MARMARA
UNIVERSITY

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
FACULTY OF ENGINEERING
MARMARA UNIVERSITY

Embedded Digital Image Processing

Final Project Report

Rüzgar Batı Okay
Student No: 150722048

Semih Yıldız
Student No: 150721029

EE-4065 Embedded Digital Image Processing

January 15, 2026

Abstract

This report presents the implementation of image processing and basic computer vision methods on the ESP32-CAM, a low-cost microcontroller with an integrated camera and limited computational resources. Due to these constraints, all algorithms are designed to be efficient and suitable for embedded systems.

The project includes three main components. First, a histogram-based thresholding method is implemented to detect bright objects on dark backgrounds. Second, a handwritten digit detection system is developed using a YOLO based model trained to recognize the digits 0, 4 and 7. Third, image resizing operations are implemented to support both upsampling and downsampling with non-integer scale factors such as 1.5x and 2/3.

All implementations rely on integer arithmetic instead of floating-point operations to ensure compatibility with the ESP32-CAM hardware. The development follows a two-stage workflow: algorithms are first designed and tested in Python on a PC, then ported and optimized in C/C++ for execution on the ESP32-CAM. Experimental results show that efficient integer-based methods can achieve practical image processing performance on resource-constrained hardware. The thresholding algorithm successfully extracts target objects, the YOLO model detects handwritten digits with reasonable accuracy and the resizing operations produce acceptable visual results for embedded applications.

Contents

1 Introduction **6**

2 Experimental Setup and Hardware Connections **7**

3 Question 1: Thresholding for Object Detection **9**

3.1 Problem Description 9

3.2 Method 9

3.3 Python Implementation on PC 10

3.4 ESP32-CAM Implementation and Python Connector Script 11

3.5 Results and Evaluation 12

4 Question 2: Handwritten Digit Detection using YOLO **14**

4.1 Problem Description 14

4.2 Method 14

4.3 Python Implementation on PC 14

4.3.1 Dataset Preparation 14

4.3.2 Model Training 15

4.3.3 Model Evaluation 16

4.3.4 Hyperparameter Search Results 16

4.3.5 Training Progress 16

4.3.6 Test Set Detection Results 17

4.3.7 Evaluation Curves 18

4.3.8 Overfitting Strategy 20

4.3.9 Deployment Considerations 20

4.4 Results and Evaluation 20

5 Question 3: Image Upsampling and Downsampling **22**

5.1 Problem Description 22

5.2 Method 22

5.3 Python Implementation on PC 23

5.4 ESP32-CAM Implementation and Python Connector Script 24

5.5 Results and Evaluation 26

6 Conclusion **27**

A Question 1: Complete Code Listings **28**

A.1 Python Implementation 28

A.2 ESP32-CAM Implementation 30

A.3 Python Connector Script 33

B Question 2: Relevant Code Snippets **35**

B.1 Dataset Preparation Key Functions 35

B.2 Training Script Configuration and Logic 37

B.3 Model Evaluation Logic 38

B.4 Inference for Report Figures 40

B.5 Note on Code Size 41

C Question 3: Complete Code Listings **41**

C.1 Python Implementation 41

C.2 ESP32-CAM Implementation 45

C.3 Python Connector Script 49

List of Figures

1 ESP32-CAM (AI Thinker) connected to USB-to-Serial FTDI adapter for power, programming and serial communication. 8

2 Reference image taken with a mobile phone camera: sugar cube on a darker calculator case and wooden background. Identifying text on the sugar wrapper (company address, etc.) is censored with a white edit. 11

3 Python test: grayscale image (left) and binary output (right) after histogram-based thresholding. Due to viewing angle and lighting conditions from the mobile phone camera and environment, the exact shape is slightly different from the ESP32 result, but in both cases the sugar cube is successfully extracted as a bright object. 11

4 Binary image received from the ESP32-CAM thresholding pipeline. Even though the camera angle and lighting differ from the calculator-case image, the algorithm again extracts the sugar cube as a compact bright region with at most 1000 pixels. 12

5 Training curves for the best experiment (`lr0.001_batch8`). Loss decreases steadily while precision, recall and mAP metrics increase, showing stable learning over 50 epochs. 17

6 Example detections from the test set using the best model. The detector correctly finds and labels digits 0, 4 and 7 under different augmentations. 17

7 Validation batch with ground truth labels for digits 0, 4 and 7. This view helps confirm that the dataset is annotated correctly. 18

8 Precision-recall curves for each digit class. All three classes achieve high area under the curve, with overall mAP@0.5 = 0.954. 19

9 F1 score as a function of confidence threshold. The best trade-off between precision and recall occurs around a confidence of 0.29, where the overall F1 score is about 0.80. Digit 7 shows a stronger drop at high confidence, matching the observed confusion with digit 4. 19

10 Python test results showing original image (160×120), upsampled (400×300 at 2.5×) and downsampled (64×48 at 2/5) versions. The Python implementation uses more extreme scale factors for demonstration; the ESP32 implementation uses 1.5× and 2/3 as specified. 24

11 Original image from ESP32-CAM: 160×120 pixels (no resizing) 25

12 Upsampling result: 160×120 image resized to 240×180 (1.5×) 25

13 Downsampling result: 160×120 image resized to 106×80 (2/3×) 26

List of Tables

1 Validation results for different learning rate and batch size combinations. Values are taken from `validation_metrics.csv`. 16

2 Best model configuration and performance summary. 21

1 Introduction

This project focuses on the implementation of several image processing and computer vision tasks within the constraints of an embedded system. The goal is to design, implement and evaluate algorithms that can operate efficiently using integer arithmetic and limited computational resources.

Three tasks are addressed in this work. The first task implements histogram-based thresholding for object detection, where bright regions are extracted from dark backgrounds under a constraint on the maximum number of detected pixels, limited to 1000 pixels. The second task explores handwritten digit detection using a YOLO-based model trained to recognize the digits 0, 4 and 7. To reduce model complexity and memory requirements, the detection task is intentionally limited to three representative digit classes, increasing the feasibility of deployment on resource-constrained embedded hardware. The third task implements image upsampling and downsampling with non-integer scale factors using nearest-neighbor interpolation.

All tasks follow a two-stage development workflow. Algorithms are first developed and tested in Python on a PC to validate correctness and behavior. After validation, the implementations are ported to the embedded platform using C/C++, with optimizations focused on integer arithmetic, memory usage and execution efficiency. This workflow allows systematic evaluation of each method while maintaining a clear connection between high-level design and embedded implementation.

