MARMARA
UNIVERSITY

Embedded Digital Image Processing

EE4065

Final Project

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

CNN Convolutional Neural Network

EE4065 Embedded Digital Image Processing

MCU Microcontroller Unit

MNIST Modified National Institute of Standards and Technology

RAM Random Access Memory

YOLO You Only Look Once

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report shows the implementation and analysis of the final project for the Embedded Digital Image Processing (EE4065) course. The primary objective is to implement computer vision and machine learning techniques on an embedded system for ESP32-CAM module. The project is around four questions each addressing some embedded vision pipelines.

The first question focuses on image preprocessing through the implementation of an Adaptive Thresholding algorithm. This technique is indispensable for real-world applications where lighting conditions vary unpredictably. The second question is the implementation of an object detection using the You Only Look Once (YOLO) architecture. We deploy a lightweight version of the YOLO model to detect handwritten digits. The third question deals with the image resizing by upsampling and downsampling. We implement custom algorithms directly on the Microcontroller Unit (MCU). These operations are essential for normalizing image inputs to the specific dimensions required by neural networks and for creating image pyramids in multi-scale processing. Finally, the fourth question involves a comparative study of various Convolutional Neural Network (CNN) architectures, including ResNet, MobileNet and SqueezeNet. We train these models on the Modified National Institute of Standards and Technology (MNIST) dataset and convert them for deployment using TensorFlow Lite Micro. This section evaluates the performance trade-offs between classification accuracy, inference latency, and memory footprint (Flash and Random Access Memory (RAM)), providing insights into the most suitable models for edge AI applications.

Throughout the project, we utilize the standard embedded development ecosystem, including the Arduino IDE and Python-based training scripts, to demonstrate a complete workflow from model training to hardware deployment.

2. QUESTION 1: ADAPTIVE THRESHOLDING

2.1. THEORY

2.1.1. Digital Image Representation and Thresholding

In computer vision an image is fundamentally represented as a two-dimensional matrix $f(x, y)$, where (x, y) denotes the spatial coordinates and the amplitude f represents the intensity or gray level at that point [1]. For an 8-bit grayscale image, this intensity takes integer values in the range $[0, 255]$, where 0 represents black and 255 represents white.

Thresholding is one of the simplest yet most effective segmentation techniques. It creates a binary image $g(x, y)$ from a grayscale image $f(x, y)$ based on a threshold value T:

$$g(x,y) = \begin{cases} 255 & \text{if } f(x,y) \geq T \\ 0 & \text{if } f(x,y) < T \end{cases} \tag{2.1}$$

While a fixed threshold is computationally inexpensive, it is often insufficient for real-world applications where lighting conditions fluctuate. Variations in illumination, camera exposure, or object reflectivity can cause a fixed T to either miss the object entirely or merge it with the background.

2.1.2. Size-Based Adaptive Thresholding

The problem addressed in this project involves detecting a single bright object on a darker background, where the object is known to occupy a specific area (approximately $P_{target} = 1000$ pixels). This prior knowledge allows us to formulate an adaptive thresholding problem: Find the optimal threshold T^* such that the number of foreground pixels (count of non-zero elements in $g(x, y)$) approximates P_{target}.

Let $N(T)$ be the function representing the number of white pixels for a given threshold T. $N(T)$ is a decreasing function because as T increases, the condition $f(x, y) \geq T$ becomes stricter and fewer pixels survive the thresholding operation.

$$T_1 < T_2 \implies N(T_1) \geq N(T_2) \tag{2.2}$$

This property allows us to employ a **Binary Search** algorithm to find T^*. The algorithm operates in $O(\log_2(256)) = O(8)$ iterations making it efficient for real-time systems compared to $O(256)$ search.

2.2. IMPLEMENTATION

2.2.1. Part A: Python Implementation (PC)

The first part of the quesetion involved verifying the algorithm on a PC using Python and OpenCV. The function `find_threshold_by_area` takes a grayscale image and a target pixel count, iteratively adjusting the threshold until the white pixel count falls within a specified tolerance.

Listing 2.1: Python Function for Size-Based Adaptive Thresholding

```
def find_threshold_by_area(gray_image, target_area=1000,
    tolerance_percent=1, max_iterations=16):
    """
    Find the threshold value that results in approximately target_area
        white pixels.
    Uses binary search for efficiency.
    """
    low, high = 0, 255
    best_threshold = 128
    best_error = gray_image.size
    best_binary = None

    tolerance = max(1, int(target_area * tolerance_percent / 100))

    for iteration in range(max_iterations):
        mid = (low + high) // 2

        # Apply threshold
        _, binary = cv2.threshold(gray_image, mid, 255, cv2.
            THRESH_BINARY)

        # Count white pixels
        pixel_count = cv2.countNonZero(binary)
        error = abs(pixel_count - target_area)

        # Track best result
        if error < best_error:
            best_error = error
            best_threshold = mid

```

```
        # Check if within tolerance
        if error <= tolerance:
            return mid, binary, pixel_count

        # Binary search adjustment
        if pixel_count > target_area:
            # Too many white pixels, increase threshold (stricter)
            low = mid + 1
        else:
            # Too few white pixels, decrease threshold (looser)
            high = mid - 1

        if low > high:
            break

    # Recompute with best threshold
    _, best_binary = cv2.threshold(gray_image, best_threshold, 255, cv2.
        THRESH_BINARY)
    return best_threshold, best_binary, cv2.countNonZero(best_binary)
```

2.2.2. Part B: ESP32-CAM Implementation

The deployment to the ESP32-CAM required handling the specific hardware constraints and memory architecture of the microcontroller.

2.2.2.1. Image Acquisition Pipeline

The ESP32-CAM utilizes the OV2640 sensor. Due to the limited internal RAM (520KB) we utilize the external PSRAM (4MB) to store image buffers. The pipeline is like this:

1. **Capture**: The camera captures a frame in JPEG format at QVGA resolution (320×240). JPEG is preferred over raw RGB capture as it allows the hardware to handle noise reduction during compression. Also to show a better image preview to user.

2. **Decode**: The JPEG is decoded into an RGB565 buffer ($320 \times 240 \times 2$ bytes).

3. **Grayscale Conversion**: We implement a `convertToGrayscale` function to transform RGB565 to 8-bit luminance values (≈ 76 KB buffer).

$$Y = 0.299R + 0.587G + 0.114B \tag{2.3}$$

2.2.2.2. C++ Algorithm

The binary search logic is ported to C++ for the ESP32. We utilize a `WebServer` interface to initiate captures and visualize results.

