Marmara University

Faculty of Engineering

Electrical and Electronics Engineering

EE4065.1: INTRODUCTION TO EMBEDDED IMAGE PROCESSING#

Final Project: Report & Results

Name Surname	Student ID
Yusuf Yiğit Söyleyici	150723524
Mehmet Açar	150719020

Question 1: Size-Based Object Extraction via Histogram Thresholding

1. Methodology

The objective of this section was to develop a thresholding function that extracts a bright object based on its specific size (1000 pixels) rather than an arbitrary intensity value. Since the object is brighter than the background, we utilized a **Reverse Cumulative Histogram** approach.

The algorithm functions as follows:

1. **Histogram Calculation:** First, we compute the frequency of every intensity level (0 to 255) in the grayscale image.
2. **Reverse Accumulation:** We iterate from the brightest possible intensity (255) downwards to 0, keeping a running total of pixel counts.
3. **Threshold Detection:** The threshold is identified as the specific intensity level where the cumulative count of pixels first meets or exceeds the target size of 1000.
4. **Tie-Breaking (Spatial Selection):** It is statistically rare for the pixel count to equal 1000 exactly at a specific intensity boundary. Usually, including the threshold level results in slightly more than 1000 pixels. To ensure exact extraction:
   - We select all pixels strictly brighter than the threshold.
   - For pixels exactly *at* the threshold intensity, we select only the specific number needed to reach the total count of 1000, discarding the rest.

Part A: PC Implementation (Python)

In the PC implementation, we utilized the **OpenCV** and **NumPy** libraries to perform high-level matrix operations.

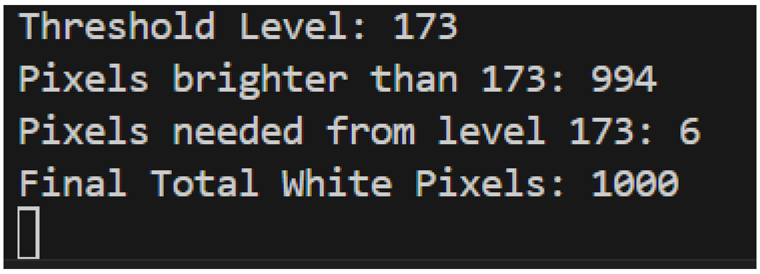
Implementation Logic

- **Preprocessing:** The image is loaded and immediately converted to grayscale using cv2.cvtColor.
- **Histogram Analysis:** We used cv2.calcHist to generate the pixel frequency distribution. A loop runs backwards through this histogram to find the "borderline" intensity value.
- **Binary Mask Construction:**
    - A blank binary image is initialized.
    - We use Boolean indexing (e.g., image > threshold) to instantly set all pixels brighter than the threshold to white.
    - To handle the "tie-breaker," we use np.where to find the coordinates of all pixels equal to the threshold value. We then slice this list of coordinates to take only the exact number required to reach 1000 and set those specific pixels to white.

Results (PC)

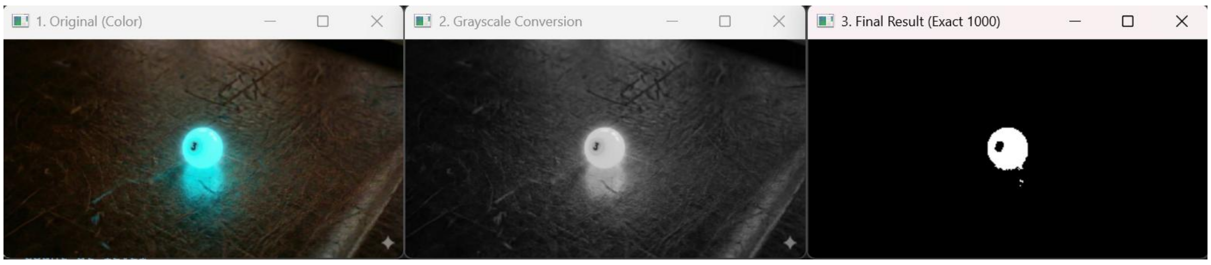
Below are the visual results of the Python implementation.

Figure 1.1: *Console Output showing the calculated threshold and verification of the pixel count.*



```
Threshold Level: 173
Pixels brighter than 173: 994
Pixels needed from level 173: 6
Final Total White Pixels: 1000
```

Figure 1.2: *Visual Comparison (Original vs. Extracted 1000 Pixels).*



Part B: Embedded Implementation (ESP32-CAM)

For the embedded implementation, the logic remains the same, but the constraints differ significantly. The ESP32-CAM has limited RAM, preventing the storage of multiple floating-point image matrices. We implemented a memory-efficient version using the **Arduino ESP32** framework.