2 Experimental Setup and Hardware Connections

This section describes the physical hardware setup used for the ESP32-CAM experiments. The setup uses an ESP32-CAM (AI Thinker module) connected to a USB-to-Serial FTDI adapter, jumper wires and a PC for power, programming and data capture.

An external USB-to-Serial adapter is required because the ESP32-CAM does not provide a native USB interface. The FTDI adapter supplies both power and the serial link needed for flashing and runtime communication.

The essential wiring connections are:

- **FTDI TX → ESP32-CAM U0R**
- **FTDI RX → ESP32-CAM U0T**
- **FTDI GND → ESP32-CAM GND**
- **FTDI 5V → ESP32-CAM 5V**

The FTDI adapter RTS (and/or DTR) control lines are used to handle reset and boot mode automatically during programming, instead of manually wiring IO0 to GND. This allows the ESP32-CAM to enter flashing mode and reset without manual intervention.

The physical setup and wiring are shown in Figure 1.

The ESP32-CAM pin mapping follows the AI Thinker reference design. The pin definitions are based on the official ESP32-CAM examples in the Espressif documentation and the Arduino ESP32 core. This matches the standard AI Thinker layout and ensures correct operation with the OV2640 camera module.

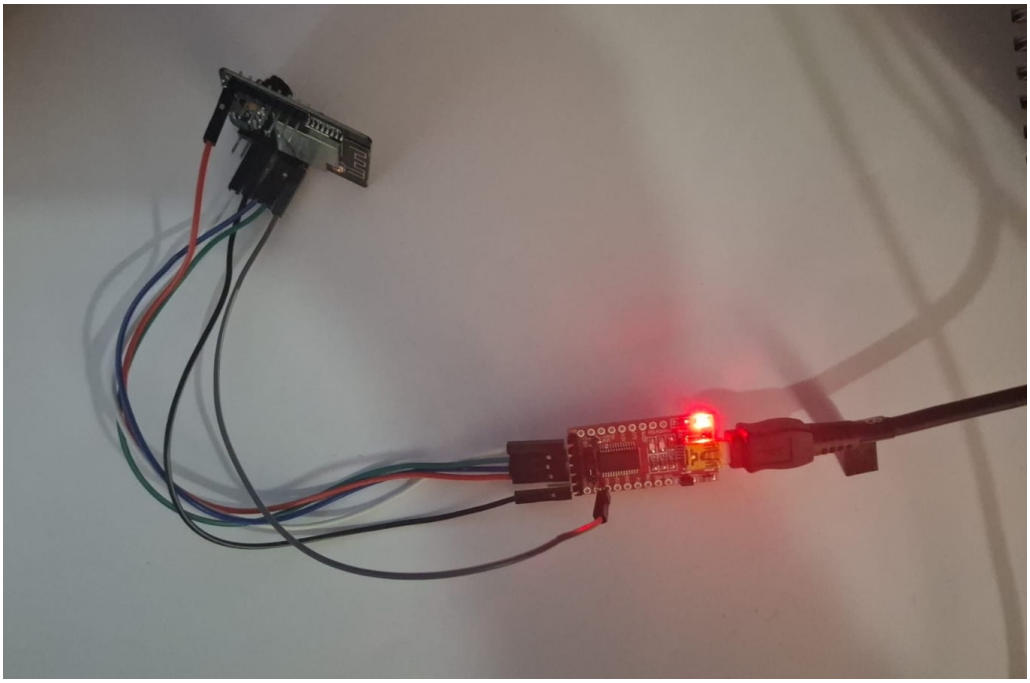


Figure 1: ESP32-CAM (AI Thinker) connected to USB-to-Serial FTDI adapter for power, programming and serial communication.

3 Question 1: Thresholding for Object Detection

This task focuses on detecting bright objects in images captured under low-light conditions. The objective is to extract bright regions from a dark background while limiting the output to a maximum of 1000 bright pixels. The algorithm is first implemented and tested in Python on a PC, then ported to the ESP32-CAM using C for embedded execution.

3.1 Problem Description

The input consists of images where the background is predominantly dark and the object of interest appears significantly brighter. The goal is to generate a binary output image in which bright pixels corresponding to the object are retained, while all other pixels are set to zero. To satisfy memory and processing constraints, the number of bright pixels in the output is limited to at most 1000.

3.2 Method

A histogram-based thresholding approach is used to identify the brightest pixels in the image. First, the grayscale intensity distribution of the image is computed by counting the number of pixels at each brightness level (0-255). Starting from the highest intensity value, pixels are accumulated until the total number of selected pixels reaches 1000. The corresponding intensity level is then used as the threshold.

The algorithm proceeds as follows:

1. Convert the input image to grayscale.
2. Compute a histogram of pixel intensities.
3. Traverse the histogram from the brightest intensity downward until 1000 pixels are accumulated.
4. Select all pixels brighter than or equal to the computed threshold.
5. Generate a binary output image where selected pixels are set to 255 and all others to 0.

This method ensures that the brightest regions are preserved while enforcing a strict limit on the number of output pixels.

3.3 Python Implementation on PC

The algorithm is first implemented in Python to verify correctness and behavior. The complete Python implementation is shown in Appendix A.1. The code closely follows the structure intended for the embedded implementation, avoiding unnecessary abstractions and floating-point operations where possible.

For testing, we used a real photograph of a **sugar cube** placed on the back of a dark calculator case and a wooden table, taken with a mobile phone camera. The printed identifying information (such as company name and address) on the sugar cube is censored in the image, which explains the visible white edit in the reference picture.

The main components of the Python implementation (see Appendix A.1) include:

- `rgb_to_grayscale()`: Converts RGB images to grayscale using the weighted sum

  `gray = (30R + 59G + 11B) / 100`

- `extract_bright_pixels_histogram()`: Constructs the histogram and determines the adaptive threshold.
- `visualize()`: Displays the original and thresholded images for evaluation.

The histogram is computed in a single pass over the image, followed by a second pass to determine the threshold. This design directly mirrors the logic used in the embedded implementation. The Python script loads images from a specific file location and saves visualization results.

Listing 1: Histogram-based threshold selection (Python)

```
hist = [0] * 256
for p in gray_pixels:
    hist[p] += 1

count = 0
threshold = 255
for i in range(255, -1, -1):
    count += hist[i]
    if count >= 1000:
        threshold = i
        break
```

Figure 2: Reference image taken with a mobile phone camera: sugar cube on a darker calculator case and wooden background. Identifying text on the sugar wrapper (company address, etc.) is censored with a white edit.

