algorithms utilized in this project.

OVERVIEW OF THE SYSTEM

The code detects hand gestures like grabbing, rotation, pinching and thumbs up using a webcam and the OpenCV library. The coding language used is python and the IDLE is PyCharm. The system detects the gestures by calculating the angles between fingers where the defects are detected. The system also utilizes the use of masking and trackbar which helps in tuning the system real time. Below is an overall explanation of the coding structure and the results of the system.

TECHNIQUES AND APPROACHES

The implementation of this hand gesture recognition system combines several advanced computing vision techniques using the OpenCV library and Python. These techniques encompass:

Setting the Region of Interest (ROI): This involves defining a particular area within the video frame (ROI) where the hand gestures are expected and analysed. This step concentrates the processing power on a smaller frame segment, enhancing the gesture detection's accuracy and speed.

Filtering Skin Colour: By converting the ROI to the HSV colour space and applying a skin colour filter, the system isolates the hand from other objects in the frame. This step proves crucial for effectively detecting the hand's contours.

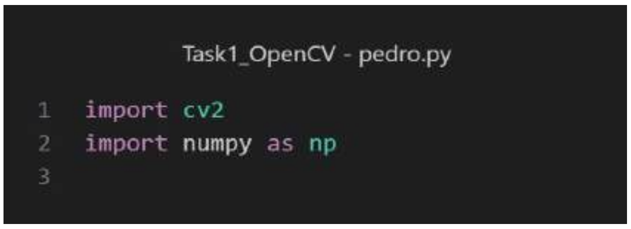
Reducing Noise and Detecting Contours: After applying the skin colour filter, the system utilizes morphological operations (erosion and dilation) and Gaussian blurring to clean the image, followed by contour detection to identify the hand's outline.

Gesture Buffering and Analysis: A special feature stores recent hand motions. It helps confirm gestures over multiple video frames. This reduces chances of incorrect gesture recognition due to misreading or noise.These steps are crucial. They ensure the hand gesture system works accurately. It also ensures the system functions well in different lighting and environmental conditions.

CODE STRUCTURE EXPLANATION

The code for this hand gesture system was made using PyCharm IDE. PyCharm provides an efficient platform for coding, debugging, and testing Python apps. This makes it ideal for complex computer vision projects.

Libraries Used:



Task1_OpenCV - pedro.py

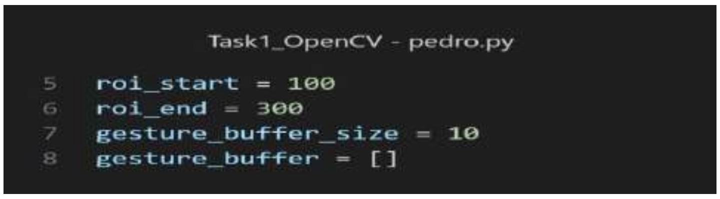
```python
import cv2
import numpy as np

```

Figure 105: Imported Libraries

NumPy: Used for high-performance math computing. It provides efficient data structures and operations. These are essential for handling image and video data in computer vision.

OpenCV (cv2): This library is key for any computer vision project. It offers tools for image processing, video capture, and analysis. These are crucial for detecting and understanding hand gestures.

Defining Constants and Variables:



Task1_OpenCV - pedro.py

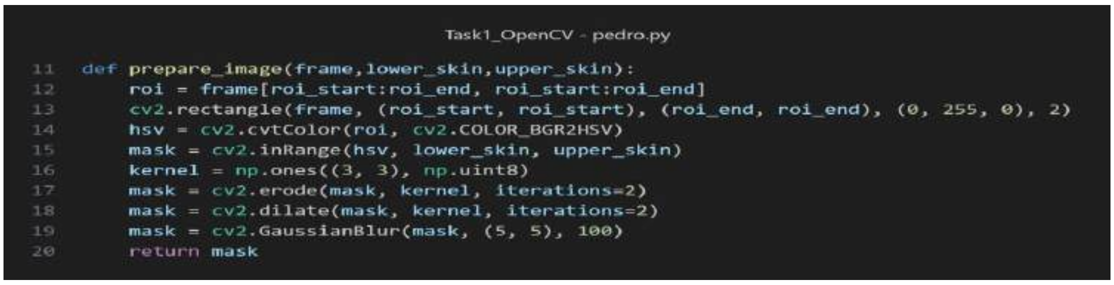
```python
roi_start = 100
roi_end = 300
gesture_buffer_size = 10
gesture_buffer = []
```

Figure 106: region of interest

roi_start/roi_end: Defines Region of Interest (ROI) coordinates in video frame. This is where hand is expected to be placed. Focusing analysis here reduces processing load.

gesture_buffer_size/gesture_buffer: Helps to make gesture recognition steady. They store the last few recognized gestures. This ensures that occasional misrecognitions do not impact the overall output.

Image Preparation:



Task1_OpenCV - pedro.py

```python
def prepare_image(frame,lower_skin,upper_skin):
    roi = frame[roi_start:roi_end, roi_start:roi_end]
    cv2.rectangle(frame, (roi_start, roi_start), (roi_end, roi_end), (0, 255, 0), 2)
    hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, lower_skin, upper_skin)
    kernel = np.ones((3, 3), np.uint8)
    mask = cv2.erode(mask, kernel, iterations=2)
    mask = cv2.dilate(mask, kernel, iterations=2)
    mask = cv2.GaussianBlur(mask, (5, 5), 100)
    return mask
```

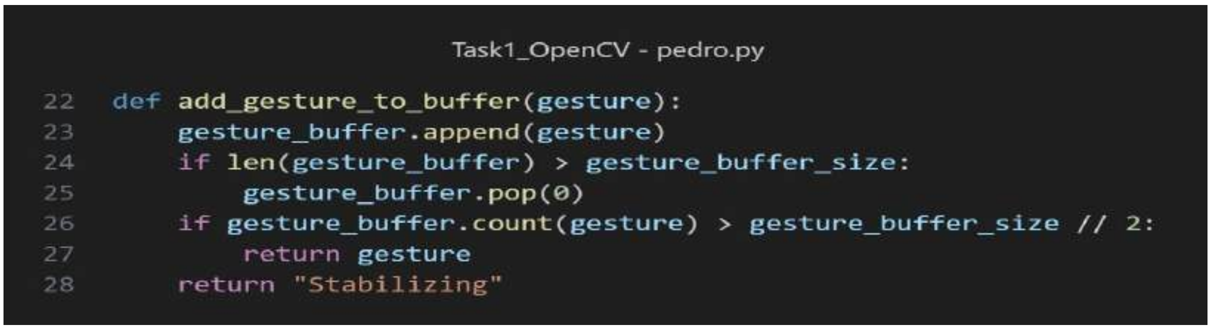
Figure 107: Image Preparation

def Prepare_image: Isolates the hand from the video frame. First, it picks out the ROI (Region of Interest). Then, it converts it to HSV colour space for better colour segmentation.

cv2.inRange: Makes a binary mask. In this mask, the skin colour (defined by lower skin and upper_skin) appears white. The rest appears black.

cv2.erode/cv2.dilate: are morphological operations. They remove noise and fill gaps in the detected skin region. **cv2.GaussianBlur:** smooths the mask reduces noise further and improves contour detection.

Gesture Buffering Function:


Task1_OpenCV - pedro.py

```
def add_gesture_to_buffer(gesture):
    gesture_buffer.append(gesture)
    if len(gesture_buffer) > gesture_buffer_size:
        gesture_buffer.pop(0)
    if gesture_buffer.count(gesture) > gesture_buffer_size // 2:
        return gesture
    return "Stabilizing"
```

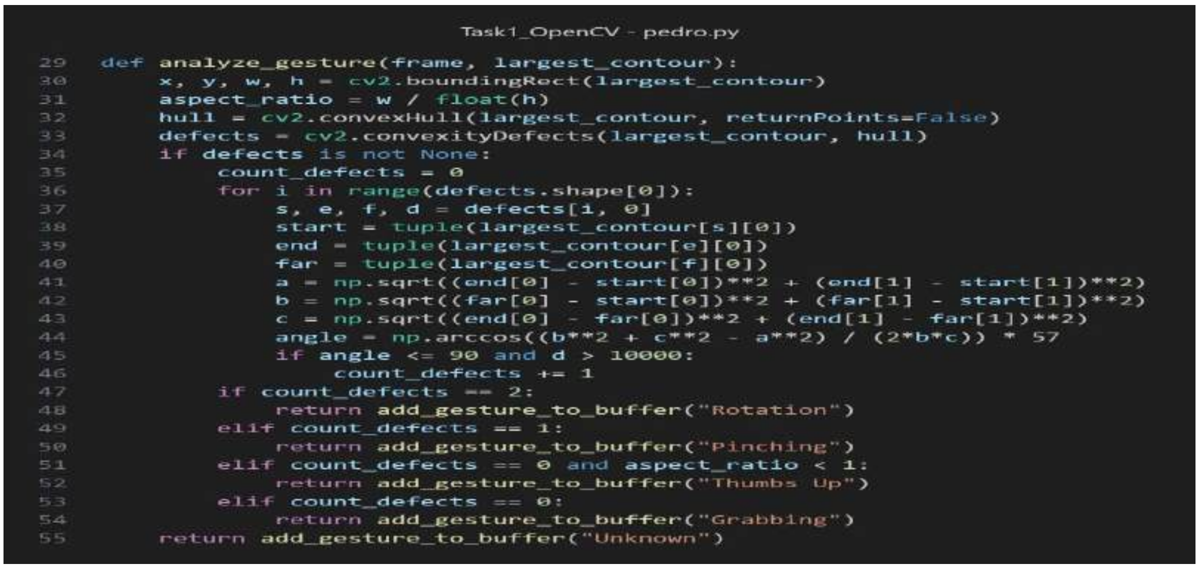
Figure 108: add gesture to buffer

Adding Gesture: As the gestures are detected it get added to the array gesture_buffer which stores the different gestures.

Checking Size: It then checks the length of the gesture_buffer exceeds the length set in the variable gesture_buffer_size. If it exceeds the length the gesture is taken out of the list using pop
().

Checking count: This is used to check the number of times the gesture was added to the list. If it is more than half gesture_buffer_size it returns the gesture if not it will return "stabilizing".
This helps in confusing the system.

