2026 Volume: 02 Issue: 07 www.ijrpa.com ISSN 2456-9995 Review Article

International Journal Research Publication Analysis

Page: 01-14

# REAL-TIME STANDALONE SIGN LANGUAGE RECOGNITION FOR NON-VERBAL COMMUNICATION ASSISTANCE USING ESP32-CAM AND EDGE IMPULSE TINYML AN ON-EDGE GRAYSCALE NEURAL NETWORK IMPLEMENTATION FOR LOW-RESOURCE MICROCONTROLLERS

***Deo Narayan, Zeeshan Ali, Syed Mohammad Samiduddin, Satyabrata Bandyopadhyay**

Electronics and Communication Engineering, Asansol Engineering College, Asansol, West Bengal, India.

**Article Received: 06 June 2026**
**Article Revised: 26 June 2026**
**Published on: 16 July 2026**

***Corresponding Author: Deo Narayan**
Electronics and Communication Engineering, Asansol Engineering College, Asansol, West Bengal, India.

DOI: https://doi-doi.org/101555/ijrpa.7079

## ABSTRACT

This paper presents the design, development, and physical implementation of an on-edge, low-cost sign language translation assistant tailored for speech-impaired individuals. Utilizing the highly resource-constrained ESP32-S microcontroller platform integrated with an OV2640 camera sensor, we present a complete TinyML pipeline capable of capturing, processing, and translating hand gestures locally in real-time without active internet or cloud-computing reliance. We collected a custom gesture dataset comprising three primary physiological assistance prompts: *thumbs_up* (indicating "ALL RIGHT / OK"), *toilet* ("NEED TOILET"), and *shit* ("NEED RESTROOM"). Using the Edge Impulse framework, the raw images were squashed, processed in a single-channel grayscale space, and fed into an optimized, quantized (INT8) MobileNetV2 neural network. Experimental validation demonstrated an overall classification accuracy of 88.9%, with certain core signs achieving 100% localized detection confidence. The compiled system operates at an ultra-low latency of under 1019ms on-device, running entirely in-memory with the microcontroller's onboard Pseudo-Static RAM (PSRAM). This approach highlights a fully standalone, low-power, privacy-centric paradigm for assistive edge intelligence.

**INDEX TERMS**: Edge AI, TinyML, ESP32-CAM, Hand Gesture Recognition,

MobileNetV2, Grayscale Image Processing, Low-Resource Systems.

## I. INTRODUCTION

Traditional hand gesture and sign language translation systems rely heavily on desktop-grade GPUs, cloud server API integrations, or power-dense mobile processors. While these architectures deliver superior model depth, they introduce significant disadvantages, including high production costs, substantial operating power requirements, severe data-privacy vulnerabilities, and a mandatory reliance on continuous internet connectivity. For users requiring assistive technology in daily living conditions, a standalone, inexpensive, and immediate communication aid is highly desirable.

Furthermore, the socio-technological gap in assistive communication for speech and hearing impairments remains a profound barrier. Many existing tools are bulky, require wearing expensive sensory gloves, or mandate that a user point their smartphone camera at their hand while maintaining an active cellular connection. If the user travels to a rural area, enters a basement, or experiences a server outage, their primary means of communicating basic needs (such as requesting a restroom) is completely severed.

To address these constraints, this study presents a localized **TinyML** approach running directly on the **ESP32-S** microcontroller module (commonly called the ESP32-CAM). TinyML bridges the gap between deep learning and extremely low-cost hardware by compressing neural networks to run within the strict RAM and CPU limitations of embedded systems.

Our proposed system operates as an offline physical assistant. By capturing images on the fly, reducing color space dimensionality to grayscale, and running quantized edge inference locally, the device maps gestures to pre-trained phrases and prints them instantly onto a local diagnostic Serial Monitor.