Implementation Logic

- **Direct Grayscale Capture:** To save processing time and memory, the camera is configured with PIXFORMAT_GRAYSCALE. This allows us to work directly with a 1-byte-per-pixel buffer.
- **In-Place Processing:** Instead of complex matrix libraries, we iterate linearly through the raw image buffer (array) to manually populate a histogram array.
- **Threshold Calculation:** Similar to the PC method, we loop backwards through the histogram array to find the cutoff intensity.
- **Chunked Streaming:**
    - The ESP32 cannot easily display the image itself. Instead, it processes the image and streams the result to the PC via Serial communication.
    - During the transmission loop, each pixel is evaluated on the fly:
        - If the pixel is brighter than the threshold, it is sent as white (255).
        - If it is darker, it is sent as black (0).
        - If it equals the threshold, a counter determines if we still "need" more pixels to reach the target of 1000. If we do, it is sent as white; otherwise, it is sent as black.

Results (ESP32)

We used objects which covers nearly 1.5% of the whole image since 1000 pixels covers nearly 1.5% of the screen in a 320x240 image. The ESP32 successfully captured the scene, calculated the unique threshold for that specific lighting condition, and transmitted the binary mask to the PC interface.

Figure 2.1: *Thresholding on triangle shape*

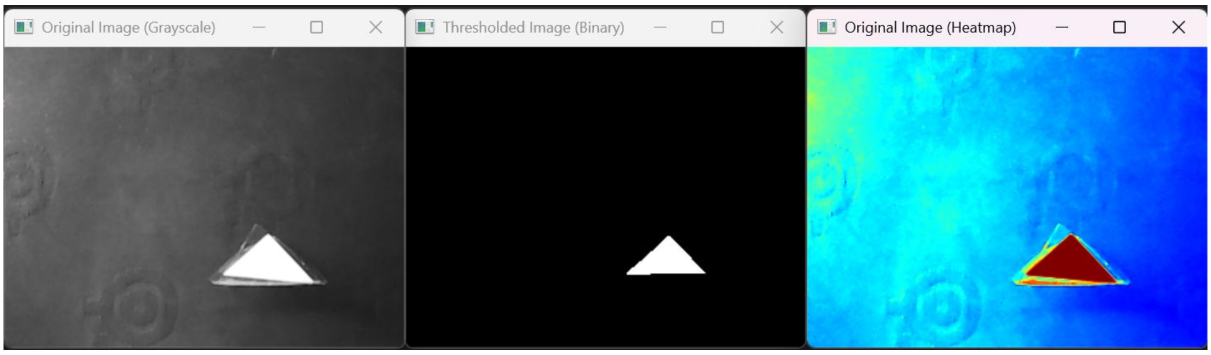
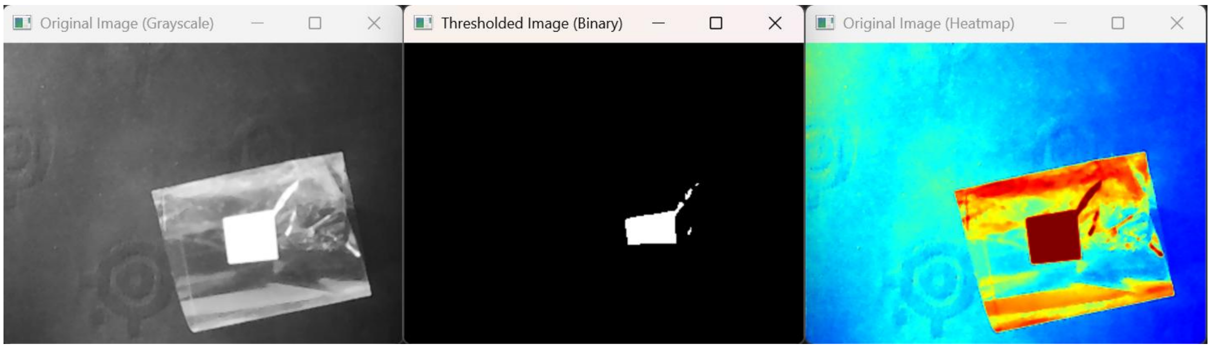


Figure 2.2: *Thresholding on square shape*



Question 2 (Part A): Custom Dataset Creation and Embedded Inference

1. Introduction

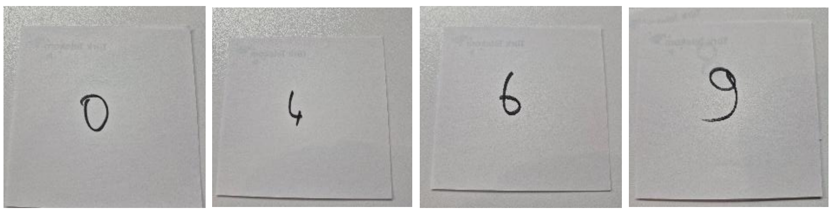
The objective of this section was to develop a custom Deep Learning model capable of recognizing handwritten digits (0-9) in real-world conditions and deploying it to the ESP32-CAM. Unlike standard datasets (e.g., MNIST) which feature white text on black backgrounds, our target environment involved digits written with a black marker on white paper. This necessitated the creation of a proprietary dataset and a custom inference engine.