Listing 2.2: C++ Implementation of Adaptive Thresholding

```
// Binary search to find threshold that gives approximately target pixel count
uint8_t findOptimalThreshold(uint8_t *gray, uint8_t *binary, int size,
                             int targetPixels, int tolerance,
                             int maxIterations) {
  int low = 0;
  int high = 255;
  uint8_t bestThreshold = 128;
  int bestError = size;

  for (int iter = 0; iter < maxIterations; iter++) {
    int mid = (low + high) / 2;
    int count = applyThresholdAndCount(gray, binary, size, (uint8_t)mid);
    int error = abs(count - targetPixels);

    // Track best result so far
    if (error < bestError) {
      bestError = error;
      bestThreshold = (uint8_t)mid;
    }

    // Check if within tolerance
    if (error <= tolerance) {
      return (uint8_t)mid;
    }

    // Adjust search range
    if (count > targetPixels) {
      low = mid + 1; // Too many white pixels, increase threshold
    } else {
      high = mid - 1; // Too few white pixels, decrease threshold
    }

    if (low > high)
      break;
  }
  return bestThreshold;
}
```

2.3. RESULTS

2.3.1. Python Results

Algorithm logic was first verified on PC (Figure 2.1) before ESP32 deployment, confirming the binary search approach.

Figure 2.1: Python simulation results.

2.3.2. ESP32-CAM Results

The algorithm tested by using a mouse against a black background. Figure 2.2 shows the adaptability of the algorithm. With $P_{target} = 1000$ (Figure 2.2b), the object is correctly isolated. Increasing P_{target} to 9000 (Figure 2.2c) forces a lower threshold, incorrectly including background noise. This confirms the importance of accurate target sizing.

(a) Original Image **(b)** $P_{target} = 1000$ **(c)** $P_{target} = 9000$

Figure 2.2: ESP32-CAM adaptive thresholding results.

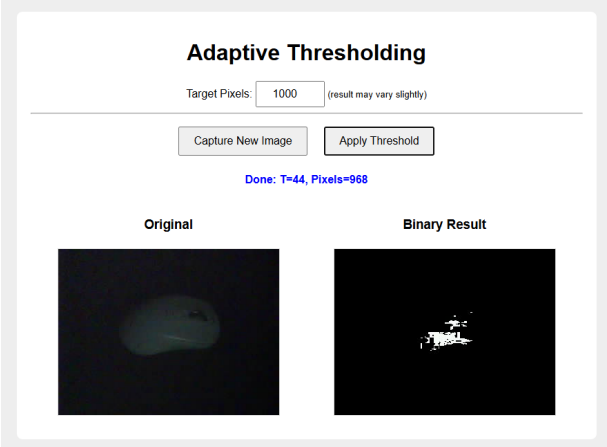


Figure 2.3: ESP32 Web Interface for Real-time Adaptive Thresholding.

The binary search proved efficient, converging in 6-8 iterations compared to 256 for a linear scan.

3. QUESTION 2: HANDWRITTEN DIGIT DETECTION VIA YOLO

3.1. THEORY

3.1.1. Object Detection and YOLO

Object detection finds and labels objects in an image. Older methods like R-CNN use two steps: first they find possible regions, then they classify each one. This works well but is slow.

YOLO (You Only Look Once) [2] uses a faster approach. It divides the image into a grid. Each grid cell predicts objects whose center is inside it. This one-step method is much faster and works well on devices like the ESP32-CAM.

3.1.2. YoloFastestV2 Architecture

We used **YoloFastestV2** from an open-source project [3]. This is a very small model made for mobile and embedded devices. Its main features are:

- **No Anchor Boxes**: Unlike older YOLO versions, it does not use fixed box shapes. This makes the output simpler.
- **Lightweight Layers**: Uses special convolutions that need fewer calculations.
- **Single Output**: Only one feature map (8×8 grid), which reduces memory use.

3.1.3. Output Format

Each grid cell outputs a vector with $5 + C$ values, where C is the number of classes (10 for digits 0-9):

$$\text{Output} = [\underbrace{p_{obj}}_{\text{Confidence}}, \underbrace{c_x, c_y, w, h}_{\text{Box Position}}, \underbrace{p_0, p_1, \ldots, p_9}_{\text{Class Scores}}] \tag{3.1}$$

Here, c_x, c_y are the box center offsets, and w, h are the box width and height.

3.2. IMPLEMENTATION

3.2.1. Dataset Generation

There is no ready dataset for multi-digit detection, so we made our own using `dataset.py`. The script creates training images with 3-6 digits from MNIST placed on a noisy background. Figure 3.1 shows examples of generated training samples.

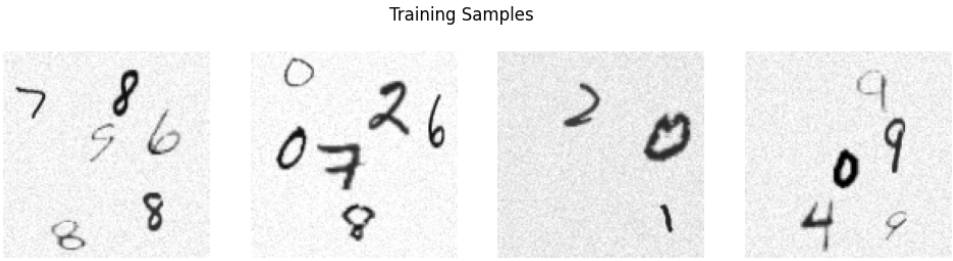


Figure 3.1: Training samples with multiple MNIST digits on synthetic backgrounds.

The core of the dataset generation is shown below. For each sample, digits are randomly placed and their positions are recorded as ground truth:

Listing 3.1: Dataset Generation (dataset.py)

```
class SyntheticDigitDataset(Dataset):
    def __init__(self, train_mode=True, root_folder="data"):
        self.source = datasets.MNIST(root=root_folder, train=train_mode,
            download=True, transform=None)
        self.pool = list(range(len(self.source)))
        self.res = RESOLUTION
        self.cells = CELLS
        self.cats = CATEGORIES

    def _make_background(self):
        shade = random.randint(220, 255)
        bg = np.full((self.res, self.res), shade, dtype=np.float32)
        jitter = np.random.normal(0, 10, (self.res, self.res))
        bg = np.clip(bg + jitter, 0, 255)
        return bg

    def _augment_digit(self, raw_img):
        if random.random() > 0.5:
```

```
            kern = np.ones((2, 2), np.uint8)
            raw_img = cv2.erode(raw_img, kern, iterations=1)
        new_h = random.randint(24, 48)
        new_w = random.randint(24, 48)
        return cv2.resize(raw_img, (new_w, new_h)), new_w, new_h
        ...
```

3.2.2. Model Architecture

Table 3.1 shows the model layers. The backbone has five convolutional blocks. Each block has convolution, batch normalization, and ReLU activation. Most blocks also have max-pooling to make the image smaller. The last layer predicts 15 values for each grid cell.