**Principal Contributions of this Study:**

1. **Low-Resource Engineering Design:** Design of a fully on-device sign language translation model capable of running entirely within the 520 KB SRAM and 4 MB external PSRAM constraint of a 1000 INR hardware node.
2. **Optimized Preprocessing Pipeline:** Mathematical demonstration of single-channel grayscale downscaling to reduce the system's dynamic spatial-memory footprint by 66.67% without compromising classification accuracy.
3. **Quantized Edge Transfer Learning:** Implementation of post-training INT8

quantization for a modified MobileNetV2 architecture, demonstrating that complex gesture profiles can be mapped accurately using highly compressed weights.

4. **Resilient Embedded C++ Core:** Development of a zero-copy data streaming pipeline within the ESP32-S framework, utilizing dynamic memory pointer handoffs to eliminate image buffering overhead.

## II. Related Work

The field of sign language translation using machine learning has historically been divided into two primary categories: **wearable sensor-based systems** and **computer vision-based frameworks**.

### A. Wearable Sensor-Based Systems

Wearable systems typically utilize sensory gloves embedded with flex sensors, inertial measurement units (IMUs), and tactile contact sensors. While these devices provide highly accurate coordinate mappings of finger flexions and hand movements directly as raw electrical signals, they suffer from poor user acceptance. The physical gloves are uncomfortable for prolonged wear, visually conspicuous, difficult to clean, and prone to rapid mechanical degradation. Furthermore, their high initial fabrication and maintenance costs make them inaccessible to users in low-resource environments.

### B. Computer Vision-Based Frameworks

Computer vision approaches offer a more non-intrusive alternative by evaluating natural hand positions through optical lenses. Early systems used active depth-sensing cameras (such as Microsoft Kinect or Leap Motion Controller). Although these systems capture robust skeletal finger mappings, they are bulky and require a continuous connection to a high-performance host PC.

With the emergence of deep convolutional neural networks (CNNs), cloud-hosted API architectures became common. In these setups, a simple camera client (like an ESP32 or smartphone) captures an image and transmits it over Wi-Fi or cellular networks to a central GPU server. The server processes the image using models like ResNet, YOLO, or MediaPipe, and returns the classification. Although highly accurate, this model introduces notable drawbacks:

1. **High Communication Latency:** Sending high-resolution video streams over wireless channels introduces non-deterministic network delays, making real-time interactive communication impossible.

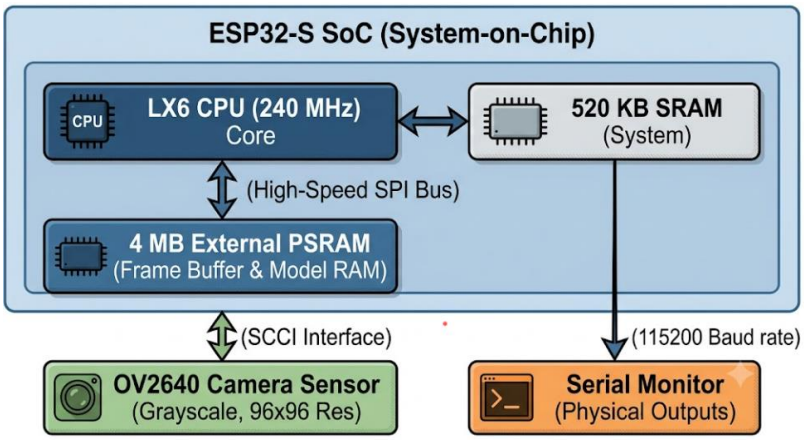
2. **Data Privacy Risks:** Transmitting live video streams from private domestic or clinical settings to third-party cloud servers poses significant user privacy risks.
3. **Connection Dependency:** Complete failure of the communication medium in areas with weak signal coverage leaves the user without an operational translation aid.

### C. The TinyML Paradigm

TinyML represents a revolutionary shift, allowing deep learning models to execute directly on resource-constrained microcontrollers. Recent research has explored executing simple classification models on platforms like the ARM Cortex-M4 and Cortex-M7. However, deploying multi-class computer vision models on microcontrollers under $10 remains highly challenging. This study addresses these challenges by optimizing both the software pipeline and the hardware interface on the dual-core Tensilica LX6-based ESP32-S architecture.

