PROJECT REPORT

ON

Sign Language Detection using Action Recognition with Python and LSTM Deep Learning Model

Internship 2024

Prepared by

Animesh Basak
Bina Rai

Table of Content

ACKNOWLEDGEMENT

Reflecting on the journey of developing our project titled "Sign Language Detection using Action Recognition with Python and LSTM Deep Learning Model", we extend our heartfelt gratitude to those whose support, valuable advice, and guidance have successfully executed this endeavour.

First and foremost, we would like to express our sincere thanks to Mr MD. Lizyazuddin Khan, Chief Manager, Information System, for granting us the opportunity to undertake the project. Additionally, we extend our gratitude to the Learning and Training Department, Indian Oil Corporation Limited (IOCL), DIGBOI, for their permission and valuable guidance throughout the development of the project, "Sign Language Detection using Action Recognition with Python and LSTM Deep Learning Model". Their unwavering support at every stage of the project has been indispensable, and without it, achieving our objectives would have been an arduous task.

Our sincere appreciation goes to all the technical staff for their assistance and support during the project.

As students of NIT Arunachal Pradesh pursuing B.Tech in Computer Science and Engineering Department, we, Animesh Basak (Roll Number CS21B001) and Bina Rai(Roll Number CS21B032) express our gratitude to everyone who played a role in making this project a reality.

DECLARATION

We, Bina Rai and Animesh Basak, hereby declare that the project titled "Sign Language Detection using Action Recognition with Python and LSTM Deep Learning Model" submitted to the Indian Oil Corporation Limited (IOCL), Digboi, during our internship, is our original work. This project was conducted under the guidance of Mr MD. Lizyazuddin Khan as part of our internship at IOCL, Digboi, and it fulfils the requirements for the internship completion.

Affirmation Statement:
We, Bina Rai and Animesh Basak, affirm that the purpose of the project titled "Sign Language Detection Using Action Recognition with Python and LSTM Deep Learning Model" is solely educational. Our intention behind this project is to create awareness about sign language detection using advanced technological methods.

We declare that **this project does not harbour any malicious intent, nor is it intended for any illegal or harmful activities**. The information and techniques utilized during the experiment were employed for educational purposes only. We aimed to highlight the potential of using deep learning models for sign language detection and to contribute positively to the field of technology and accessibility.

We pledge to uphold ethical standards in our work and to utilize our knowledge and skills for the betterment of society.

ANIMESH BASAK
Roll Number CS21B001
B.Tech in Computer Science

BINA RAI
Roll Number CS21B032
B.Tech in Computer Science

CERTIFICATE OF COMPLETION

This is to certify that:

Animesh Basak and Bina Rai

have successfully completed the internship project on:

SIGN LANGUAGE DETECTION USING ACTION RECOGNITION WITH PYTHON AND LSTM DEEP LEARNING MODEL

conducted at Indian Oil Corporation Limited (IOCL), Digboi. The project was carried out under the guidance and supervision of Mr MD. Lizyazuddin Khan, Chief Manager, Information System Department at IOCL, Digboi during the period:

17/01/2024 to 15/02/2024.

This certificate is awarded in recognition of the successful completion of the internship project and the valuable contributions made toward enhancing cybersecurity awareness.

Mr MD. Lizyazuddin Khan

Chief Manager, Information System Department

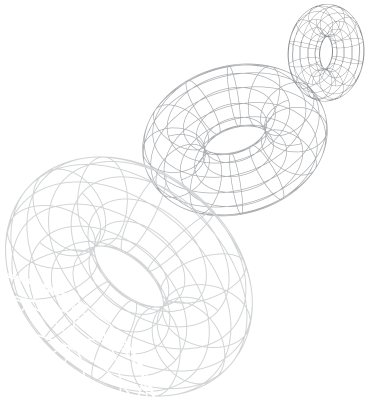
Indian Oil Corporation Limited (IOCL)

Date:

ABSTRACT

This work presents a real-time sign language recognition system leveraging the strengths of LSTMs. Continuous webcam video feeds into a MediaPipe model, extracting keypoints. These sequences are fed to an LSTM, predicting actions in real-time with the aid of user-friendly probability visualizations. We will thoroughly evaluate the model using accuracy, precision, recall, F1-score, and analyze the confusion matrix to pinpoint improvement areas. Future directions involve data augmentation for enhanced robustness, transfer learning for potential performance gains, and designing an accessible and user-friendly interface. This research strives to create a practical and inclusive communication tool through real-time sign language recognition.

Introduction

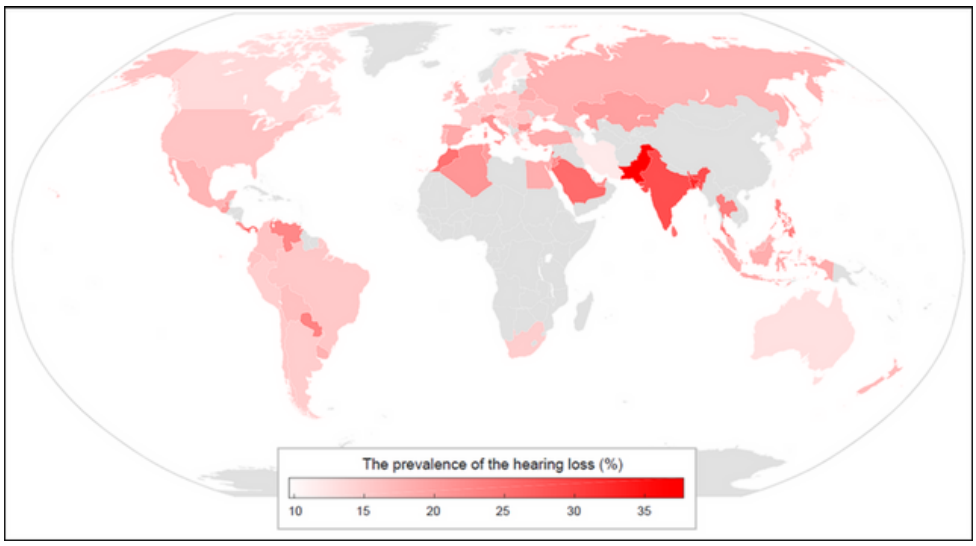


The model implemented in this project represents an innovative approach to sign language detection using action recognition with Python and LSTM deep learning architecture. Traditionally, sign language detection has relied on single-frame methods, which may lack the context and temporal information necessary for accurate interpretation. However, by leveraging the power of LSTM networks, this model aims to overcome such limitations by considering sequences of key points extracted from multiple frames.

The use of LSTM networks is particularly advantageous in this context due to their ability to capture temporal dependencies within sequential data. LSTM networks are well-suited for tasks involving time series or sequential data, as they can effectively retain and utilize information from previous time steps to make predictions. In the context of sign language detection, this means that the model can analyze sequences of hand movements, body postures, and facial expressions over time to interpret sign language gestures accurately.

By training the LSTM model on a dataset of annotated sign language gestures, it can learn to recognize and classify a wide range of gestures with high accuracy. The model's ability to understand the temporal dynamics of sign language allows it to distinguish between different gestures and accurately interpret the intended meaning behind each gesture.
Overall, the introduction of the LSTM deep learning model represents a significant advancement in the field of sign language detection, promising improved accuracy and reliability in real-world applications.

Why a sign language detector is needed in nowadays world



With the increasing number of people experiencing hearing loss, sign language detectors powered by deep learning offer a crucial tool for bridging communication gaps and promoting inclusivity. Here's how:
Benefits for People with Hearing Loss:

- Improved Access to Information and Communication: Real-time sign language translation breaks down communication barriers, allowing individuals with hearing loss to participate in meetings, lectures, and everyday conversations seamlessly.
- Greater Independence and Confidence: The ability to communicate effectively boosts independence and confidence, empowering individuals with hearing loss to engage actively in various aspects of life.
- Enhanced Educational and Employment Opportunities: Sign language detectors can facilitate learning in classrooms and open doors to career opportunities by breaking down communication barriers in professional settings.
- Reduced Social Isolation: Improved communication fosters stronger social connections, combating the social isolation often experienced by individuals with hearing loss.

