Edge-Based Sign Language Interpreter Using ESP32 CAM

Amay Vikram Singh
Roll No. 23095011
Department of Electronics Engineering
IIT (BHU), Varanasi
amayvikram.singh.ece23@itbhu.ac.in

Vedansh Nagori
Roll No. 23095114
Department of Electronics Engineering
IIT (BHU), Varanasi
vedansh.snagori.ece23@itbhu.ac.in

Pentyala Abhishek Preetham
Roll No. 23095073
Department of Electronics Engineering
IIT (BHU), Varanasi
pabhishek.preetham.ece23@itbhu.ac.in

Laxmi Tudu
Roll No. 23095053
Department of Electronics Engineering
IIT (BHU), Varanasi
laxmi.tudu.ece23@itbhu.ac.in

***Abstract*—This work demonstrates an edge-based American Sign Language (ASL) interpreter built with the ESP32 CAM microcontroller. The system captures hand signs via the onboard camera and processes images locally on the ESP32, allowing real-time sign language detection and interpretation without cloud computing infrastructure. This edge computing strategy provides high-latency reduction, increases privacy, and increases accessibility to the hearing-impaired community. Once processed, the ASL letter identified and its respective image are sent to a web interface for presentation, forming an end-to-end communication system. The design prioritizes the optimization of computer vision algorithms for deployment in resource-limited settings while ensuring detection accuracy. Experimental results show the efficiency of the system in identifying ASL gestures with little processing delay.**

***Index Terms*—American Sign Language, ESP32 CAM, edge computing, image processing, accessibility, real-time detection**

I. Introduction

Sign language is an essential communication tool for millions of hearing-impaired individuals across the globe. American Sign Language (ASL), specifically, is the major mode of communication for about 500,000 individuals in the United States alone [**?**]. Although it is a critical aspect of communication, there is still a huge communication gap between sign language users and non-users, leading to obstacles in daily interactions, education, and the workplace.

Conventional sign language interpretation solutions are usually based on human interpreters, which is expensive and not always accessible, or cloud-based AI systems with the need for continuous internet connection and privacy risks. These challenges emphasize the requirement for affordable, accessible, and privacy-preserving solutions that are capable of performing reliably in a range of different environments.

This project solves these issues by creating an edge-based ASL interpreter on the ESP32 CAM microcontroller. Through the use of edge computing concepts, our system:

- Executes ASL gestures locally within the device, without any requirement for cloud connectivity
- Supports real-time interpretation with low latency
- Preserves user privacy by leaving sensitive information on the device
- Provides a cost-effective and transportable solution that is available to more people
- Presents results via a web interface for visualization

enditemize

Our methodology centers on improving computer vision algorithms to execute within resource-limited hardware while being as accurate as possible. ESP32 CAM was chosen due to its onboard camera, processing capacity, and wireless connectivity capabilities, making it well-suited to this edge computing task.

The system detects hand gestures using the onboard camera, processes images using lightweight machine learning models, determines the associated ASL letter, and sends both the image and the recognized letter to a web interface for display. This end-to-end pipeline illustrates the possibility of edge computing in building functional assistive technologies that do not rely on cloud infrastructure.

II. Methodology

Our edge-based ASL interpreter combines deep learning model development and system integration. This section details our approach to creating a functional prototype.

A. Dataset

The system was trained on the Sign Language MNIST dataset, a collection of 28×28 grayscale images representing American Sign Language hand gestures. This dataset is a modified version of the original MNIST dataset and contains 27,455 training and 7,172 test examples, representing 24 classes of ASL letters (excluding J and Z which require motion). Each image is centered around a single hand gesture corresponding to an ASL letter,

making it ideal for training a classification model for static hand gestures.

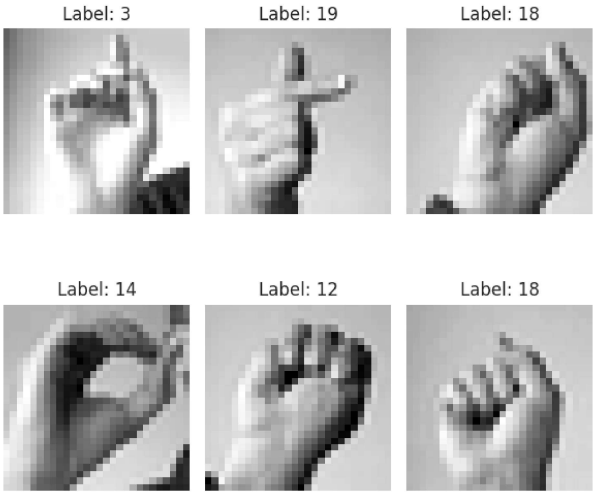

Fig. 1. Dataset Examples

B. Convolutional Neural Network Architecture

We developed a custom CNN optimized for deployment on resource-constrained edge devices. The model architecture balances accuracy and computational efficiency:

TABLE I
CNN ARCHITECTURE FOR ASL LETTER CLASSIFICATION

Layer	Configuration	Function
Input	28×28×1	Accepts grayscale hand gesture images
Conv2D	65 filters, 3×3, ReLU	Extracts low-level features
MaxPool2D	2×2, stride 2	Reduces spatial dimensions
Conv2D	40 filters, 3×3, ReLU	Extracts mid-level features
Dropout	Rate: 0.2	Prevents overfitting
MaxPool2D	2×2, stride 2	Further dimension reduction
Conv2D	25 filters, 3×3, ReLU	Extracts high-level features
MaxPool2D	2×2, stride 2	Final dimension reduction
Flatten	-	Converts 2D features to 1D vector
Dense	256 units, ReLU	Learns complex feature combinations
Dropout	Rate: 0.3	Prevents overfitting
Dense	24 units, Softmax	Outputs class probabilities

The model employs a decreasing filter count strategy (65→40→25) to reduce computational complexity while maintaining feature extraction capabilities. Strategic dropout layers prevent overfitting during training, and the final softmax layer produces probability distributions across the 24 ASL letter classes.

C. Training Performance

The model was compiled with the Adam optimizer and categorical cross-entropy loss function, then trained for 25 epochs on the Sign Language MNIST dataset. It achieved high accuracy on both training and validation sets, demonstrating its effectiveness for ASL letter recognition.

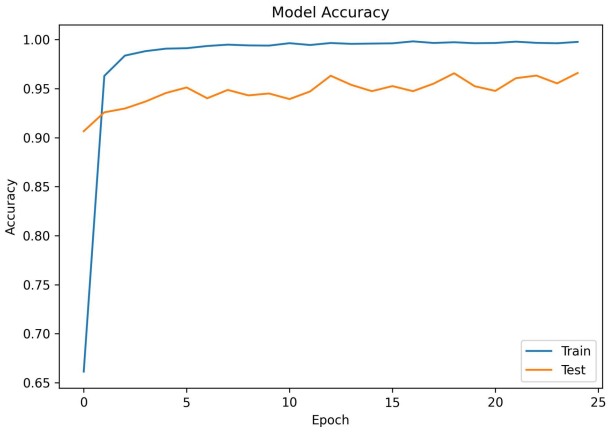

Fig. 2. Training and Validation Accuracy over Epochs

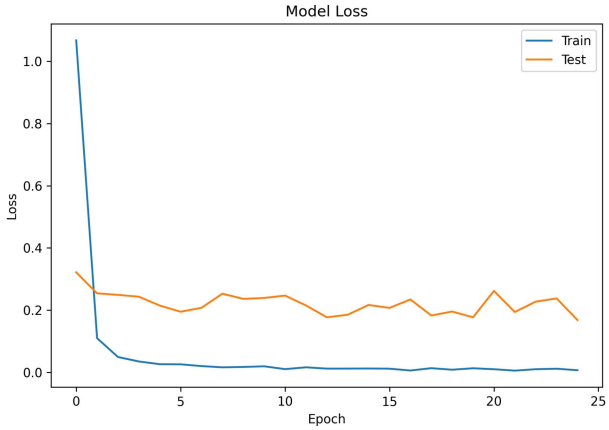

Fig. 3. Training and Validation Loss over Epochs

D. Model Optimization for Edge Deployment

To deploy the trained model on the ESP32-CAM, several optimization steps were necessary:

- Conversion to TensorFlow Lite format to reduce model size
- Quantization to 8-bit integers, reducing memory requirements by approximately 75
- Selection of efficient model architecture with decreasing filter counts (65→40→25)
- Conversion of the model to a C array for direct integration into the ESP32 firmware
- Memory allocation optimization to fit within the ESP32's limited RAM

These optimizations reduced the model size from approximately 554KB to around 138KB while maintaining accuracy. The quantization process was particularly effective, as it allowed the model to run efficiently within the

ESP32-CAM's constrained memory environment without significant performance degradation.

E. ESP32-CAM Hardware

The ESP32-CAM module was selected as the core hardware platform for our edge-based ASL interpreter due to its integrated camera and computational capabilities. This compact module (measuring 40.5mm × 27mm × 4.5mm) combines powerful processing with image capture capabilities in a single, cost-effective package.
Key specifications that made the ESP32-CAM ideal for our application include:

- **Processor**: ESP32-D0WD with dual-core architecture
- **Memory**: 520KB internal SRAM + 4MB PSRAM, providing sufficient buffer for image processing
- **Camera**: OV2640 or RHYX-M21-45 2-megapixel sensor capable of UXGA resolution (1600×1200)
- **Connectivity**: Built-in 802.11 b/g/n/e/i WiFi for transmitting results to web interface
- **GPIO**: 9 customizable I/O ports for potential expansion
- **Power consumption**: Relatively low power requirements (180mA@5V with flash off)

