CS232: Artificial Intelligence

Course project : Introduction to ai

Submitted By:

U23cs020 Dhruvi Rana

U23cs029 Brinda Soneji

U23cs054 Kena Parmar

Branch: CSE

Semester: 4th Sem

Division : A

Submitted To: Dr. Chandra Prakash

Department of Computer Science and Engineering

SV NATIONAL INSTITUTE OF TECHNOLOGY SURAT

2025

Table of contents

Abstract

This project introduces a real-time sign language interpretation system designed to bridge the communication gap between the hearing and speech-impaired communities and the general public. The system utilizes an Artificial Neural Network (ANN) to recognize static hand gestures and accurately predict corresponding alphabets and words. These predictions are displayed on an LED matrix, enhancing accessibility, while an integrated dictionary API enables users to query word meanings for better understanding. Although Random Forest classifiers were initially considered for their computational efficiency, ANN was ultimately chosen due to its accuracy and lower memory constraints. A custom dataset comprising static sign representations of alphabets and commonly used words was developed to ensure inclusivity and adaptability. The system's design and model selection were informed by extensive research and experimentation, contributing to a robust and scalable assistive communication tool.

Introduction

In today's world, bridging the communication gap for individuals with hearing or speech impairments remains a significant challenge. The Real-Time Sign Language Interpreter addresses this by providing an interactive system that recognizes and interprets static hand gestures into meaningful text and speech.

This system leverages machine learning techniques, specifically an Artificial Neural Network (ANN), to detect and classify hand gestures into corresponding alphabets and commonly used words. The model is trained on a custom-built dataset containing images of static sign language gestures, ensuring inclusivity and adaptability across various sign patterns. Once a word is recognized, the predicted word is displayed on an LED matrix for instant visual feedback, and users can also query the word's meaning through an integrated dictionary API.

By enabling real-time interpretation of sign language, this system offers an accessible and efficient means of communication for the speech and hearing-impaired community, while promoting inclusivity in both personal and public environments.

Study of modules

1.NumPy

Purpose: Fundamental library for numerical computations in Python.
Key Features:

- Supports creation and manipulation of multidimensional arrays.
- Provides efficient array-based operations and broadcasting.
- Widely used for numerical processing in machine learning pipelines.

2.Pandas

Purpose: Data manipulation and analysis library built on top of NumPy.
Key Features:

- Supports creation and manipulation of multidimensional arrays.
- Provides efficient array-based operations and broadcasting.
- Widely used for numerical processing in machine learning pipelines.

3.OpenCV(cv2)

Purpose: Real-time computer vision library.
Key Features:

- Captures and processes images from webcam in real time.
- Supports image preprocessing tasks like resizing, thresholding, and color conversion.
- Crucial for feeding gesture frames into the model.

4.MediaPipe

Purpose: Framework for building multimodal applied machine learning pipelines.
Key Features:

- Used to extract hand landmarks from webcam frames
- Provides highly accurate and fast hand tracking.
- Extracted coordinates are used as input features for gesture recognition.

5.TensorFlow / Keras

Purpose: Deep learning and neural network model development..
Key Features:

- TensorFlow provides the backend for running and training models.
- Keras offers a user-friendly API to define and load the ANN model..
- Pre-trained .**keras** model is loaded using **load_model()** for real-time prediction.

6.Scikit-learn(sklearn)

Purpose: Machine learning library used for traditional ML algorithms.
Key Features:

- Random Forest was experimented with for gesture classification.
- Offers tools for model training, evaluation, and joblib-based serialization.
- Ultimately replaced by ANN due to memory efficiency and better performance.

7.Joblib

Purpose: Serialization of Python objects, especially machine learning models.

Key Features:

- Used for saving and loading models as well as encoder and decoder functions.
- Offers faster and more efficient storage for large numpy arrays.

8.Requests

Purpose: HTTP library for sending API requests.

Key Features:

- Used to send recognized words to a dictionary API to fetch word meanings.
- Enables real-time integration of external semantic information.

9.Threading

Purpose: Enables concurrent execution of multiple tasks

Key Features:

- Used to handle API calls and ESP32 communication without freezing the main interface.
- Prevents lagging or crashing when making HTTP requests or updating the LED matrix

10.Time

Purpose: Time-related functions.

