**SIGN LANGUAGE DETECTION SYSTEM**

**1.PROJECT OVERVIEW**

**Project Name:** Sign Language Detection System

**Project Description:** A deep learning system that uses OpenCV for video processing and a YOLO-based model to identify and annotate different sign language motions in real-time.

**Objectives:**

* Real-time sign language recognition
* High detection accuracy across environments
* Low latency for seamless interaction
* Scalability for additional gestures and languages
* Support for education across users

**2. Installation and Setup**

**Prerequisites:**

1. **Hardware**:

* Ensure the system has a compatible GPU (e.g., NVIDIA RTX 3050) with CUDA support for accelerated inference.

1. **Software Requirements:**
   * + - Python 3.7 or higher
       - PyTorch (ensure compatibility with CUDA if using a GPU)
       - OpenCV
       - YOLOv5 (with dependencies)
       - CUDA and can (specific to the GPU, for accelerated processing)
       - Visual Studio Code or similar

**Environment Setup:**

* 1. **Python Environment:**

It's recommended to use a virtual environment (e.g., conda or venv) to manage dependencies.

**Bash**

conda create -n sign\_detect python=3.8

conda activate sign\_detect

* 1. **Install Dependencies:**

**Bash**

pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu117 # Update the URL as per CUDA version

pip install opencv-python

* 1. **Clone YOLOv5 Repository:**

**Bash**

git clone https://github.com/ultralytics/yolov5.git

cd yolov5

pip install -r requirements.txt

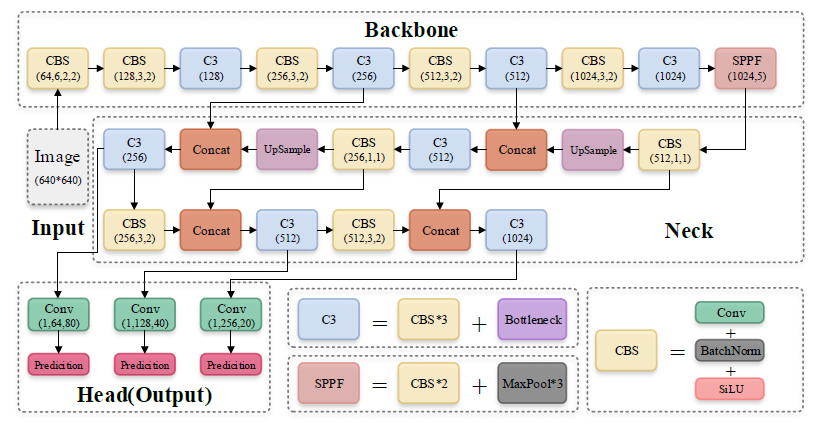
* 1. **Verify GPU Setup** (optional):

Ensure that PyTorch recognizes the GPU by running.

**Python**

import torch

print("CUDA available:", torch.cuda.is\_available())

1. **MODEL ARCHITECTURE:**
   * + **Model Choice**: Justify the choice of YOLOv5 for this project.
     + **Model Structure**: Describe the layers or modules (e.g., Conv, C3) in YOLOv5 that make it suitable for real-time detection.
     + **Customization**: Detail any modifications made to YOLOv5 to improve performance or adapt it for sign language detection.

**Components:**

* CBS: Convolution + Batch Normalization + SiLU (Swish) Activation
* C3: Cross Stage Partial Network (CSP)
* SPPF: Spatial Pyramid Pooling – Fast
* Concat: Concatenate
* UpSample: Upsampling
* Conv: Convolutional Layer

**4. Data Collection and Annotation**

**Step 1:** Capture the images

**CODE:**

**Python**

import cv2

import os

import uuid

import time

#image directory

IMAGES\_PATH="path\_to\_your\collectedImages"

labels=["hello","thanks","yes","no","iloveyou"]

number\_imgs=15

for label in labels:

    img\_path = os.path.join(IMAGES\_PATH, label)

    os.makedirs(img\_path, exist\_ok=True)  # Use makedirs to ensure intermediate directories are created

    cap = cv2.VideoCapture(0)

    print(f"Collecting images for {label}")

    time.sleep(5)

    for image\_num in range(number\_imgs):

        ret, frame = cap.read()

        if ret:

            # Construct the image name

            imagename = os.path.join(img\_path, f"{label}.{uuid.uuid1()}.jpg")

            cv2.imwrite(imagename, frame)

            cv2.imshow("frame", frame)

            time.sleep(2)