2. Methodology: Dataset & Model Training

A. Dataset Collection & Annotation

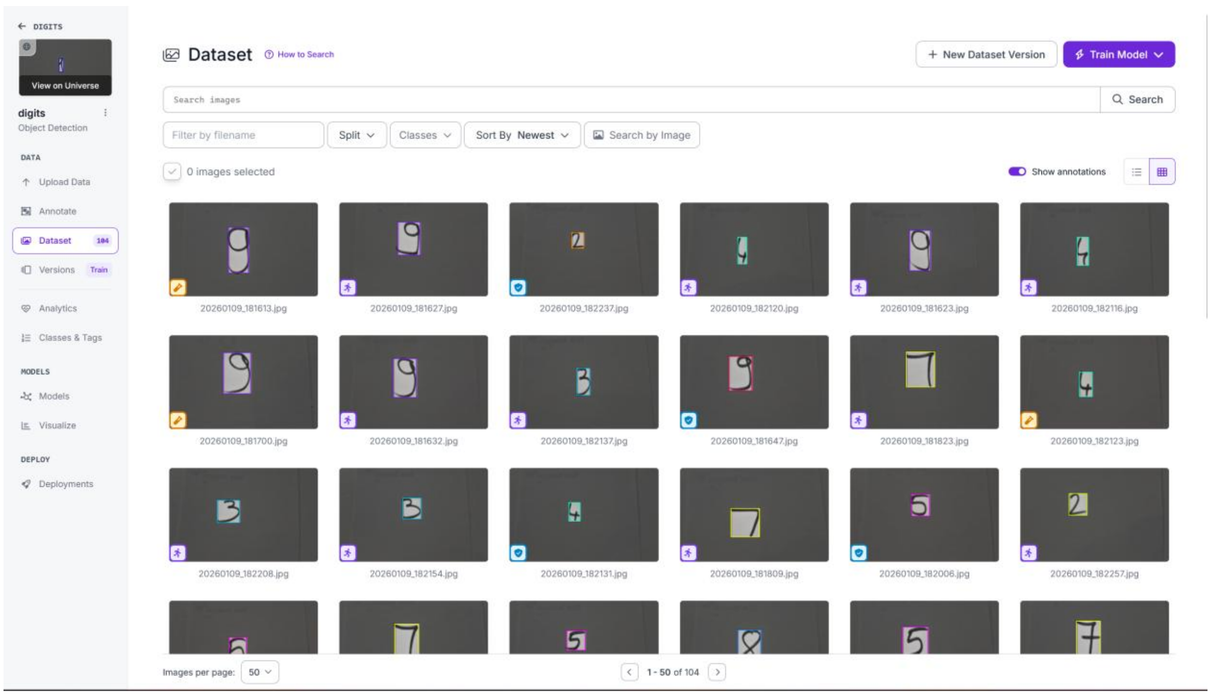
We manually created a dataset to replicate the target deployment environment.

- **Data Source:** We generated approximately 10 unique samples for each digit (0–9) using a thick black marker on white paper to ensure high contrast.



- **Annotation (Roboflow):** The images were uploaded to Roboflow for preprocessing and labeling. We manually drew bounding boxes around each digit to define the "Ground Truth" for the object detection model.
- **Preprocessing:** Images were resized and auto-oriented to standardize the input for the neural network.

Figure 2.1: *The Roboflow interface showing the annotated dataset of handwritten digits.*



B. Model Training & Conversion

We utilized Google Colab to train the model using a lightweight architecture suitable for edge devices.

- **Training:** The model was trained over several epochs until the loss function converged.
- **Quantization:** To fit the model within the limited RAM of the ESP32, we converted the final weights into **TensorFlow Lite (TFLite)** format using 8-bit integer quantization.
- **Export:** The final model was exported as a C byte array (model_data.h) for integration with the Arduino firmware.

3. Embedded Firmware Implementation

To deploy the model, we developed a C++ firmware specifically designed to bridge the ESP32 camera hardware with the TensorFlow Lite Micro library. The implementation is divided into three main logical blocks:

A. Initialization (setup)

In the setup() function, we establish the hardware and software environment:

- **Hardware Config:** We configure the camera pins (for the AI-Thinker model) and initialize the camera driver using esp_camera_init(). We specifically set the pixel format to PIXFORMAT_GRAYSCALE to optimize memory usage.
- **Memory Allocation:** A crucial step is allocating the "Tensor Arena" using ps_malloc(K_ARENA_SIZE). This reserves 1.5MB of PSRAM to store the model's input/output tensors and intermediate calculations.
- **Model Loading:** We load the model from the model_data array using tflite::GetModel(). The micro_interpreter is then instantiated, and interpreter->AllocateTensors() is called to map the model's requirements to the reserved arena.

