COE-379L

Final Project Report

Introduction and Problem Statement

Our project aims to develop a real-time Sign Language Interpreter that uses computer vision to recognize hand gestures corresponding to letters of the alphabet. This tool is designed to bridge the communication gap between individuals who use American Sign Language (ASL) and those who do not. By translating signed letters into their corresponding text outputs, the project supports accessibility and inclusion for the deaf and hard-of-hearing communities.

Data Sources and Technologies Used

The core of our dataset originates from a bespoke capture pipeline (`collect_imgs.py`) that leverages a standard webcam to record static ASL hand gestures under real-world conditions. At runtime, the script clears and recreates a `./data` directory, defines 24 target gesture classes, and captures 100 frames per class (2,400 total images). During collection, the operator sees a live preview ("Ready? Press 'Q'") and can adjust hand position, lighting, and background before each 100-frame batch is saved into its class-labeled subfolder (0–23, Note the letters J and Z were omitted from our model since they are dynamic letters, so you'd have to train the motion rather than the landmarks resulting in the 0-23 subfolders rather than 0-25). This approach captures natural variation in pose, illumination, and background—critical for training a model that must generalize making the model robust (All ASL letters can be seen below).

Figure 1: ASL letters

As a contingency against underrepresented or challenging gestures, we incorporated the Kaggle "ASL Dataset" (A–Z plus 0–9, 70 images per class, 2,520 images total). These serve as auxiliary examples: whenever landmark detection proves inconsistent for a given class, we merge in the corresponding Kaggle samples to balance the training set and ensure every gesture is well-represented.

All raw images—both from our pipeline and, if needed, from Kaggle—are then funneled through `create_dataset.py`. Here, MediaPipe Hands (static image mode, 0.3 confidence threshold) detects up to 21 landmarks per hand (image of the indexing can be seen below). We extract each landmark's (x, y) coordinates, normalize them by subtracting the per-image

minima (ensuring translation invariance), and concatenate them into 42-dimensional feature vectors. Only images yielding the full set of 21 landmarks are kept; partial detections are discarded to preserve data quality. The resulting `data` and `labels` arrays are serialized into `data.pickle`, enabling seamless batch loading for downstream model training. This three-tiered strategy—live capture, backup augmentation, and rigorous preprocessing—yields a robust, diverse dataset tailored for real-time ASL letter recognition.

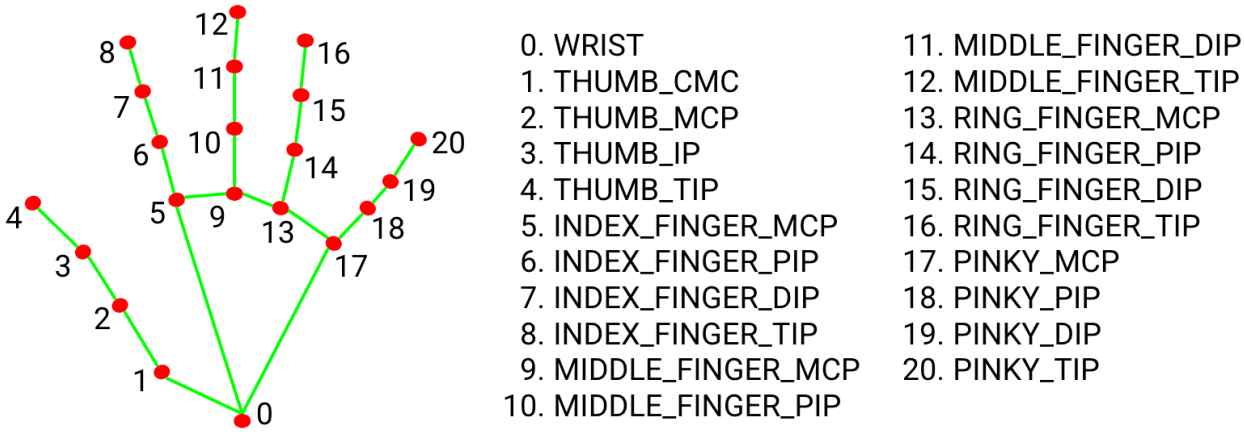


Figure 2: Mediapipe hand landmarks

Methods employed

Our Sign Language Interpreter pipeline is organized into four sequential stages—data acquisition, preprocessing, model training, and real-time inference—each implemented as a standalone Python script.