Gesture Analysis Function:



Task1_OpenCV - pedro.py

```python
def analyze_gesture(frame, largest_contour):
    x, y, w, h = cv2.boundingRect(largest_contour)
    aspect_ratio = w / float(h)
    hull = cv2.convexHull(largest_contour, returnPoints=False)
    defects = cv2.convexityDefects(largest_contour, hull)
    if defects is not None:
        count_defects = 0
        for i in range(defects.shape[0]):
            s, e, f, d = defects[i, 0]
            start = tuple(largest_contour[s][0])
            end = tuple(largest_contour[e][0])
            far = tuple(largest_contour[f][0])
            a = np.sqrt((end[0] - start[0])**2 + (end[1] - start[1])**2)
            b = np.sqrt((far[0] - start[0])**2 + (far[1] - start[1])**2)
            c = np.sqrt((end[0] - far[0])**2 + (end[1] - far[1])**2)
            angle = np.arccos((b**2 + c**2 - a**2) / (2*b*c)) * 57
            if angle <= 90 and d > 10000:
                count_defects += 1
        if count_defects == 2:
            return add_gesture_to_buffer("Rotation")
        elif count_defects == 1:
            return add_gesture_to_buffer("Pinching")
        elif count_defects == 0 and aspect_ratio < 1:
            return add_gesture_to_buffer("Thumbs Up")
        elif count_defects == 0:
            return add_gesture_to_buffer("Grabbing")
    return add_gesture_to_buffer("Unknown")
```

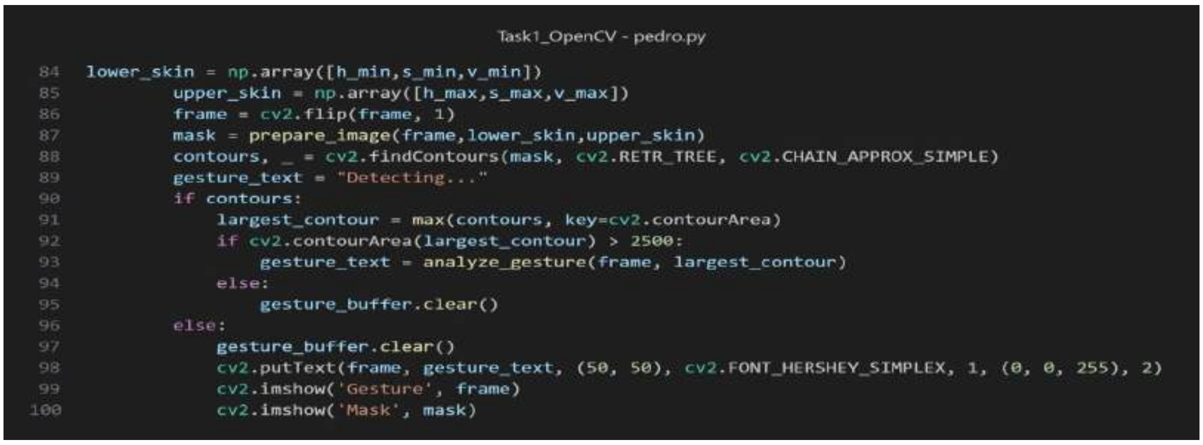
Figure 109: analyze gesture function

Convex Hull: This variable receives the bounding box of the largest contour of the object detected.

Defects: The defects of the largest contour and the aspect ratio is also calculated which is stored in the variable called defects.

Validation: if the defects are not null the system will start to iterate through the defects and calculate the angle between each finger where the defect is. Based on the angle it predicts the gesture. If there were no defects the system will return unknown.

Main Execution Loop



Task1_OpenCV - pedro.py

```python
lower_skin = np.array([h_min,s_min,v_min])
        upper_skin = np.array([h_max,s_max,v_max])
        frame = cv2.flip(frame, 1)
        mask = prepare_image(frame,lower_skin,upper_skin)
        contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        gesture_text = "Detecting..."
        if contours:
            largest_contour = max(contours, key=cv2.contourArea)
            if cv2.contourArea(largest_contour) > 2500:
                gesture_text = analyze_gesture(frame, largest_contour)
            else:
                gesture_buffer.clear()
        else:
            gesture_buffer.clear()
            cv2.putText(frame, gesture_text, (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
            cv2.imshow('Gesture', frame)
            cv2.imshow('Mask', mask)
```

Figure 110: : Lower skin

Main Loop: The main loop captures videos from the webcam. It detects shapes in each frame. The largest shape is checked to classify the hand gesture.

Stopping Loop: The loop stops when you press the 'q' key. This lets you exit easily.

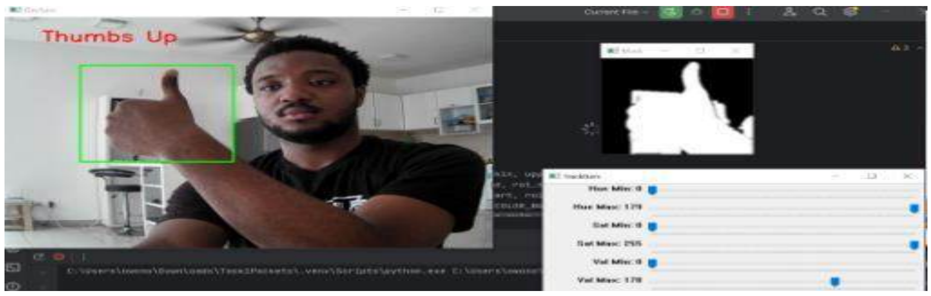
Frame Processing: The system captures video frames. It processes each frame to detect and understand hand gestures. The system shows the result on the screen. This provides real-time interaction. The system can be improved for better gesture-based user experience.

RESULTS

The Gesture Recognition System can find four hand gestures: pinching, grabbing, thumbs up, and rotation. A Python program using OpenCV checked for these gestures in real-time and showed the results. This report talks about how well the system recognized each gesture and what its limits are.

Results and Observations:

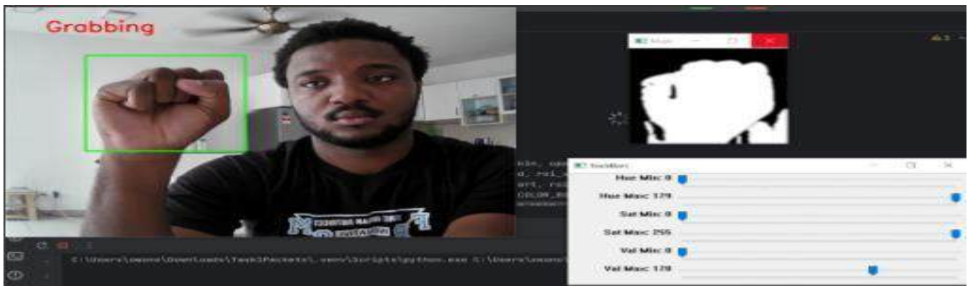
Thumbs Up Gesture: The correctly identified the "Thumbs Up" gesture. It drew a box around the hand and labelled the gesture properly. In the mask window, the thumb stood out clearly in black and white, showing that the system segmented and recognized it accurately.



```
Hue Min: 0
Hue Max: 179
Sat Min: 0
Sat Max: 255
Val Min: 0
Val Max: 179
```

Figure 111: Thumbs Up

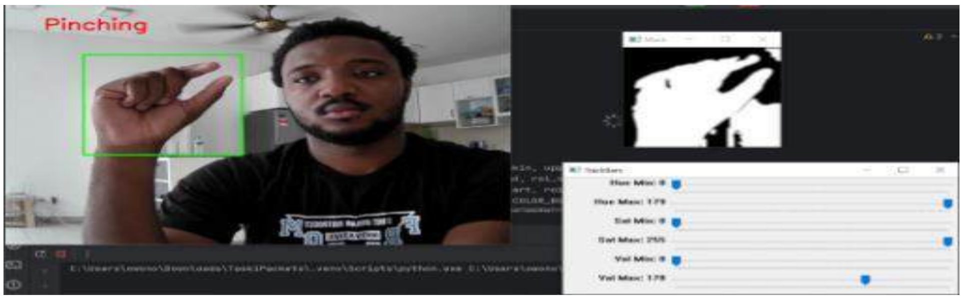
Grabbing Gesture: For the "Grabbing" gesture, the system drew the right label and box around the hand. The mask window highlighted the fingers bent inwards, confirming that the system understood the gesture.



```
Hue Min: 0
Hue Max: 179
Sat Min: 0
Sat Max: 255
Val Min: 0
Val Max: 179
```

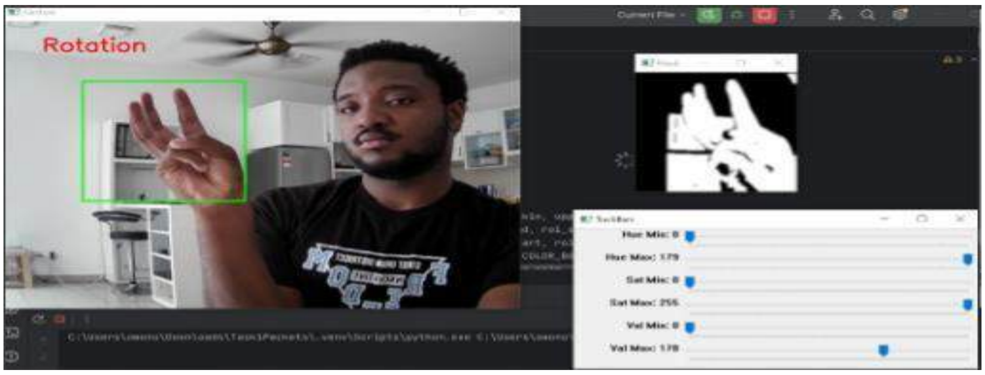
Figure 112: Grabbing

Pinching Gesture: The system spotted the "Pinching" gesture when the thumb and index fingertips came close together. It labelled the gesture correctly and made the fingertips stand out in the mask window.



```
Hue Min: 0
Hue Max: 179
Sat Min: 0
Sat Max: 255
Val Min: 0
Val Max: 178
```

Figure 113:Pinching



```
Hue Min: 0
Hue Max: 179
Sat Min: 0
Sat Max: 255
Val Min: 0
Val Max: 178
```

Figure 114: Rotation

Rotation Gesture: The system was able to detect the complex 'Rotation' gesture correctly. It recognized the specific finger orientation needed for this gesture. This suggests the pattern recognition algorithm works great.