Table 3.1: YoloFastestV2 Model Layers

Layer	Type	Input Size	Output Size	Parameters
1	Conv + BN + ReLU + Pool	$1 \times 128 \times 128$	$16 \times 64 \times 64$	160
2	Conv + BN + ReLU + Pool	$16 \times 64 \times 64$	$32 \times 32 \times 32$	4,640
3	Conv + BN + ReLU + Pool	$32 \times 32 \times 32$	$64 \times 16 \times 16$	18,496
4	Conv + BN + ReLU + Pool	$64 \times 16 \times 16$	$128 \times 8 \times 8$	73,856
5	Conv + BN + ReLU	$128 \times 8 \times 8$	$128 \times 8 \times 8$	147,584
6	Conv (1×1)	$128 \times 8 \times 8$	$15 \times 8 \times 8$	1,935
			Total	**$\sim$246K**

The model definition in PyTorch shows how the backbone and predictor are structured:

Listing 3.2: Model Definition (model.py)

```
def conv_block(in_ch, out_ch, pool=True):
    layers = [
        nn.Conv2d(in_ch, out_ch, 3, 1, 1, bias=False),
        nn.BatchNorm2d(out_ch),
        nn.ReLU(inplace=True)
    ]
    if pool:
        layers.append(nn.MaxPool2d(2, 2))
    return nn.Sequential(*layers)

class YoloFastestV2(nn.Module):
    def __init__(self, num_classes=CATEGORIES):
        super(YoloFastestV2, self).__init__()
        self.backbone = nn.Sequential(
            conv_block(1, 16),
            conv_block(16, 32),
            conv_block(32, 64),
            conv_block(64, 128),
            conv_block(128, 128, pool=False)
        )
        self.predictor = nn.Conv2d(128, 5 + num_classes, 1)

    def forward(self, x):
```

```
        features = self.backbone(x)
        raw = self.predictor(features)
        activated = torch.sigmoid(raw)
        return activated.permute(0, 2, 3, 1)
```

3.2.3. Training

We trained the model using PyTorch with a custom loss function. The loss combines coordinate error, confidence error, and classification error. Coordinate errors are weighted higher ($\lambda_{coord} = 7.0$) while empty cells have lower weight ($\lambda_{noobj} = 0.5$).

Listing 3.3: Detection Loss Function (model.py)

```
class DetectionLoss(nn.Module):
    def __init__(self, coord_weight=7.0, empty_weight=0.5):
        super().__init__()
        self.mse_fn = nn.MSELoss(reduction="sum")
        self.w_coord = coord_weight
        self.w_empty = empty_weight

    def forward(self, pred, truth):
        has_obj = truth[..., 0] == 1
        no_obj = truth[..., 0] == 0

        coord_err = self.mse_fn(pred[..., 1:5][has_obj], truth[...,
            1:5][has_obj])

        conf_pos = self.mse_fn(pred[..., 0][has_obj], truth[..., 0][
            has_obj])
        conf_neg = self.mse_fn(pred[..., 0][no_obj], truth[..., 0][
            no_obj])

        cat_err = self.mse_fn(pred[..., 5:][has_obj], truth[..., 5:][
            has_obj])

        total = self.w_coord * coord_err + conf_pos + self.w_empty *
            conf_neg + cat_err
        return total / SAMPLES_PER_BATCH
```

Training was done for 50 epochs using Adam optimizer. Figure 3.2 shows the loss curve. The loss decreases but has some fluctuations, which is normal for detection tasks.

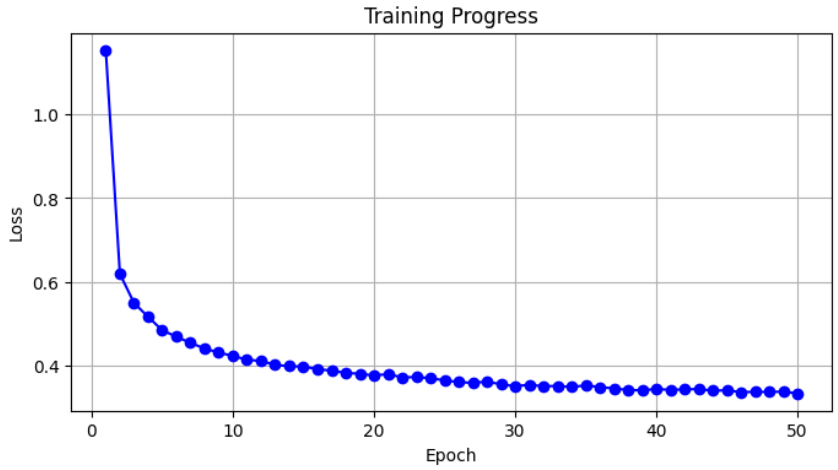


Figure 3.2: Training loss over 50 epochs.

3.2.4. Model Conversion

The original YoloFastestV2 uses **ONNX Runtime**. To run it on ESP32, we converted it through a multi-step pipeline shown in Figure 3.3.

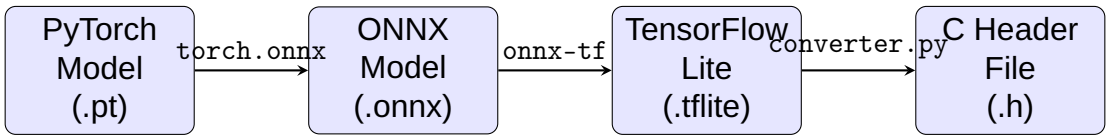


Figure 3.3: Model conversion pipeline from PyTorch to ESP32-compatible C header.

3.2.5. ESP32-CAM Deployment

The inference engine on the ESP32-CAM uses the **tflm_esp32** library, a port of TensorFlow Lite Micro for ESP32. A 2MB tensor arena is allocated in PSRAM, and the required operators are registered manually.

Listing 3.4: TFLite Initialization (ESP32_YOLO_Server.ino)

```
#define ARENA_SIZE (1024 * 1024 * 2)
const tflite::Model *model = nullptr;
tflite::MicroInterpreter *interpreter = nullptr;
tflite::MicroMutableOpResolver<10> *resolver = nullptr;
uint8_t *tensor_arena = nullptr;
TfLiteTensor *input = nullptr;
TfLiteTensor *output = nullptr;

void initTFLite() {
  if (!psramInit()) {
    Serial.println("PSRAM Init Failed!");
    return;
  }
  Serial.printf("PSRAM OK. Free: %d bytes\n", ESP.getFreePsram());

```

```
  tensor_arena = (uint8_t *)ps_malloc(ARENA_SIZE);
  if (!tensor_arena) {
    Serial.println("Arena Allocation Failed!");
    return;
  }

  model = tflite::GetModel(model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    Serial.println("Model Version Mismatch!");
    return;
  }

  resolver = new tflite::MicroMutableOpResolver<10>();
  resolver->AddConv2D();
  resolver->AddMaxPool2D();
  resolver->AddRelu();
  resolver->AddReshape();
  resolver->AddTranspose();
  resolver->AddAdd();
  resolver->AddLogistic();
  resolver->AddMean();
  resolver->AddQuantize();
  resolver->AddDequantize();

  interpreter = new tflite::MicroInterpreter(model, *resolver,
      tensor_arena,
                                             ARENA_SIZE, nullptr,
                                                nullptr);

  if (interpreter->AllocateTensors() != kTfLiteOk) {
    Serial.println("AllocateTensors Failed!");
    return;
  }

  input = interpreter->input(0);
  output = interpreter->output(0);

  Serial.println("TFLite Ready.");
  Serial.printf("Input: %dx%dx%d\n", input->dims->data[1], input->dims->
      data[2],
                input->dims->data[3]);
}
```