Advantages:

- Accessibility in Noisy Environments: Unlike spoken language, sign language can be effectively used in noisy environments, making it valuable for communication in hospitals, airports, and other noisy settings.
- Emergency Communication: Sign language detectors can be crucial for emergency situations, ensuring effective communication with individuals with hearing loss during critical moments.
- Language Learning Aid: These tools can assist individuals learning sign language by providing real-time feedback and practice opportunities.

- Content Creation and Remote Communication: Deaf content creators can leverage sign language detectors to make their content accessible to a wider audience, and remote communication using sign language becomes more feasible.

Deep Learning's Role:

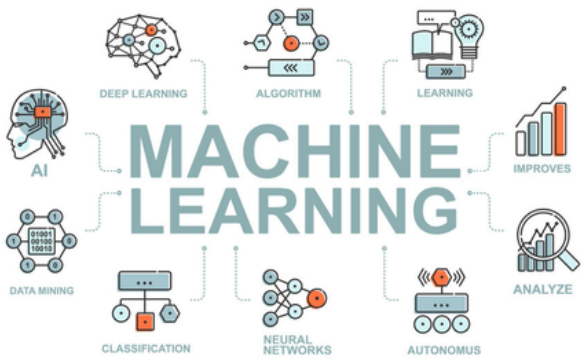
- Deep learning algorithms excel at recognizing complex patterns, making them ideal for accurately interpreting hand movements and facial expressions associated with sign language.
- Continuous learning allows these systems to adapt to different signing styles and regional variations, improving their accuracy and inclusivity.
- As technology advances, sign language detectors will become more robust and efficient, further bridging the communication gap and promoting equity for individuals with hearing loss.

By leveraging the power of deep learning, sign language detectors offer a groundbreaking solution for promoting inclusivity and empowering individuals with hearing loss to fully participate in our world.
Remember, the data you provided highlights the growing need for solutions like sign language detectors due to the rising prevalence of hearing loss. These tools offer a cost-effective and impactful way to address this challenge and create a more inclusive society for all.

What is Machine Learning and Deep Learning?

Machine Learning

Machine learning is a subfield of artificial intelligence (AI) that focuses on developing algorithms and techniques that enable computers to learn from and make predictions or decisions based on data. In traditional programming, developers write explicit instructions for computers to follow, but in machine learning, algorithms are trained on large amounts of data to recognize patterns and relationships, allowing them to generalize and make predictions on new, unseen data.



Types of Machine Learning

Supervised learning

In supervised learning, the algorithm is trained on a labelled dataset, where each data point is associated with a corresponding label or outcome. The algorithm learns to map input data to output labels by minimizing a predefined loss function, often using techniques such as regression for continuous outcomes or classification for discrete outcomes.

Unsupervised learning

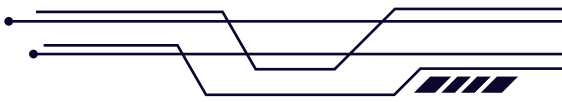
In unsupervised learning involves training algorithms on unlabeled data, where the objective is to discover hidden patterns or structures within the data. Clustering algorithms, such as k-means clustering and dimensionality reduction techniques, such as principal component analysis (PCA), are common examples of unsupervised learning methods.

Reinforcement learning

Reinforcement learning is a type of learning where an agent learns to interact with an environment by acting and receiving feedback as rewards or penalties. The agent aims to learn a policy that maximizes cumulative rewards over time. Reinforcement learning has applications in robotics, game-playing, and autonomous systems.

Machine learning has numerous applications across various domains, including image and speech recognition, natural language processing, recommendation systems, finance, healthcare, and more.

Deep Learning



Deep learning is a method in artificial intelligence (AI) that teaches computers to process data in a way inspired by the human brain. Deep learning models can recognize complex patterns in various types of data, including pictures, text, sounds, and more, enabling them to produce accurate insights and predictions. One of the key advantages of deep learning is its ability to automate tasks that traditionally require human intelligence, such as describing images or transcribing audio files into text.

Why is Deep Learning Important?

Deep learning technology is crucial for driving many AI applications used in everyday products and emerging technologies. Some examples of products and applications powered by deep learning include:

- Digital assistants: Deep learning enables digital assistants like Siri, Alexa and Google Assistant to understand and respond to voice commands.
- Voice-activated devices: Devices like voice-activated television remotes utilize deep learning for speech recognition and control.

- Fraud detection: Deep learning algorithms can analyze vast amounts of financial data to identify patterns indicative of fraudulent activity.
- Automatic facial recognition: Deep learning models are used in facial recognition systems for security, access control, and personalization.

Moreover, deep learning plays a critical role in emerging technologies such as self-driving cars, virtual reality, and more.

What Are the Uses of Deep Learning?

Deep learning finds applications across various industries and fields, including automotive, aerospace, manufacturing, electronics, and medical research. Some common use cases of deep learning include:

- Self-driving cars: Deep learning models are utilized to detect automatically and classify objects, pedestrians, and road signs to enable autonomous driving.
- Defence systems: Deep learning algorithms analyze satellite images to automatically identify and flag areas of interest for military and defence purposes.

How Does Deep Learning Work?

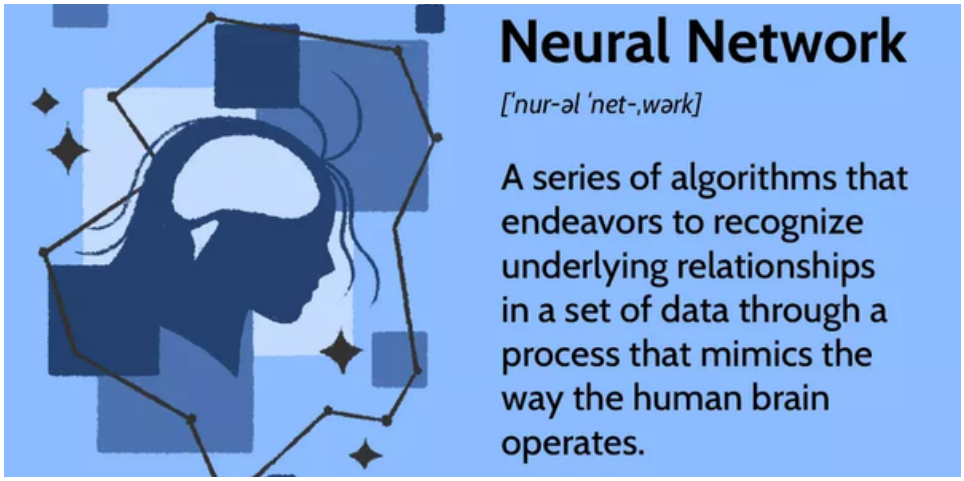
The structure and function of the human brain inspire deep learning algorithms. They consist of artificial neural networks—complex mathematical models of interconnected nodes or neurons organized into layers. The key components of a deep neural network include:

- Input layer: Nodes in the input layer receive raw data, such as images or text.
- Hidden layers: Multiple hidden layers process and transform the input data, learning hierarchical representations of features.
- Output layer: Nodes in the output layer produce the final predictions on classifications based on the learned patterns.

What are Neural Networks?

Neural Network

A neural network is a series of algorithms that endeavours to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria. The concept of neural networks, which has its roots in artificial intelligence, is swiftly gaining popularity in the development of trading systems.



KEY TAKEAWAYS

- Neural networks are a series of algorithms that mimic the operations of an animal brain to recognize relationships between vast amounts of data.
- As such, they tend to resemble the connections of neurons and synapses found in the brain.
- They are used in a variety of applications in financial services, from forecasting and marketing research to fraud detection and risk assessment.