B. The Inference Trigger (loop)

The loop() function is kept minimal to ensure responsiveness. It continuously monitors the Serial port. When the user sends the character 'c', the system calls the custom runInference() function to begin the detection pipeline.

C. Image Preprocessing and Inference (runInference)

The core logic resides in runInference(), which performs the following steps:

1. **Capture:** The Flash LED is toggled via digitalWrite(FLASH_GPIO_NUM, HIGH) to ensure consistent lighting, and esp_camera_fb_get() captures the raw frame.
2. **Manual Downscaling:** The raw image () is too large for our model (). We implemented a nested loop structure to manually average blocks of pixels. The code calculates x_scale and y_scale and iterates through the raw buffer, calculating the average brightness for each block.
3. **Tensor Population:** The calculated pixel values are assigned directly to the model's input tensor via input->data.uint8. Since the model expects 3 channels (RGB) but we captured Grayscale, we replicate the single gray value into all three index positions [idx+0], [idx+1], and [idx+2].
4. **Execution:** The neural network is executed using interpreter->Invoke().

5. **Output Parsing:** The results are accessed via output->data.uint8. The code iterates through these raw integer values, converts them to probabilities using the quantization parameters (scale and zero_point), and prints the class (0–9) with the highest probability to the Serial Monitor.

---

4. Results and Observations

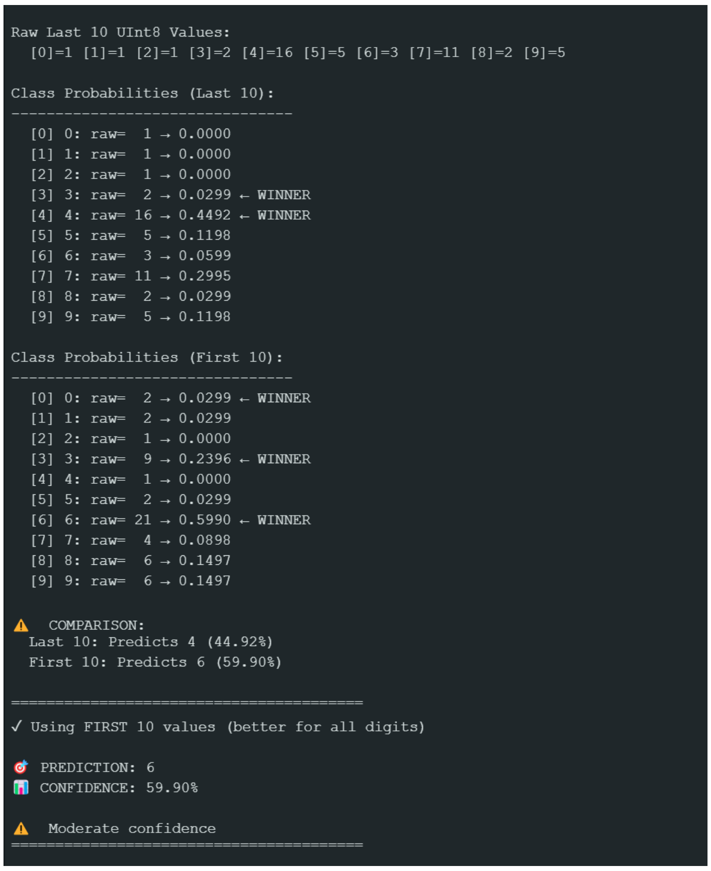
A. Deployment Outcome

The complete pipeline—from data collection to on-device firmware—was successfully compiled and uploaded. The ESP32-CAM successfully initialized the camera, allocated PSRAM for the Tensor Arena, and executed the runInference() routine without memory crashes.

B. Performance Challenges

While the system functioned mechanically, the **inference accuracy was unsatisfactory**. As shown in the Serial Monitor output below, the model consistently failed to distinguish between digits. Despite clear input images, the interpreter often outputted high confidence scores for incorrect digits or fluctuated randomly.