The inference function uses auto-contrast normalization to prepare the input, then parses the grid output to find detections above the confidence threshold:

Listing 3.5: Inference and Detection Parsing (ESP32_YOLO_Server.ino)

```
void runInference() {
  if (!hasImage || !captured_gray) {
    Serial.println("No image to process!");
    return;
  }

  detection_count = 0;

  // Auto-Contrast Normalization
  float min_val = 255.0;
  float max_val = 0.0;
  for (int i = 0; i < IMG_W * IMG_H; i++) {
    float val = captured_gray[i];
    if (val < min_val) min_val = val;
    if (val > max_val) max_val = val;
```

```
  }
  float range = max_val - min_val;
  if (range < 1.0) range = 1.0;

  Serial.printf("Input Stats - Min: %.0f, Max: %.0f\n", min_val, max_val
      );

  // Normalize and Fill Input Tensor
  for (int i = 0; i < IMG_W * IMG_H; i++) {
    float norm = (captured_gray[i] - min_val) / range;
    input->data.f[i] = norm;
  }

  // Run Inference
  uint32_t t = millis();
  if (interpreter->Invoke() != kTfLiteOk) {
    Serial.println("Invoke Failed!");
    return;
  }
  Serial.printf("Inference Time: %d ms\n", millis() - t);

  // Parse Detections
  for (int i = 0; i < GRID; i++) {
    for (int j = 0; j < GRID; j++) {
      int idx = (i * GRID + j) * TOTAL_PER_CELL;
      float conf = output->data.f[idx + 0];

      if (conf > CONF_THRESHOLD) {
        float max_score = 0;
        int cls = -1;
        for (int c = 0; c < CLASS_COUNT; c++) {
          float score = output->data.f[idx + 5 + c];
          if (score > max_score) {
            max_score = score;
            cls = c;
          }
        }

        float cx = (j + output->data.f[idx + 1]) / GRID;
        float cy = (i + output->data.f[idx + 2]) / GRID;
        float w = output->data.f[idx + 3];
        float h = output->data.f[idx + 4];

        if (detection_count < 20) {
          detections[detection_count].cls = cls;
          detections[detection_count].conf = conf;
          detections[detection_count].cx = cx;
          detections[detection_count].cy = cy;
          detections[detection_count].w = w;
          detections[detection_count].h = h;
          detection_count++;
        }
      }
    }
  }
  Serial.printf(">> Total detections: %d\n", detection_count);
}
```

3.3. RESULTS

3.3.1. Web Interface

Figure 3.4 shows the web interface. Users can take a photo, turn the flash on or off, and run detection with one click.

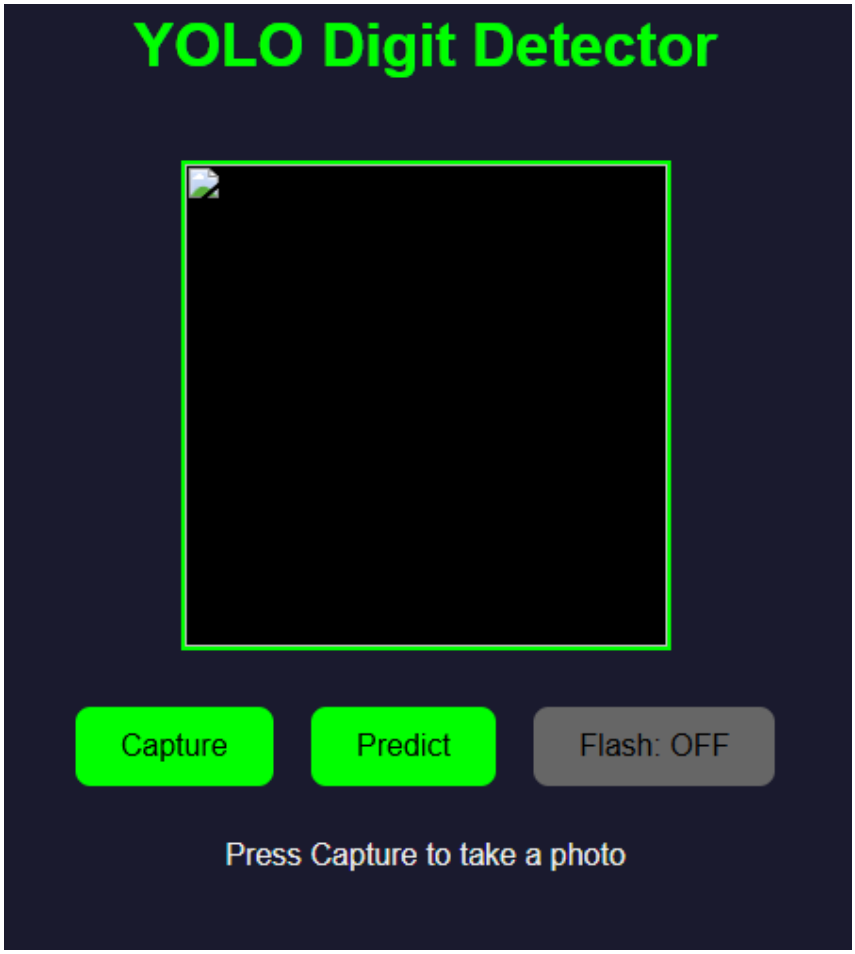
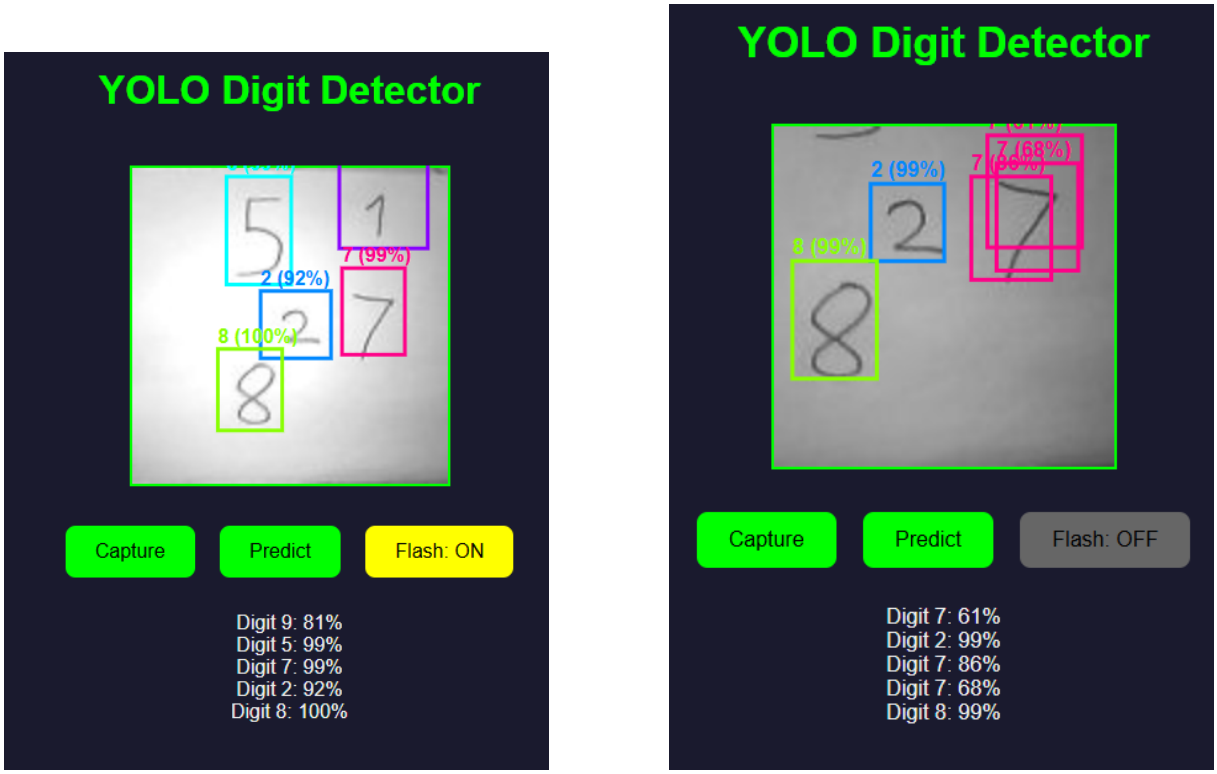