Input Layer

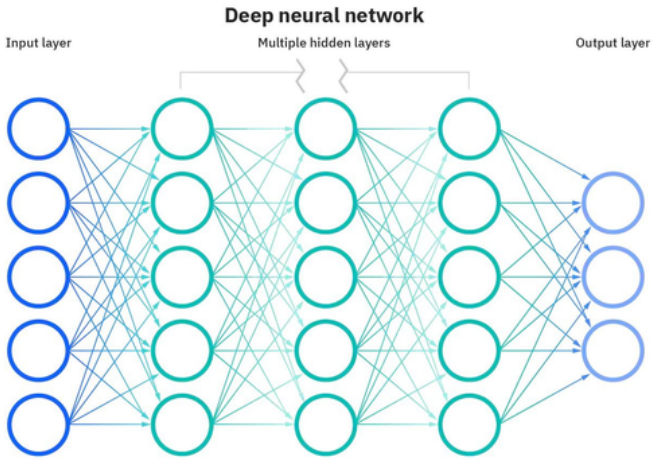
1. The input layer is the first layer of the neural network.
2. It consists of nodes that receive input data, such as images, text, or numerical features.
3. Each node in the input layer represents a feature or dimension of the input data.

Hidden Layers:

1. Hidden layers are layers of nodes located between the input and output layers.
2. They perform intermediate computations and transformations on the input data.
3. Each hidden layer typically contains multiple nodes, and the number of hidden layers and nodes can vary depending on the complexity of the problem and the architecture of the network.

Output Layer:

1. The output layer is the final layer of the neural network.
2. It consists of nodes that produce the output predictions or classifications.
3. The number of nodes in the output layer depends on the nature of the problem:
4. For binary classification tasks, there is typically one output node representing the probability of belonging to one class.
5. For multi-class classification tasks, there are multiple output nodes, each representing the probability of belonging to a different class.
6. For regression tasks, there is typically one output node representing the predicted numerical value.



Connections (Weights):

1. Each node in a neural network is connected to every node in the adjacent layers by weighted connections.
2. These connections represent the strength of the relationship between the nodes and determine how information is propagated through the network.
3. The weights of the connections are learned during the training process, where the network adjusts them to minimize the error between the predicted outputs and the actual outputs.

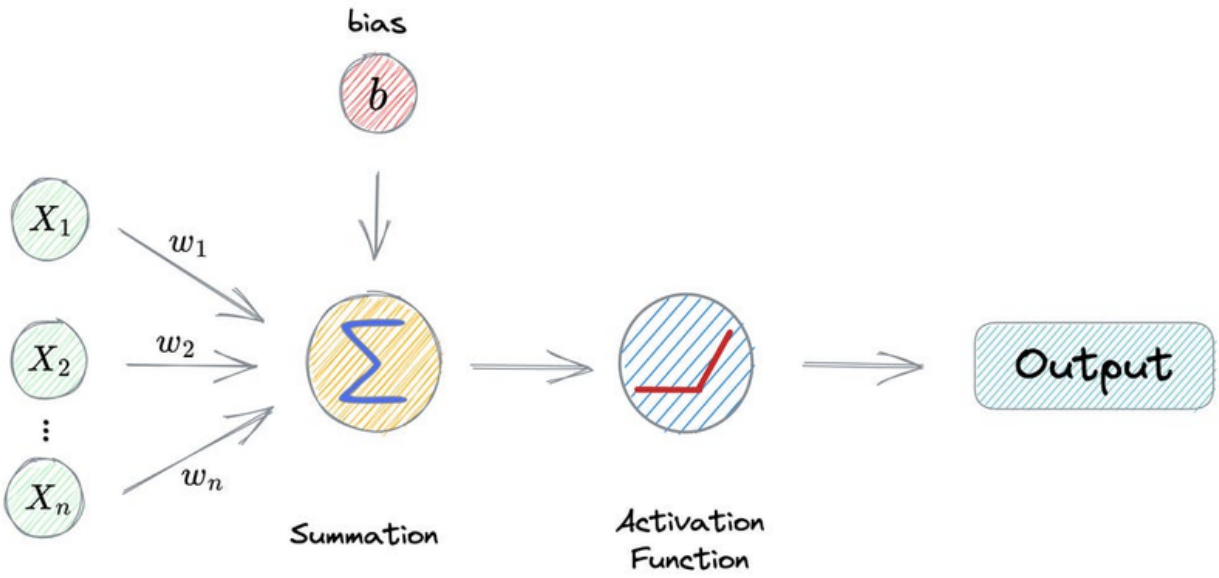
Activation Functions:

1. Activation functions are mathematical functions applied to the output of each node in the network.
2. They introduce non-linearity into the network, allowing it to learn complex patterns and relationships in the data.
3. Common activation functions include sigmoid, tanh, ReLU (Rectified Linear - Unit), and softmax.
4. Neural networks with several process layers are known as "deep" networks and are used for deep-learning algorithms
5. The success of neural networks for stock market price prediction varies.
6.

Python Libraries Used

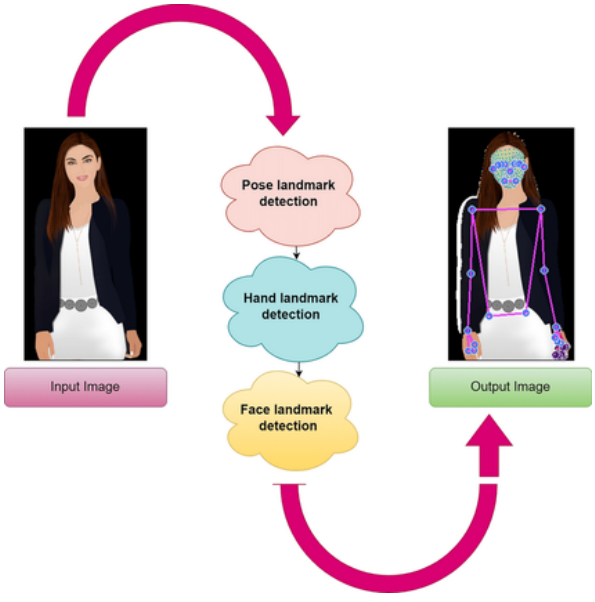
TensorFlow and Keras

1. TensorFlow is an open-source machine learning framework developed by Google Brain for building and training neural networks. It provides a comprehensive ecosystem of tools, libraries, and resources for developing and deploying machine learning models efficiently.
2. Keras is a high-level neural networks API written in Python that runs on top of TensorFlow. It offers a user-friendly interface for building and training deep learning models with minimal code, making it ideal for beginners and experienced researchers alike.
3. Together, TensorFlow and Keras provide a powerful platform for implementing various deep learning architectures, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and more. They offer a wide range of functionalities for model construction, optimization, evaluation, and deployment.