Key Features:

- Used to manage delays between frame capture and prediction.
- Ensures smooth execution and prevents data overflow in real-time processing.

Research and Approach Selection

---

1. CNN-Based Image Classification

- Explored Idea: Use a Convolutional Neural Network to classify hand gesture images.
- Reference:
    - *Development of a Sign Language Recognition System Using ML*
    - *Najib (2024), Neural Computing and Applications*

      https://www.researchgate.net/publication/364185120_Real-Time_Sign_Language_Detection_Using_CNN

- Reason for Rejection: CNNs are computationally heavy and not suitable for real-time performance on limited hardware like ESP32.

2. MediaPipe + ANN (Finalized Approach)

- Explored Idea: Use MediaPipe to extract 21 hand landmarks and classify them using an Artificial Neural Network.
- Why We Chose It:
    - Lightweight, fast, and ideal for real-time applications.
    - ANN required less memory than CNN or Random Forest.
- Reference:
    - *Random Forest Classifier in SIBI Sign Language System*

3. YOLO-Based Detection

- Explored Idea: Use YOLO for real-time detection and classification as it can help predict various words on the basis of relative position of hand with face and body.
- Reason for Rejection: requires high-end GPU support.

4.Dynamic vs Static Gestures

- Initially planned to use LSTM for motion-based gesture prediction (like letters 'J' and 'Z').
- Challenge: Building a large sequential dataset was not feasible.
- Final Decision: Shifted to static gesture recognition using ANN for better efficiency and ease of implementation.

Model architecture

---

1. Importing Necessary Libraries

The system uses essential Python libraries to enable real-time sign language interpretation:

- **NumPy** for numerical operations and array handling.
- **Pandas** for managing and preprocessing dataset-related information.
- **OpenCV (cv2)** to capture and process live video frames.
- **MediaPipe** to detect and extract hand landmarks efficiently.
- **TensorFlow & Keras** to define, train, and load the ANN model.
- **Requests & Threading** to handle asynchronous API calls and ESP32 communication.

2. Data Preprocessing and Loading

- **Landmark Extraction:** Used MediaPipe to detect 21 hand landmarks and extract (x, y) coordinates from webcam frames.
- **Feature Flattening:** These coordinates were flattened into a 1D vector to serve as input features for the model.
- **Label Encoding:** Class labels for alphabets and static words were converted into numeric categories.
- **Normalization:** Landmark coordinate values were normalized to ensure uniform input scaling.
- **Train-Test Split:** The dataset was split into training and validation sets to evaluate model performance fairly.

3. Model Architecture (ANN)

The model used for sign language classification is a simple yet effective **Artificial Neural Network**:

- **Input Layer:** Accepts a 40-dimensional input vector (20 landmarks × 2 coordinates) (all the 21 landmarks were normalised with respect to a base landmarks thus it can be ignored during training) .
- **Hidden Layers:** Multiple dense layers with ReLU activation to learn high-level representations.
  1.Layer 1: 128 neurons with ReLU
  2.Layer 2: 64 neurons with ReLU
- **Dropout Layer:** Applied dropout to reduce overfitting and encourage generalization.
- **Output Layer:** Softmax activation to classify among multiple gesture classes (like A-Z, Hello, Yes, No, etc.).
  Note: Random Forest was initially tested for classification due to its speed and interpretability, but it was replaced by the ANN model owing to better memory efficiency and real-time performance.

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	5,248
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 29)	1,885

```
Total params: 46,169 (180.35 KB)
Trainable params: 15,389 (60.11 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 30,780 (120.24 KB)
```

4. Model Compilation

- **Loss Function:** Sparse Categorical Crossentropy – appropriate for multi-class classification.
- **Optimizer:** Adam – adaptive learning optimizer for fast convergence.
- **Metrics:** Accuracy – used to evaluate model correctness during training.

5. Model Training

- The ANN model was trained on the custom dataset using `model.fit()` with:
    1. **Epochs:** 100 (though the model eventually converged at 53th epoch due to implementation of early stopping)
    2. Batch Size:32
- Real-time gesture samples were collected and augmented to improve robustness.