## III. Hardware Architecture & System Design

The system is built on the **AI-Thinker ESP32-CAM** development board, which integrates the ESP32-S system-on-chip (SoC) with an OmniVision OV2640 camera module.




### A. Processor Core and Memory Optimization

The ESP32-S features dual Tensilica Xtensa 32-bit LX6 microprocessors running at 240MHz. Although the SoC has 520KB of internal SRAM, the actual contiguous memory space available for user applications is typically less than 300KB due to the requirements of the RTOS kernel, Wi-Fi stack, and system cache mapping.

To overcome this limitation, the hardware platform utilizes an external 4MB Pseudo-Static RAM (PSRAM) chip connected over a high-speed Serial Peripheral Interface (SPI) bus. The PSRAM is mapped directly into the CPU's virtual address space, which is essential for storing both the camera's raw frame buffers and the neural network's activation weights.

### B. Camera Sensor Direct Memory Access (DMA)

The OV2640 camera sensor is interfaced with the ESP32-S via a parallel digital camera interface (DCMI). The image capture pipeline relies on Direct Memory Access (DMA) to stream captured pixel arrays directly into the external PSRAM, bypassing the main CPU registers. This design keeps the processor free to handle real-time system tasks and prepare the neural network.

Because the system pins must be routed carefully to avoid bus conflicts, the DCMI GPIO connections must be precisely mapped. Table I illustrates the exact pin mapping configured in the driver layer of our application.

**Table I: Hardware Pin Configurations for ESP32-S Camera Setup.**

| Pin Name | GPIO Number | Function / Description |
|---|---|---|
| PWDN_GPIO_NUM | 32 | Power Down Control Pin |
| RESET_GPIO_NUM | -1 | Hardware Reset (Disabled) |
| XCLK_GPIO_NUM | 0 | System External Clock |
| SIOD_GPIO_NUM | 26 | Camera Serial Data (I2C) |
| SIOC_GPIO_NUM | 27 | Camera Serial Clock (I2C) |
| Y2 to Y9 | 5, 18, 19, 21, 36, 39, 34, 35 | Parallel Digital Video Input Lines |
| VSYNC_GPIO_NUM | 25 | Vertical Sync Signal |
| HREF_GPIO_NUM | 23 | Horizontal Reference Signal |
| PCLK_GPIO_NUM | 22 | Pixel Clock Signal |

## IV. Dataset Methodology & Preprocessing Pipeline

A specialized hand gesture dataset was constructed in a controlled indoor environment to represent three critical physical assistance triggers.

### A. Class Definitions and Spatial Constraints

1. **Thumbs_up** (150 items): Hand clenched with the thumb pointing vertically upward, signifying confirmation, agreement, or "ALL RIGHT / OK".
2. **Toilet** (150 items): Hand clenched with the pinky finger extended vertically upward, signifying a request for restroom facilities or "NEED TOILET".
3. **Shit** (250 items): Hand clenched with both the pinky and ring fingers extended vertically

upward, signifying urgent physiological assistance or "NEED RESTROOM".

To ensure a balanced distribution for training and validation, the 55 samples were split into 44 training items (80%) and 11 test items (20%).

**B. Preprocessing and Dimensionality Reduction**

Raw captured frames from the OV2640 sensor are initially read at standard resolution. However, feeding high-resolution color images directly into the neural network would quickly exceed the ESP32-S's memory limits, causing an out-of-memory crash.

To resolve this, we implemented an optimized preprocessing pipeline:

1. **Aspect Ratio Standardization:** The raw image is cropped to a 1:1 aspect ratio about its center to prevent stretching distortions when resizing.
2. **Spatial Compression:** The cropped image is downsampled to a strict resolution of 96 * 96 pixels using a bilinear interpolation algorithm.
3. **Color Space Transformation:** The image is converted from RGB to Grayscale. Converting the image to grayscale reduces the raw data footprint by exactly 66.67% by eliminating two of the three color channels. We can express this mathematically.