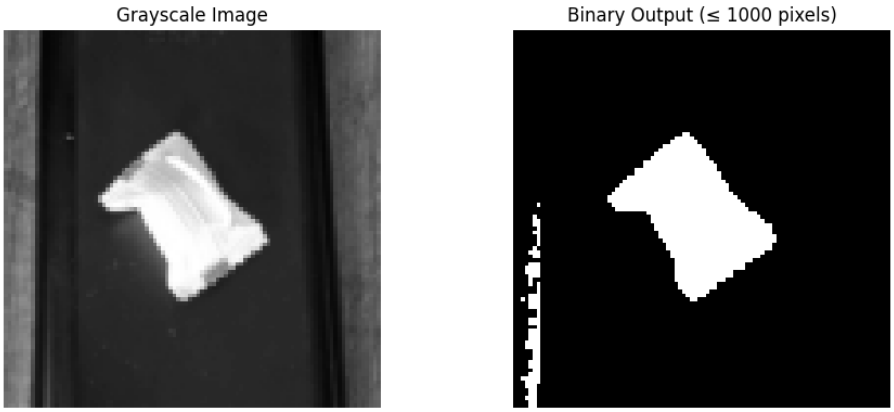


Figure 3: Python test: grayscale image (left) and binary output (right) after histogram-based thresholding. Due to viewing angle and lighting conditions from the mobile phone camera and environment, the exact shape is slightly different from the ESP32 result, but in both cases the sugar cube is successfully extracted as a bright object.

3.4 ESP32-CAM Implementation and Python Connector Script

The validated algorithm is then ported to the ESP32-CAM using C. The complete ESP32 implementation is shown in Appendix A.2. The embedded implementation follows the same logical structure as the Python version, with additional attention paid to memory usage and execution efficiency.

Key characteristics of the ESP32 implementation include:

- Use of grayscale images to reduce memory consumption.

- Single-pass histogram construction to minimize processing time.
- Exclusive use of integer arithmetic.
- High-speed serial transmission (921600 baud) for transferring results to a PC.

Images are captured at a resolution of 96×96 pixels in grayscale format, which provides sufficient spatial detail while remaining well within memory limits.

The embedded processing pipeline consists of the following steps:

1. Construct a grayscale intensity histogram.
2. Determine the adaptive threshold by accumulating pixels from the brightest level downward.
3. Generate a binary image limited to 1000 bright pixels.
4. Transmit the resulting image to the PC with synchronization bytes (0xAA 0x55) and size information.

A Python receiver script (see Appendix A.3) running on the PC receives the serial data, reconstructs the image and saves it as `binary.png` for visualization and evaluation.

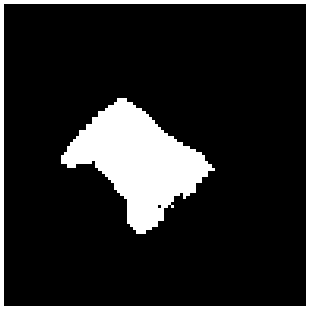


Figure 4: Binary image received from the ESP32-CAM thresholding pipeline. Even though the camera angle and lighting differ from the calculator-case image, the algorithm again extracts the sugar cube as a compact bright region with at most 1000 pixels.

3.5 Results and Evaluation

The histogram-based thresholding algorithm successfully extracts bright objects from dark backgrounds when sufficient contrast is present. By prioritizing the brightest pixels, the method reliably isolates the object of interest while respecting the imposed pixel limit.

The main observations are as follows:

- The output is consistently limited to a maximum of 1000 bright pixels.

- The algorithm executes fast enough to process approximately one frame per second on the ESP32-CAM.
- Memory usage is kept low through grayscale processing and simple data structures.
- The embedded results closely match those obtained from the Python implementation.

4 Question 2: Handwritten Digit Detection using YOLO

This task focuses on detecting and localizing handwritten digits using a YOLO-based object detection model. The objective is to train and evaluate a lightweight detector capable of recognizing the digits 0, 4 and 7, while considering the constraints of eventual deployment on an embedded platform.

4.1 Problem Description

The goal is to train an object detection model that can accurately locate and classify handwritten digits 0, 4 and 7 in grayscale images. The workflow includes dataset preparation, model training and validation, evaluation on a held-out test set and an investigation of deployment feasibility on the ESP32-CAM.

4.2 Method

A YOLOv8-based object detection approach is used. To keep the model lightweight and suitable for embedded deployment, the task is intentionally limited to three digit classes. The overall workflow consists of the following steps:

1. Dataset preparation with controlled data augmentation
2. Model training and hyperparameter selection
3. Evaluation on a separate test set
4. Investigation of model conversion and deployment constraints

4.3 Python Implementation on PC

All training and evaluation are performed using Python 3.11 and the Ultralytics YOLOv8 framework. The implementation logic and key code sections are shown in Appendix B.

4.3.1 Dataset Preparation

Dataset preparation is performed using a Python script. The key functions are shown in Appendix B.1. The preparation process carries out the following operations:

- Splits the dataset into training, validation and test sets based on image indices
- Applies controlled data augmentations, including noise, blur, small rotations (±8°), brightness and contrast changes

- Assigns class labels based on file naming conventions (e.g., `0_1.png` corresponds to digit 0)
- Automatically extracts bounding boxes around digits and adds 10% padding

Data augmentation is used to increase robustness to variations in handwriting and image quality. Flipping and aggressive cropping are intentionally avoided to preserve digit readability.

4.3.2 Model Training

Model training is implemented as shown in Appendix B.2. A YOLOv8n model is selected due to its small size and suitability for resource-constrained environments. The following training strategy is applied:

- Learning rates of 0.001 and 0.01 are evaluated
- Batch sizes of 2, 4 and 8 are tested
- Training is performed for 50 epochs
- Model performance is monitored using validation mAP50

Training is conducted using the training set, hyperparameters are evaluated on the validation set and the test set is used only for final evaluation. All images are resized to 320×320 pixels during training.