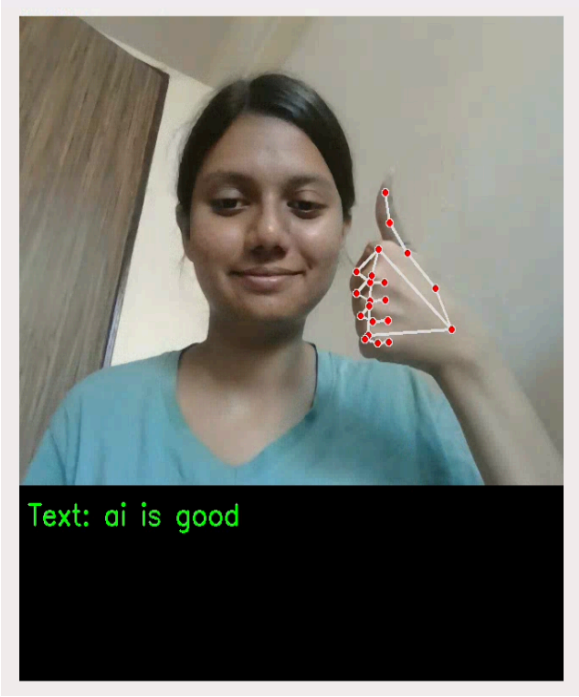
```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_acc:.2f}")
```

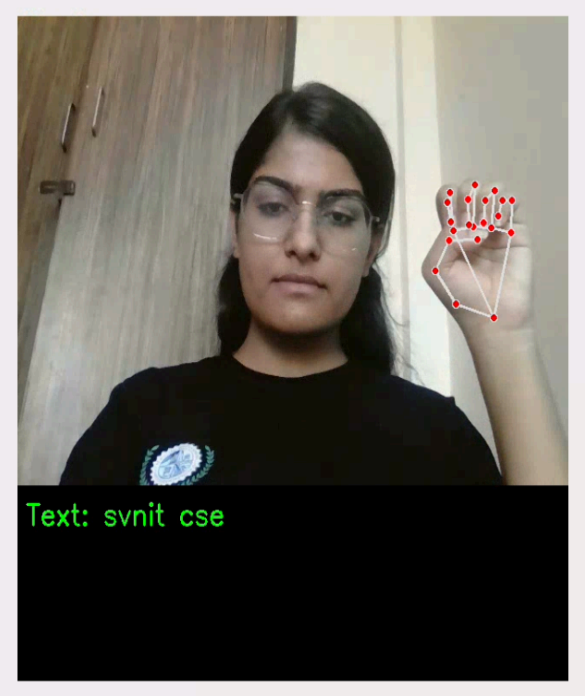
```
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9458 - loss: 0.1941
Test Accuracy: 0.95
```

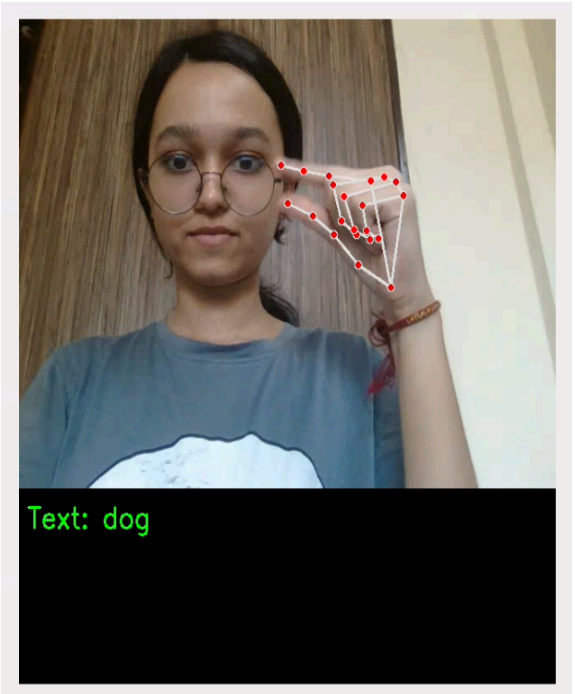
6. Model Inference and Real-Time Deployment