DISCUSSION

Challenges and Limitations: The more complex set of hand gestures recognized by my system is prone to high ambiguity when one tries to recognize one gesture where similar gestures share common features, hence hindering accurate classification. Thus, skin colour filtering would rely on the performance variation with different skin and light colour. Moreover, the present setup can only detect gestures from a single hand, a situation that would not be feasible for applications involving interactions with both hands.

Approach and Comparative Analysis: Wherever I have done a comparative analysis, the approach of the implementation of the filter based on skin colour and detection of gestures through contours was quite effective for different gestures but comparably less effective if gestures somehow look alike in appearance. This approach is evidently far from methods analysing gestures with time series, which otherwise can differentiate one gesture from another based-on movement pattern in time, thereby possibly yielding better results in discriminating similar gestures.

Technical/Performance: Trackbars for HSV (Hue, Saturation, Value) colour space let us adjust settings. This was key for accurate skin detection. Skin detection was crucial for segmentation. The mask views helped ensure the hand was isolated from the rest of the image. Segmentation is very important for analysing gestures. The system was highly accurate for the given gestures. Its performance was limited by OpenCV's capabilities and camera resolution. Adjusting HSV values with trackbars improved accuracy. This lets the system adapt to different skin tones and lighting.

Improvements and Alternatives: Gesture recognition accuracy could be enhanced if deep learning methods, which extract feature learning from huge volumes of annotated data for each specific gesture, are included. This would also include temporal gestures in the analysis, where movements like rotation can be found from a sequence of frames rather than from static images. On the other hand, the system can use the hybrid model to combine skin colour detection and motion tracking to improve varied lighting and different colours of the skin.

TASK 2

INTRODUCTION

This part of the report describes Task 2, which involves creating or using a premade AI model that can detect hand gestures like grabbing, pinching, rotation, and thumbs up. For this the team came to mutual agreement on using Yolo which is well-known for its object detection speed and accuracy. In this case, the AI needs to provide an accurate count of the number of fingers in the given image. Before the we go to the overview of task2 let's talk a bit about what yolo is.

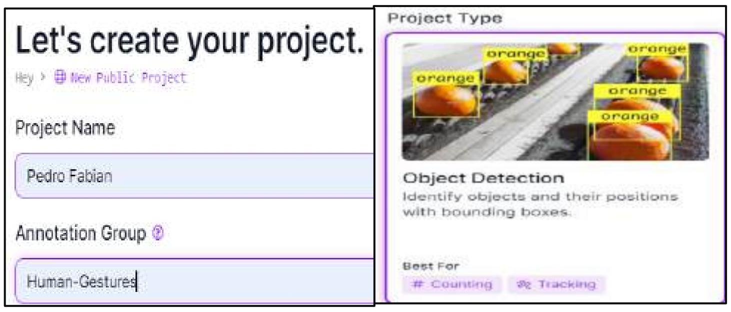
The YOLO algorithm, whose full form is "You Only Look Once," is an innovative approach to the problem of object detection as it does the detections during a single neural network passing. This is different from the traditional techniques which required numerous stages to detect and classify the objects. In YOLO, the object detection process is framed as a regression issue to one large execution, allowing arbitrary bounding box and rendering probability calculation. The most recent release in the YOLO series is YOLOv8, which provides a major enhancement in efficiency and performance. This model has been optimized to handle a higher range of objects, variant sizes of bounding boxes, and address difficulties like occluded objects and objects with intersections. YOLOv8 is designed to be easily scaled and efficient for edge computing scenarios with constrained computational resources.

OVERVIEW OF TASK 2

The whole of task 2 consist of three main steps which will be explained in detail in the report. The steps consist of data collection, annotation, training and then testing. So, the data is in form of images. Then the image is annotated which then later used to train the model which is Yolo V8. Then the model trained is then used to test to see if the prediction is correct. So next part will be explaining the process from data collection till the model being trained until the results from the testing.

PROCEDURES IN ROBOFOW

The process of starting a project in Roboflow is straightforward: the user is to define a project name and specify the annotation group type corresponding to the dataset. In this example, the project name is "Pedro Fabian," and it is under the group "Human-Gestures," which is indicative of the fact that it deals with an interpretation involving movements given from the hand of a human.


Let's create your project.

Hey > New Public Project

Project Name: Pedro Fabian

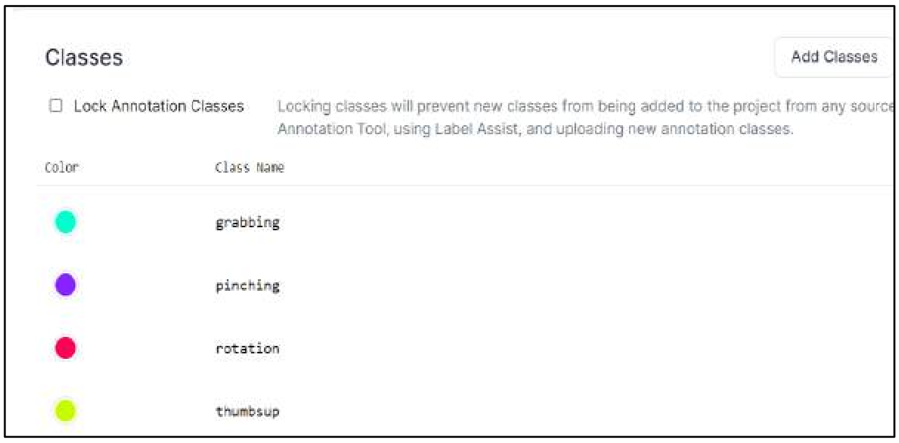
Annotation Group: Human-Gestures

Project Type: Object Detection — Identify objects and their positions with bounding boxes.

Figure 115: Creating a project

Classes for Hand Gestures

The definition of categories is another important step of the "Pedro Fabian" project. It represents the gestures that will represent different types. The categories, which were quite carefully chosen to represent different gestures, include grabbing, pinching, rotation, and thumbs up. We then must develop classes that encapsulate the basic knowledge of a computer vision model and can recognize different gestures.


Classes

Add Classes

Lock Annotation Classes — Locking classes will prevent new classes from being added to the project from any source Annotation Tool, using Label Assist, and uploading new annotation classes.

Color	Class Name
●	grabbing
●	pinching
●	rotation
●	thumbsup

Figure 116: Classes Hand Gesture

Annotation and Labelling Process

In the next step, these images are annotated very carefully inside the Roboflow interface. Bounding boxes are drawn around the gestures in each picture, with different classes using distinct colours. That visual cue becomes a part of the training data for the model, acting as the 'ground truth' for the model in knowing how to locate and recognize the set of defined gestures.

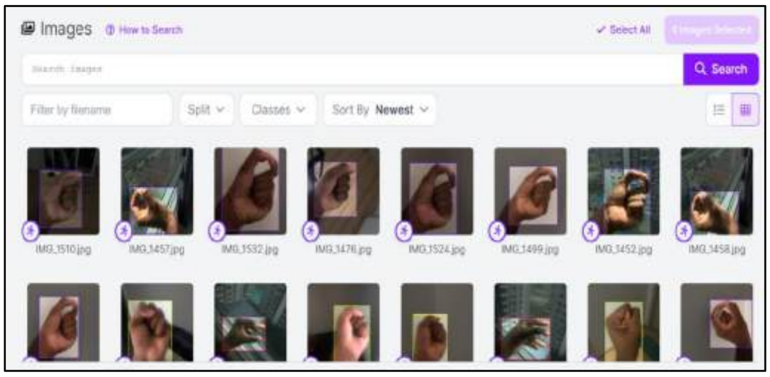


Figure 117:Annoated Data Set

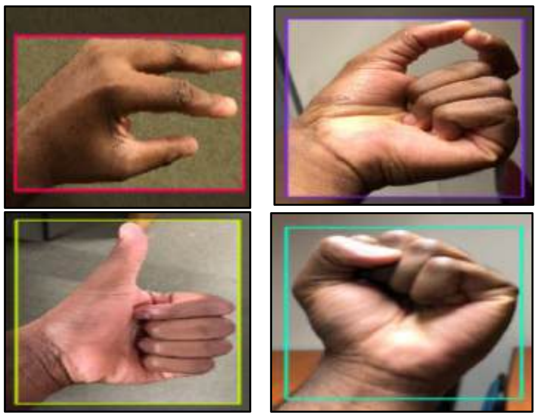


Figure 118 Annotating

Exporting the Annotated Data

The annotations with a corresponding configured file for training the model are then exported in the YOLOv8 format. It has good integration with the YOLOv8 architecture, where in return, both performance and accuracy in object detection are very high.

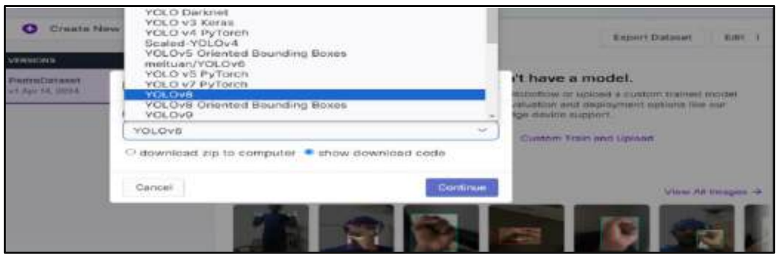
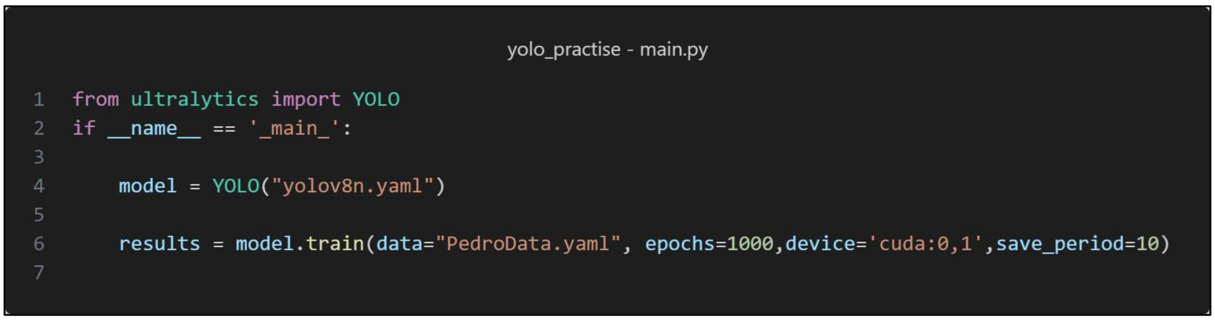