Libraries Used

- MediaPipe - Detects hands in video frames and assigns 21 precise landmark points that represent key positions on the hand.
- OpenCV - Captures live video from the webcam and displays frames with real-time annotations and overlays.
- Pickle - Saves and loads Python objects, such as machine learning models, by converting them into a file format that can be reused later.

- Numpy - Python library that provides fast, efficient tools for numerical computing, especially for working with large arrays, matrices, and performing mathematical operations on them.

1. **Data Acquisition (`collect_imgs.py`)**
   We first gather our core dataset via a live capture pipeline. Upon execution, the script initializes a standard webcam feed and prompts the operator through 24 target gesture classes (ASL letters A–Y, excluding J and Z which require motion). For each class, the user positions their hand in front of the camera, confirms readiness by pressing **Q**, and the script then automatically records 100 frames into a class-labeled directory (totaling 2,400 images). This approach captures real-world variability—differences in hand orientation, lighting, and background—which is critical for downstream model generalization.

2. **Feature Extraction & Dataset Construction (`create_dataset.py`)**
   All captured images (plus any supplemental Kaggle samples, if needed) are processed to extract a compact, translation-invariant representation of hand shape. Using MediaPipe Hands in static-image mode (min_detection_confidence=0.3), we detect up to 21 landmarks per frame. For each successful detection, the script:

   - Collects all landmark **x** and **y** coordinates and computes their minima.
   - Normalizes each coordinate by subtracting the respective minima—ensuring that the feature vector is invariant to absolute hand position.
   - Concatenates the 21 normalized (x, y) pairs into a 42-dimensional feature vector. Frames with incomplete landmark sets are discarded to maintain data integrity. The resulting feature matrix and corresponding labels are serialized to `data.pickle` for efficient loading.

3. **Model Training (`train_classifier.py`)**
   In the model training phase, we transform our preprocessed landmark data into a

robust statistical classifier. First, the script unpickles `data.pickle`, which contains an array of shape (N,42)(N, 42)(N,42) holding the normalized (x,y)(x,y)(x,y) coordinates for 21 hand landmarks per frame, alongside a matching label array of length NNN. We then use scikit-learn's `train_test_split` with an 80/20 split, `shuffle=True`, and `stratify=labels` to ensure each of the 24 gesture classes remains proportionally represented in both training and test sets. With the data partitioned, a `RandomForestClassifier()` (100 trees, Gini impurity, default feature sampling) is instantiated and fit to the training split. The reasoning behind Random Forests is they provide fast, low-latency inference on our 42-feature inputs, capture complex nonlinear hand-pose patterns without extensive tuning, and resist overfitting on our moderate-sized dataset—making them ideal for real-time ASL classification. Each tree is built on a bootstrap sample, and at each node a random subset of the 42 features is considered, which collectively mitigates overfitting and captures diverse decision boundaries across the high-dimensional gesture space. After training, the ensemble's performance is quantified by predicting on the held-out test set and computing an overall accuracy score; this quick diagnostic can be extended to per-class precision and recall analyses to identify challenging gestures. Finally, the fully trained model object is serialized to `model.p` via pickle, allowing subsequent real-time inference to bypass retraining entirely and load the ensemble in under a second.