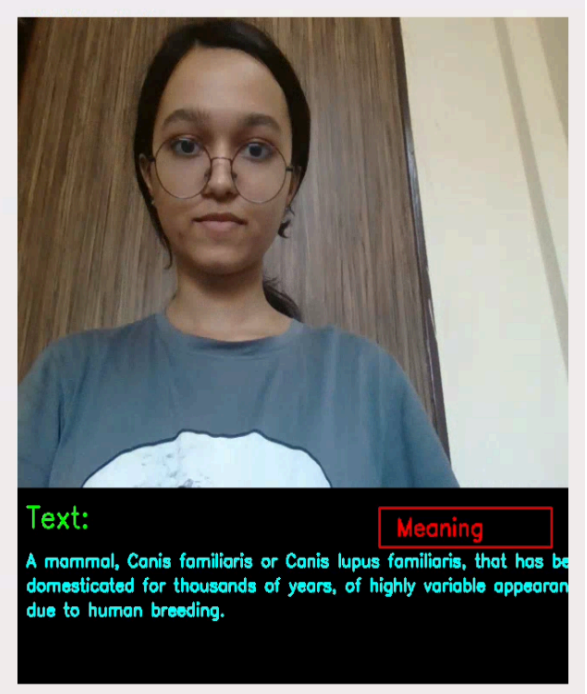
- Live webcam feed is captured using OpenCV for continuous frame processing.
- MediaPipe's Hands module is utilized to detect and extract 21 hand landmarks from each frame in real time.
- The extracted hand landmarks are normalized relative to the wrist joint and flattened to create a feature vector representing the hand gesture.
- This processed vector is then passed to a trained Artificial Neural Network (ANN) model that predicts the most probable alphabet class corresponding to the gesture.

- The predicted characters are accumulated into a word, which is then:
    - Displayed on an LED grid via ESP32 by sending an HTTP request with the word as a parameter.
    - Simultaneously, a dictionary API is queried to fetch the meaning of the detected word for educational support.
- Threading is used to perform API requests and ESP32 communication asynchronously, ensuring that the real-time webcam feed and inference loop remain smooth and responsive









Errors faced and solutions

Throughout the development of our sign language interpreter system, we faced several technical and logical challenges. These hurdles not only shaped the final architecture of our project but also deepened our understanding of real-time systems, machine learning limitations, and hardware integration. Below is a detailed account of the issues we encountered and how we overcame them.

1. Motion Gesture Prediction Using LSTM

Problem:
Our initial plan was to recognize dynamic hand gestures by capturing 20 continuous frames using MediaPipe and processing them with an LSTM (Long Short-Term Memory) model. The idea was to detect motion patterns to predict complete words based on sequential hand movements.
Challenge:
LSTMs are highly dependent on large and well-labeled sequential datasets to perform accurately. Since we were working with a custom dataset, it was not feasible to collect thousands of labeled video sequences for each gesture class. This lack of data significantly limited the model's learning capacity.
Solution:
We decided to pivot from dynamic to static gesture recognition, focusing on recognizing individual letters and static words. This allowed us to build a lighter model that required fewer samples per class, fit our real-time constraints, and reduced overall complexity without compromising the effectiveness of our system.

2. Model Architecture Decision: Random Forest vs ANN

Problem:
To identify the best classification algorithm, we experimented with both Random Forest and Artificial Neural Network (ANN) approaches.
Challenge:
We noticed that while Random Forest trained the model significantly faster than ANN, it also consumed substantially more memory, which became a concern for real-time performance and deployment on devices like the ESP32 or other constrained environments.
Solution:
After evaluating both models, we chose to proceed with ANN because of its lower memory footprint and smoother integration with real-time pipelines. While ANN took longer to train, it offered more consistent and optimized performance during inference, especially when deployed in hardware-limited settings.

3. Autocorrection using BERT

Problem:
To handle minor inaccuracies in gesture recognition (e.g., if a user gestured "helo" instead of "hello"), we initially tried integrating a BERT-based autocorrection system.

Challenge:
BERT often misinterpreted short or partial inputs and suggested semantically unrelated words. For example, instead of correcting "helo" to "hello," it might suggest "help" or even "helmet." This created confusion and undermined the usability of the system.
Solution:
We scrapped the autocorrection logic and instead implemented a dictionary API that displays the meaning of the predicted word when a "Meaning" button is clicked. This simplified the pipeline and ensured that the user always saw a valid, unambiguous result corresponding to their hand gesture.

4. ESP32 Hardware Issues: Wi-Fi & Flash Memory

Problem:
During the hardware integration phase, we encountered issues with the ESP32 board, specifically in connecting it to Wi-Fi and handling LED display memory.