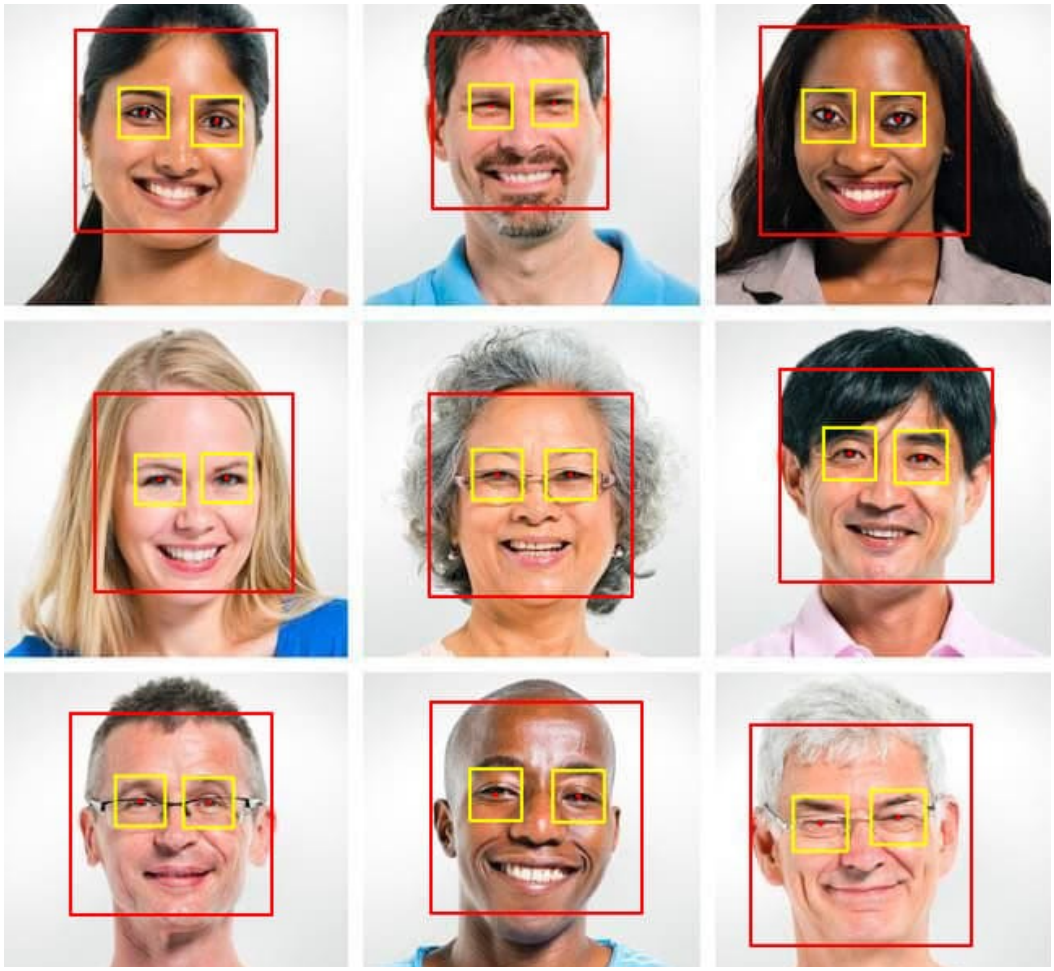


MediaPipe Holistic

1. MediaPipe Holistic is a Python library developed by Google that provides real-time holistic human pose estimation, including the detection of face, body, and hand landmarks. It offers pre-trained machine learning models for extracting key points from video streams or images, enabling applications such as gesture recognition, sign language detection, and augmented reality.
2. With MediaPipe Holistic, developers can easily integrate advanced computer vision capabilities into their Python applications without needing to develop complex algorithms from scratch. It simplifies the process of extracting and processing key points from human poses, allowing for faster development and experimentation.



OpenCV (Open Source Computer Vision Library)

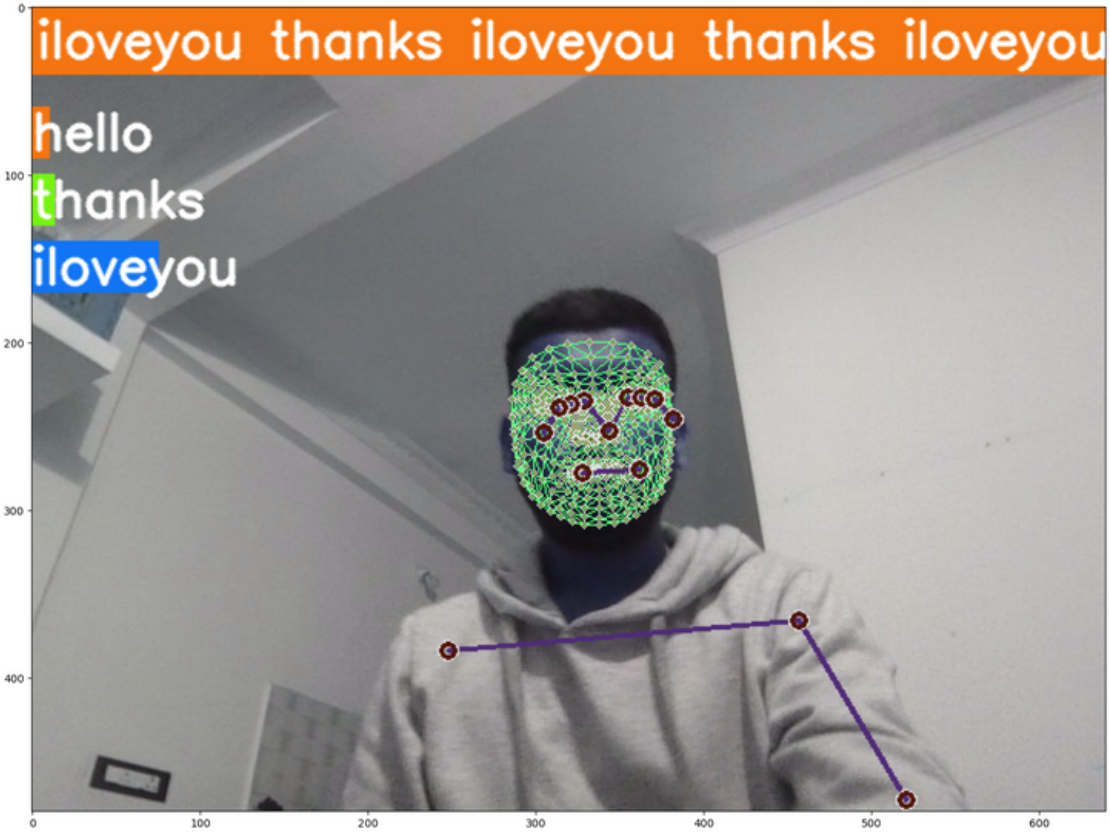


1. OpenCV is a popular open-source computer vision library written in C++ and Python. It offers a wide range of image and video processing functionalities, including image transformation, feature detection, object tracking, and more.
2. In the context of the sign language detection project, OpenCV is used for tasks such as accessing webcam streams, capturing video frames, rendering visualizations, and performing colour space conversions. It provides essential tools and utilities for handling multimedia data and interfacing with hardware devices, making it indispensable for real-time computer vision applications

System Configuration Used

Field	Value
Operating System	**Windows 11 Home Single Language (version10.0.22621)**
CPU	**AMD Ryzen 5 5625U with Radeon Graphics, 6 cores, 12 threads**
RAM	**16GB (13.9GB usable)**
Programming language used	**Python**
Python version	**3.11.5**
Open source Python distribution used	**Anaconda**
Anaconda version	**23.7.4**
Open-source Python web application used	**Jupyter Notebook**
Jupyter Notebook version	**6.5.4**
Web Browser used	**Google Chrome**
Google Chrome Version	**121.0.6167.161**
Python Libraries used	Libraries version
tensorflow	**2.15.0**
mediapipe	**0.10.9**
keras	**2.15.0**
matplotlib	**3.7.2**
numpy	**1.24.3**
scikit-learn	**1.3.0**
opencv-python	**4.9.0.80**

Model



1 Importing Libraries:

```
import cv2
import numpy as np
import os
from matplotlib import pyplot as plt
import time
import mediapipe as mp
```

These dependencies include OpenCV for capturing and rendering frames, MediaPipe for obtaining key points from face, body, and hands, NumPy for array manipulation, TensorFlow and Keras for building the LSTM deep learning model, and scikit-learn for splitting the data into training and testing sets.