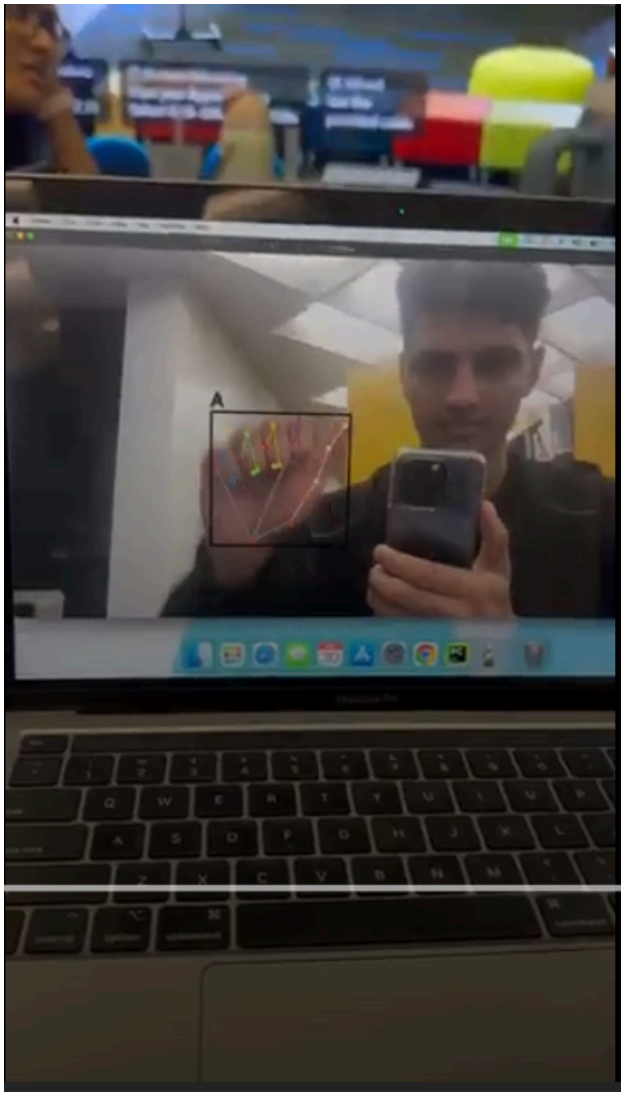
4. **Real-Time Inference (`inference_classifier.py`)**
   In the real-time inference phase, the `inference_classifier.py` script couples OpenCV video capture with MediaPipe landmark detection and our pre-trained Random Forest to deliver smooth, frame-by-frame ASL recognition. Upon launch, the script loads `model.p` into memory and opens the default webcam. MediaPipe Hands is configured in static-image mode with a 0.3 detection confidence threshold, trading temporal smoothing for immediate responsiveness on each frame. Inside the main loop, each BGR frame is converted to RGB and passed to `hands.process`; if no hand is detected, the frame is skipped. When a hand is found, we draw its 21 landmarks and corresponding connections directly onto the frame for user feedback. We then extract the raw (x,y)(x,y)(x,y) coordinates of all landmarks, compute their per-frame minima, and subtract these minima from each coordinate

to produce a translation-invariant 42-dimensional feature vector. Only frames yielding the complete set of 42 features proceed to classification; any partial detections are quietly discarded after a single warning. The normalized vector is fed to `model.predict`, returning a class index that we map to its corresponding ASCII letter. We compute a tight bounding box around the scaled landmark positions (with a small pixel margin), overlay this rectangle, and render the predicted character just above it using OpenCV's text functions. The annotated frame is displayed in real time, and pressing "q" cleanly terminates the loop, releasing the camera and closing all windows. This end-to-end integration ensures that, at typical webcam resolutions, gesture predictions appear with negligible lag—enabling a fluid, interactive ASL translation experience.

Collectively, these four scripts implement a turn-key pipeline—from raw image capture through to live gesture recognition—designed for robustness and ease of use in accessible, real-time ASL interpretation.

Results

Our results can be seen in the quick demonstration video below (Click on image it should give a link to view the video if not there is also the link at the bottom).



Quick Video Demo

Video 1: A quick demo of our results

As shown in the video, our system delivers live gesture detection via webcam using MediaPipe Hands/OpenCV, accurately translates static ASL signs (A–Z) into text, and provides immediate on-screen visual feedback— all with a little to no latency inference to maintain the balance between speed and prediction accuracy. There were a no times of wrong letters outputted when testing giving us around a 100% accuracy, but this could be due to the data being trained on our hands only and in certain environments which would cause for us to do more testing to fully see the robustness of our model, however

sometimes it would see another hand in frame and just guess a letter even if they weren't making one as seen towards the end of the video where it picks up the hand holding the phone as a W and this is also shown by Arsh in our video.

Future improvements or Extensions

Future improvements will include support for simultaneous left- and right-hand detection to handle two-handed signs, a sequence model (e.g., LSTM or lightweight transformer) paired with a language lexicon for continuous word recognition, and the addition of dynamic gesture classification—using trajectory tracking—to recognize the motion-based letters J and Z.

ChatGPT Uses

ChatGPT was used for debugging errors and understanding error messages as they were presented. When using OpenCV to collect the image data to train our model, the camera on our laptop would not open. After running our error through ChatGPT, we learned that our video capture was set to the wrong initial setting for our internal webcam, which resulted in our webcam not being opened when running the file.

References

We didn't use any references for this project.