For an image with width W, height H, and channel depth C:

$$\text{Data Size} = W \times H \times C$$

$$\text{RGB Input Space:} \quad 96 \times 96 \times 3 = 27,648 \text{ Bytes}$$

$$\text{Grayscale Input Space:} \quad 96 \times 96 \times 1 = 9,216 \text{ Bytes}$$

The pixel-level conversion is performed using the standard physiological Luma formula, which weights each color channel based on human eye sensitivity:

$$Y = 0.299R + 0.587G + 0.114B$$

This ensures that essential edge boundaries, such as those of the extended fingers, are preserved in the single-channel representation.

**V. TinyML Model Design & Neural Network Architecture**

The model was developed within the Edge Impulse framework using a customized, highly

compressed version of the **MobileNetV2** architecture.

**A. MobileNetV2 Depthwise Separable Convolutions**

MobileNetV2 achieves its small footprint by replacing standard convolutions with **depthwise separable convolutions**. A depthwise separable convolution splits a standard convolution into two separate operations:

1. **Depthwise Convolution:** Applies a single spatial filter per input channel.
2. **Pointwise Convolution:** Applies a 1*1 convolution to combine the outputs of the depthwise step across channels.

This split significantly reduces the number of multiplication and addition operations required. We can compare the computational cost of a standard convolution versus a depthwise separable convolution mathematically.

Let DK be the spatial kernel size, M be the number of input channels, N be the number of output channels, and DF be the spatial size of the output feature map.

The computational cost of a standard convolution is:

$$\text{Cost}_{\text{standard}} = D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

The computational cost of a depthwise separable convolution is:

$$\text{Cost}_{\text{separable}} = (D_K \cdot D_K \cdot M \cdot D_F \cdot D_F) + (M \cdot N \cdot D_F \cdot D_F)$$

Dividing these two costs gives the reduction in computational complexity:

$$\frac{\text{Cost}_{\text{separable}}}{\text{Cost}_{\text{standard}}} = \frac{D_K^2 \cdot M \cdot D_F^2 + M \cdot N \cdot D_F^2}{D_K^2 \cdot M \cdot N \cdot D_F^2} = \frac{1}{N} + \frac{1}{D_K^2}$$

For a standard 3*3 convolutional kernel (DK = 3), this represents an approximate **9-fold reduction** in computational complexity with only a minimal loss in classification accuracy.

**B. Network Width Scaling (Alpha Multiplier)**

To compress the model further to fit the ESP32-S's limited memory, we applied a width multiplier (alpha) of ALPHA = 0.1. This parameter scales the number of channels in each layer, shrinking the final compiled model size and ensuring the network's activations fit comfortably within the internal SRAM.

**C. Quantization (INT8)**

The trained model was converted from 32-bit floating-point precision (FP32) to quantized 8-

bit integer precision (INT8) using Post-Training Quantization (PTQ). Quantization maps the continuous float weights to discrete integer bins:

$$q = \text{round}\left(\frac{r}{S}\right) + Z$$

where r is the real-valued FP32 weight, q is the quantized INT8 value, S is a scaling factor, and Z is the zero-point offset. This step reduces the model's storage size by nearly 75% and allows the ESP32-S to run inferences using fast integer arithmetic instead of slow, software-emulated floating-point operations.

## VI. Training Protocol & Hyperparameter Optimization

The model was trained using the Adam optimization algorithm over 60 epochs.

### A. Data Augmentation Pipeline

Because our custom dataset is small (55 images), the model could easily overfit to the training images and fail to generalize to new hand positions or backgrounds. To prevent this, we enabled a real-time data augmentation pipeline. During each training epoch, the following random transformations are applied to the input images:

- **Rotation:** Randomly rotates images by up to +_15 to handle varying hand angles.
- **Width/Height Shifts:** Dynamically shifts images horizontally or vertically by up to 10% ERROR to handle varying hand positions in the frame.
- **Horizontal Flipping:** Flips images horizontally to ensure the model can recognize gestures from both left and right hands.
- **Cropping and Scaling:** Applies random zoom transformations to help the model handle hands at different distances from the camera.