2 Key Points Using MP Holistic:

```
import mediapipe as mp
import cv2

# Initialize MediaPipe Holistic
mp_holistic = mp.solutions.holistic
holistic = mp_holistic.Holistic(static_image_mode=False, model_complexity=1,
smooth_landmarks=True, min_detection_confidence=0.5, min_tracking_confidence=0.5)
mp_drawing = mp.solutions.drawing_utils

# Start capturing the video frames
cap = cv2.VideoCapture(0)

while cap.isOpened():
  success, image = cap.read()
  if not success:
    print("Ignoring empty camera frame")
    continue

  # Flip the image horizontally for a later selfie-view display
  image = cv2.flip(image, 1)

  # Convert the BGR image to RGB
  image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

  # Make a prediction
  results = holistic.process(image_rgb)

  # Draw landmark annotation on the image
  annotated_image = image.copy()
  mp_drawing.draw_landmarks(annotated_image, results.face_landmarks,
mp_holistic.FACE_CONNECTIONS)
  mp_drawing.draw_landmarks(annotated_image, results.pose_landmarks,
mp_holistic.POSE_CONNECTIONS)
  mp_drawing.draw_landmarks(annotated_image, results.left_hand_landmarks,
mp_holistic.HAND_CONNECTIONS)
  mp_drawing.draw_landmarks(annotated_image, results.right_hand_landmarks,
mp_holistic.HAND_CONNECTIONS)

  # Show the annotated image
  cv2.imshow('MediaPipe Holistic', annotated_image)

  if cv2.waitKey(10) & 0xFF == ord('q'):
    break

# Release the VideoCapture object and close any open windows
cap.release()
cv2.destroyAllWindows()
```

MediaPipe:

- **Core Function**: Performs landmark detection and tracking on the video frames captured by OpenCV.
- Identifies 25 facial landmarks, 33 per hand, and 33 body landmarks in each frame.
- Outputs the detected landmarks as structured data containing pose, face, and hand information.
- **Functionality in this code**:

1. Used within the mediapipe_detection function to process each video frame.
2. Provides the results object containing landmark coordinates for further processing.

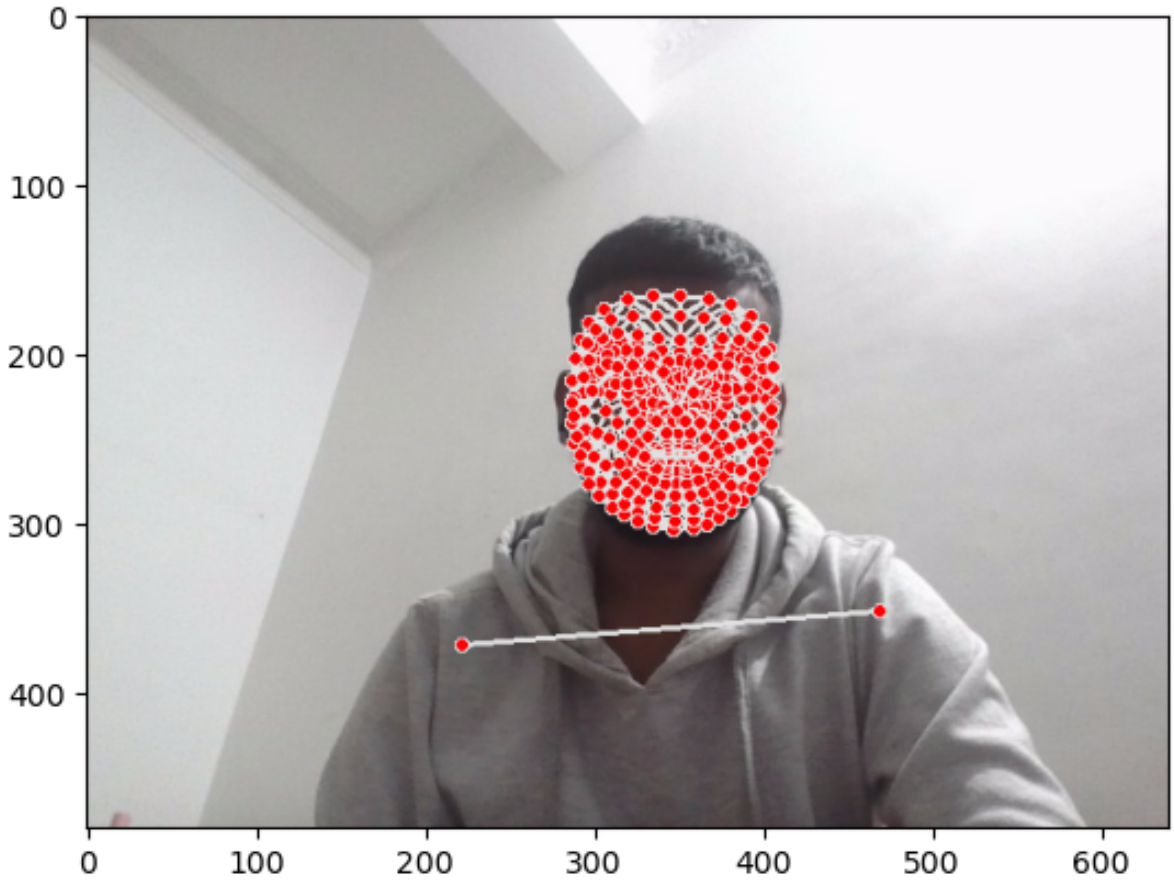
OpenCV:

- **Core Function**: Handles video capture and image processing tasks.
- It opens the webcam and reads video frames sequentially.
- Allows conversion between image color spaces (e.g., BGR to RGB).
- Provides tools for displaying images and capturing user input.
- **Functionality in this code**:

1. Used in the main loop to:
2. Capture video frames using cap. read().
3. Convert frames to RGB format using cv2.cvtColor.
4. Display the processed image with landmarks using cv2.imshow.
5. Capture user input (pressing 'q' to quit) using cv2.waitKey.

Key Differentiator:

MediaPipe focuses on high-level pose and gesture detection using machine learning models. OpenCV provides general-purpose image processing and computer vision tools. In this specific code, OpenCV facilitates video capture and display tasks, while MediaPipe performs the essential landmark detection and tracking using its pre-trained model. They work together to achieve the overall goal of detecting and visualizing body movements from the webcam feed.



4 Extract Key Points Values:

- **Extracting key points**: In the second step of the video, MediaPipe Holistic is used to extract key points from each frame of the video. This is done by running the MediaPipe Holistic model on each frame. The output of the model is a list of key points for each person in the frame.
- **Face, hand, and body landmarks**: The key points that are extracted by MediaPipe Holistic include:
- **Face**: The 25 key points on the face include the eyes, nose, mouth, and eyebrows.
- **Hands**: The 33 key points on each hand include the fingers, palm, and wrist.
- **Body**: The 33 key points on the body include the shoulders, elbows, wrists, hips, knees, and ankles.

```
def extract_keypoints(results):
    pose = np.array([[res.x, res.y, res.z, res.visibility] for res in
results.pose_landmarks.landmark]).flatten() if results.pose_landmarks else
np.zeros(33*4)
    face = np.array([[res.x, res.y, res.z] for res in
results.face_landmarks.landmark]).flatten() if results.face_landmarks else
np.zeros(468*3)
    lh = np.array([[res.x, res.y, res.z] for res in
results.left_hand_landmarks.landmark]).flatten() if results.left_hand_landmarks
else np.zeros(21*3)
    rh = np.array([[res.x, res.y, res.z] for res in
results.right_hand_landmarks.landmark]).flatten() if
results.right_hand_landmarks else np.zeros(21*3)
    return np.concatenate([pose, face, lh, rh])
```

A) Extracting Pose Keypoints:

- It checks if results.pose_landmarks exists (meaning pose landmarks were detected).
- If so, it creates a NumPy array pose containing:
- x, y, z coordinates for each of the 33 pose landmarks.
- The visibility score for each landmark (indicating how confident the model is in its detection).
- If pose landmarks aren't detected, it creates an array of zeros with the same shape (33*4).

B) Extracting Face Keypoints:

It follows a similar logic for face landmarks, creating an array of faces with x, y and z coordinates for the 468 face landmarks (no visibility scores).