Listing 2: Training and metric extraction (Python)

```python
from ultralytics import YOLO

model = YOLO("yolov8n.pt")
results = model.train(
    data="data.yaml",
    imgsz=320,
    epochs=50,
    batch=8,
    lr0=0.001
)

metrics = results.results_dict
map50 = metrics.get("metrics/mAP50(B)", 0.0)
```

4.3.3 Model Evaluation

The trained models are evaluated using the evaluation logic shown in Appendix B.3. The evaluation computes standard object detection metrics, including mAP50, mAP50-95, precision and recall. Results are saved to CSV files (`validation_metrics.csv`, `test_results.csv`) for analysis.

An additional inference script is used to generate qualitative results by drawing bounding boxes and confidence scores on test images.

4.3.4 Hyperparameter Search Results

Six combinations of learning rate and batch size are evaluated. Validation results are summarized in Table 1. The configuration with a learning rate of 0.001 and a batch size of 8 achieves the best validation performance, with an mAP50 of 0.954. These results are recorded in the file `validation_metrics.csv`.

Table 1: Validation results for different learning rate and batch size combinations. Values are taken from `validation_metrics.csv`.

Name	LR	Batch	mAP50	mAP50-95	Precision	Recall
lr0.001_batch2	0.001	2	0.834	0.769	0.738	0.940
lr0.001_batch4	0.001	4	0.905	0.876	0.784	0.889
lr0.001_batch8	**0.001**	**8**	**0.954**	**0.954**	**0.813**	**0.822**
lr0.01_batch2	0.01	2	0.767	0.700	0.719	0.722
lr0.01_batch4	0.01	4	0.905	0.876	0.784	0.889
lr0.01_batch8	0.01	8	0.954	0.954	0.813	0.822

The best model configuration is recorded in `best_model_info.txt`, confirming that the selected model is `lr0.001_batch8` with learning rate 0.001, batch size 8 and validation mAP50 of 0.9542. The trained weights are saved as `best.pt` in the corresponding experiment directory.

4.3.5 Training Progress

Figure 5 illustrates the training behavior over 50 epochs. The loss curves show a steady decrease, while precision, recall and mAP metrics increase consistently. Both mAP50 and mAP50–95 reach high values, indicating successful convergence of the model.

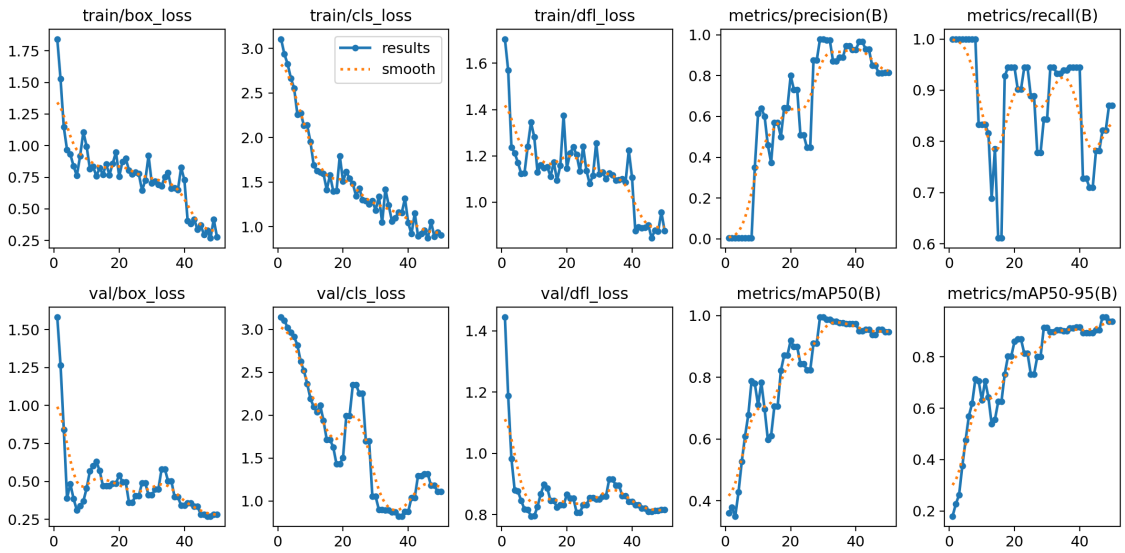
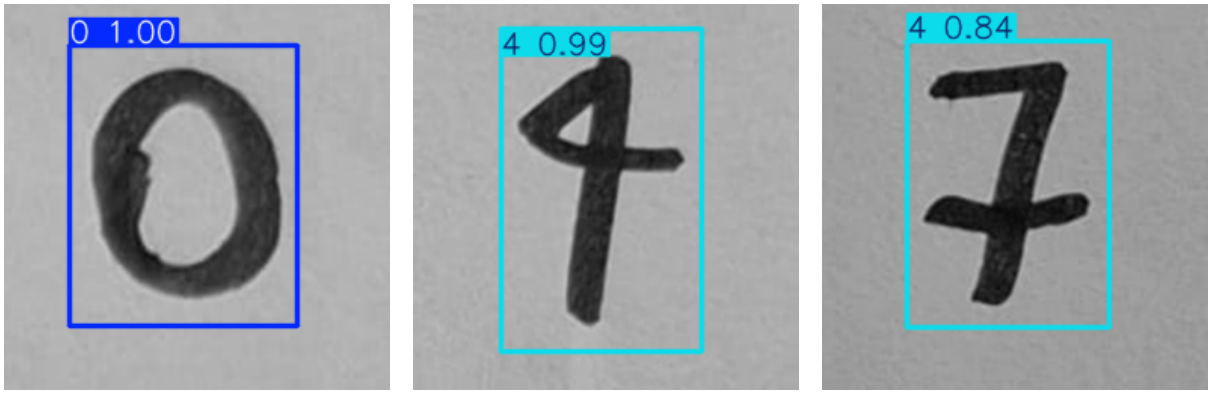


Figure 5: Training curves for the best experiment (`lr0.001_batch8`). Loss decreases steadily while precision, recall and mAP metrics increase, showing stable learning over 50 epochs.

4.3.6 Test Set Detection Results

The best-performing model is evaluated on the test set. Sample detection results are shown in Figure 6. The model successfully detects and localizes handwritten digits across different variations, including rotation, contrast changes and noise.

Misclassification occurs primarily between digits 4 and 7. This behavior is expected, as these digits share similar structural elements in handwritten form, such as vertical strokes and diagonal lines.



(a) Digit 0 (b) Digit 4 (rotated) (c) Digit 7 (contrast change)

Figure 6: Example detections from the test set using the best model. The detector correctly finds and labels digits 0, 4 and 7 under different augmentations.