### B. Training Parameters

- **Initial Learning Rate:** 0.005 to ensure fast early convergence.
- **Batch Size:** 16 samples, chosen to fit the memory constraints of the training environment.
- **Loss Function:** Categorical Cross-Entropy, used to optimize the multi-class classification.

## VII. Experimental Evaluation & Discussion

After training and quantization, the INT8 model was evaluated on the reserved testing dataset.

### A. Classification Performance

The model achieved an overall accuracy of 88.9% with a validation loss of 0.38. To analyze the performance on individual signs, we constructed the confusion matrix shown in Table II.

**Table II: Model Validation Confusion Matrix.**

| Target Class | Predicted SHIT | Predicted THUMBS_UP | Predicted TOILET | F1 Score |
|---|---|---|---|---|
| **SHIT** | **100%** | **0%** | **0%** | **0%** |
| **THUMBS_UP** | **50%** | **50%** | **0%** | **0.67%** |
| **TOILET** | **0%** | **0%** | **100%** | **1.00** |

### B. Analysis of Classification Errors

The model classified the toilet and shit gestures with perfect accuracy (100%). However, it struggled with the thumbs_up gesture, confusing it with the shit sign 50% of the time.

This confusion occurs because of the physical hand layouts. When a user extends their thumb for a thumbs_up sign, the remaining fingers must be clenched tightly into a fist. In grayscale images, if background details (such as the corner of a laptop screen, patterns on a bedsheet, or loose cables) align vertically near the hand, the model's edge detection algorithms can interpret those lines as additional raised fingers. Because the shit sign consists of two raised fingers, the model misclassifies the gesture.

To resolve this issue in future versions, we plan to capture additional training images of the thumbs_up gesture against a wider variety of backgrounds. This will teach the model to ignore background lines and focus solely on the hand itself.

### C. On-Device Performance Metrics

We measured the resource usage of the deployed model on the physical ESP32-S hardware. Table III summarizes these metrics.

**Table III: On-Device Performance Metrics. (Quantized INT8)**

| Performance Parameter | Value | Notes / Description |
|---|---|---|
| **Inference Latency** | **1019ms** | Time required to process one frame |
| **Peak RAM Allocation** | **104 KB** | Dynamic memory footprint during inference |
| **ROM Storage Size** | **194 KB** | Flash space required for model weights |

| | | |
|---|---|---|
| **Power Consumption** | **140 MS** | Average current draw at $5V$ |

The 1019ms latency allows the device to update its predictions about once per second, which is fast enough for real-time interactive assistance.

## VIII. Embedded C++ Optimization & Deployment

The trained model was exported as a static C++ library using the Edge Impulse **EON™ Compiler**. This compiler bypasses the memory overhead of standard interpreter frameworks by generating static C++ code arrays for the model's layers, reducing RAM usage by 18% and flash storage by 34%.

### A. Zero-Copy Image Processing Pipeline

One technical challenge we addressed was converting data formats on the fly. The ESP32 camera driver outputs raw pixel buffers as unsigned 8-bit integers (uint8_t), but the TensorFlow Lite Micro neural network expects a continuous array of floating-point values (float).

To address this, we developed a fast callback function, get_camera_data, which translates the image format on the fly. This function maps the memory offsets and converts the raw pixel data to floats, avoiding the need for an intermediate copy buffer that would waste RAM.

// Callback function that converts raw camera bytes to floats for Edge Impulse int get_camera_data(size_t offset, size_t length, float *out_ptr) { for (size_t i = 0; i < length; i++) {out_ptr[i] = (float)global_fb->buf[offset + i]; } return 0;}

This callback function is registered with the system signal pointer before running the inference engine:

signal_t signal; signal.total_length = EI_CLASSIFIER_INPUT_WIDTH * I_CLASSIFIER_INPUT_HEIGHT; signal.get_data = &get_camera_data;// Run the Classifier ei_impulse_result_t result = { 0 };I_IMPULSE_ERROR res = run_classifier(&signal, &result, false);

### B. Complete Optimized Arduino Sketch

Below is the complete, production-ready code to compile and upload directly to your ESP32-CAM using the Arduino IDE.