Figure 119:Exporting the Annotated Images

MODEL TRAINING

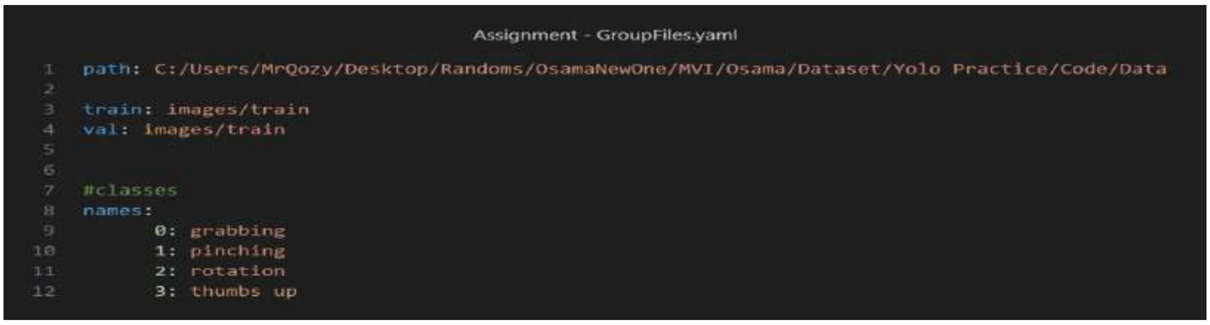


yolo_practise - main.py

```python
from ultralytics import YOLO
if __name__ == '_main_':

    model = YOLO("yolov8n.yaml")

    results = model.train(data="PedroData.yaml", epochs=1000,device='cuda:0,1',save_period=10)

```

Figure 120: Training code

- The code starts by getting the YOLO class from ultralytics. Ultralytics has YOLO object detection models. Next, it checks if the code runs directly or is imported. Code inside this part only runs when executed directly. Then, it creates a YOLO class instance. It uses the yolov8n.yaml file for configuration like model details. After that, it starts training the YOLO model. It uses data and settings from PedroData.yaml. The training runs for 1000 epochs on CUDA device 0 (first GPU). It saves checkpoints every 10 epochs.



Assignment - GroupFiles.yaml

```yaml
1   path: C:/Users/MrQozy/Desktop/Randoms/OsamaNewOne/MVI/Osama/Dataset/Yolo Practice/Code/Data
2
3   train: images/train
4   val: images/train
5
6
7   #classes
8   names:
9         0: grabbing
10        1: pinching
11        2: rotation
12        3: thumbs up
```

Figure 121: Data yaml file

- The root folder path stores the training and validation data (line 1). The training images are in a subfolder of the root. (3 and 4). These lines define the class names and indices. For example, class 0 is grabbing, 1 is pinching, and so on. The YOLO model learns to detect and differentiate the same classes (line 9 until 12)

GRAPH RESULTS

train/box_loss, train/cls_loss, train/dfl_loss:

These graphs track the training losses for the bounding box regression, classification, and distribution focal loss. All three losses start relatively high with much fluctuation, dropping over the first 200 epochs and then tending slowly down. The losses of Box and CLS converge close to 0, while the loss of DFL reaches a minimum value of around 1. It shows good training. Though raw data has some noise, overall trends appear on smooth lines.

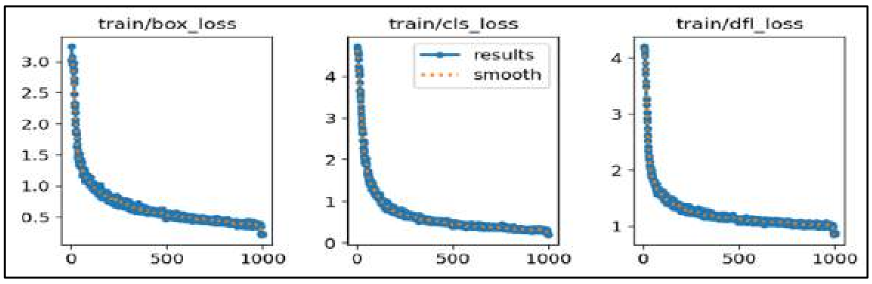


Figure 122: graphs train and loss

Val/box,val/cls, val/dfl loss

These track the same three losses as above but are evaluated on the validation set during training. Validation losses decreased in a curve like training losses across epochs. The validation losses are a bit higher than the training losses, as one would expect; they also reach low values, but not as low as the training ones, consolidating good generalization. The gap between small training and validation loss shows that there is no serious overfitting.

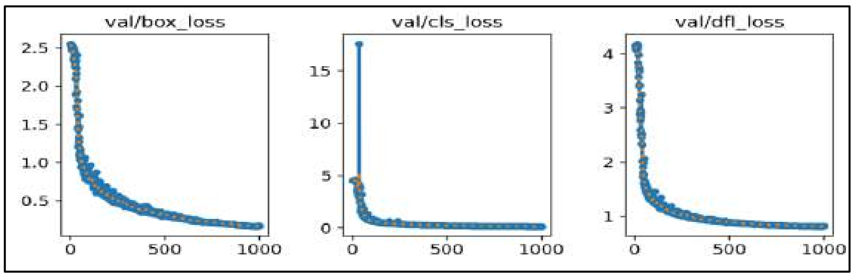


Figure 123: graphs val/box, val/cls loss, val/dfl loss

metrics/precision(B), metrics/recall(B):

These metrics for precision and recall are likely evaluated at an IoU threshold of 0.5 on the validation set during training. Precision and recall start relatively low at around 0.2, but both start increasing fast and reach good values—over 0.8 already around 500 epochs. Precision is slightly higher than recall, indicating a low false positive rate relative to the false negatives. High final values for both metrics point to the excellent model performance in accurately detecting hand gestures.

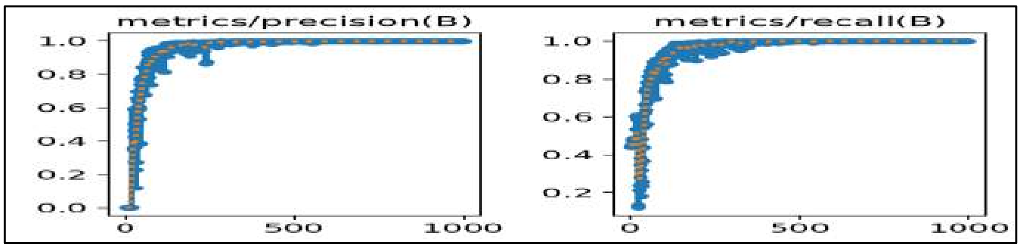


Figure 124: Graphs metrics & recall

metrics/mAP50(B), metrics/mAP50-95(B):

These metrics report the mean Average Precision (mAP) at IoU=0.5 averaged over classes. mAP0.5 starts very low but quickly improves and achieves a tremendous final value at around 0.95. mAP from 0.5 to 0.95 shows a similar trend but with lower values because it includes more challenging high IoU thresholds. Still, it gets a final solid value near 0.8. Meanwhile, the mAP graphs indicate the qualitative characteristic of this model's robust detection performance across all quality thresholds that simultaneously consider precision and recall.

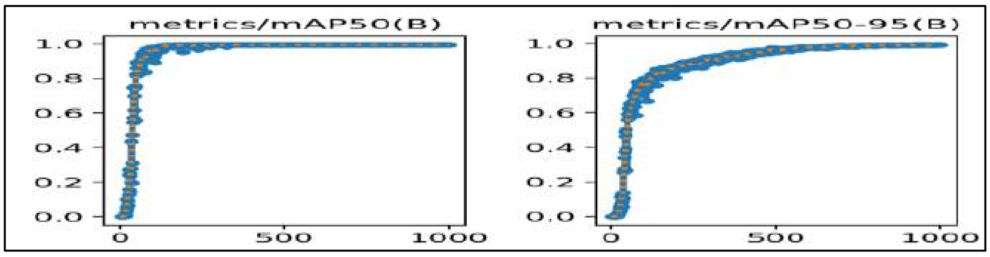


Figure 125: metrics mAP50 graphs

Number of Instances per Class (bar graph):

This bar graph displays the number of samples for each hand gesture class in the dataset. The class "Pinch Hand" has the most instances, followed by "Rotation_Hand" and "Grab_Hand. The imbalanced class distribution indicates that more training data are available for some gestures than others, which could affect the model's performance on underrepresented classes. Class imbalance is a critical factor to consider as it can lead to biased models that perform better on overrepresented classes.

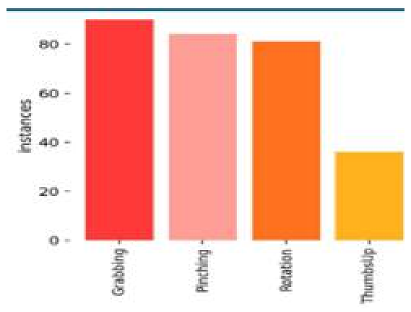


Figure 126: Instances_bar

Confidence Distribution (scatterplot):

This scatterplot shows the density of instances in the distribution of predicted confidences. The x-axis represents the confidence score, while the y-axis indicates the density of instances at each confidence level. Color-coded points represent each instance, showcasing the model's confidence in its predictions. Most instances exhibit high predicted confidence, clustering towards the right side of the plot (confidence > 0.8), which demonstrates strong assurance in the model's classifications for most of the data. However, a considerable number of instances with low confidence are scattered on the left side of the plot, likely indicating uncertainty or difficulty in classifying certain hand gestures. This could be due to ambiguous gestures, occlusion, or the model's poor handling of variations in hand appearance.

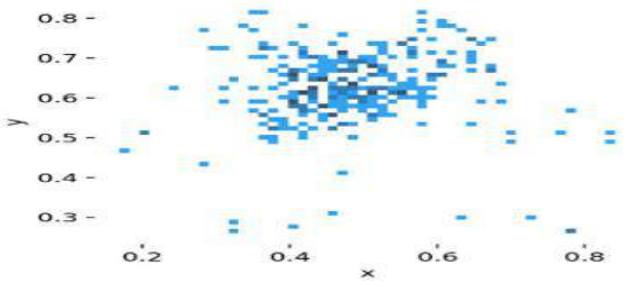


Figure 127: Confidence distribution graph

Width-Height Scatterplot (inset):

This plot illustrates the relationship between the width and height of detected hand regions in the images. There is a strong positive correlation between width and height, indicating that hand regions typically maintain similar aspect ratios across the dataset. The clustering of points suggests that the hand regions generally fall within a consistent size range, which could be useful for further analysis or post-processing steps. A few outliers with larger widths or heights might represent more challenging or unusual hand appearances that the model needs to accommodate.