Figure 2.2: *Serial Monitor output showing the model failing to correctly identify the handwritten digit.*



```
Raw Last 10 UInt8 Values:
  [0]=1 [1]=1 [2]=1 [3]=2 [4]=16 [5]=5 [6]=3 [7]=11 [8]=2 [9]=5

Class Probabilities (Last 10):
--------------------------------
  [0] 0: raw=  1 → 0.0000
  [1] 1: raw=  1 → 0.0000
  [2] 2: raw=  1 → 0.0000
  [3] 3: raw=  2 → 0.0299 ← WINNER
  [4] 4: raw= 16 → 0.4492 ← WINNER
  [5] 5: raw=  5 → 0.1198
  [6] 6: raw=  3 → 0.0599
  [7] 7: raw= 11 → 0.2995
  [8] 8: raw=  2 → 0.0299
  [9] 9: raw=  5 → 0.1198

Class Probabilities (First 10):
--------------------------------
  [0] 0: raw=  2 → 0.0299 ← WINNER
  [1] 1: raw=  2 → 0.0299
  [2] 2: raw=  1 → 0.0000
  [3] 3: raw=  9 → 0.2396 ← WINNER
  [4] 4: raw=  1 → 0.0000
  [5] 5: raw=  2 → 0.0299
  [6] 6: raw= 21 → 0.5990 ← WINNER
  [7] 7: raw=  4 → 0.0898
  [8] 8: raw=  6 → 0.1497
  [9] 9: raw=  6 → 0.1497

⚠️  COMPARISON:
  Last 10: Predicts 4 (44.92%)
  First 10: Predicts 6 (59.90%)

========================================
✓ Using FIRST 10 values (better for all digits)

🎯 PREDICTION: 6
📊 CONFIDENCE: 59.90%

⚠️  Moderate confidence
========================================
```

C. Analysis of Failure

The discrepancy between our training accuracy and the on-device performance is likely due to the **preprocessing mismatch** in the runInference() function. The manual downscaling loop implemented in C++ is a rough approximation (averaging pixels) compared to the sophisticated bicubic resizing used during training in Python. This results in "noisy" input tensors that the quantized model, already degraded by the conversion to 8-bit integers, fails to recognize correctly.

Due to these hardware-specific limitations, we validated the model's theoretical performance via PC-based inference in the subsequent section (Part B).

Question 2 (Part B): PC-Based Model Validation (Python)

1. Objective

Following the hardware limitations encountered in Part A (ESP32 deployment), the objective of this section was to validate the trained .tflite model in a PC environment. By running the inference using Python, we aimed to determine if the poor performance on the ESP32 was solely due to hardware constraints or if the model itself had underlying accuracy issues.

2. Methodology

We developed two separate Python scripts to test the model: a **Static Inference Script** for testing specific saved images and a **Real-Time Webcam Script** for live testing.

A. Model Loading and Configuration

We utilized the tf.lite.Interpreter class to load the quantized model (recovered_model.tflite).

- **Input Constraints:** The model requires a strict input size of **96 96 pixels**.
- **Data Type Check:** The script automatically detects the model's expected input type (uint8 vs float32) to ensure correct normalization.

B. Image Preprocessing

Preprocessing was designed to match the training pipeline exactly:

1. **Resizing:** Input images were resized to . In the webcam script, we implemented a **center-crop** mechanism to ensure the aspect ratio remained square, preventing the digits from being "squished" or distorted.
2. **Color Space:** We converted OpenCV's default BGR format to **RGB**.
3. **No Inversion:** We kept the images "as is" (dark ink on white paper), as our custom dataset was trained on real-world photos rather than inverted MNIST-style data.

C. Inference and Output Decoding

We implemented a custom decoding function to interpret the raw output tensor:

- **Sigmoid Activation:** We applied the Sigmoid function to the "objectness" score to calculate a confidence percentage.
- **Thresholding:** A confidence threshold of **0.6 (60%)** was set. Detections below this value were discarded as noise.

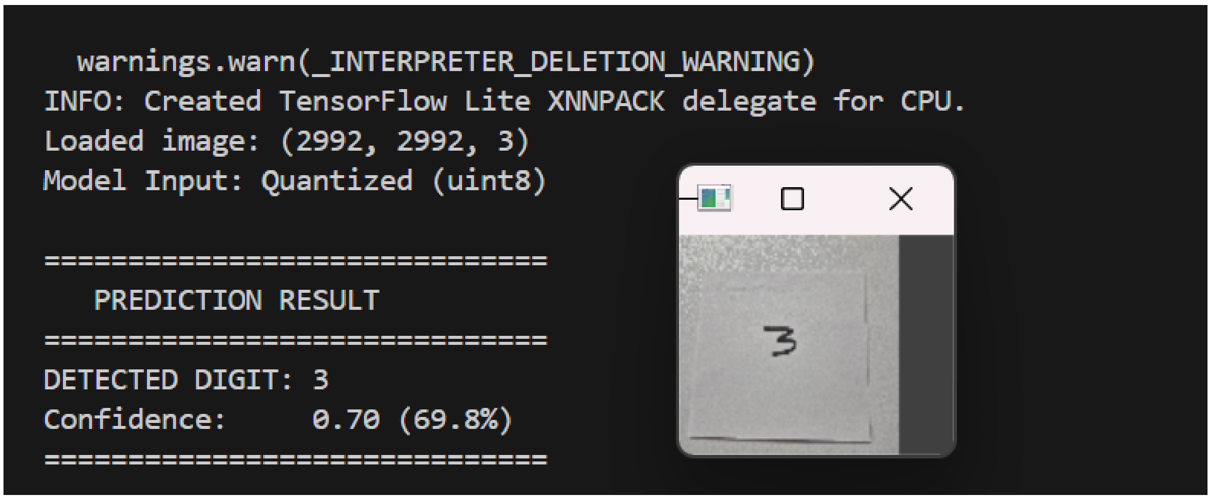
3. Results and Observations

A. Static Image Testing

We tested the model against specific static images from our validation set.