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

                break

        else:

            print(f"Failed to capture image {image\_num} for {label}")

cap.release()

cv2.destroyAllWindows()

A black background with white text

Description automatically generatedA person holding up his hand

Description automatically generated A person with his hands over his mouth

Description automatically generatedA person pointing his finger

Description automatically generatedA person holding his hand to his head

Description automatically generatedA person holding up his hand

Description automatically generated

**STEP 2:** ANNOTATION

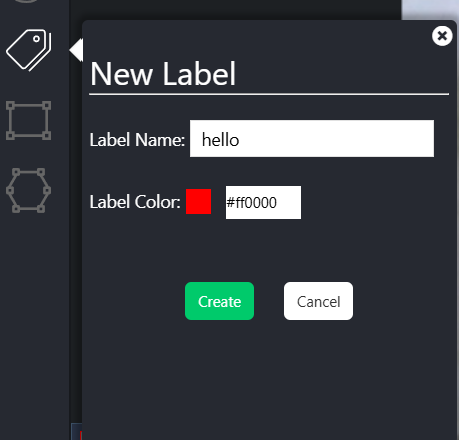
Annotation can be done using any tool of your choice. I have used **pixlab** here. <https://annotate.pixlab.io/>

1. Upload the images

A screenshot of a computer

Description automatically generated

1. Go to label manager and create 5 labels for hello, yes, no, thanks, iloveyou.

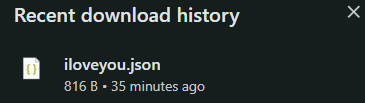


1. Annotate in the images

A person holding up a green screen

Description automatically generated

1. Download the annotations in json file.



1. Convert the json files to txt file using the below code.

**Python**

import json

import os

# Class mapping from the uploaded notebook

class\_mapping = {

'helllo': 0,

'iloveyou': 1,

'no': 2,

'thanks': 3,

'yes': 4

}

# Specify the input directory containing JSON files and the output directory for text files

input\_dir = '/path/to/json/files' # Replace with your directory path containing JSON files

output\_dir = '/path/to/output/txt/files' # Replace with your desired output directory

# Define image dimensions for normalization

image\_width = 640 # Replace with actual image width

image\_height = 480 # Replace with actual image height

# Ensure the output directory exists

os.makedirs(output\_dir, exist\_ok=True)

# Process each JSON file in the input directory

for filename in os.listdir(input\_dir):

if filename.endswith('.json'):

json\_path = os.path.join(input\_dir, filename)

# Read JSON data

with open(json\_path, 'r') as json\_file:

data = json.load(json\_file)

# Define output text file path

output\_file = os.path.join(output\_dir, filename.replace('.json', '.txt'))

# Write YOLO format annotations to text file

with open(output\_file, 'w') as txt\_file:

for item in data:

# Extract label and bounding box data

label\_name = item["labels"]["labelName"]

if label\_name in class\_mapping:

label\_id = class\_mapping[label\_name]

else:

print(f"Warning: Unknown label '{label\_name}' in file {filename}")

continue

# Get bounding box data and normalize

rect\_mask = item.get("rectMask", {})

x\_min = rect\_mask.get("xMin", 0)

y\_min = rect\_mask.get("yMin", 0)

width = rect\_mask.get("width", 0)

height = rect\_mask.get("height", 0)

# Calculate normalized values for YOLO format

x\_center = (x\_min + width / 2) / image\_width

y\_center = (y\_min + height / 2) / image\_height

norm\_width = width / image\_width

norm\_height = height / image\_height

txt\_file.write(f"{label\_id} {x\_center} {y\_center} {norm\_width} {norm\_height}\n")

print("Conversion complete for all JSON files in the directory!")

A screenshot of a computer

Description automatically generatedA person holding up his hand

Description automatically generatedThe label file for corresponding image will look like this:

**5.Organise the Dataset**

In the **yolov5** directory:

1. Locate a folder called **data** for your dataset.
2. Inside data, create two folders: **train** and **val**.
3. Inside each of these folders, create images and labels subfolders.
   * train/images should contain all training images.
   * train/labels should contain corresponding YOLO-format label .txt files.
   * val/images and val/labels should similarly contain validation data.