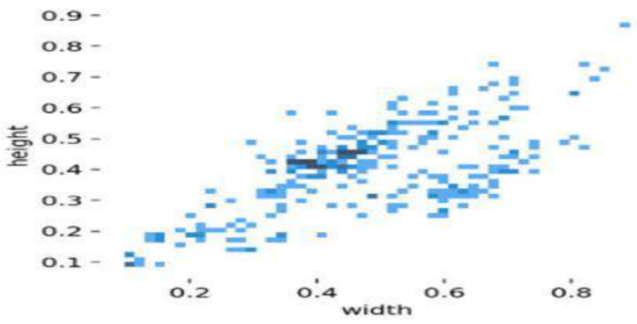


Figure 128: Width-Height Scatterplot graph

F1 Confidence curve

The F1-Confidence curve indicates a model that provides a consistent F1 score for all gesture classes across all confidence thresholds. The uniform nature of the model indicates that the model is robust and can detect the gestures uniformly irrespective of the fingers displayed or the occlusion that may cause false positive detections.

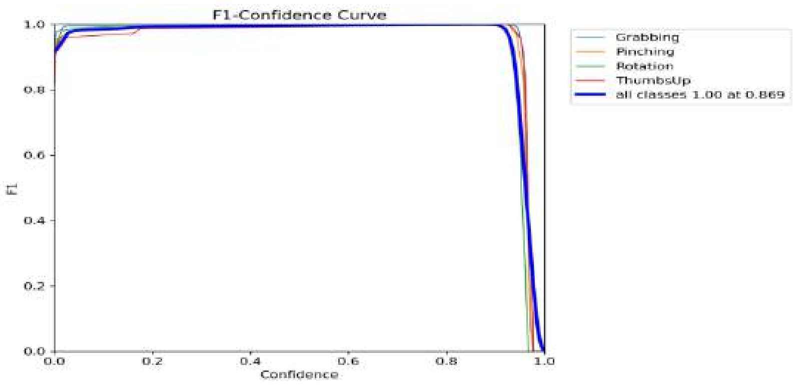


Figure 129: Confidence curve

RESULTS

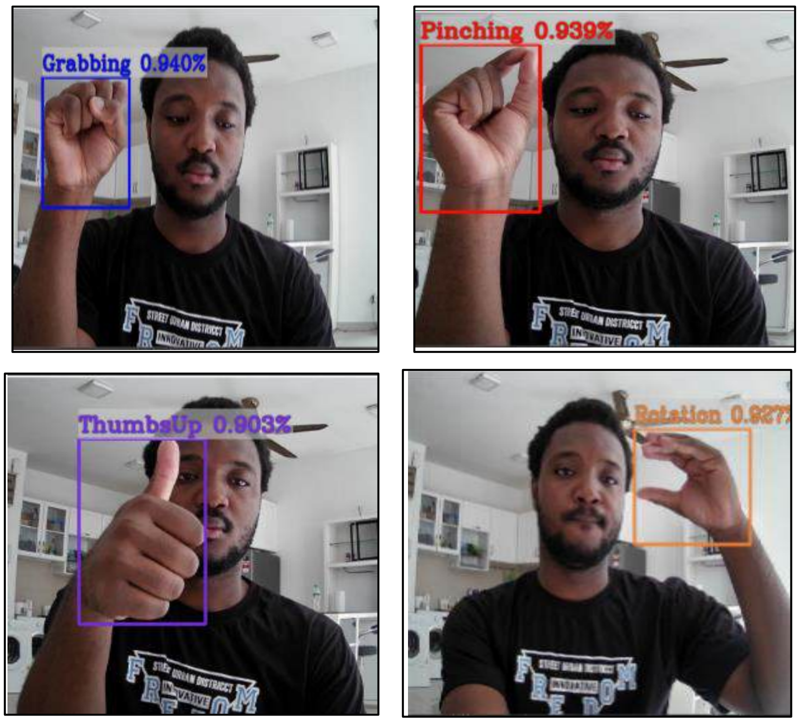


Figure 130;Detected Results

- The images above are from testing the model. Above each bonding box for each image is the confidence level of the detection:

1. Grabbing: 0.940 or 94%.
    a. Pinching: 0.939 or 93.9%
    b. Thumbs up: 0.903 or 90.3%
    c. Rotation: 0.927 or 92.7%

- The confidence level for Hand Gesture count is close or almost 1. This helps to say that the model was trained successfully, and it can detect and classify the finger count. This can also be because of no overfitting of data. Another evidence to support the accuracy of the model is its ability to differentiate between the gestures grabbing and pinching since they both have similar size and look.

DISCUSSION

The design of such a system for recognizing hand gestures introduces some challenges, which in most cases include complexities concerning gesture variance and environmental conditions. The project utilized the YOLO (You Only Look Once) object detection framework with PyTorch to capture four different hand movements: grabbing, pinching, rotation, and thumbs-up gestures. High-resolution pictures have initially faced significant challenges to the system in the way they affect efficiency. This has been overcome by reducing the size of the images so that the training may be much faster.

Concerning this case, the greatest challenge proved to be the imbalance brought about in the training dataset. That is to say, the 'thumbs up' gesture was highly underrepresented in numbers, with only 34 images out of 283 images. Nonetheless, the recognition capability was good. This effectiveness had been attributed to the careful choice of qualitative, representative images for an underrepresented gesture, such that even less could give a reliable performance in detecting training data.

For better concentration on the relevant features, contour-focused and grayscale application techniques that the Roboflow platform offers were applied during the training session. The last one or grayscale application has been implemented to decrease the possibility of the background becoming a nuisance by eliminating color information. However, in its put-to-use, it was not exactly quite the hit that it was supposed to be—the contrast offered between hand and background was not marked enough to have isolated the gesture effectively, hence recognition failed. This highlighted the limited applicability of simple grayscale processing under some conditions, and it emphasized the urgent need for developing some adaptive background segmentation techniques that can adaptively change in the varying lighting and color contrasts.

Setting the labels was also an essential step in Roboflow that required accurate adjustments so that the labels enclosed the area of interest effectively—the hand that displays the gesture. Proper labeling is essential, as it directly influences the training accuracy of the model, which in turn directly influences the ability for appropriate identification and classification of gestures in real-time applications.

This made the model training, and it expanded its application because of the simple reason that the system is flexible in recognizing gestures from either hand without conditioning on hand orientation or position. Training the model to detect gestures across both orientations of the hands, irrespective of the hand used for detection, the developed system thus assures robust performance in diversified real-world scenarios.

The twin-GPU setup, for example, brings the training time down significantly, thus affording the researchers enough room to experiment widely and have even faster iteration cycles. Proper use of computation resources further underscores the benefits parallel processing brings in the management of such computationally resource-intensive tasks like real-time hand gesture recognition.

This, therefore, points to an iterative development process of an effective hand gesture recognition system that underlines the balance between quality and quantity of training data, the importance of effective preprocessing techniques, and the advantages that are promised by using advanced computational resources in coping with the real-time requirements of image processing.

TASK 3

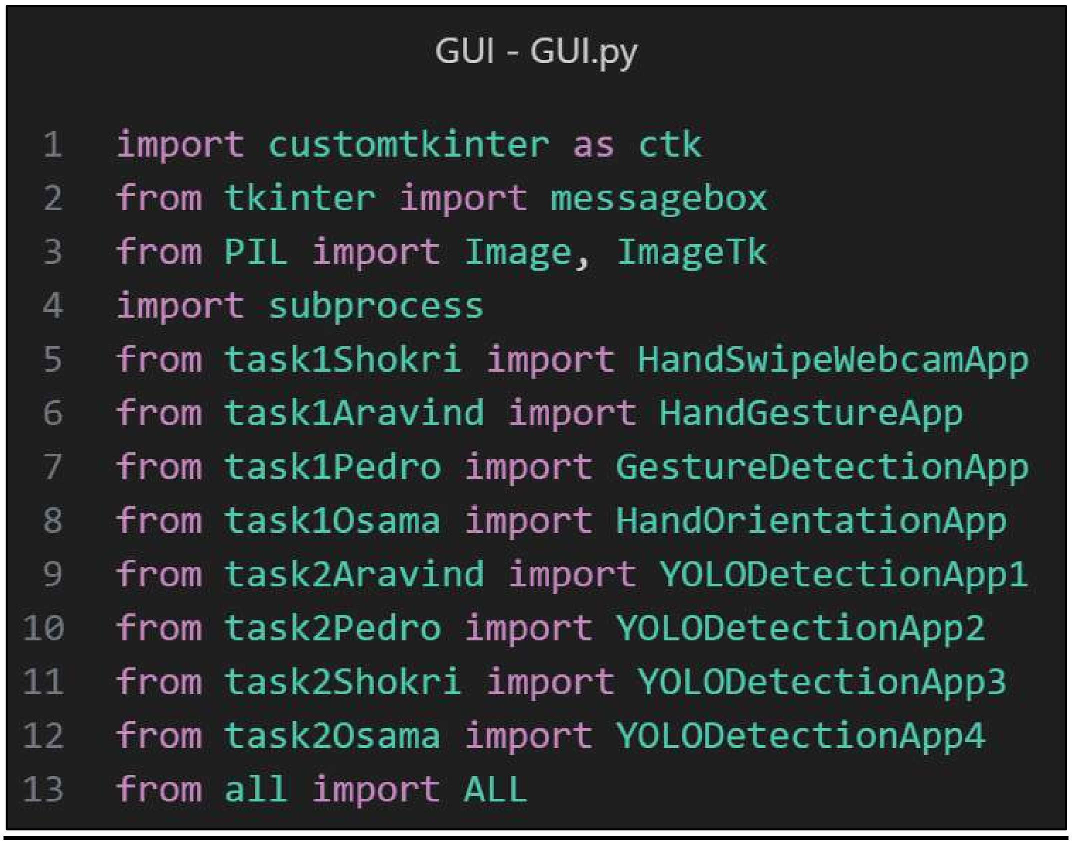
Introduction

Task 3 involved combining the task 1 and task 2 of all the members and display them in a Gui format. so, for the creation of the Gui Tkinter library was used to design the layout of the Gui and. OpenCV was used later to link the frames in the Gui. Below is the step-by-step process of the Gui and the coding structure.