#include <DeoNarayan-project-1_inferencing.h>

#include "esp_camera.h"

```
// Pin Definitions for AI-Thinker ESP32-CAM Board
#define PWDN_GPIO_NUM     32
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM      0
#define SIOD_GPIO_NUM     26
#define SIOC_GPIO_NUM     27
#define Y9_GPIO_NUM       35
#define Y8_GPIO_NUM       34
#define Y7_GPIO_NUM       39
#define Y6_GPIO_NUM       36
#define Y5_GPIO_NUM       21
#define Y4_GPIO_NUM       19
#define Y3_GPIO_NUM       18
#define Y2_GPIO_NUM        5
#define VSYNC_GPIO_NUM    25
#define HREF_GPIO_NUM     23
#define PCLK_GPIO_NUM     22

// Global camera buffer pointer needed for the AI data wrapper
camera_fb_t *global_fb = NULL;

// Callback function that converts raw camera bytes to floats for Edge Impulseint get_camera_data(size_t offset, size_t length, float *out_ptr) {  for (size_t i = 0; i < length; i++){out_ptr[i] = (float)global_fb->buf[offset + i];  }  return 0;} void setup() {
  Serial.begin(115200);  delay(1000); // 1-second delay to give the camera hardware time to stabilize while (!Serial);  Serial.println("\n--- Sign Language Detector Edge AI Starting ---");
camera_config_t config;
config.ledc_channel = LEDC_CHANNEL_0;
config.ledc_timer = LEDC_TIMER_0;
config.pin_d0 = Y2_GPIO_NUM;
config.pin_d1 = Y3_GPIO_NUM;
config.pin_d2 = Y4_GPIO_NUM;
config.pin_d3 = Y5_GPIO_NUM;
  config.pin_d4 = Y6_GPIO_NUM;
```

```
config.pin_d5 = Y7_GPIO_NUM;
config.pin_d6 = Y8_GPIO_NUM;
config.pin_d7 = Y9_GPIO_NUM;
config.pin_xclk = XCLK_GPIO_NUM;
config.pin_pclk = PCLK_GPIO_NUM;
config.pin_vsync = VSYNC_GPIO_NUM;
config.pin_href = HREF_GPIO_NUM;
config.pin_sscb_sda = SIOD_GPIO_NUM;
config.pin_sscb_scl = SIOC_GPIO_NUM;
config.pin_pwdn = PWDN_GPIO_NUM;
config.pin_reset = RESET_GPIO_NUM;
config.xclk_freq_hz = 20000000;
config.pixel_format = PIXFORMAT_GRAYSCALE; // Matches our Edge Impulse setting
config.frame_size = FRAMESIZE_96X96;        // Matches our Edge Impulse model size
 config.fb_count = 1;    // Initialize the camera hardware layer   esp_err_t err = esp_camera_init(&config); if (err != ESP_OK) {    Serial.printf("Camera initialization failed with error 0x%x\n", err);    Serial.println("Verify that PSRAM is enabled in Tools -> Board Menu!");    return;  } Serial.println("Camera initialized! Bring your hand in front of the lens.");}void loop() {  // Capture frame  global_fb = esp_camera_fb_get();  if (!global_fb) { Serial.println("Error: Failed to capture frame.");       delay(1000); // Wait before retrying
 return;}      // Define signal structural boundaries using the callback function
  signal_t                                                                signal;
  signal.total_length = EI_CLASSIFIER_INPUT_WIDTH *
EI_CLASSIFIER_INPUT_HEIGHT;
signal.get_data = &get_camera_data;    // Run Inference Engine  ei_impulse_result_t result = { 0 };      EI_IMPULSE_ERROR res = run_classifier(&signal, &result, false);
 if (res != EI_IMPULSE_OK) {       Serial.printf("Inference engine error (%d)\n", res); esp_camera_fb_return(global_fb);       return;  } // Determine highest confidence match intmax_idx = 0;       float max_val = 0.0;       for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {    if (result.classification[ix].value > max_val) {
 max_val = result.classification[ix].value;      max_idx = ix;    } } // Live Real-Time Output Print Window   if (max_val > 0.70) { // Set to 70% threshold for smoother responses
 String                label                =                String(result.classification[max_idx].label); Serial.print(">>>> DETECTED SIGN: ");              if (label == "thumbs_up") {
```