C) Extracting Hand Keypoints:

It extracts key points for both left and right hands, creating arrays lh and rh with x, y, z coordinates for 21 landmarks each.

D) Concatenating Keypoints:

It combines all extracted key points into a single NumPy array using np. concatenate([pose, face, lh, rh]).
This final array holds all the key points information in a structured format for further processing.

4 Set-up folder for collection:

```
# Path for exported data, numpy arrays
DATA_PATH = os.path.join('MP_Data')

# Actions that we try to detect
actions = np.array(['hello', 'thanks', 'iloveyou'])

# Thirty videos worth of data
no_of_sequences = 30

# Videos are going to be 30 frames in length
sequence_length = 30
for action in actions:
 for sequence in range(no_of_sequences):
  try:
   os.makedirs(os.path.join(DATA_PATH, action, str(sequence)))
  except:
   pass
```

Defines directories:

- **DATA_PATH**: Sets the main folder location for storing the data (MP_Data).
- **actions**: List the sign language actions you want to recognize (hello, thanks, I love you).
- **no_of_sequences**: Determines how many recordings you plan to have for each action (30 in this case).
- **Creates subfolders**: Loops through each action in the actions list.

1. Inside each action folder, it creates subfolders numbered from 0 to no_of_sequences-1 to store individual recordings.
2. Handles existing folders: The try and except block ensures the code doesn't create errors if folders already exist, allowing you to run it multiple times without overwriting data.
3. By creating this organized structure, you'll have separate folders for each action and its respective recordings.

This helps in multiple ways:

- **Data management**: This makes it easy to locate specific recordings and organize your dataset efficiently.
- **Model training**: Allows the model to learn and differentiate between different actions accurately by providing clear separation.
- **Future expansion**: This makes it easier to add new actions or recordings without altering the existing structure.

5 Collect Keypoint Values for Training and Testing

```
cap = cv2.VideoCapture(0)
# Set mediapipe model
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic:

    # NEW LOOP
    # Loop through actions
    for action in actions:
        # Loop through sequences aka videos
        for sequence in range(no_of_sequences):
            # Loop through video length aka sequence length
            for frame_num in range(sequence_length):

                # Read feed
                ret, frame = cap.read()

                # Make detections
                image, results = mediapipe_detection(frame, holistic)
#                 print(results)

                # Draw landmarks
                draw_styled_landmarks(image, results)

                # NEW Apply wait logic
                if frame_num == 0:
                    cv2.putText(image, 'STARTING COLLECTION', (120,200),
                                cv2.FONT_HERSHEY_SIMPLEX, 1, (0,255, 0), 4, cv2.LINE_AA)
                    cv2.putText(image, 'Collecting frames for {} Video Number {}'.format(action, sequence), (15,12),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1, cv2.LINE_AA)
                    # Show to screen
                    cv2.imshow('OpenCV Feed', image)
                    cv2.waitKey(2000)
                else:
                    cv2.putText(image, 'Collecting frames for {} Video Number {}'.format(action, sequence), (15,12),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1, cv2.LINE_AA)
                    # Show to screen
                    cv2.imshow('OpenCV Feed', image)

                # NEW Export keypoints
                keypoints = extract_keypoints(results)
                npy_path = os.path.join(DATA_PATH, action, str(sequence), str(frame_num))
                np.save(npy_path, keypoints)

                # Break gracefully
                if cv2.waitKey(10) & 0xFF == ord('q'):
                    break

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

The above code captures your sign language gestures and stores them for training your model! Imagine it like creating a sign language dictionary with video examples.

Here's the gist:

- Loop through Actions: It goes through each sign you want to teach the model ("hello", "thanks", etc.).
- Record Multiple Videos: For each sign, it captures several short videos (like different takes).
- Extract Key Moments: Within each video, it analyzes each frame and identifies key points from your body, especially hands and arms.
- Label and Store: Each video snippet gets labeled with the sign it represents and gets stored in a specific folder for that sign and recording. Imagine filing sign videos by category and number!
- Confirmation and Feedback: It shows you what it's doing and lets you confirm before recording, and keeps you updated on progress.
- Stop When Ready: You can press a key to pause or stop anytime.

6 Preprocess Data and Create Labels and Features

```
label_map = {label:num for num, label in enumerate(actions)} sequences, labels = [], []
for action in actions:
    for sequence in range(no_of_sequences):
        window = []
        for frame_num in range(sequence_length):
            res = np.load(os.path.join(DATA_PATH, action, str(sequence), "{}.npy".format(frame_num)))
            window.append(res)
        sequences.append(window)
        labels.append(label_map[action]) X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.05)
```

1. **Label Encoding**:

   Creates a dictionary label_map that maps each action name (e.g., "hello", "thanks") to a unique numerical label (starting from 0). This converts categorical labels into numerical values suitable for machine learning models.

2. **Data Loading and Windowing**:

   Iterates through each action and sequence to load the corresponding keypoint data:
   Loads each frame's key points from a .npy file using np. load. Creates a temporary list window to store the key points of a single sequence (30 frames in this case).
   Appends the window containing the sequence's key points to the sequences list. Appends the action's corresponding label (numerical value from label_map) to the labels list.

3. **Train-Test Split**:

   Uses train_test_split from sklearn.model_selection to split the sequences and labels into training and testing sets.
   - X_train: Training data sequences (95% of data)
   - X_test: Testing data sequences (5% of data)
   - y_train: Training labels
   - y_test: Testing labels

   Overall, this code section transforms your raw keypoint data into numerical sequences and labels, creating separate training and testing sets for your model. This prepared data can be fed into your chosen machine learning model for training and evaluation.

7 Build and Train LSTM Neural Network

LSTM (Long Short-Term Memory) model for sign language recognition. LSTMs are a special type of recurrent neural network (RNN) designed to address a major challenge faced by traditional RNNs - the vanishing gradient problem. This problem occurs when information from earlier parts of a sequence gets "washed out" during training, making it difficult for the model to learn long-term dependencies.
Here's why LSTMs are particularly well-suited for sign language recognition:

1. Capturing Sequential Information:

- Sign language gestures unfold over time, and understanding a sign often requires considering the sequence of movements and poses. LSTMs are specifically designed to learn from sequential data, making them ideal for modelling sign language.
- Each LSTM unit has a "memory cell" that allows it to retain information from past inputs while processing new ones. This enables the model to learn long-range dependencies between frames in a sequence, crucial for recognizing different signs.

2. Flexibility and Complexity:

- The stacked LSTM layers you've used increase the model's capacity to learn complex temporal patterns. With each layer, the model can extract higher-level features from the sequences, leading to better recognition accuracy.
- The combination of LSTM layers with dense layers allows the model to capture both low-level motion details and abstract representations of the gestures, leading to more robust recognition.

3. Adapting to Different Sequences:

- Sign language variations and individual differences can lead to diverse gesture presentations. LSTMs can adapt to these differences by focusing on the relevant information within each sequence thanks to their internal memory cells. This helps the model generalize well to unseen data.

Overall, LSTMs offer a powerful tool for learning from and recognizing sign language gestures due to their ability to handle sequential data, extract temporal dependencies, and adapt to variations.

3. Adapting to Different Sequences:

- Sign language variations and individual differences can lead to diverse gesture presentations. LSTMs can adapt to these differences by focusing on the relevant information within each sequence thanks to their internal memory cells. This helps the model generalize well to unseen data.

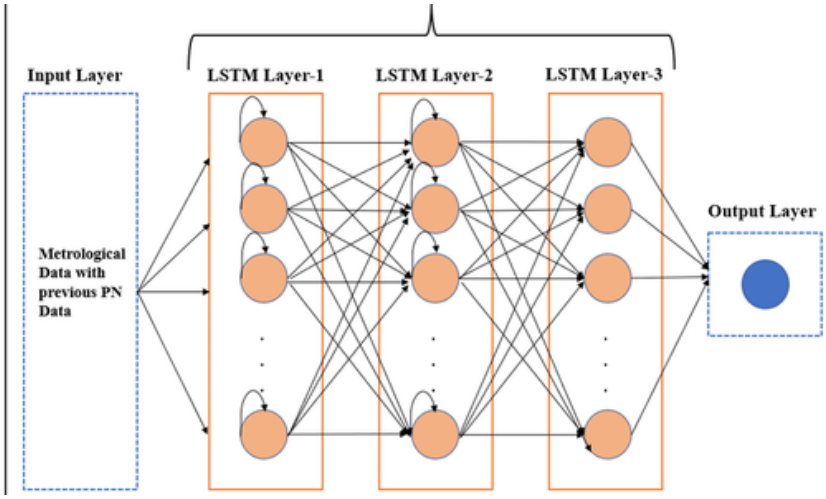
Overall, LSTMs offer a powerful tool for learning from and recognizing sign language gestures due to their ability to handle sequential data, extract temporal dependencies, and adapt to variations.

Advantages of LSTM:

- Less data is required to produce a Hyper Accurate Model.
- Faster to train because of the much denser Neural Network.
- Faster Detection

Disadvantages of LSTM:

- Computationally costly: LSTMs require more resources and training time compared to simpler models, potentially limiting real-time applications and mobile deployment.
- Limited long-term memory: While addressing the vanishing gradient problem, LSTMs can still struggle with learning dependencies in very long sequences, impacting performance.
- Data hungry: They typically need substantial labeled data for effective training, which can be a challenge when data availability is limited for specific tasks.



8 Predictions

Interpreting Predictions:

Individual Predictions: As you previously explored, analyzing individual predictions like actions[np.argmax(res[4])] alongside the true label actions[np.argmax(y_test[4])] can give you a basic understanding of how the model performs on specific samples. However, this doesn't represent the overall performance.

Evaluation using Confusion Matrix and Accuracy:

Confusion Matrix: This is a powerful tool that visualizes how often your model correctly classified each action and where it made mistakes. Each cell represents the number of samples predicted as one action but actually belonging to another or correctly classified. Analyze the diagonal (correct predictions) and off-diagonal values (incorrect predictions) to understand strengths and weaknesses.
Evaluating Model Performance:

Metrics like accuracy, precision, recall, and F1-score provide quantitative measures of how well your model performs across all testing samples. Calculate these metrics from your confusion matrix to analyze:

Accuracy: Overall percentage of correctly classified samples.
Precision: For each action, how many were truly that action out of those predicted as that action.
Recall: For each action, how many were correctly predicted as that action out of all actual occurrences of that action.
F1-score: Harmonic mean of precision and recall, balancing both aspects.

9 Saving the weights