<u>Overview of the GUI</u>

The qui firstly allows the user to start the individual task for task 1 and 2 for each member. It also allows for seamless transition between the members. Below is the coding structure of the Gui.

Design Code Explanation



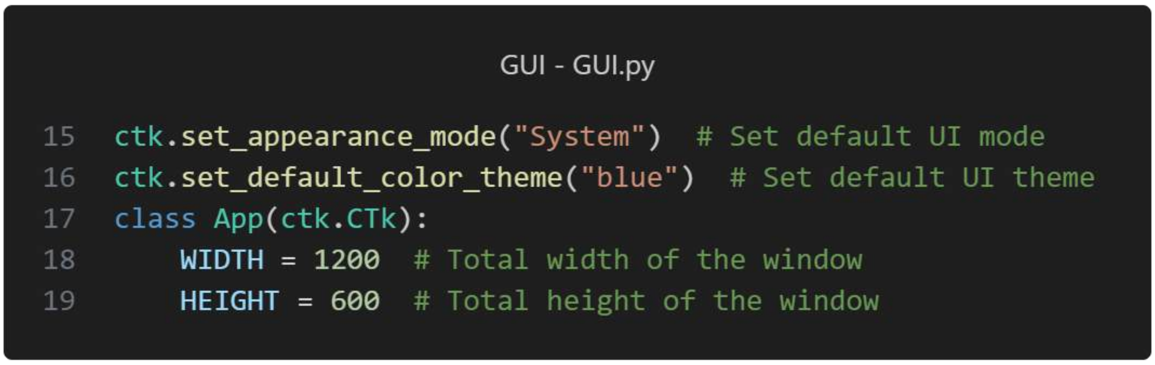
GUI - GUI.py

```python
import customtkinter as ctk
from tkinter import messagebox
from PIL import Image, ImageTk
import subprocess
from task1Shokri import HandSwipeWebcamApp
from task1Aravind import HandGestureApp
from task1Pedro import GestureDetectionApp
from task1Osama import HandOrientationApp
from task2Aravind import YOLODetectionApp1
from task2Pedro import YOLODetectionApp2
from task2Shokri import YOLODetectionApp3
from task2Osama import YOLODetectionApp4
from all import ALL
```

Figure 131 Gui Libraies

The image above shows the initializations of every team member task for the GUI. Example: “from task1Shokri import HandSwipeWebcamApp. This will import hand swipe for task 1 and can be used when the user selects it.

Libraries were imported like customtkinter and Tkinter which allows in the design and creation of the app’s widget. For example: Adding buttons, Adding frames, Adding background and etc.

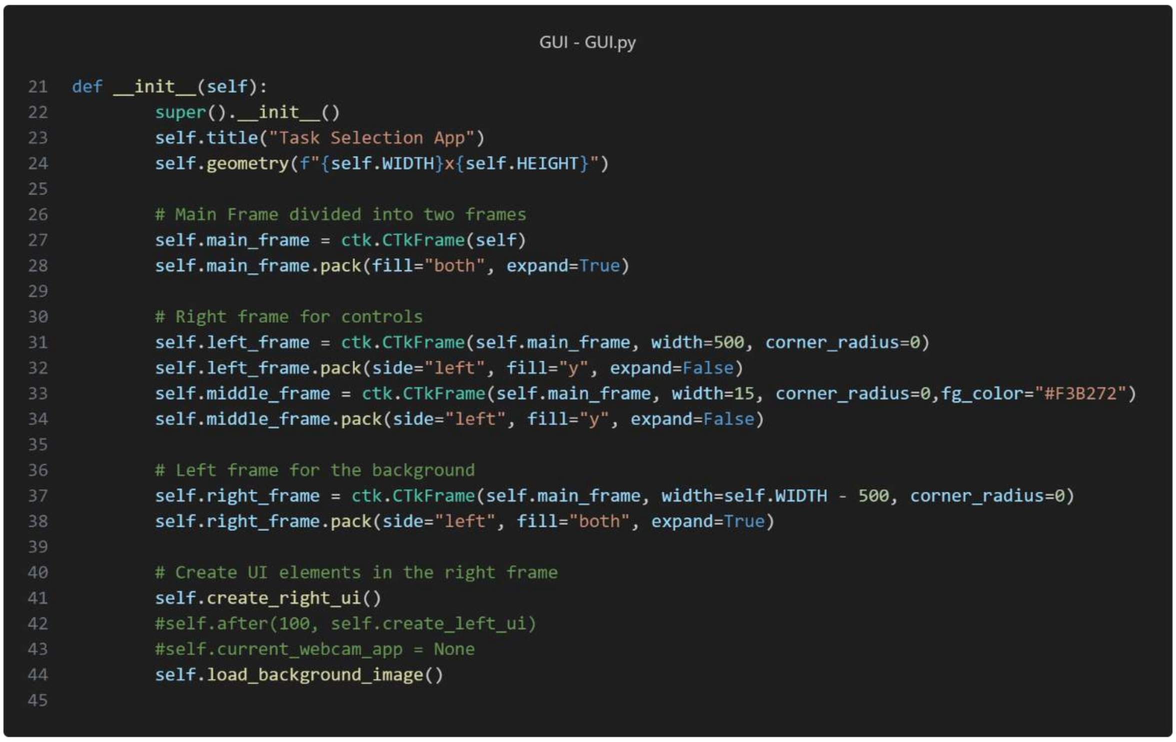


GUI - GUI.py

```python
ctk.set_appearance_mode("System")  # Set default UI mode
ctk.set_default_color_theme("blue")  # Set default UI theme
class App(ctk.CTk):
    WIDTH = 1200  # Total width of the window
    HEIGHT = 600  # Total height of the window
```

Figure 132 Gui Width and Height

Firstly, the width of the app is set. Our team decided on width of 1200 and a height of 600.



GUI - GUI.py

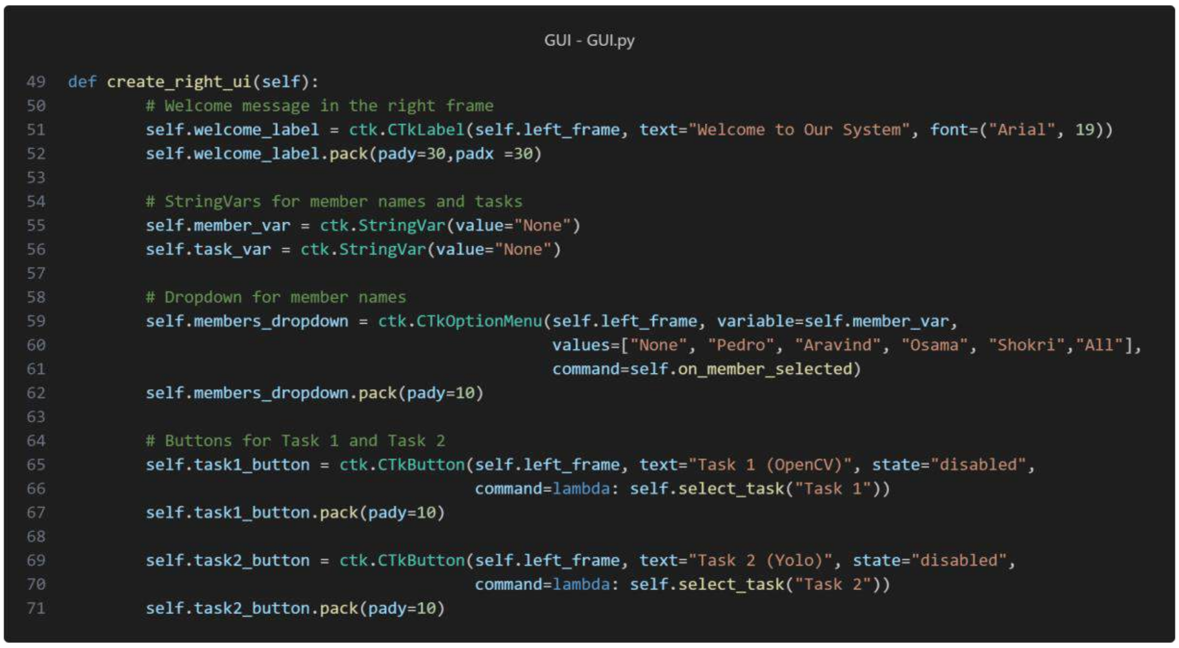
```python
def __init__(self):
        super().__init__()
        self.title("Task Selection App")
        self.geometry(f"{self.WIDTH}x{self.HEIGHT}")

        # Main Frame divided into two frames
        self.main_frame = ctk.CTkFrame(self)
        self.main_frame.pack(fill="both", expand=True)

        # Right frame for controls
        self.left_frame = ctk.CTkFrame(self.main_frame, width=500, corner_radius=0)
        self.left_frame.pack(side="left", fill="y", expand=False)
        self.middle_frame = ctk.CTkFrame(self.main_frame, width=15, corner_radius=0,fg_color="#F3B272")
        self.middle_frame.pack(side="left", fill="y", expand=False)

        # Left frame for the background
        self.right_frame = ctk.CTkFrame(self.main_frame, width=self.WIDTH - 500, corner_radius=0)
        self.right_frame.pack(side="left", fill="both", expand=True)

        # Create UI elements in the right frame
        self.create_right_ui()
        #self.after(100, self.create_left_ui)
        #self.current_webcam_app = None
        self.load_background_image()

```

Figure 133 Def init Function

The appearance of the Gui was set to System and the theme of the of the app was set to blue.

This main function of the Gui app. This where all the widgets like the buttons, frames and images are called. So, when the program runs all the widgets are loaded into the frame.

The use of custom tkinter is visible in this snippet, which is being used to call in the frames of the app.