Figure 3.4: ESP32-CAM YOLO Web Interface.

3.3.2. Detection Examples

Figure 3.5 shows the system finding handwritten digits. Colored boxes show where each digit is, with the class and confidence score.



(a) Example 1 **(b)** Example 2

Figure 3.5: Digit detection results on ESP32-CAM.

3.3.3. Serial Output

Figure 3.6 shows the serial monitor during inference. It prints the inference time and details for each detection.

```
[1] digit=7 score=0.61 box=(81,4,116,46)
[2] digit=2 score=0.99 box=(37,22,65,51)
[3] digit=7 score=0.86 box=(75,19,105,58)
[4] digit=7 score=0.68 box=(84,14,115,55)
[5] digit=8 score=0.99 box=(7,51,39,95)
>> Total detections: 5
```

Figure 3.6: Serial output showing detection results.

3.3.4. Performance

- **Inference Time**: About **15 seconds** per image. This is slow because ESP32 has no hardware acceleration.

- **Model Size**: About 1MB, which fits in PSRAM.
- **Accuracy**: Figure 3.7 shows the confusion matrix. Most digits are classified correctly.

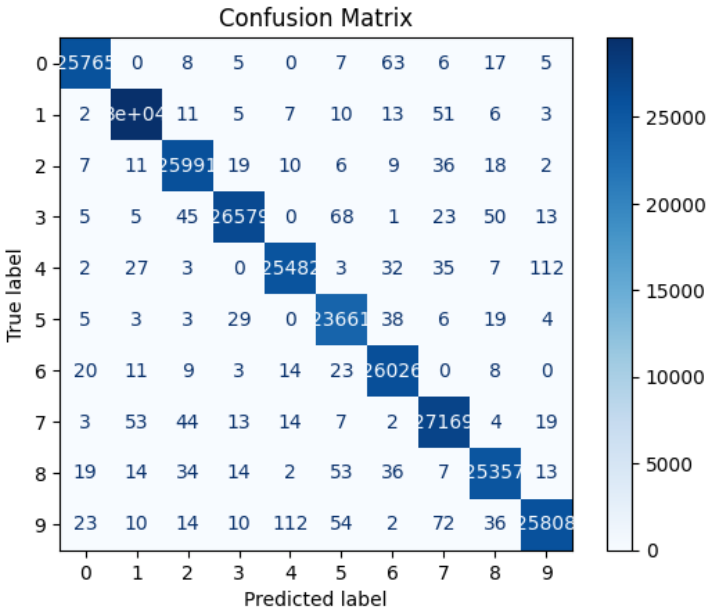


Figure 3.7: Confusion matrix for digit classification.

3.3.5. Limitations

The system works but has some problems:

- **Duplicate Detections**: Sometimes the same digit is detected twice by nearby cells.
- **Wrong Classifications**: Due to the training loss fluctuations, 1-2 digits per image may be wrong.
- **Slow Speed**: 15 seconds is too slow for real-time use. Future work could try model compression.

Despite these issues, the project shows that we can run a customized YOLO model on ESP32-CAM for digit detection.

4. QUESTION 3: IMAGE RESIZING (UPSAMPLING & DOWNSAMPLING)

4.1. THEORY

4.1.1. Sampling and Discrete Signal Processing

In Digital Signal Processing (DSP), images are 2D discrete signals obtained by sampling a continuous scene at regular intervals. The **Sampling Theorem** (Nyquist-Shannon) says that to perfectly reconstruct a signal, the sampling rate must be at least two times of maximum frequency present in the signal ($f_s \geq 2f_{max}$).

4.1.2. Upsampling (Interpolation)

Upsampling is the process of increasing the sampling rate of a signal. In DSP theory, this is typically achieved in two steps:

1. **Expander (Zero-Stuffing)**: Inserting $L - 1$ zeros between original samples.
2. **Interpolation Filter**: Applied to the expanded signal to replace zeros with estimated values.

In this project, we utilize **Nearest Neighbor Interpolation**, which effectively corresponds to a "box" or "sample-and-hold" filter. Instead of inserting zeros, we replicate the original pixel value L times. While computationally extremely efficient ($O(1)$), it introduces high-frequency artifacts visible as blockiness as illustrated in Figure 4.1.

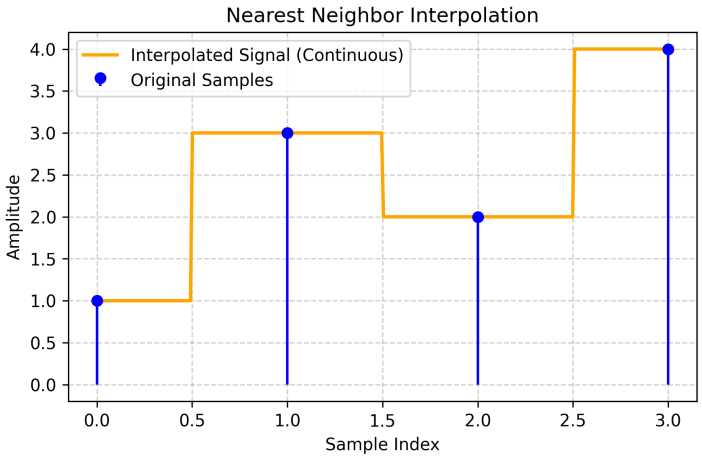


Figure 4.1: Conceptual visualization of Nearest Neighbor Interpolation.