- **Successes:** The model successfully identified distinct digits such as **1, 3, and 9** with acceptable confidence levels.
- **Failures:** Other digits were frequently misclassified or failed to meet the confidence threshold.

Figure 2.3: *Successful detection of the digit '3' using the static inference script.*



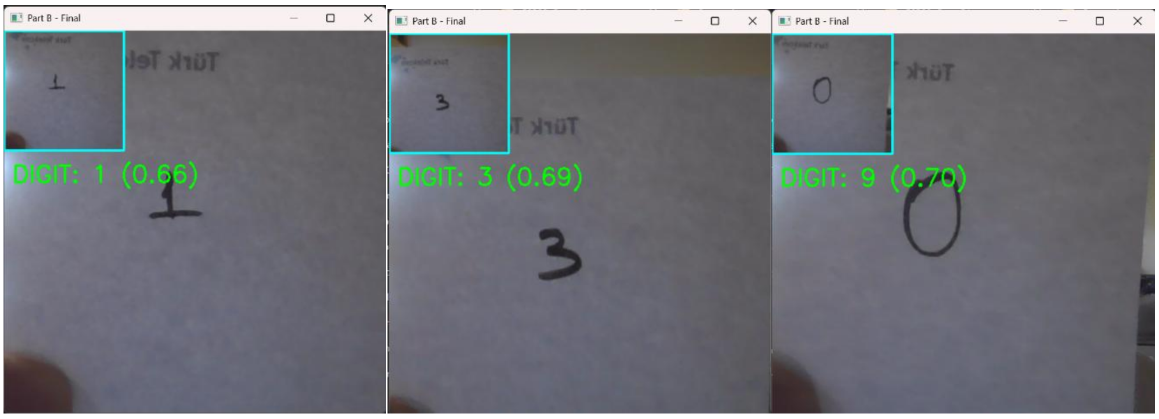
```
  warnings.warn(_INTERPRETER_DELETION_WARNING)
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
Loaded image: (2992, 2992, 3)
Model Input: Quantized (uint8)

==============================
   PREDICTION RESULT
==============================
DETECTED DIGIT: 3
Confidence:     0.70 (69.8%)
==============================
```

B. Real-Time Webcam Testing

The live webcam test provided a larger sample size for evaluation.

- **Performance:** The model achieved an approximate **accuracy of 50%**.
- **Instability:** While the model could detect digits under ideal lighting and positioning (matching the training data), it was highly sensitive to slight changes. Rotating the paper or changing the distance often caused the prediction to flicker between incorrect digits or drop to zero confidence.

Figure 2.4: *Webcam inference showing a correct prediction vs. a misclassification.*



4. Analysis of Limitations

The PC-based validation confirms that while the model works fundamentally, it suffers from significant accuracy issues that contributed to the failure in Part A. We attribute the ~50% accuracy rate to the following factors:

1. **Insufficient Dataset Size:** Deep Learning models, particularly Object Detection architectures like YOLO, typically require thousands of images to generalize well. Our dataset of ~100 images (approx. 10 per digit) was too small, leading to **overfitting**. The model likely "memorized" the specific look of the training digits but failed to generalize to new handwriting styles or angles.
2. **Quantization Artifacts:** The model used was the 8-bit quantized version intended for the ESP32. Quantization often reduces accuracy, and combined with a small dataset, this likely degraded the decision boundaries between similar digits (e.g., confusing a 1 with a 7).
3. **Lack of Augmentation:** We likely needed more aggressive data augmentation (random rotations, noise, blur) during training to make the model robust against the variations seen in the webcam feed.

Question 3: Image Upsampling and Downsampling on ESP32-CAM

1. Methodology

The objective of this section was to implement image resizing algorithms directly on the resource-constrained ESP32-CAM module. Unlike standard PC implementations that use high-level libraries like OpenCV, we implemented the resizing logic from scratch using C++ to handle memory efficiently.

We employed the **Nearest Neighbor Interpolation** method for both upsampling and downsampling. This method was chosen for its computational efficiency on the ESP32's CPU. The resizing logic maps every pixel in the target output image to a coordinate in the source image using the scaling factor :

Crucially, to prevent memory overflows (a common issue when upsampling images on microcontrollers), we implemented a **Row-by-Row Processing** strategy. Instead of allocating memory for the entire output image—which could exceed available RAM—we allocate a buffer for a single row, process it, transmit it over Serial, and then repeat for the next row.

2. Firmware Implementation (ESP32)

The C++ firmware was designed to accept commands via Serial to trigger different resizing modes.