GUI - GUI.py

```python
def create_right_ui(self):
        # Welcome message in the right frame
        self.welcome_label = ctk.CTkLabel(self.left_frame, text="Welcome to Our System", font=("Arial", 19))
        self.welcome_label.pack(pady=30,padx =30)

        # StringVars for member names and tasks
        self.member_var = ctk.StringVar(value="None")
        self.task_var = ctk.StringVar(value="None")

        # Dropdown for member names
        self.members_dropdown = ctk.CTkOptionMenu(self.left_frame, variable=self.member_var,
                                                  values=["None", "Pedro", "Aravind", "Osama", "Shokri","All"],
                                                  command=self.on_member_selected)
        self.members_dropdown.pack(pady=10)

        # Buttons for Task 1 and Task 2
        self.task1_button = ctk.CTkButton(self.left_frame, text="Task 1 (OpenCV)", state="disabled",
                                          command=lambda: self.select_task("Task 1"))
        self.task1_button.pack(pady=10)

        self.task2_button = ctk.CTkButton(self.left_frame, text="Task 2 (Yolo)", state="disabled",
                                          command=lambda: self.select_task("Task 2"))
        self.task2_button.pack(pady=10)
```

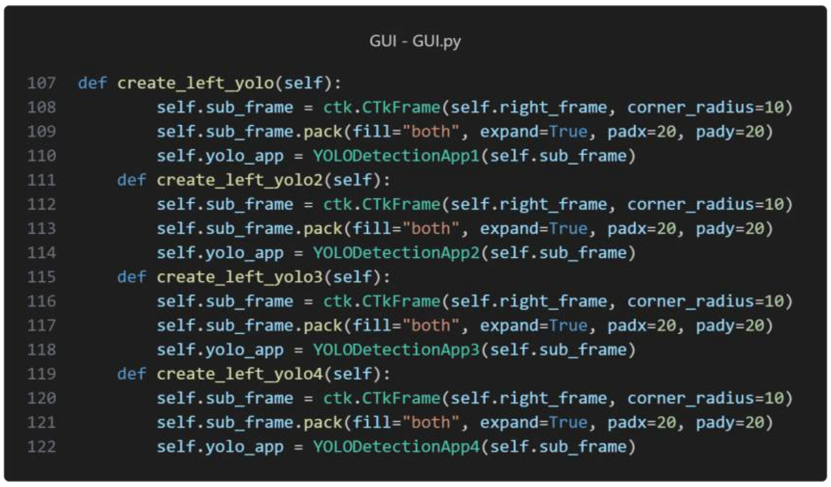
Figure 134: Create Right UI Function

This function is in charge with dealing with all the widgets on the frames which is on the left.

Dropdown menu is being used in the Gui. this allows users to choose from the name list provided. Example: If “ALL” is chosen from the dropdown list then the user will have access to the combined trained model of all the members.

Buttons for task 1 and task 2 were set, which can be used to choose from task 1 or task 2. So, if task 1 is chosen then task 1 will run and vice versa.

Functionality Code Explanation

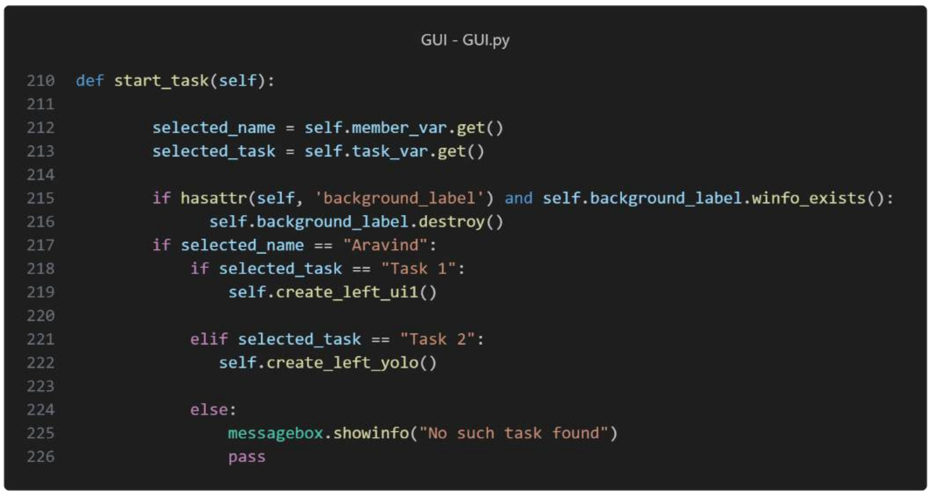

GUI - GUI.py

```python
def create_left_yolo(self):
        self.sub_frame = ctk.CTkFrame(self.right_frame, corner_radius=10)
        self.sub_frame.pack(fill="both", expand=True, padx=20, pady=20)
        self.yolo_app = YOLODetectionApp1(self.sub_frame)
    def create_left_yolo2(self):
        self.sub_frame = ctk.CTkFrame(self.right_frame, corner_radius=10)
        self.sub_frame.pack(fill="both", expand=True, padx=20, pady=20)
        self.yolo_app = YOLODetectionApp2(self.sub_frame)
    def create_left_yolo3(self):
        self.sub_frame = ctk.CTkFrame(self.right_frame, corner_radius=10)
        self.sub_frame.pack(fill="both", expand=True, padx=20, pady=20)
        self.yolo_app = YOLODetectionApp3(self.sub_frame)
    def create_left_yolo4(self):
        self.sub_frame = ctk.CTkFrame(self.right_frame, corner_radius=10)
        self.sub_frame.pack(fill="both", expand=True, padx=20, pady=20)
        self.yolo_app = YOLODetectionApp4(self.sub_frame)
```

Figure 135: Create YOLO

The snippet provided above shows some of the functions that is used to give the flow of the code. In the snippet provided the function are linked to the task 2 yolo models of each member.

This function will be called when the buttons are pressed. the individual member tasks are loaded and packed into a sub frame which is on the right side of the screen.



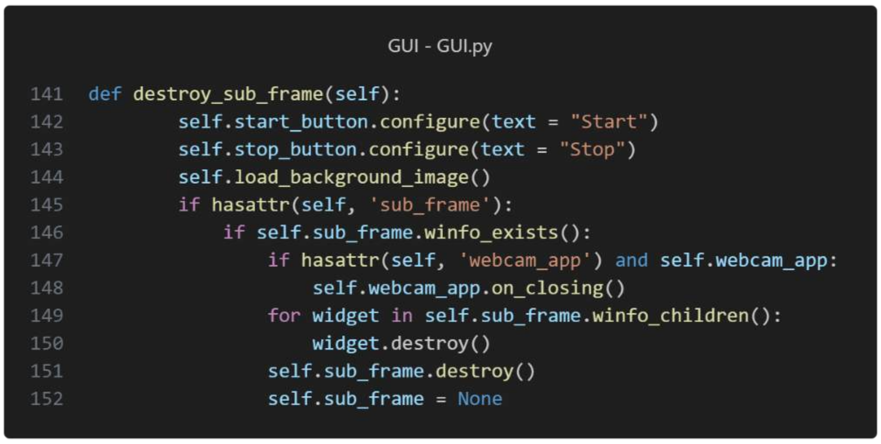
GUI - GUI.py

```python
def start_task(self):

        selected_name = self.member_var.get()
        selected_task = self.task_var.get()

        if hasattr(self, 'background_label') and self.background_label.winfo_exists():
            self.background_label.destroy()
        if selected_name == "Aravind":
            if selected_task == "Task 1":
                self.create_left_ui1()

            elif selected_task == "Task 2":
               self.create_left_yolo()

            else:
                messagebox.showinfo("No such task found")
                pass
```

Figure 136 Start task

The snippet above the main function which controls the flow of the code. So, when the name from the drop list is picked and the type of task is selected and when the start button is pressed this function is called.

Using simple if and else statements we can control which task gets displayed. So, if “Osama” and “Task1” is picked, and the start button is pressed Osama’s task 1 will run. The code just runs the function for that task based on the task picked in the Gui app.



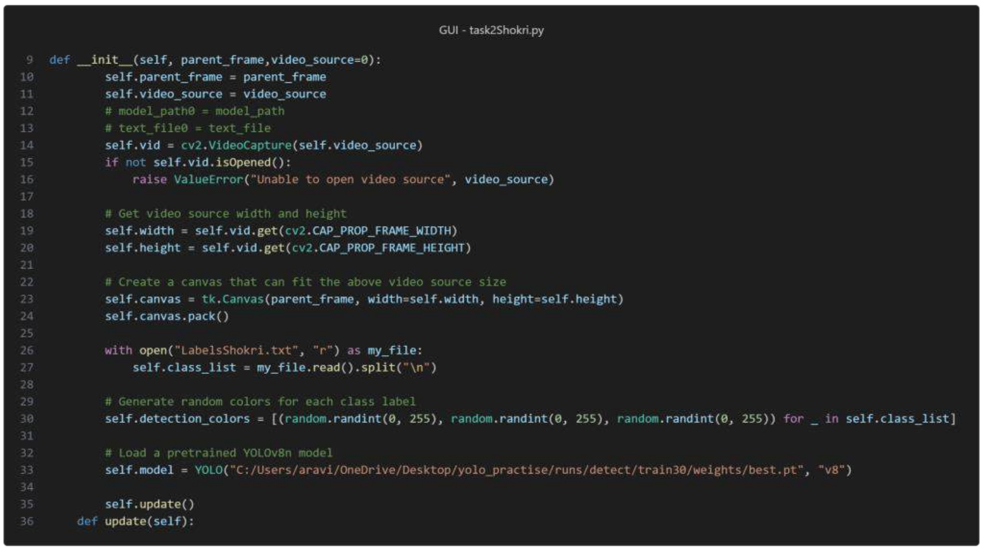
GUI - GUI.py

```python
def destroy_sub_frame(self):
        self.start_button.configure(text = "Start")
        self.stop_button.configure(text = "Stop")
        self.load_background_image()
        if hasattr(self, 'sub_frame'):
            if self.sub_frame.winfo_exists():
                if hasattr(self, 'webcam_app') and self.webcam_app:
                    self.webcam_app.on_closing()
                for widget in self.sub_frame.winfo_children():
                    widget.destroy()
                self.sub_frame.destroy()
                self.sub_frame = None
```

Figure 137: Destroy Frames

The code snippet above shows what happens when the stop button is pressed. when the task for the members run it opens a open cv frame. So, to open another task without interfering with each other.

The code checks to see if there is a sub frame still in the Gui. If there is the sub frame is destroyed and when the new task is called a new sub frame is created. This function also helps to reset the start and stop button text.