4.1.3. Downsampling

Downsampling decreases the sampling rate (minification) by a factor M. The standard DSP process involves:

1. **Anti-Aliasing Filter**: Low-pass filtering the signal to remove frequencies above the new Nyquist limit ($f_s/2M$).
2. **Decimator**: Discarding $M - 1$ samples for every retained sample.

Our implementation performs direct decimation (skipping pixels). Since we bypass the low-pass filtering stage to save processing power on the ESP32, the resulting images are susceptible to **aliasing**, where high-frequency patterns (like textures) appear as lower-frequency noise.

4.1.4. Rational Scaling

To support non-integer scaling factors (e.g., $1.5\times$), we combine these operations to perform **Rational Rate Conversion** by a factor $S = L/M$:

$$\text{Signal} \xrightarrow{\uparrow L} \text{Intermediate} \xrightarrow{\downarrow M} \text{Output}$$

4.2. IMPLEMENTATION

The implementation on the ESP32-CAM handles memory dynamically using PSRAM to support the expanded intermediate buffers.

4.2.1. Upsampling Function

The `upsampleImage` function implements the geometric expansion. It allocates a large buffer in PSRAM and maps the destination pixels back to the source using integer division (y/L), effectively replicating rows and columns.

Listing 4.1: Upsampling Implementation (ESP32)

```
// Function to UPSAMPLE by integer factor L
bool upsampleImage(uint8_t *src, int srcW, int srcH, int L, uint8_t **
    outBuf,
                   int *outW, int *outH, size_t *outLen) {
  int newW = srcW * L;
  int newH = srcH * L;
  size_t newSize = newW * newH * 2; // RGB565 = 2 bytes/pixel

  // Safety Limit for Intermediate Buffer
  if (newSize > 4000000) return false;

  uint8_t *newBuf = (uint8_t *)heap_caps_malloc(newSize,
      MALLOC_CAP_SPIRAM);
  if (!newBuf) return false;

  for (int y = 0; y < newH; y++) {
    // Map to source Y (Repeat rows)
    int srcY = y / L;
    int srcRowOffset = srcY * srcW;
    int dstRowOffset = y * newW;

    for (int x = 0; x < newW; x++) {
      // Map to source X (Repeat cols)
      int srcX = x / L;
      int srcIndex = (srcRowOffset + srcX) * 2;
      int dstIndex = (dstRowOffset + x) * 2;

      newBuf[dstIndex] = src[srcIndex];
      newBuf[dstIndex + 1] = src[srcIndex + 1];
    }
  }
  *outBuf = newBuf;
  *outW = newW;
  *outH = newH;
  return true;
}
```

4.2.2. Downsampling Function

The `downsampleImage` function performs decimation by skipping pixels ($y * M$), selecting every M-th pixel from the source.

Listing 4.2: Exact Downsampling Implementation (ESP32)

```
// Function to DOWNSAMPLE by integer factor M
bool downsampleImage(uint8_t *src, int srcW, int srcH, int M, uint8_t **
    outBuf,
                     int *outW, int *outH, size_t *outLen) {
  int newW = srcW / M;
```

```
  int newH = srcH / M;
  size_t newSize = newW * newH * 2;

  uint8_t *newBuf = (uint8_t *)heap_caps_malloc(newSize,
      MALLOC_CAP_SPIRAM);
  if (!newBuf) return false;

  for (int y = 0; y < newH; y++) {
    // Map to source Y (Skip rows)
    int srcY = y * M;
    int srcRowOffset = srcY * srcW;
    int dstRowOffset = y * newW;

    for (int x = 0; x < newW; x++) {
      // Map to source X (Skip cols)
      int srcX = x * M;
      int srcIndex = (srcRowOffset + srcX) * 2;
      int dstIndex = (dstRowOffset + x) * 2;

      newBuf[dstIndex] = src[srcIndex];
      newBuf[dstIndex + 1] = src[srcIndex + 1];
    }
  }
  *outBuf = newBuf;
  *outW = newW;
  *outH = newH;
  return true;
}
```

4.3. RESULTS

The system was tested with various rational scaling factors. Figure 4.2 shows the web interface controls where L and M can be dynamically adjusted.

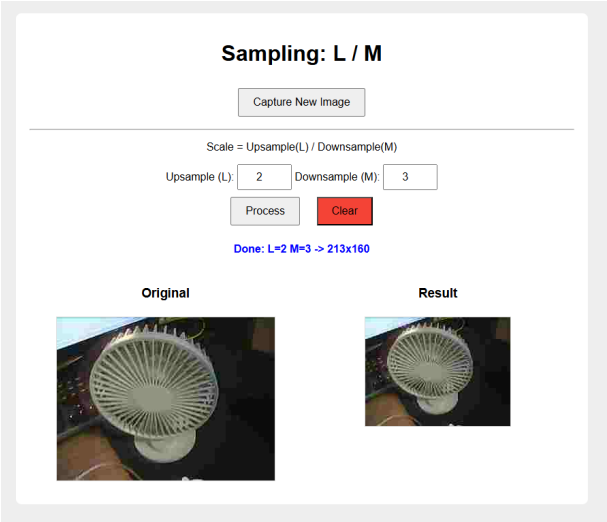


Figure 4.2: ESP32-CAM Web Interface for Image Resizing.

Figure 4.3 demonstrates the results. The original image (Figure 4.3a) was resized using four different L/M combinations.

- **0.5x** ($L = 1, M = 2$): Resolution is halved.
- **2.0x** ($L = 2, M = 1$): Resolution is doubled, showing pixelation blocks.
- **0.66x** ($L = 2, M = 3$): Non-integer reduction.
- **1.5x** ($L = 3, M = 2$): Non-integer magnification.

(a) Original Input Image

(b) 0.5x ($L = 1, M = 2$)

(c) 2.0x ($L = 2, M = 1$)

(d) 0.66x ($L = 2, M = 3$)

(e) 1.5x ($L = 3, M = 2$)

Figure 4.3: Rational resizing results. Non-integer scaling is achieved by combining upsampling and downsampling operations.

5. QUESTION 4: MULTI-MODEL DIGIT RECOGNITION & FUSION

5.1. THEORY

This question requires implementing handwritten digit recognition using multiple CNN architectures on the ESP32-CAM. We trained three models from the suplementary codes from the book [1]: **ResNet20v1**, **SqueezeNetV1.1**, and **MobileNetV1**. Each model was quantized to INT8 and deployed on the ESP32-Cam. The final prediction can be obtained from any single model or by **merging** the probability outputs of all three.

5.2. IMPLEMENTATION

5.2.1. Model Architectures

Table 5.1 summarizes the three architectures. All models use 64×64 grayscale input and output 10 class probabilities.