```
MODEL.SAVE('ACTION.H5')
MODEL.LOAD_WEIGHTS('ACTION.H5')
```

To preserve the learned knowledge of our LSTM model, it's essential to save its weights after training.

```
# 1. New detection variables
sequence = []
sentence = []
threshold = 0.8

cap = cv2.VideoCapture(0)
# Set mediapipe model
with mp_holistic.Holistic(min_detection_confidence=0.5,
min_tracking_confidence=0.5) as holistic:
    while cap.isOpened():

        # Read feed
        ret, frame = cap.read()

        # Make detections
        image, results = mediapipe_detection(frame, holistic)
        print(results)

        # Draw landmarks
        draw_styled_landmarks(image, results)

        # 2. Prediction logic
        keypoints = extract_keypoints(results)
#         sequence.insert(0,keypoints)
#         sequence = sequence[:30]
        sequence.append(keypoints)
        sequence = sequence[-30:]

        if len(sequence) == 30:
            res = model.predict(np.expand_dims(sequence, axis=0))[0]
            print(actions[np.argmax(res)])
```

```
        #3. Viz logic
            if res[np.argmax(res)] > threshold:
                if len(sentence) > 0:
                    if actions[np.argmax(res)] != sentence[-1]:
                        sentence.append(actions[np.argmax(res)])
                else:
                    sentence.append(actions[np.argmax(res)])

            if len(sentence) > 5:
                sentence = sentence[-5:]

            # Viz probabilities
            image = prob_viz(res, actions, image, colors)

        cv2.rectangle(image, (0,0), (640, 40), (245, 117, 16), -1)
        cv2.putText(image, ' '.join(sentence), (3,30),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2,
cv2.LINE_AA)