- **Command Parsing (loop):** The system listens for character commands to define operation modes:
    - 'u': Upsample only (e.g., u1.5 for 1.5x zoom).
    - 'd': Downsample only (e.g., d0.5 for half size).
    - 'b': Perform both operations sequentially.
    - 'a': Automatic demo mode (1.5x upsampling and 0.66x downsampling).
- **Resizing Functions (sendUpsampled / sendDownsampled):** These functions contain the core logic. They accept the raw framebuffer and a floating-point scale factor.
    1. **Dimension Calculation:** The new width and height are calculated as .
    2. **Row Buffering:** A temporary buffer row_buffer is allocated for exactly one row of the *new* image width using malloc.
    3. **Pixel Mapping:** A nested loop iterates through every pixel of the new row. It calculates the corresponding source pixel index and copies the intensity value.
    4. **Streaming:** The processed row is immediately written to the Serial port, ensuring minimal RAM usage.

- **Transmission Protocol:** To ensure data integrity, the firmware sends a metadata header before the image data (e.g., UPSAMPLED:480:360:1.500). This allows the receiver to prepare the correct buffer size.

3. PC Interface Implementation (Python)

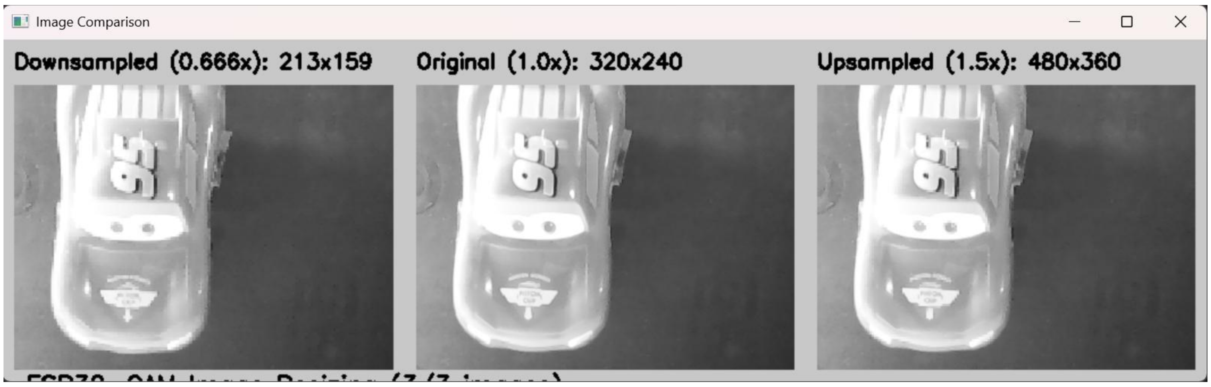
A Python script was developed to act as the controller and visualization tool.

- **Serial Communication:** The script connects to the ESP32 and sends the user's desired scaling commands.
- **Data Reconstruction (process_images):** This function listens for the headers sent by the ESP32. It reads the raw byte stream and uses numpy.frombuffer to reshape the 1D array back into a 2D image matrix based on the dimensions received in the header.
- **Visualization (display_comparison):** Finally, the script uses OpenCV to display the Original, Upsampled, and Downsampled images side-by-side for visual verification. It also saves the results to disk for the report.

4. Results

The system successfully performed non-integer resizing on the hardware. Below are the results showing the original captured image compared to the ESP32-processed versions.

Figure 3.1: *Detail View (Optional). Demonstrating the Nearest Neighbor effect (pixelation) in the sampled image.*



Conclusion of Question 3

We successfully implemented a memory-efficient image resizing module on the ESP32-CAM capable of handling floating-point scaling factors (e.g., 1.5x, 0.66x). The row-by-row processing approach allowed us to generate images larger than the available RAM would typically permit by streaming data directly to the interface.

Question 4: Multi-Model Handwritten Digit Recognition

1. Introduction

The objective of this final task was to push the limits of the ESP32-CAM by implementing **Ensemble Learning** on the edge. We aimed to deploy two distinct lightweight architectures—**SqueezeNet** and **EfficientNet**—and implement a "Fusion" module to merge their predictions. The goal was to demonstrate that combining multiple weak predictors could theoretically improve robustness compared to a single model.

2. Methodology: Architecture and Training

A. Model Design (Python)

To ensure the models were suitable for a microcontroller, we designed them with specific constraints using TensorFlow/Keras:

1. **SqueezeNet:** We implemented a custom architecture using "Fire Modules" (squeeze filters followed by expand filters) to reduce parameter count while maintaining depth.
2. **EfficientNet:** We designed a simplified version utilizing **Depthwise Separable Convolutions**, which significantly reduces the computational cost (FLOPs) required per inference.

B. Training and Preprocessing

We utilized the MNIST dataset, resizing all images to **RGB**. Crucially, we inverted the colors (black text on white background) during training to match the camera's input. The models were trained for 10 epochs, quantized to **INT8**, and exported as C header files (squeezenet_model.h, efficientnet_model.h).