Table 5.1: Model Comparison

Model	Params	Inference Time	Key Features
MobileNetV1 (α=0.25)	$\sim$220K	250 ms	Depthwise separable convolutions
SqueezeNetV1.1	$\sim$730K	1,200 ms	Fire modules (squeeze + expand)
ResNet20v1	$\sim$270K	12,700 ms	Residual skip connections

5.2.2. Training

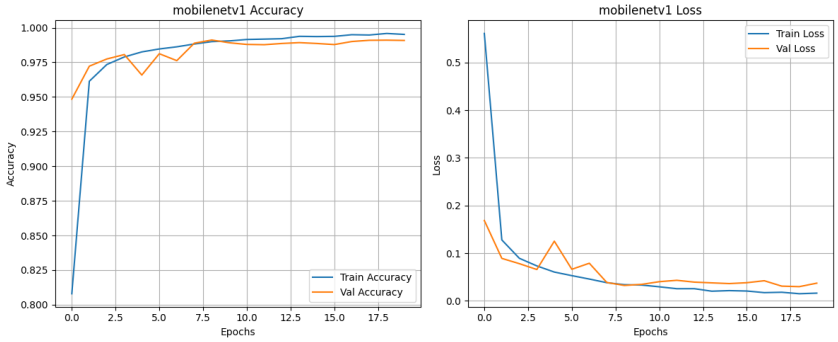
All models were trained on the MNIST dataset resized to 64×64 using `train_models.py`. Training used the Adam optimizer for 20 epochs with sparse categorical cross-entropy loss.

Listing 5.1: Training Configuration (train_models.py)

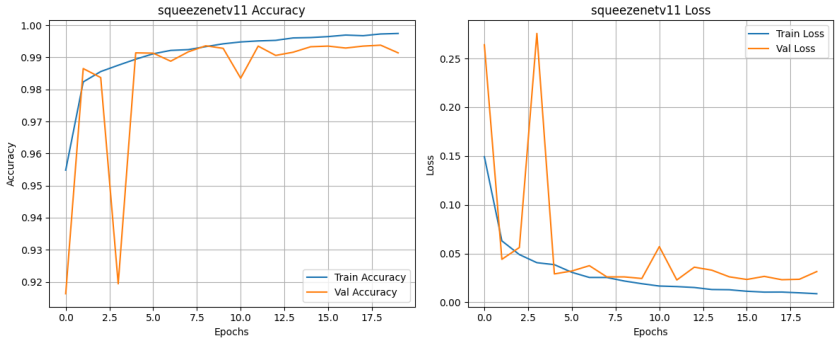
```
IMG_SIZE = 64
BATCH_SIZE = 32
EPOCHS = 20
NUM_CLASSES = 10

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
history = model.fit(train_ds, epochs=EPOCHS, validation_data=test_ds)
model.save(os.path.join(MODELS_DIR, f"{model_name}.keras"))
```

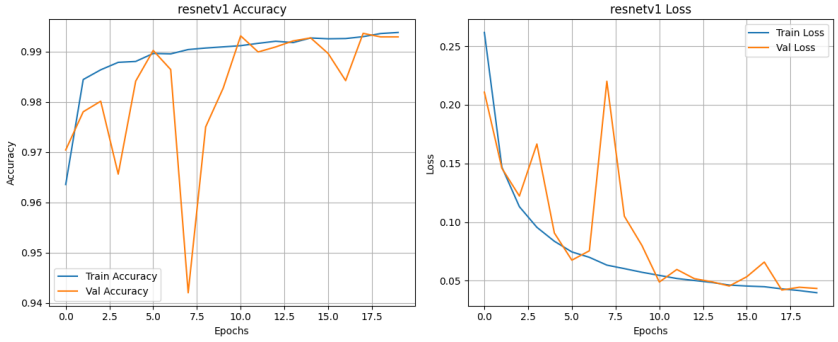
Figure 5.1 shows the training metrics for each model.



(a) MobileNet



(b) SqueezeNet



(c) ResNet

Figure 5.1: Training accuracy and loss curves for each model.

5.2.3. INT8 Quantization

Models were quantized to INT8 using TensorFlow Lite with a calibration dataset from MNIST. The `convert_models.py` script handles the full pipeline:

Listing 5.2: INT8 Quantization (convert_models.py)

```
def convert_to_tflite_int8(model_path, calibration_data):
    model = tf.keras.models.load_model(model_path)
    converter = tf.lite.TFLiteConverter.from_keras_model(model)

    # INT8 Quantization settings
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    converter.representative_dataset = rep_dataset
    converter.target_spec.supported_ops = [tf.lite.OpsSet.
       TFLITE_BUILTINS_INT8]
    converter.inference_input_type = tf.int8
    converter.inference_output_type = tf.int8

    return converter.convert()
```

5.2.4. ESP32 Deployment

We use the **ArduTFLite** library for model inferencing. Models can be used seperately and the web interface provides buttons for each model plus a merged prediction option.

Listing 5.3: Model Loading and Prediction (ESP32_Inference.ino)

```
bool loadModel(const unsigned char *model_data) {
  if (interpreter != nullptr) {
    delete interpreter;
    interpreter = nullptr;
  }

  model = tflite::GetModel(model_data);
  interpreter = new tflite::MicroInterpreter(model, resolver,
     tensor_arena,
                                             kArenaSize, nullptr);
  interpreter->AllocateTensors();
  input = interpreter->input(0);
  output = interpreter->output(0);
  return true;
}
```

For the merged prediction, all three models are run sequentially and their output probabilities are averaged:

Listing 5.4: Merged Prediction Logic

```
Prediction p1 = predict(mobilenet_model_model, "MobileNet");
Prediction p2 = predict(resnet_model_model, "ResNet");
Prediction p3 = predict(squeezenet1_model_model, "SqueezeNet");

for (int i = 0; i < 10; i++) {
  final_probs[i] = (p1.all_probs[i] + p2.all_probs[i] + p3.all_probs[i])
      / 3.0;}
```

5.3. RESULTS

5.3.1. Web Interface

Figure 5.2 shows the web interface with buttons for each model and the merged prediction option.