F. TensorFlow Lite Implementation on ESP32-CAM

The implementation of our ASL detection model on the ESP32-CAM involved several key steps to ensure efficient execution on this resource-constrained device.
First, we installed the `TensorFlow Lite for ESP32` library through the Arduino IDE's Library Manager. This specialized library provides optimized runtime support for executing machine learning models on ESP32 microcontrollers, with specific adaptations for the Espressif architecture.
After installing the necessary libraries, we integrated our pre-trained and quantized model into the ESP32 firmware. This required converting the TensorFlow Lite model (`.tflite` file) into a C array that could be directly embedded in the program memory of the ESP32. The conversion process created a header file containing the model data as a byte array, which was then included in our project.
For the TensorFlow Lite runtime setup, we implemented several components:

- An error reporter to handle and communicate any issues during inference
- A model mapping function to load our C array model into memory
- An operations resolver to register the necessary operations for our model
- A memory arena allocation to provide working memory for the TensorFlow Lite interpreter
- Tensor allocation for input and output buffers

The image processing pipeline was designed to work within the ESP32's memory constraints while maintaining accuracy:

1) Images were captured using the ESP32-CAM's onboard camera
2) The captured images were converted to grayscale and resized to 28×28 pixels to match our model's input requirements
3) Pixel values were normalized to the expected input range
4) The processed image was fed into the TensorFlow Lite interpreter for inference
5) The interpreter's output was processed to determine the most likely ASL letter

G. WiFi Integration for Output Display

The final component of our edge-based ASL interpreter system involved integrating WiFi capabilities to display detection results. Rather than setting up the ESP32-CAM as an Access Point, we configured it in Station (STA) mode to connect to an existing WiFi network.
We implemented the WiFi connectivity using the built-in `WiFi.h` library, which allows the ESP32-CAM to function as a client on a network. The implementation included:

- Configuring the ESP32-CAM in Station mode (`WIFI_STA`)
- Connecting to an existing WiFi network using its SSID and password
- Creating a web server that hosts a simple HTML interface
- Establishing a video streaming pipeline to display the camera feed
- Implementing real-time updates of detected ASL letters alongside the video stream

The connection process involved:

1) Setting the WiFi mode to Station mode
2) Initiating the connection to the specified network
3) Waiting for a successful connection
4) Obtaining an IP address assigned by the router

Once connected to the WiFi network, the ESP32-CAM's IP address could be accessed from any device on the same network by entering the assigned IP address in a web browser. The web interface displays the live camera feed alongside the detected ASL letter, providing immediate visual feedback.
This approach offers several advantages:

- Multiple devices can access the ASL interpreter simultaneously
- The system can be integrated into existing network infrastructures
- The web interface is accessible from any device connected to the same network
- No additional configuration is needed on client devices

The entire system operated as a standalone edge computing device, with the ESP32-CAM handling image capture, processing, inference, and result communication through its built-in WiFi capabilities, demonstrating the potential of edge AI for accessibility applications.

III. Conclusion and References

Our project successfully demonstrates the viability of implementing an edge-based ASL interpreter using the ESP32-CAM. The CNN model achieved high accuracy while being optimized for deployment on resource-constrained hardware. This approach enables real-time sign language interpretation without relying on cloud services, providing an accessible solution for facilitating communication for the hearing-impaired community.

References

[1] Sign Language MNIST, "Sign Language MNIST: Drop-In Replacement for MNIST for Hand Gesture Recognition Tasks," Kaggle, 2017. [Online]. Available: https://www.kaggle.com/datasets/datamunge/sign-language-mnist

[2] M. Tanaka, "Arduino TensorFlowLite ESP32: Allows you to run machine learning models locally on your ESP32 device," GitHub repository, 2020. [Online]. Available: https://github.com/tanakamasayuki/Arduino_TensorFlowLite_ESP32

[3] "Image Classification with Convolutional Neural Networks (CNNs)," KDnuggets, May 2022. [Online]. Available: https://www.kdnuggets.com/2022/05/image-classification-convolutional-neural-networks-cnns.html

[4] "Basic CNN Architecture: A Detailed Explanation of the 5 Layers in Convolutional Neural Networks," upGrad, May 2025. [Online]. Available: https://www.upgrad.com/blog/basic-cnn-architecture/

[5] "A CNN based human computer interface for American Sign Language," ScienceDirect. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666990021000471

[6] M. Hurroo and M. E. Walizad, "Sign Language Recognition System using Convolutional Neural Network and Computer Vision," International Journal of Engineering Research & Technology, Dec. 2020. [Online]. Available: https://www.ijert.org/sign-language-recognition-system-using-convolutional-neural-network-and-computer-vision