To visualize the training data and labels, Figure 7 shows a validation batch with ground truth boxes for all three digit classes.

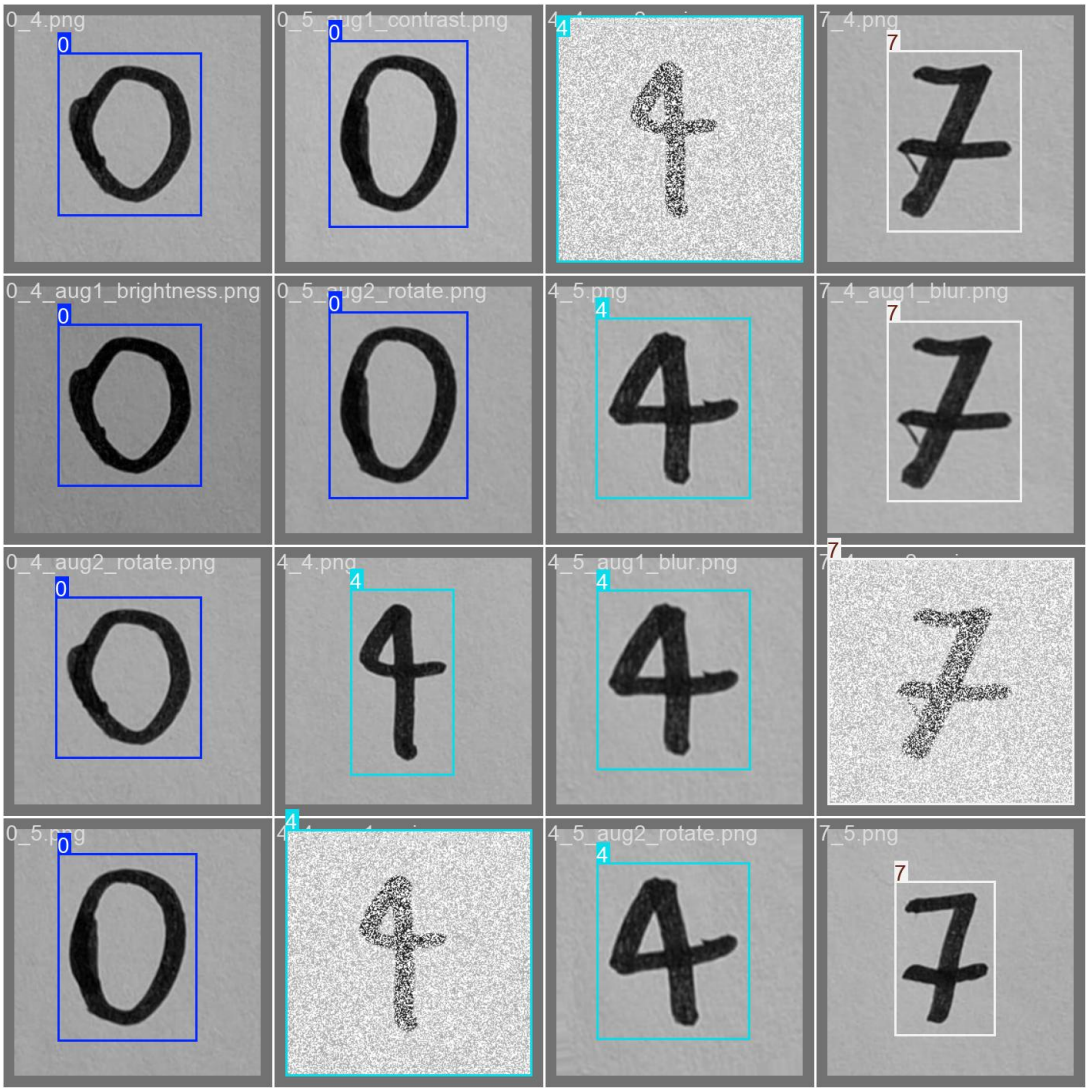


Figure 7: Validation batch with ground truth labels for digits 0, 4 and 7. This view helps confirm that the dataset is annotated correctly.

4.3.7 Evaluation Curves

Model performance across different confidence thresholds is analyzed using precision-recall and F1-confidence curves. The precision-recall curves in Figure 8 show strong performance for all three classes, with an overall mAP@0.5 of 0.954.

The F1-confidence curves in Figure 9 indicate that the optimal confidence threshold for balanced precision and recall is approximately 0.29, yielding a maximum F1 score of 0.80. The digit 7 class shows a more pronounced drop in F1 at higher confidence thresholds, reflecting confusion with digit 4.

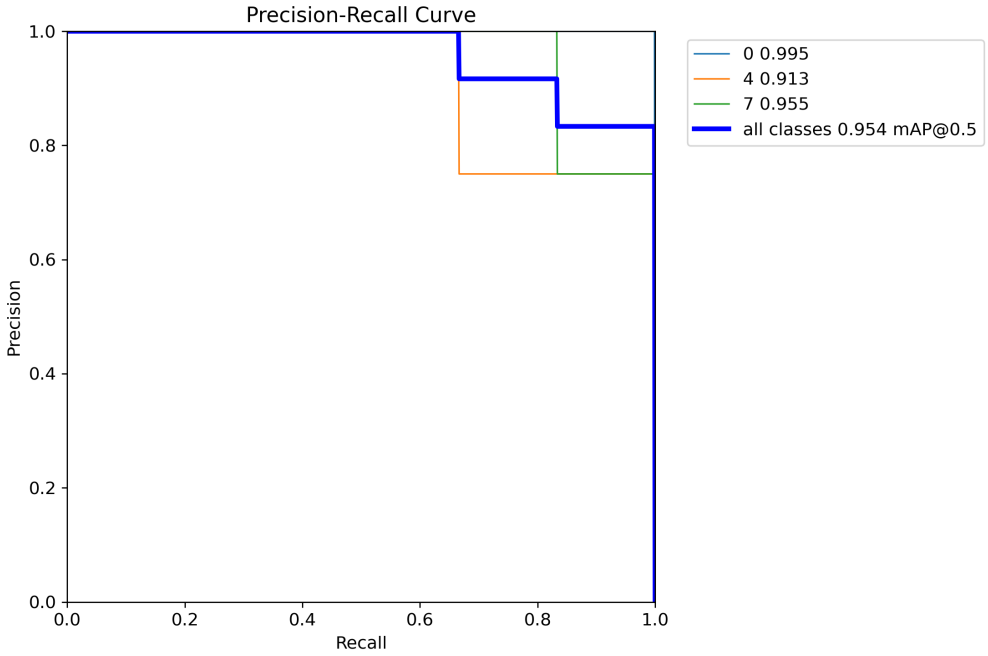


Figure 8: Precision-recall curves for each digit class. All three classes achieve high area under the curve, with overall mAP@0.5 = 0.954.