3. Embedded Implementation (ESP32 Firmware)

The firmware was designed to be modular, allowing the user to trigger individual models or the fused ensemble via Serial commands.

A. Dynamic Memory Management

Running two deep learning models simultaneously exceeds the ESP32's RAM. We solved this by implementing **Sequential Loading**:

1. A single "Tensor Arena" is allocated in PSRAM.
2. When a specific model is requested, the code dynamically instantiates the interpreter for that model, runs inference, saves the results, and then clears the memory to make room for the next model.

B. Fusion Logic

We implemented a **Soft Voting** mechanism (Command '3'):

1. The system captures an image buffer.
2. It runs **SqueezeNet** on the buffer stores probabilities .
3. It runs **EfficientNet** on the same buffer stores probabilities .
4. It calculates the average score: .
5. The digit with the highest combined score is selected.

---

4. Results and Observations

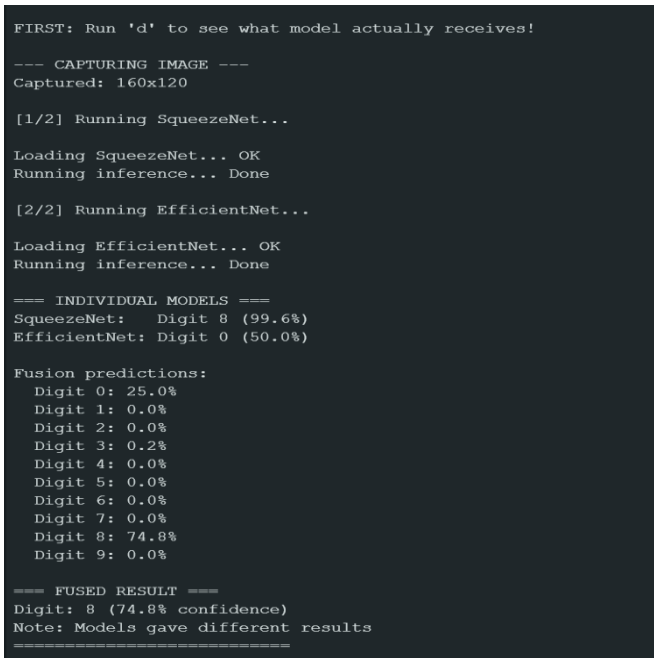
A. Deployment Outcome

The complex firmware successfully compiled and uploaded. The ESP32 was able to capture images, sequentially load both neural networks into the Tensor Arena, and execute the fusion logic without crashing.

B. Performance Analysis

Despite the successful software implementation, the **inference accuracy was poor**. As seen in the results below, both individual models and the fused system consistently failed to correctly identify the handwritten digits, often hallucinating incorrect classes with high confidence.

Figure 4.1: *Individual Model Performance. Both SqueezeNet and EfficientNet produced incorrect predictions for the input digit.*



```
FIRST: Run 'd' to see what model actually receives!

--- CAPTURING IMAGE ---
Captured: 160x120

[1/2] Running SqueezeNet...

Loading SqueezeNet... OK
Running inference... Done

[2/2] Running EfficientNet...

Loading EfficientNet... OK
Running inference... Done

=== INDIVIDUAL MODELS ===
SqueezeNet:   Digit 8 (99.6%)
EfficientNet: Digit 0 (50.0%)

Fusion predictions:
  Digit 0: 25.0%
  Digit 1: 0.0%
  Digit 2: 0.0%
  Digit 3: 0.2%
  Digit 4: 0.0%
  Digit 5: 0.0%
  Digit 6: 0.0%
  Digit 7: 0.0%
  Digit 8: 74.8%
  Digit 9: 0.0%

=== FUSED RESULT ===
Digit: 8 (74.8% confidence)
Note: Models gave different results
===========================
```

C. Analysis of Failure

We attribute the poor accuracy to several critical factors:

1. **Domain Shift:** The models were trained on MNIST (resampled to 32x32), which consists of high-quality scans. The ESP32 camera images, even after preprocessing, contain noise, shadows, and perspective distortions that the models never saw during training.
2. **Quantization Loss:** Converting the lightweight SqueezeNet/EfficientNet models to 8-bit integers likely destroyed the fine-grained features necessary to distinguish digits at such a low resolution ().
3. **Alignment:** MNIST digits are perfectly centered. In our manual tests, even slight off-centering of the digit in the camera frame likely caused the Convolutional Neural Networks (CNNs) to misinterpret features.

5. Conclusion

In this section, we successfully demonstrated the **software architecture** required for Edge AI Ensemble Learning. We proved it is possible to run multiple complex neural networks sequentially on a single ESP32-CAM using dynamic memory management. However, the functional recognition performance highlights the difficulty of transferring models trained on synthetic/ideal data (MNIST) to noisy real-world sensors without extensive data augmentation or transfer learning on real camera samples.