**Directory Structure**

yolov5/

├── data/

│ ├── train/

│ │ ├── images/

│ │ │ ├── image1.jpg

│ │ │ ├── image2.jpg

│ │ │ └── ...

│ │ └── labels/

│ │ ├── image1.txt

│ │ ├── image2.txt

│ │ └── ...

│ └── val/

│ ├── images/

│ └── labels/

**6. Configure the Dataset YAML file**

**Yaml**

# sign\_language.yaml

train: data/train/images

val: data/val/images

nc: <number\_of\_classes> # Replace with the number of classes

names: ['Hello', 'Thank You', 'Yes', 'No'] # List all your classes

**7. Fine-Tuning YOLOv5s on Your Dataset**

**Bash**

python train.py --img 640 --batch 16 --epochs 100 --data data/sign\_language.yaml --weights yolov5s.pt

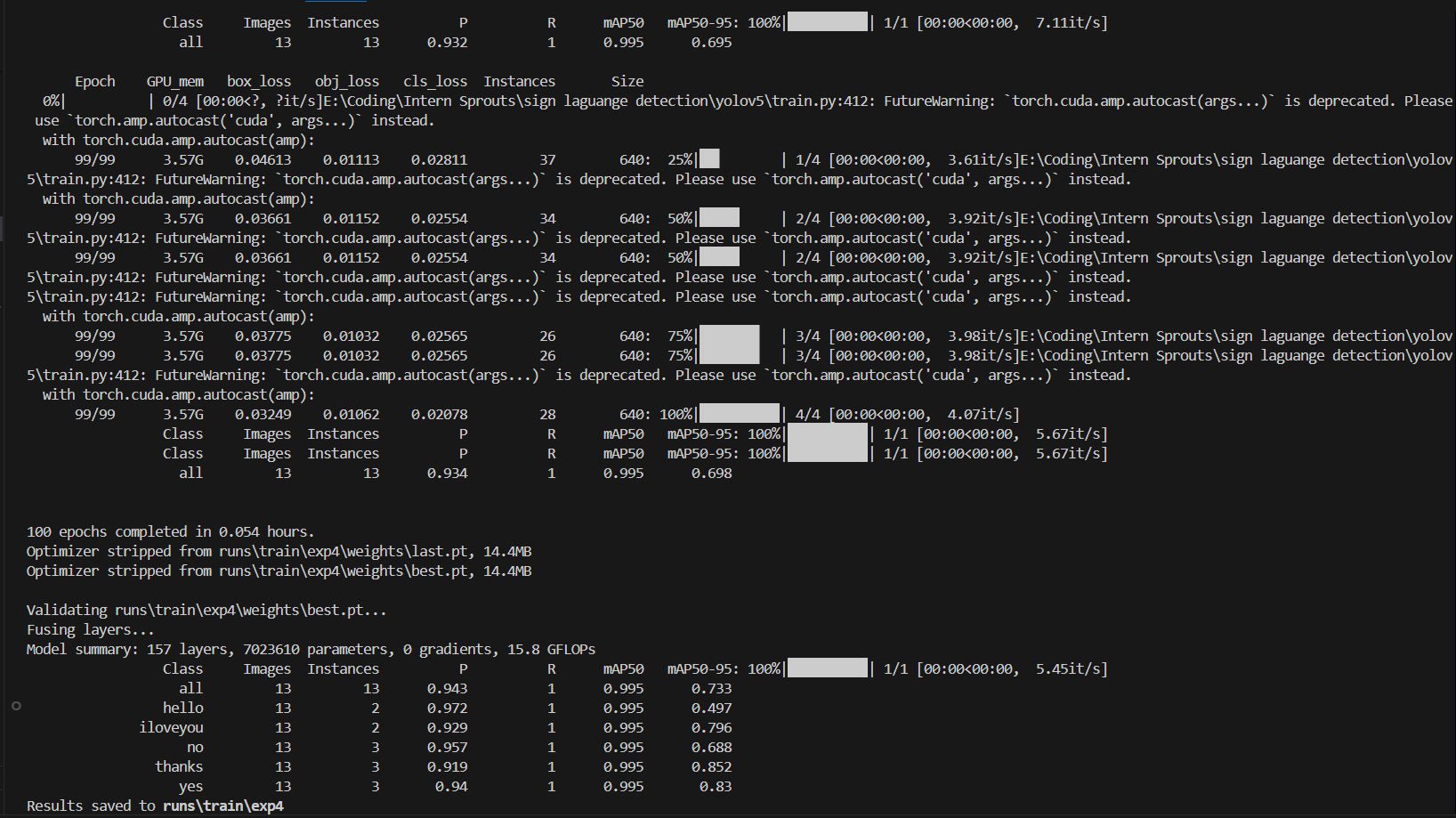
--**img**: The input image size (640 is generally a good balance between accuracy and speed).