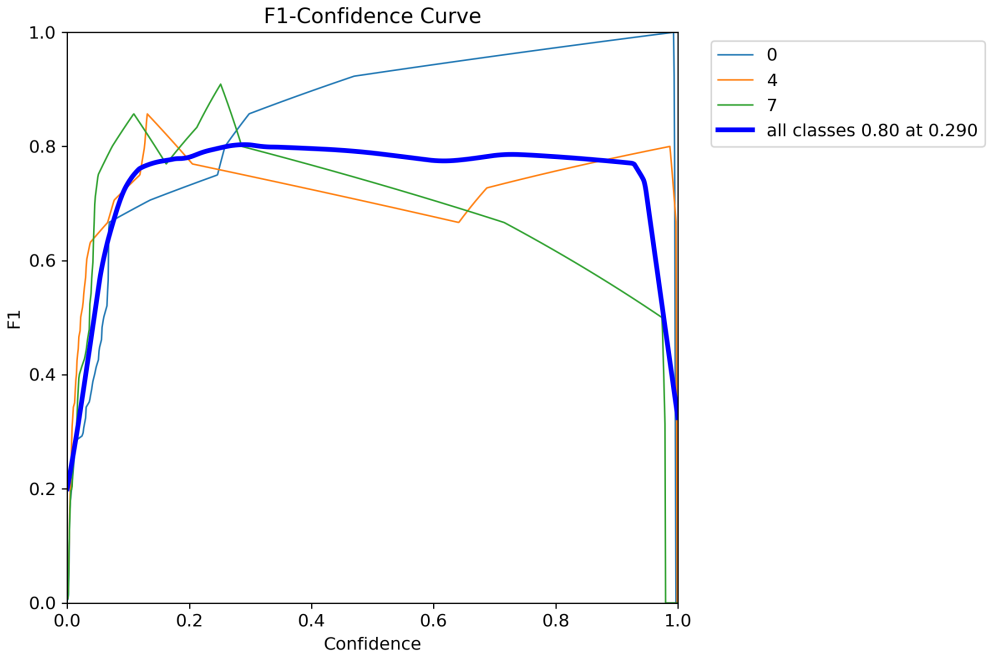


Figure 9: F1 score as a function of confidence threshold. The best trade-off between precision and recall occurs around a confidence of 0.29, where the overall F1 score is about 0.80. Digit 7 shows a stronger drop at high confidence, matching the observed confusion with digit 4.

4.3.8 Overfitting Strategy

Training is intentionally performed for the full 50 epochs without early stopping, even after signs of overfitting appear. This decision is motivated by the planned deployment of the model on highly constrained hardware.

A fully trained model tends to learn stronger feature representations, which are more likely to survive aggressive compression and integer quantization. Since the primary objective is reliable digit detection rather than perfect classification, limited overfitting is considered acceptable for this application.

4.3.9 Deployment Considerations

Several attempts are made to convert the trained model for embedded deployment:

- Conversion attempts using Python 3.10 and older YOLO versions
- Exploration of the PyTorch -¿ ONNX -¿ TensorFlow -¿ TFLite -¿ TFLite Micro pipeline
- Encountered compatibility issues related to library versions

The current implementation remains PC-based, with deployment to the ESP32-CAM identified as future work. The chosen training and overfitting strategy is expected to improve robustness after heavy quantization.

4.4 Results and Evaluation

The final model achieves strong detection performance for the selected digit classes. Digit 0 is detected most reliably, while digits 4 and 7 exhibit occasional confusion due to visual similarity. Overall, the results demonstrate that a lightweight YOLO-based model can effectively detect handwritten digits in controlled conditions, providing a strong foundation for future embedded deployment.

The best model configuration and its performance metrics are summarized in Table 2. The validation metrics are taken from `best_model_info.txt` and `validation_metrics.csv`, while test metrics are recorded in `test_results.csv`.

Table 2: Best model configuration and performance summary.

Parameter	**Value**
Model name	lr0.001_batch8
Learning rate	0.001
Batch size	8
Validation mAP50	0.9542
Validation Precision	0.813
Validation Recall	0.822

The test evaluation results are documented in `test_results.csv`. The qualitative detections shown in Figure 6 demonstrate that the model successfully detects and classifies digits on the held-out test set, with the strong validation scores from Table 1 confirming overall model performance.

5 Question 3: Image Upsampling and Downsampling

This task focuses on implementing image resizing operations on an embedded system. Both upsampling and downsampling are performed using non-integer scale factors while operating under strict constraints on memory usage and arithmetic precision.

5.1 Problem Description

The objective is to resize images captured by the ESP32-CAM using two operations:

- **Upsampling**: Increasing image size by a factor of 1.5
- **Downsampling**: Reducing image size by a factor of 2/3

All computations must use integer arithmetic only, as floating-point operations are not suitable for the target platform.

5.2 Method

Nearest-neighbor interpolation is selected due to its simplicity, low computational cost and suitability for embedded systems. Non-integer scaling is implemented using rational scale factors of the form:

$$\text{scale} = \frac{\text{scale\_num}}{\text{scale\_den}}$$

For upsampling (1.5×), the scale factors are defined as:

- `scale_num = 3`
- `scale_den = 2`

For downsampling (2/3×), the scale factors are:

- `scale_num = 2`
- `scale_den = 3`

Inverse mapping is used to determine the source pixel corresponding to each output pixel:

$$\text{src\_x} = \frac{\text{out\_x} \times \text{scale\_den}}{\text{scale\_num}}$$

This approach ensures that all source indices remain valid and avoids gaps or undefined pixels in the output image.

Note: The Python implementation uses more extreme scale factors (2.5× for upsampling and 2/5 for downsampling) for demonstration purposes to make the differences more visible. The ESP32 implementation uses the original scale factors (1.5× and 2/3) as specified in the requirements.

5.3 Python Implementation on PC

The resizing algorithm is first implemented and tested in Python. The complete Python implementation is shown in Appendix C.1. The code closely mirrors the embedded implementation and uses only integer arithmetic to ensure direct portability.