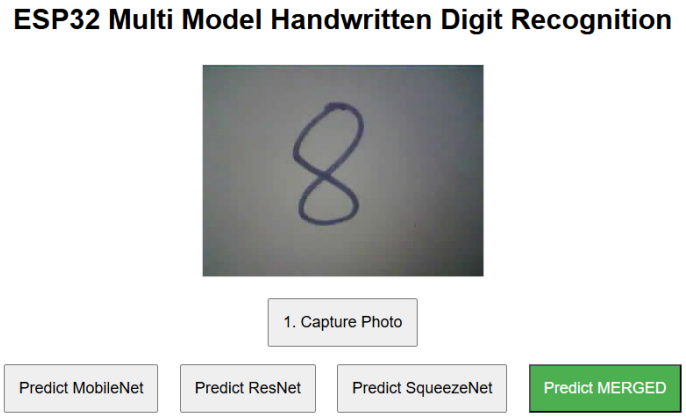
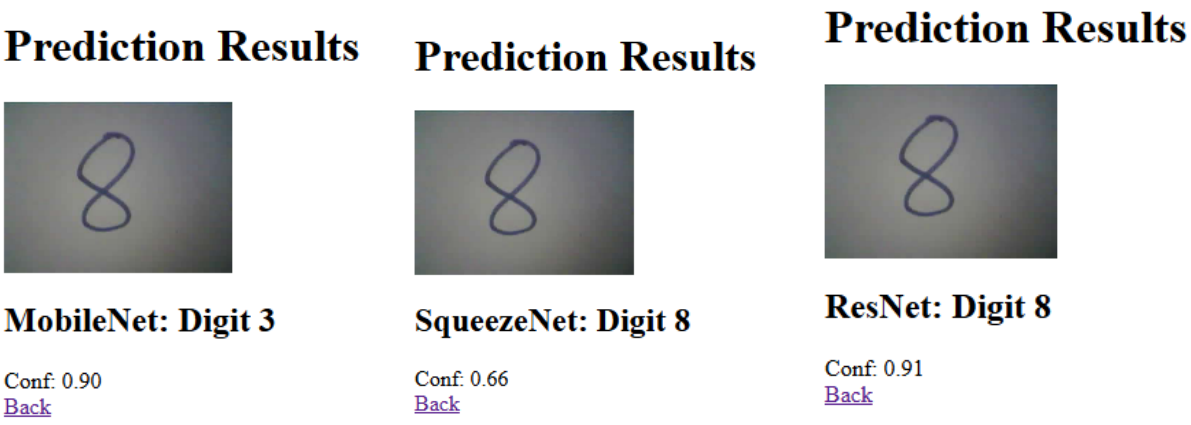


Figure 5.2: ESP32 Web Interface for multi-model digit recognition.

5.3.2. Individual Model Results

Figure 5.3 shows the prediction results from each model.



(a) MobileNet (250ms) **(b)** SqueezeNet (1.2s) **(c)** ResNet (12.7s)

Figure 5.3: Individual model prediction results.

5.3.3. Merged Prediction

Figure 5.4 shows the merged result, which combines all three model outputs by averaging their probability vectors.

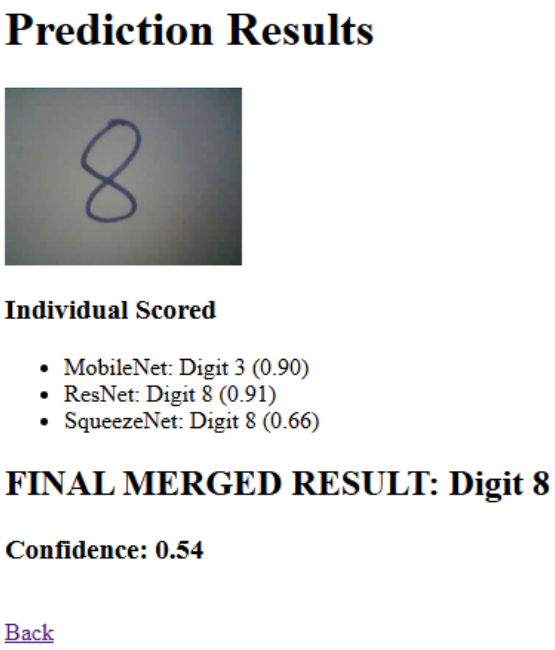
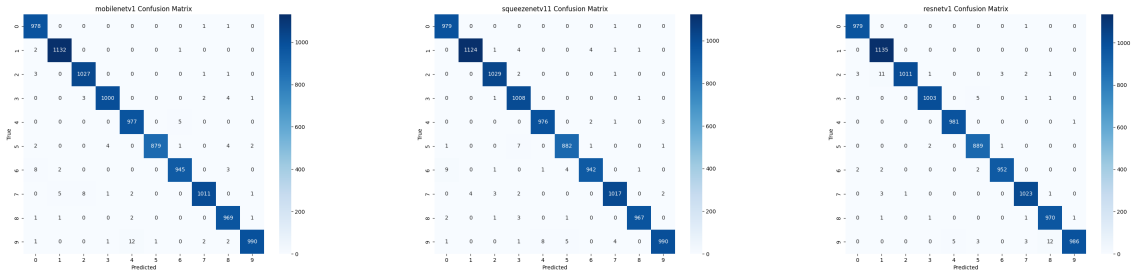


Figure 5.4: Merged prediction showing individual scores and final result.

5.3.4. Confusion Matrices

Figure 5.5 shows the confusion matrices from the training evaluation.



(a) MobileNet **(b)** SqueezeNet **(c)** ResNet

Figure 5.5: Confusion matrices for each model.

5.3.5. Serial Output

Figure 5.6 shows the serial monitor output during inference.

Figure 5.6: Serial output showing inference process and timing.

5.3.6. Performance Summary

Looking at the inference times, MobileNet is by far the fastest at around 250ms, which makes sense since it uses depthwise separable convolutions designed for mobile devices. SqueezeNet comes in at about 1.2 seconds, and ResNet is the slowest at 12.7 seconds due to its deeper architecture.

Interestingly, in our test with the digit 8, MobileNet actually gave a wrong prediction (it predicted 3), while both ResNet and SqueezeNet correctly identified it as 8. This shows that being fast does not always mean being accurate. The merged prediction ended up being correct because it averages the outputs from all three models, even when one model fails, the other two can outvote it.

If you need speed, MobileNet is the way to go. But if accuracy matters more, running all three models and merging their results (about 14 seconds total) gives you a more reliable answer.

6. CONCLUSION

This project was a great learning experience for working with embedded machine learning on the ESP32-CAM. We implemented four different applications, each with its own challenges and solutions.

For Question 1 Adaptive Thresholding, the binary search approach worked really well for finding the optimal threshold. The main challenge was dealing with camera noise which we solved by capturing images in JPEG format first and then decoding them.

In Question 2 YOLO Detection was probably the most challenging part. Training a custom YOLO model from scratch on a synthetic dataset took some trial and error. The 15-second inference time is quite slow but considering we are running a full object detection model on a microcontroller, it is still impressive that it works.

For Question 3 Image Resizing, we implemented scaling image by using integer upsampling and downsampling was a simple workaround for achieving non-integer scale factors without floating-point division in the main loop.

Finally for the Question 4 Multi-Model Classification showed an interesting trade-off between speed and accuracy. MobileNet was super fast at 250ms but made a wrong prediction in our test, while ResNet was slow at 12.7 seconds but got it right. The merged prediction approach turned out to be useful because even when one model fails, the others can correct it.

In summary we successfully implemented all four embedded image processing and machine learning applications on the ESP32-CAM. All implementations run entirely on the device without requiring any cloud connection showing that useful computer vision applications can work on cheap, low-power hardware.

All source codes for this project are available on GitHub: `https://github.com/mustrelax/EE4065`

BIBLIOGRAPHY

[1] C. Ünsalan, B. Höke, E. Atmaca, Embedded Machine Learning with Microcontrollers: Applications on STM32 Boards, Springer Nature, 2025.

[2] G. Jocher, J. Qiu, Ultralytics yolo11 (2024).
URL https://github.com/ultralytics/ultralytics

[3] dog qiuqiu, Yolo-fastestv2 (2021).
URL https://github.com/dog-qiuqiu/Yolo-FastestV2