--**batch**: Batch size (depends on your GPU memory).

--**epochs**: Number of epochs (increase if needed).

--**data**: Path to your YAML dataset configuration file.

--**weights**: Pre-trained weights to start with (e.g., yolov5s.pt for YOLOv5 small model).



**8. Real-Time Detection Code**

**Python**

1. **Install necessary libraries**

import sys

sys.path.append(r"path\_to\_yolov5")

model\_path=r"path\_to\_model\_best.pt”

import cv2

import torch

from yolov5.models.common import DetectMultiBackend

from yolov5.utils.general import non\_max\_suppression, scale\_boxes

1. **Setup yolov5s model**

detection\yolov5\runs\train\exp4\weights\best.pt"

device=torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model=DetectMultiBackend(model\_path,device=device)

model.eval()

1. **Define class names**

class\_names=["hello","iloveyou","no","thanks","yes"]

1. **Real time video captioning**

def iou(box1, box2):

    x1\_inter = max(box1[0], box2[0])

    y1\_inter = max(box1[1], box2[1])

    x2\_inter = min(box1[2], box2[2])

    y2\_inter = min(box1[3], box2[3])

    inter\_area = max(0, x2\_inter - x1\_inter) \* max(0, y2\_inter - y1\_inter)

    box1\_area = (box1[2] - box1[0]) \* (box1[3] - box1[1])

    box2\_area = (box2[2] - box2[0]) \* (box2[3] - box2[1])

    return inter\_area / float(box1\_area + box2\_area - inter\_area + 1e-6)

cap = cv2.VideoCapture(0)

if not cap.isOpened():

    print("Error: Could not open webcam.")

    sys.exit()

def draw\_bounding\_box(img, box, label, color=(0, 0, 255), thickness=2):

    x1, y1, x2, y2 = map(int, box)

    cv2.rectangle(img, (x1, y1), (x2, y2), color, thickness)

    cv2.putText(img, label, (x1, y1 - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, color, thickness)

while cap.isOpened():

    ret, frame = cap.read()

    if not ret:

        print("Error: Failed to capture frame.")

        break

    img = cv2.resize(frame, (640, 640))

    img\_rgb = img[..., ::-1]  # Convert BGR to RGB

    img\_tensor = torch.from\_numpy(img\_rgb.copy()).float().to(device).permute(2, 0, 1).unsqueeze(0) / 255.0

    results = model(img\_tensor, augment=False)

    nms\_conf\_threshold = 0.4

    detections = non\_max\_suppression(results, conf\_thres=nms\_conf\_threshold, iou\_thres=0.5)[0]

    if detections is not None and len(detections):

        detections[:, :4] = scale\_boxes(img\_tensor.shape[2:], detections[:, :4], frame.shape).round()

        boxes = []

        for \*xyxy, conf, cls in detections:

            box = xyxy

            if not any(iou(box, existing\_box) > 0.5 for existing\_box in boxes):

                boxes.append(box)

                label = f"{class\_names[int(cls)]} {conf:.2f}"

                draw\_bounding\_box(frame, box, label)

    cv2.imshow('Sign Language Detection', frame)

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

        break

cap.release()

cv2.destroyAllWindows()

**9. Testing and Results**

**Testing Scenarios**

To evaluate the robustness and performance of the sign language detection model, various scenarios were tested. This ensures that the model can handle different real-world conditions. Key scenarios include:

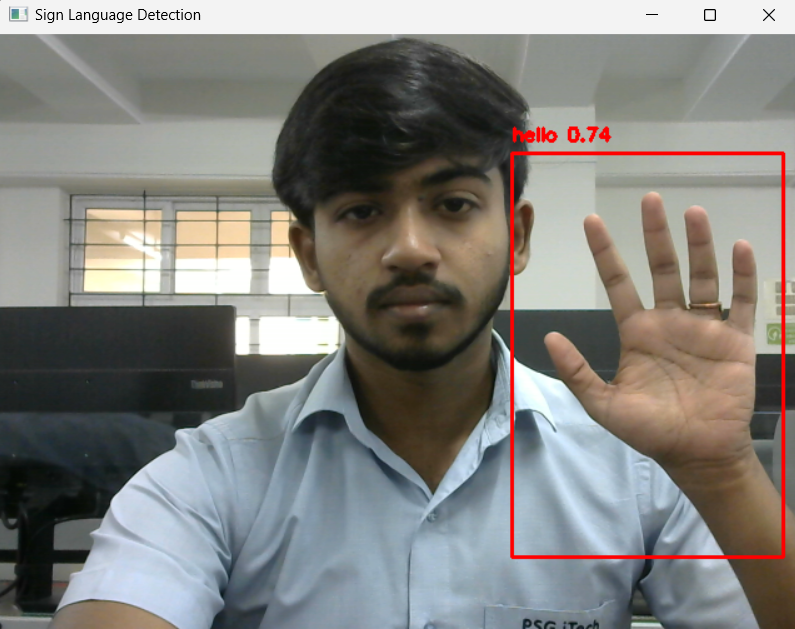
1. **Lighting Conditions:**
   * **Bright Lighting:** Tested in outdoor environments and well-lit rooms to ensure the model can accurately detect gestures in bright light.
   * **Dim Lighting:** Tested in dimly lit rooms to assess the model's performance under low-light conditions.
   * **Backlighting:** Tested with the light source behind the subject to observe any issues in gesture detection when there’s a silhouette effect.
2. **Hand Movement Speed:**
   * **Slow Movements:** Tested with slow, deliberate hand movements to check if the model consistently detects gestures.
   * **Fast Movements:** Tested with fast hand gestures to see if the model can track quick movements without losing accuracy.
3. **Multiple Hands in Frame:**
   * **Single Hand: Testing with only one hand in the frame.**
   * **Multiple Hands (Two or More):** Testing scenarios with multiple people or multiple hands in the frame to evaluate if the model can distinguish between different gestures and avoid confusion or overlapping detections.
4. **Obstructed Hands:**
   * **Partial Obstructions:** Tested with objects partially blocking the hand to see if the model can still recognize the gesture.
   * **Complete Obstruction:** Hands fully obstructed for a brief moment, assessing if the model can quickly recover once the hand is visible again.

A person holding his hands up

Description automatically generatedA person with a beard and mustache

Description automatically generated**A person holding up his hand

Description automatically generatedA person holding his fist

Description automatically generatedResults:**

A person holding his mouth to his mouth

Description automatically generated

**Performance Analysis**

The model was evaluated based on several performance metrics and qualitative observations:

1. **Gesture Recognition Accuracy:**
   * The model achieved high accuracy of 94.3% in detecting gestures across varying lighting conditions and hand movement speeds.
   * The accuracy was higher in bright lighting conditions and for slow to moderate hand movements, while performance slightly dropped in dim lighting or with very fast hand gestures.
2. **Bounding Box Precision:**
   * Bounding boxes were generally well-fitted around the hands and gestures, capturing the hand movement with minimal deviation.
   * In scenarios with multiple hands, the model occasionally produced overlapping boxes, especially when gestures were performed close to each other.
3. **Common Issues and Misclassifications:**
   * **Misclassifications:** On rare cases, the model misclassified **“iloveyou”** and **“hello”,** particularly in low-light conditions or when the gestures were performed quickly.
4. **Latency and Real-time Performance:**
   * The model takes around a minute for initializing the real-time video capturing.
   * Once the webcam turns on and the real-time video is displayed no latency is observed in captioning or live relay.

**10. Challenges and Limitations**

**Challenges Faced During Development**

1. **Lack of Structured Guidance**:
   * One of the primary challenges was the absence of a structured, step-by-step guide for building the sign language detection system. This led to a significant amount of trial and error, as resources and tutorials often provided only partial information relevant to the project.
   * The process of identifying the necessary tools, integrating the YOLOv5 model, and configuring it for real-time detection required extensive research and experimentation.
2. **Annotation and Data Preparation**:
   * Choosing the right annotation tool was initially challenging. The project started with **LabelImg**, which generated annotations in XML format, and **PixLab**, which provided JSON annotations. However, YOLO requires annotations in a specific text format, so converting these files into the YOLO-compatible text format was an extra, time-consuming step.
   * Finding an efficient method to standardize annotations for YOLO usage added complexity to the data preparation phase, making the workflow less streamlined than anticipated.