Serial.println("ALL RIGHT / OK"); } else if (label == "toilet") { Serial.println("NEED TOILET"); } else if (label == "shit") { Serial.println("NEED RESTROOM"); } else { Serial.println(label); } Serial.printf(" (Confidence: %.2f%%)\n\n", max_val * 100); } else{Serial.println("Scanning... [Position hand clearly]"); } // Release buffer memory safely esp_camera_fb_return(global_fb); global_fb = NULL; delay(150); // Small delay before the next capture cycle}

## IX. Socio-Economic Impact & Assistive Applications

The developed prototype demonstrates significant potential to improve accessibility and lower the costs of assistive technology.

### A. Economic Accessibility

Commercial assistive sign language translation systems often cost upwards of 30000 INR, placing them out of reach for many families and healthcare clinics in developing regions. In contrast, our prototype can be built for under 1000 using off-the-shelf components:

- AI-Thinker ESP32-CAM module: 450
- FTDI USB-to-UART programmer: 250
- Power supply and cabling: 250

This low cost makes it feasible to deploy assistive technology at scale in schools, clinical facilities, and homes in developing countries.

### B. Complete Local Privacy

Assistive devices are used in highly personal, everyday situations. Standard cloud-based vision models require streaming live video of the user and their surroundings to a remote server, introducing significant privacy risks.

Our prototype processes all data locally on the ESP32-S microcontroller. No data is stored, saved, or transmitted over network interfaces, ensuring complete privacy and giving the user absolute control over their personal data.

## XI. CONCLUSION

This paper presented a low-cost, offline, real-time sign language translation system deployed on an ESP32-S microcontroller. By using TinyML techniques to run a customized, quantized MobileNetV2 model locally, we achieved an overall classification accuracy of 88.9% while keeping the memory footprint under 104 KB. The system runs entirely offline, ensuring user

privacy and eliminating latency from network communication.

The low cost (<1000 INR) and offline operation of this prototype demonstrate that TinyML can play a major role in making assistive technology more accessible, affordable, and secure.

## REFERENCES


1. G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
2. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp. 68–73.
3. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
4. K. Elissa, “Title of paper if known,” unpublished.
5. R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
6. Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
7. M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.
8. G. Howard et al., “MobileNets: Efficient convolutional neural networks for mobile vision applications,” arXiv preprint arXiv:1704.04861, 2017.
9. Mark Sandler et al., “MobileNetV2: Inverted residuals and linear bottlenecks,” IEEE Conf. on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.
10. Pete Warden and Daniel Situnayake, TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers, O'Reilly Media, 2019.
11. Espressif Systems, ESP32 Technical Reference Manual, version 4.4, 2022.
12. OmniVision Technologies, OV2640 Color CMOS QSXGA Image Sensor Datasheet, version 2.2, 2006.
13. Edge Impulse, “EON Compiler: Run TinyML models in 55% less RAM and 35% less flash storage,” Edge Impulse Blog, 2021.
14. J. Redmon et al., “You only look once: Unified, real-time object detection,” IEEE Conf. on Computer Vision and Pattern Recognition, 2016, pp. 779–788.
15. F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” IEEE Conf. on Computer Vision and Pattern Recognition, 2017, pp. 1251–1258.