Challenge:
Even after successful predictions, the previous word often remained on the MAX7219 LED matrix, creating confusion and overlapping displays. Initially, we didn't realize that the ESP32 was retaining the old data in its flash memory.

Solution:
After researching and testing, we discovered that reformatting the flash memory before each new input was necessary to clear previous predictions. Implementing this fix resolved the issue and ensured only the latest predicted word was shown on the display. We also optimized our Wi-Fi setup to ensure smooth and consistent connectivity during testing and demos.

5. Random Predictions on Hand Removal & Reappearance

Problem:
During real-time testing, we observed that when a user removed their hand from the camera view and then brought it back, the system would sometimes generate a random prediction momentarily before recognizing the correct word.

Challenge:
This flicker of incorrect predictions created a poor user experience and occasionally triggered unwanted outputs on the LED grid or API requests.

Solution:
We implemented a variable for counting the number of frames to be ignored after reappearance. This variable was used to skip prediction on the next 3 frame, allowing the system to stabilize before accepting new input. After one skipped frame, the variable was decremented. This simple mechanism smoothed the transition between gestures and eliminated unwanted predictions.

6. Limited Access to Advanced Dictionary APIs

Problem:
As part of enhancing the system's output, we explored various dictionary APIs that could provide richer definitions, synonyms, or multilingual support to improve the word meaning experience for users.

Challenge:
While evaluating multiple APIs, we discovered that most of the high-quality or comprehensive APIs—such as Oxford, Merriam-Webster, or WordsAPI—required paid subscriptions for extended usage.

Solution:
To keep the system fully functional and cost-effective, we decided to stick with a free-tier API that provided basic yet reliable definitions of words. Although it offers limited functionality, it served the core purpose of our application.

7. Lag During API Calls and ESP32 Communication

Problem:
We observed significant lag or freezing in the system while performing certain tasks—particularly when fetching word meanings from the dictionary API or sending the predicted word to the ESP32 LED matrix for display.
Challenge:
These operations—especially the API call and network-based communication—were blocking the main thread. As a result, the gesture recognition loop was interrupted, causing delays in prediction and a poor real-time experience for the user. The system would appear unresponsive until the task was completed.
Solution:
To resolve this, we implemented multithreading in our Python code. By running tasks like API calls and ESP communication on separate threads, we were able to prevent blocking the main gesture prediction process. This significantly improved the responsiveness and fluidity of our application, enabling it to operate smoothly even during background operations.

TimeLine

- **Week 1–2:** Conducted problem statement analysis and explored gesture recognition through research.
- **Week 3–4:** Collected and preprocessed a custom static gesture dataset using MediaPipe; researched ESP32 for hardware integration.
- **Week 5–6:** Created a dynamic gesture dataset and implemented LSTM for predictions, but results were unsatisfactory.
- **Week 7–8:** Explored ANN and Random Forest architectures; finalized ANN after comparative analysis of all models.

- **Week 9:** Completed final integration of the trained model with ESP32 and LED dot matrix display.
- **Week 10:** Built the interface between model and hardware, optimized performance using multithreading, performed testing, and resolved bugs and errors.

Conclusion

This project successfully developed a real-time sign language interpreter capable of recognizing static hand gestures and converting them into readable text. By leveraging MediaPipe for accurate hand landmark detection and training an Artificial Neural Network (ANN) on normalized coordinate data, the system achieved efficient gesture classification.

The integration of OpenCV enabled seamless webcam-based input, while ESP32 facilitated real-time display of interpreted words on an LED grid. Additionally, a dictionary API was incorporated to provide instant word meanings, enhancing the system's educational utility.

Threading ensured uninterrupted performance by offloading network and hardware communication tasks.

This solution demonstrates strong potential for improving communication accessibility, especially for individuals with hearing or speech impairments, and can be extended further to support dynamic gestures and multilingual interpretation.

References

[1] Najib, F. M. (2024). Sign language interpretation using machine learning and AI. Neural Computing and Applications.
[2] Najib et al. (2025). Development of a Sign Language Recognition System Using Machine Learning.
[3] PLoS ONE. Sign language recognition using lightweight attentive CNN with Random Forest. https://pmc.ncbi.nlm.nih.gov/articles/PMC10994320/