**Limitations of the System**

1. **Hardware Dependency**:
   * The model's performance is heavily dependent on GPU availability. While it performs efficiently on a dedicated GPU (e.g., NVIDIA RTX 3050), running the model on a CPU results in slower inference times, which may limit its applicability for real-time detection on devices without GPU support.
2. **Model Size and Complexity**:
   * YOLOv5 models, though optimized for real-time applications, still require substantial resources for training and inference. This dependency on higher computational power may restrict the model’s usage on low-power devices or in resource-constrained environments.

**11. Future Enhancements**

1. **Live Translation to Complete Sentences**:
   * The primary goal for future development is to enable real-time translation of detected gestures into full sentences, allowing seamless, lag-free communication between individuals who use sign language and those who do not. This enhancement would make the system more user-friendly and practical for day-to-day interactions, effectively bridging communication gaps for the deaf and hard-of-hearing community.
2. **Adding More Gestures**:
   * Expanding the gesture library by incorporating a wider range of signs would increase the versatility and usefulness of the system. This could include region-specific gestures, as well as gestures for more complex or specialized vocabulary. Training the model on a larger set of gestures would make it more capable of supporting nuanced conversations.
3. **Optimization for Mobile and Web Platforms**:
   * To increase accessibility, optimizing the model for deployment on mobile and web platforms would allow users to access the system on their personal devices. Lightweight model compression techniques, such as quantization and pruning, could be applied to improve the model’s speed and efficiency on lower-powered devices, enabling real-time performance even without high-end hardware.
4. **Implementing Multi-Language Support**:
   * Incorporating multi-language support would allow users to translate detected gestures into various spoken or written languages. This feature would make the system accessible globally, facilitating communication across different linguistic backgrounds and enhancing inclusivity for users worldwide.

**12. Conclusion**  
 This research effectively uses the YOLOv5 model and deep learning techniques to create a real-time sign language identification system. This method seeks to improve communication between the deaf or hard-of-hearing community and non-sign language users by recognising and labelling sign language gestures. The project lays the groundwork for overcoming communication obstacles by converting gestures into meaningful information through OpenCV video processing and the high accuracy of YOLO-based detection.  
  
 This system could have a big effect on communication and accessibility. When used as a tool, it can empower people with hearing impairments by making it easier for them to communicate in social, professional, and educational contexts. The experiment demonstrates the importance of incorporating AI and computer vision into assistive technology to break down linguistic barriers and make real-time interactions more accessible.   
  
 Future improvements like multilingual support, real-time sentence translation, and mobile and online application optimisation are also made possible by this system. This research is ultimately a step towards creating a more inclusive society where technology can give people of all abilities equal access to possibilities for engagement and connection.

**13. References**

**1. GitHub Repository**

**Link**: [GitHub repository](https://github.com/Prithiv99/sign-language-detection.git)

**2. YOLOv5 Documentation**Official YOLOv5 documentation provides guidance on setting up, training, and deploying YOLO models.  
**Link**: [YOLOv5 Documentation](https://docs.ultralytics.com/yolov5)

**3. OpenCV Documentation**The OpenCV library for Python has extensive documentation on image and video processing, which is fundamental for capturing and preprocessing frames for model inference.  
**Link**: [OpenCV Documentation](https://docs.opencv.org/4.x/index.html)

**4. PyTorch Documentation**PyTorch's official documentation provides essential information on installing and using the deep learning library, which is the backbone for YOLOv5 model training and deployment.  
**Link**: [PyTorch Documentation](https://pytorch.org/docs/stable/index.html)

1. **LabelImg Annotation Tool**LabelImg is an open-source graphical image annotation tool used to label images for object detection training. The initial XML annotations generated from LabelImg were used and later converted to a compatible format for YOLOv5.  
   **Link**: [LabelImg GitHub](https://github.com/tzutalin/labelImg)
2. **PixLab Annotation Tool**PixLab's online annotation tool provides annotations in JSON format, which can be converted to YOLO-compatible formats.  
   **Link**: [PixLab](https://pixlab.io/)
3. **YOLOv5 Custom Training Tutorial**This step-by-step guide provides insights into training YOLOv5 on custom datasets, covering dataset preparation, model configuration, and training parameters.  
   **Link**: [YOLOv5 Custom Training Guide](https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data)