The main function, `resize_nearest_neighbor()`, performs the following steps:

- Computes the output image size using integer arithmetic
- Applies inverse mapping to determine source pixel indices
- Ensures all coordinates remain within valid bounds
- Copies RGB pixel values using nearest-neighbor selection

The Python implementation works directly with RGB color images, matching the ESP32 implementation which processes RGB565 color images.

Using the same arithmetic and logic as the ESP32 implementation allows validation of correctness before deployment.

For demonstration purposes, the Python implementation uses more extreme scale factors (2.5× for upsampling and 2/5 for downsampling) to make the differences more visible. The ESP32 implementation uses the original scale factors (1.5× and 2/3) as specified in the requirements.

Figure 10: Python test results showing original image (160×120), upsampled (400×300 at 2.5×) and downsampled (64×48 at 2/5) versions. The Python implementation uses more extreme scale factors for demonstration; the ESP32 implementation uses 1.5× and 2/3 as specified.

5.4 ESP32-CAM Implementation and Python Connector Script

The resizing algorithm is ported to the ESP32-CAM and implemented in C/C++. The complete ESP32 implementation is shown in Appendix C.2. The system operates on RGB565 images (16 bits per pixel) and applies nearest-neighbor interpolation using integer arithmetic.

Key implementation features include:

- Capture of 160×120 RGB565 images
- Integer-only resizing using inverse mapping
- JPEG encoding to reduce transmission size
- Serial transmission of images to a PC

The function `resize_nearest()` performs resizing directly on RGB565 pixel data by copying source pixels to the appropriate output locations.

Listing 3: Inverse mapping for nearest-neighbor resize (ESP32)

```
int src_y = (y * SCALE_DEN) / SCALE_NUM;
int src_x = (x * SCALE_DEN) / SCALE_NUM;

if (src_y >= IN_H) src_y = IN_H - 1;
if (src_x >= IN_W) src_x = IN_W - 1;

resized[y * OUT_W + x] = in_rgb565[src_y * IN_W + src_x];
```

A compile-time setting (`IMAGE_MODE`) controls the resizing mode:

- `IMAGE_MODE = 0`: Original image (160×120, no resizing)
- `IMAGE_MODE = 1`: Upsampling to 240×180 (1.5×)

- `IMAGE_MODE = 2`: Downsampling to 106×80 (2/3×)

The Python receiver script (see Appendix C.3) receives the transmitted JPEG images and saves them to disk for visualization and analysis.

Figure 11: Original image from ESP32-CAM: 160×120 pixels (no resizing)

Figure 12: Upsampling result: 160×120 image resized to 240×180 (1.5×)

Figure 13: Downsampling result: 160×120 image resized to 106×80 (2/3×)

5.5 Results and Evaluation

Both upsampling and downsampling operations function correctly using non-integer scale factors. Nearest-neighbor interpolation provides sufficient image quality while maintaining low computational complexity.

Key observations include:

- Integer arithmetic correctly supports non-integer scaling ratios
- RGB565 color integrity is preserved after resizing
- JPEG encoding reduces transmission size effectively
- Performance is sufficient for near real-time operation
- Embedded results closely match Python reference outputs

6 Conclusion

This project shows that useful image processing tasks can be built on low-cost embedded hardware when the logic is kept simple and memory-aware. All three tasks were first verified in Python and then aligned with the embedded implementation using integer-only operations.

Key outcomes are:

- Histogram thresholding reliably extracts bright objects with a fixed pixel limit.
- A lightweight YOLO model detects digits 0, 4 and 7 with strong validation results.
- Nearest-neighbor resizing works for both upsampling and downsampling on RGB565 images.

Future work includes model quantization for ESP32 deployment and testing under more varied lighting conditions. Overall, the workflow provides a clear and repeatable path from PC validation to embedded execution.

A Question 1: Complete Code Listings

A.1 Python Implementation

Listing 4: Python Implementation for Thresholding (`python/q1.py`)

```python
from PIL import Image
import matplotlib.pyplot as plt
from pathlib import Path


def rgb_to_grayscale(image):
    """
    Convert RGB image to grayscale using integer arithmetic.
    Formula matches embedded-friendly luminance approximation.
    gray = (30*R + 59*G + 11*B) / 100
    """

    width, height = image.size
    gray = [[0 for _ in range(width)] for _ in range(height)]

    pixels = image.load()

    for y in range(height):
        for x in range(width):
            r, g, b = pixels[x, y]
            gray[y][x] = (30 * r + 59 * g + 11 * b) // 100

    return gray


def extract_bright_pixels_histogram(gray, max_pixels=1000):
    """
    ESP32-friendly adaptive thresholding using histogram
       accumulation.
    Ensures output contains at most 'max_pixels' bright pixels.
    All other pixels are set to 0.
    """

    height = len(gray)
    width = len(gray[0])

```

```
    # 1. Build grayscale histogram
    hist = [0] * 256
    for y in range(height):
        for x in range(width):
            hist[gray[y][x]] += 1

    # 2. Find threshold from brightest to darkest
    cumulative = 0
    threshold = 255
    for intensity in range(255, -1, -1):
        cumulative += hist[intensity]
        if cumulative >= max_pixels:
            threshold = intensity
            break

    # 3. Create binary output image
    output = [[0 for _ in range(width)] for _ in range(height)]

    selected = 0
    for y in range(height):
        for x in range(width):
            if gray[y][x] >= threshold and selected < max_pixels:
                output[y][x] = 255
                selected += 1
            else:
                output[y][x] = 0

    return output, threshold


def visualize(gray, binary):
    """
    Visualization for PC validation only.
    Not part of embedded logic.
    """

    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    plt.title("Grayscale Image")
    plt.imshow(gray, cmap="gray")
```

```
    plt.axis("off")

    plt.subplot(1, 2, 2)
    plt.title("Binary Output (    1000 pixels)")
    plt.imshow(binary, cmap="gray")
    plt.axis("off")

    plt.tight_layout()
    plt.show()


def main():
    # Load image (RGB) relative to this script's folder
    base_dir = Path(__file__).resolve().parent
    img_path = base_dir / "question1_images" / "
        reference_taken_from_phone.jpg"
    image = Image.open(img_path).convert("RGB")

    # For Python testing only: resize to match ESP32 resolution
    target_size = (96, 96)
    image = image.resize(target_size, Image.BILINEAR)

    # Convert to grayscale (ESP32-style)
    gray = rgb_to_grayscale(image)

    # Apply histogram-based thresholding
    binary, threshold = extract_bright_pixels_histogram(
        gray, max_pixels=1000
    )

    print("Selected threshold intensity:", threshold)

    # Visualization (PC only)
    visualize(gray, binary)


if __name__ == "__main__":
    main()
```