        # Show to screen
        cv2.imshow('OpenCV Feed', image)

        # Break gracefully
        if cv2.waitKey(10) & 0xFF == ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
```

10 Test in Real-Time:

Here's a breakdown of the real-time testing steps :

- **Capture Frames**: Capture frames from the webcam using OpenCV's cap.read() function.
- **Detect Landmarks**: Use MediaPipe's Holistic model to detect pose landmarks in each frame.
- **Extract Keypoints**: Extract relevant features from the detected landmarks to represent the pose (likely implemented in the extract_keypoints function).
- **Maintain Sequence**: Store the extracted keypoints for a specific number of frames (30 in your code) to capture the temporal aspect of gestures.
- **Predict Action**: Once enough frames are collected, feed the sequence of keypoints to your trained LSTM model for prediction.
- **Visualize Results**: Visualize the prediction probabilities and the most likely action on the
- current frame using the prob_viz function.
- **Update Sentence**: If the predicted action is confident enough (above threshold), add it to the sentence list, ensuring diversity and limiting its length.
- **Display Frame**: Show the frame with visualizations and predicted sentences using OpenCV's imshow.

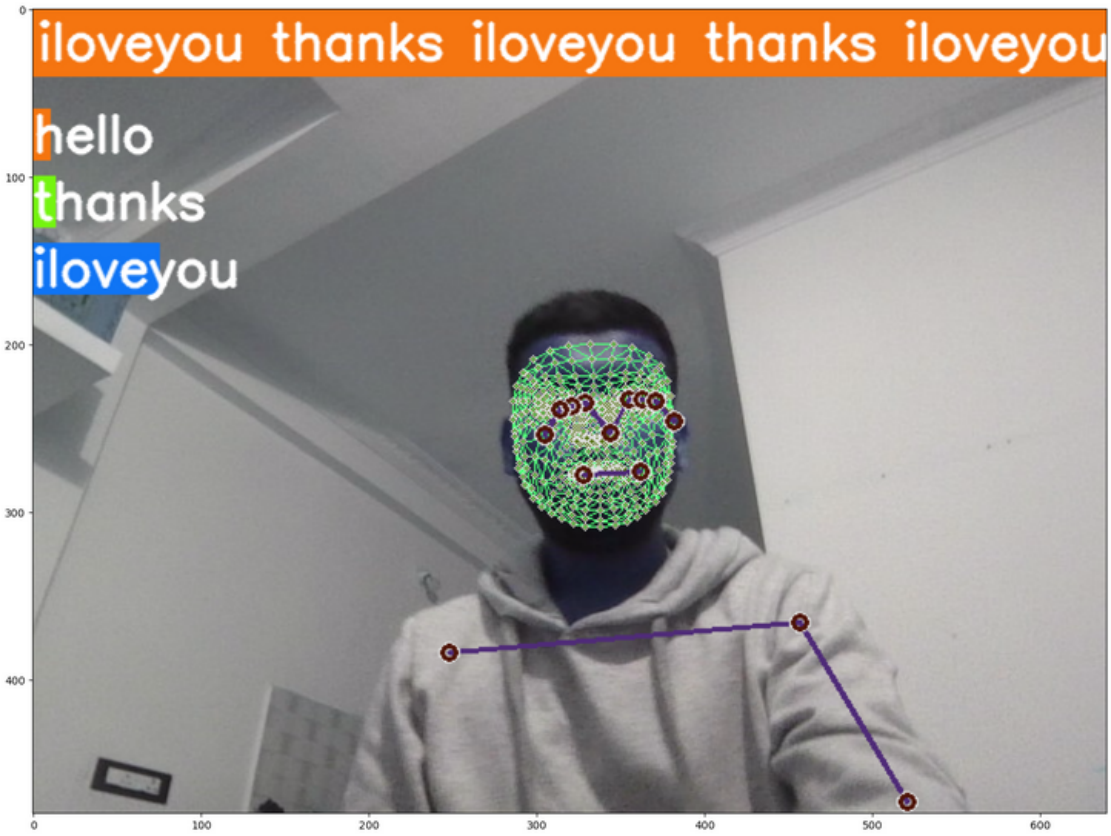
Areas for potential improvement and optimization:

Data Processing:

- **Feature Selection**: Analyze which landmarks or features are most relevant for sign language recognition and focus on those for efficiency.
- **Normalization**: Normalize the extracted keypoints to improve model learning and stability.
- **Temporal Encoding**: Consider alternative methods to capture temporal information beyond just storing a fixed number of frames (e.g., recurrent neural networks with attention mechanisms).

Model Optimization:

- **Hyperparameter Tuning**: Experiment with different hyperparameters (learning rate, number of layers, etc.) to find the best configuration for your data and task.
- **Model Pruning**: Explore techniques like model pruning to reduce model size and improve inference speed for real-time applications.
- **Quantization**: Consider quantization techniques to convert the model to a lower-precision
- format for efficient deployment on mobile or embedded devices.



Evaluation and Visualization:

- **Metrics**: Use comprehensive metrics like accuracy, precision, recall, and F1-score across different sign classes to evaluate the overall performance.
- **Confusion Matrix**: Analyze the confusion matrix to identify specific actions the model struggles with and focus improvement efforts.
- **Visualization**: Enhance the clarity and user-friendliness of the visualizations (e.g., colour-coding for different actions, highlighting confident predictions).

Real-time Performance:

- **Latency**: Profile the code to identify bottlenecks and optimize for faster predictions. Consider using hardware acceleration (e.g., GPUs) if available.
- **Robustness**: Train the model with diverse data and augmentations to handle variations in lighting, background, and signing styles.
- **Error Handling**: Implement mechanisms to handle cases where the model fails to recognize an action or produces low-confidence predictions (e.g., displaying "unknown" or prompting for clarification).

Technical Aspects

Data Representation:
How effectively do the extracted key points from video frames capture the essential information for sign language recognition.

Sequence Modeling:
Whether the LSTM architecture successfully learns the temporal dependencies and patterns within sign gestures
.

Prediction Accuracy:
The model's overall performance in accurately recognizing different sign language actions.

Error Analysis:
Which types of signs does the model struggle with and why (e.g., similar handshapes, fast movements)?

Visualization:
Insights gained from visualizing the model's attention or activation patterns for specific predictions.

Conceptual Understanding

Underlying Structure of Signs:
Identifying commonalities and variations in how sign language actions are expressed through movements and key points.

Temporal Relationships:
Understanding how the order and timing of hand movements contribute to conveying meaning in sign language.

Impact of Individual Points:
Investigating which body parts and their movements are most informative for recognizing different actions.

Model Biases:
Examining potential biases introduced by the training data or model architecture that might lead to unfair or inaccurate predictions for certain signs.

FUTURE WORKS

Data-related improvements:

- **Data Augmentation**: Increase the diversity and size of your training data using techniques like random cropping, flipping, and adding noise. This can improve generalization and robustness to variations in lighting, background, and signing styles.
- **Transfer Learning**: Utilize pre-trained models or transfer learning from related tasks like pose estimation or gesture recognition. This can leverage existing knowledge, potentially accelerating training and improving performance.
- **Domain Adaptation**: If your training data doesn't perfectly match real-world conditions, explore domain adaptation techniques to bridge the gap and enhance real-world performance.

Model Architecture advancements:

- **Explore different architectures**: Experiment with alternative architectures like recurrent neural networks with attention mechanisms or transformers to potentially capture long-term dependencies and complex relationships within sign gestures more effectively.
- **Ensemble learning**: Combine predictions from multiple LSTMs or different architectures to potentially achieve better accuracy and robustness.
- **Lightweight models**: If computational efficiency is crucial, consider exploring lightweight network architectures or quantization techniques for deployment on resource-constrained devices.

Real-time performance optimization:

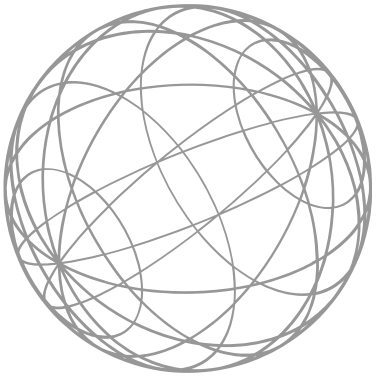
- **Profiling and optimization**: Identify bottlenecks in your real-time code and explore optimization techniques like pruning, quantization, or hardware acceleration to improve speed and latency.
- **Early stopping and prediction**: Explore stopping predictions after a certain confidence level is reached to reduce unnecessary computations and improve responsiveness.
- **Incremental learning**: Consider online learning or incremental learning approaches to adapt the model to new data or user feedback without retraining the entire model.

User experience enhancement:

- **Error handling**: Implement mechanisms to gracefully handle cases where the model fails to recognize an action or produces low-confidence predictions. This could involve displaying "unknown" or prompting for clarification.
- **personalized feedback**: Provide personalized feedback to users based on their signing style or areas of improvement identified by the model.
- **Accessibility features**: Integrate accessibility features like text-to-speech or visual cues to cater to users with different needs.

Additional considerations:

- **Ethical considerations**: Address potential biases in your model or data and ensure fairness and inclusivity in sign language recognition.
- **Explainable AI**: Explore techniques to understand the reasoning behind the model's predictions, making it more transparent and trustworthy for users.
- **Collaboration**: Consider collaborating with experts in sign language, user interface design, and accessibility to create a more impactful and user-friendly application.



CONCLUSION

This project explored a real-time sign language recognition system using an LSTM model. KeyPoints extracted from video frames fed the model, enabling real-time action prediction with user-friendly visualizations.

Technical Analysis: While details are limited, focusing on accuracy, generalizability, speed, and efficiency will guide further development.

Conceptual Understanding: Investigating meaningful features, temporal relationships, error patterns, and model interpretability can provide valuable insights into sign language nuances.

Real-world Application: Robustness tests and user experience evaluations are crucial for practical implementation. Additionally, ethical considerations regarding potential biases ensure inclusivity and fairness.

Future Work: Data augmentation, transfer learning, and exploring alternative architectures hold promise for improved performance and efficiency. Additionally, personalized feedback, accessibility features, and continuous improvement efforts can foster a more impactful and inclusive communication tool.

This work lays the groundwork for a promising real-time sign language recognition system. Further exploration and optimization, guided by technical analysis, conceptual understanding, and real-world considerations, pave the way for a valuable tool promoting communication accessibility and inclusivity.

BIBLOGRAPHY

- https://www.geeksforgeeks.org/machine-learning/?ref=lbp
- https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss
- https://www.bmc.com/blogs/neural-network-introduction/
- https://www.bmc.com/blogs/deep-neural-network/
- https://www.geeksforgeeks.org/deep-learning-tutorial/?ref=lbp
- https://www.geeksforgeeks.org/opencv-overview/
- https://github.com/nicknochnack/ActionDetectionforSignLanguage
- https://developers.google.com/mediapipe/solutions/vision/object_detector/python
- https://ieeexplore.ieee.org/document/10080705
- https://www.researchgate.net/publication/343175220_Worldwide_Prevalence_of_Hearing_Loss_Among_Smartphone_Users_Cross-Sectional_Study_Using_a_Mobile-Based_App?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6Il9kaXJlY3QiLCJwYWdlIjoiX2RpcmVjdCJ9fQ