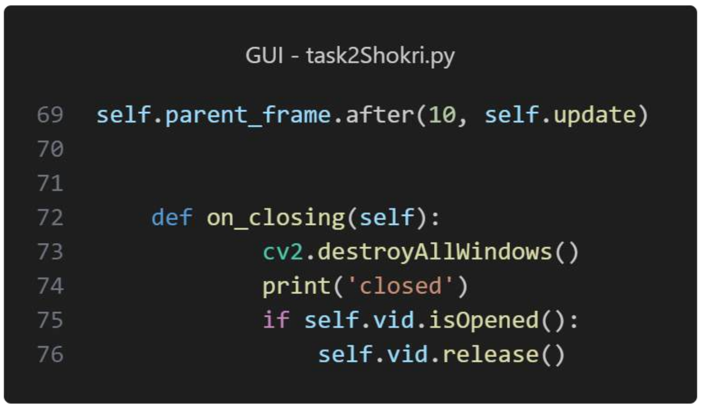


GUI - task2Shokri.py

```python
def __init__(self, parent_frame,video_source=0):
        self.parent_frame = parent_frame
        self.video_source = video_source
        # model_path0 = model_path
        # text_file0 = text_file
        self.vid = cv2.VideoCapture(self.video_source)
        if not self.vid.isOpened():
            raise ValueError("Unable to open video source", video_source)

        # Get video source width and height
        self.width = self.vid.get(cv2.CAP_PROP_FRAME_WIDTH)
        self.height = self.vid.get(cv2.CAP_PROP_FRAME_HEIGHT)

        # Create a canvas that can fit the above video source size
        self.canvas = tk.Canvas(parent_frame, width=self.width, height=self.height)
        self.canvas.pack()

        with open("LabelsShokri.txt", "r") as my_file:
            self.class_list = my_file.read().split("\n")

        # Generate random colors for each class label
        self.detection_colors = [(random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)) for _ in self.class_list]

        # Load a pretrained YOLOv8n model
        self.model = YOLO("C:/Users/aravi/OneDrive/Desktop/yolo_practise/runs/detect/train30/weights/best.pt", "v8")

        self.update()
    def update(self):
```

Figure 138: Attributes Task 2

This is the one of the members task2 file. The file is link to the main Gui using the Def init function which links the individual yolo app to the main Gui. the Hui acts as a parent frame and the individual code being the child frame.



GUI - task2Shokri.py

```python
self.parent_frame.after(10, self.update)


    def on_closing(self):
            cv2.destroyAllWindows()
            print('closed')
            if self.vid.isOpened():
                self.vid.release()
```

Figure 139: Closing_GUI

The def on_closing function is what allows the frames of each task to be destroyed allowing the next tsk to be displayed. This is allows shows how OpenCV can be used in Gui as well. In this scenenario it can be sued to close all the windows and frames of each task.

WORKING GUI

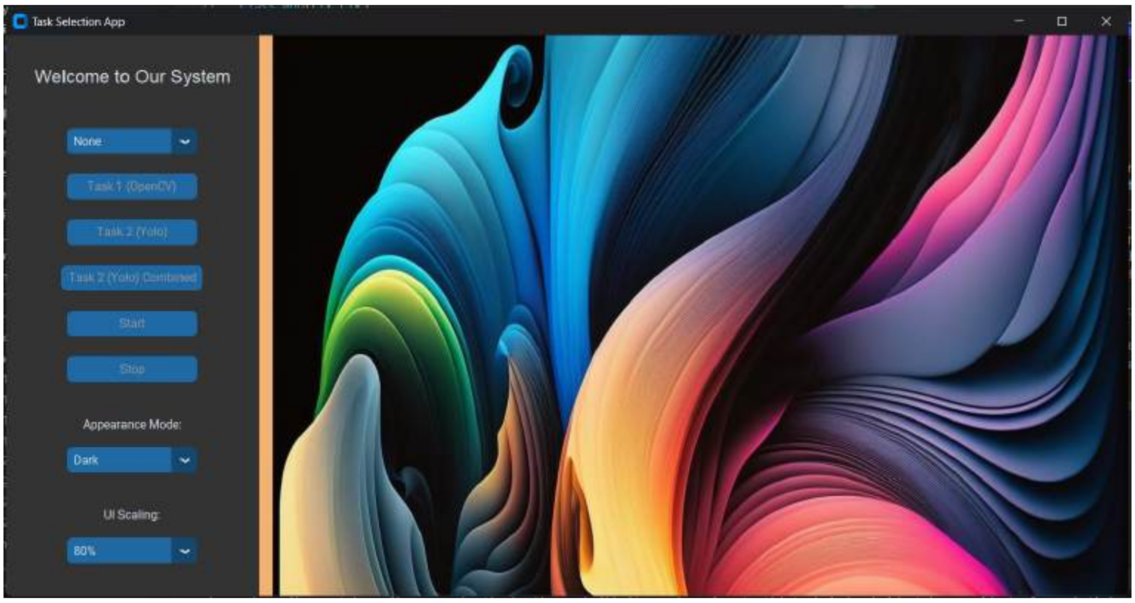


Figure 140: Main Screen GUI

This starting window of the combined Gui app. An image as been placed in the main frame to give a bit of design making the Gui stand out.

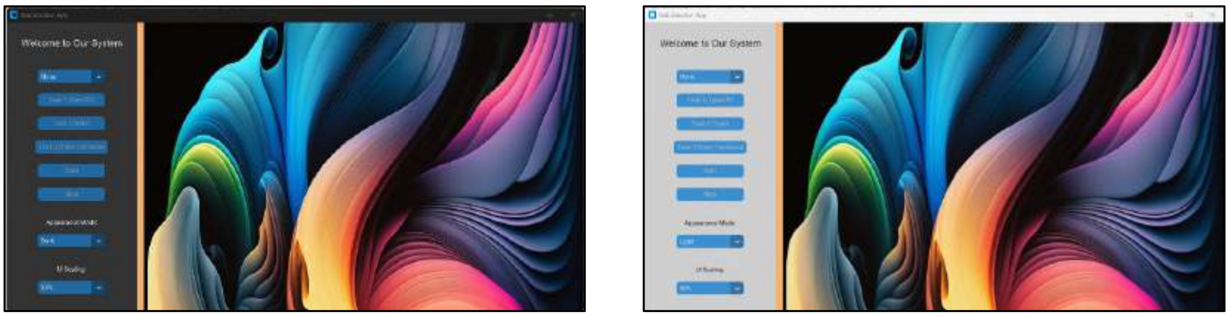


Figure 141:Gui Theme

The Gui also can change theme colour. It can change through two colours. The snippet of the Gui in three theme colours is shown below.The theme of changing the Gui using the same method as choosing the member's name which is the use of a drop-down list.

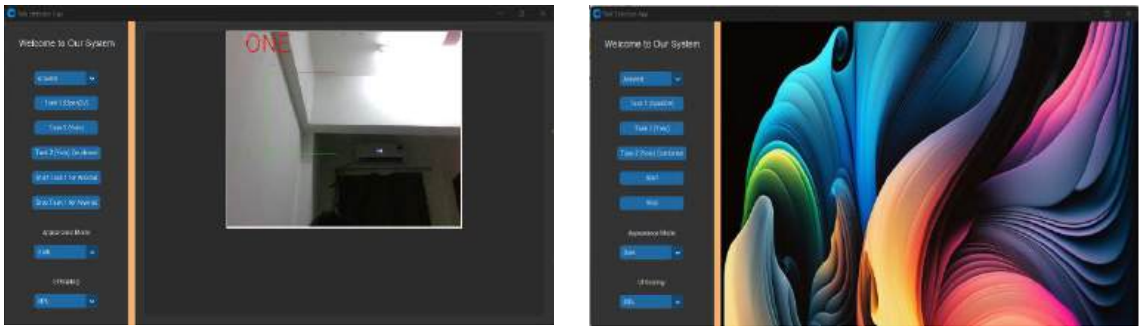


Figure 142: Finger Counting Frame_GUI

The image gets destroyed and when the start button is pressed and when the stop function is pressed the frame is closed and the image is placed back on the Gui. Overall this the main flow of the Gui of each member combined.

OVERALL DISCUSSION

General Discussion on Gesture Recognition System Group Project:

This project involved designing and developing a comprehensive gesture recognition system that includes functionalities like finger counting, hand motion, specific gestures, and hand orientation. Different team members collaborated, focusing on various aspects of the system using machine learning and image processing techniques. The primary goal was to create a robust and versatile system capable of operating with high accuracy and efficiency in real-world conditions.

Key Challenges and Methodological Approaches:

Data Collection and Preprocessing: Common challenges included acquiring and preprocessing the dataset. Issues encountered involved the quality and quantity of data, such as mislabeling, background variability, and inadequate representation of gestures. Efforts to improve dataset quality involved increasing image diversity and balancing the representation of gestures, along with resizing images to expedite training.

Model Training and Implementation: Models were trained and implemented using tools and frameworks like OpenCV, PyTorch, and YOLO. Projects benefited from CUDA-accelerated training, facilitating efficient handling of large datasets. Techniques like color segmentation, motion history, and optical flow were utilized to address gesture recognition challenges.

Integration and System Performance: Integrating various modules like finger counting, hand motion detection, and specific gesture orientations into a single system presented unique challenges. Each component needed optimization to function cohesively, requiring careful tuning of model parameters and threshold settings to maintain overall system performance.

Technological Insights and Improvements:

Adaptive Methods and Machine Learning: Background and lighting issues were addressed using adaptive background subtraction and machine learning techniques for hand segmentation, enabling real-time adjustments and more accurate hand and gesture detection.

Deep Learning Enhancements: Deep learning models significantly improved accuracy and robustness over traditional image processing methods, though they required more computational resources, illustrating a trade-off between performance and resource utilization.

GUI Design and System Implementation:

User Interface: A user-friendly graphical interface was designed to simplify interactions with the system, providing real-time feedback and adjustable settings, such as HSV value tuning, to enhance user experience and accessibility.

Combined Functionality: The final implementation integrated all individual functionalities into a unified system capable of detecting and interpreting multiple gestures and hand orientations simultaneously, suitable for advanced applications like VR, AR, and smart home systems.

Conclusion and Future Directions:

The project demonstrated that a sophisticated gesture recognition system is feasible through collaborative efforts and addressing various challenges. Incorporating more adaptive algorithms to better handle diverse environments and user conditions. Developing libraries to include more complex and subtle gestures, enhancing the system's applicability in advanced interaction scenarios. Reducing the computational demand without compromising accuracy, potentially through more efficient network architectures or edge computing solutions.