A.2 ESP32-CAM Implementation

Listing 5: ESP32 CAM Thresholding Code (`esp32_cam_link/esp32_cam_q1/esp32_cam_q1.ino`)

```
#include "esp_camera.h"
#include <string.h>

// ===== AI Thinker ESP32-CAM pin map =====
#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

#define FLASH_GPIO        4

#define WIDTH  96
#define HEIGHT 96
#define MAX_PIXELS 1000

void setup() {
  Serial.begin(921600);
  delay(2000);

  pinMode(FLASH_GPIO, OUTPUT);
  digitalWrite(FLASH_GPIO, LOW);

  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;
  config.frame_size   = FRAMESIZE_96X96;
  config.fb_count     = 1;

  esp_camera_init(&config);
}

void loop() {
  digitalWrite(FLASH_GPIO, HIGH);   // Flash ON
  delay(20);                        // allow light to stabilize

  camera_fb_t *fb = esp_camera_fb_get();
  digitalWrite(FLASH_GPIO, LOW);    // Flash OFF

  if (!fb) return;

  // Histogram
  uint32_t hist[256] = {0};
  for (int i = 0; i < fb->len; i++) {
    hist[fb->buf[i]]++;
  }

  // Threshold selection
  uint32_t sum = 0;
  uint8_t threshold = 255;
  for (int i = 255; i >= 0; i--) {
```

```
    sum += hist[i];
    if (sum >= MAX_PIXELS) {
      threshold = i;
      break;
    }
  }

  // Binary output
  static uint8_t binary[WIDTH * HEIGHT];
  uint32_t selected = 0;

  for (int i = 0; i < fb->len; i++) {
    if (fb->buf[i] >= threshold && selected < MAX_PIXELS) {
      binary[i] = 255;
      selected++;
    } else {
      binary[i] = 0;
    }
  }

  // Send to PC
  uint8_t sync[2] = {0xAA, 0x55};
  uint32_t size = fb->len;

  Serial.write(sync, 2);
  Serial.write((uint8_t*)&size, 4);
  Serial.write(binary, size);

  esp_camera_fb_return(fb);

  delay(1000);   // 1 frame per second (stable & safe)
}
```

A.3 Python Connector Script

Listing 6: Python Receiver Script (esp32_cam_link/esp32_cam_q1/receive.py)

```
import serial
import struct
import numpy as np
from PIL import Image
```

```
5
6  PORT = "COM4"
7  BAUD = 921600
8
9  WIDTH = 96
10 HEIGHT = 96
11
12 ser = serial.Serial(
13     PORT,
14     BAUD,
15     timeout=5,
16     dsrdtr=False,
17     rtscts=False
18 )
19
20 ser.setDTR(False)
21 ser.setRTS(False)
22
23 img_count = 0
24 print("Receiving binary images... Press Ctrl+C to stop.")
25
26 try:
27     while True:
28         # Sync
29         if ser.read(1) != b'\xAA':
30             continue
31         if ser.read(1) != b'\x55':
32             continue
33
34         # Length
35         size_bytes = ser.read(4)
36         size = struct.unpack("<I", size_bytes)[0]
37
38         # Data
39         data = ser.read(size)
40         if len(data) != size:
41             print("Incomplete frame")
42             continue
43
44         img = np.frombuffer(data, dtype=np.uint8)
45         img = img.reshape((HEIGHT, WIDTH))
```

```
46
47          image = Image.fromarray(img, mode='L')
48          filename = f"binary.png"
49          image.save(filename)
50
51          print(f"Saved {filename}")
52          img_count += 1
53
54  except KeyboardInterrupt:
55      print("\nStopped by user.")
56      ser.close()
```

B Question 2: Relevant Code Snippets

Due to the large size of the YOLO implementation, only key code sections showing the logic flow are included here.

B.1 Dataset Preparation Key Functions

Listing 7: Key Functions from Dataset Preparation Script

```
1  def get_class_id_from_filename(filename):
2      """Extract class ID from filename: 0_1.png -> class 0, 4_5.png
          -> class 1, 7_6.png -> class 2"""
3      if filename.startswith('0_'):
4          return 0  # Class 0 = digit "0"
5      elif filename.startswith('4_'):
6          return 1  # Class 1 = digit "4"
7      elif filename.startswith('7_'):
8          return 2  # Class 2 = digit "7"
9      return None
10
11 def get_image_number(filename):
12     """Extract image number from filename: 0_1.png -> 1, 4_6.png ->
          6"""
13     parts = filename.split('_')
14     if len(parts) >= 2:
15         try:
16             return int(parts[1].split('.')[0])
17         except:
```

```
            return None
    return None

def detect_digit_region(image):
    """Detect bounding box of digit in image using contour detection"""
    _, binary = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
    contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        x, y, w, h = cv2.boundingRect(largest_contour)

        # Add 10% padding
        h_img, w_img = image.shape
        padding_x = int(w * 0.1)
        padding_y = int(h * 0.1)
        x = max(0, x - padding_x)
        y = max(0, y - padding_y)
        w = min(w_img - x, w + 2 * padding_x)
        h = min(h_img - y, h + 2 * padding_y)
        return x, y, w, h

    # Fallback: centered box covering 80% of image
    h_img, w_img = image.shape
    margin = int(w_img * 0.1)
    return margin, margin, w_img - 2*margin, h_img - 2*margin

def create_label_file(image_path, label_dir, image):
    """Create YOLO format label file: class_id center_x center_y width height (all normalized)"""
    filename = os.path.basename(image_path)
    label_path = os.path.join(label_dir, os.path.splitext(filename)[0] + '.txt')

    class_id = get_class_id_from_filename(filename)
    if class_id is None:
        return

```

```
    x, y, w, h = detect_digit_region(image)
    img_h, img_w = image.shape

    # Convert to YOLO format (normalized)
    center_x = max(0, min(1, (x + w / 2) / img_w))
    center_y = max(0, min(1, (y + h / 2) / img_h))
    norm_width = max(0, min(1, w / img_w))
    norm_height = max(0, min(1, h / img_h))

    with open(label_path, 'w') as f:
        f.write(f"{class_id} {center_x:.6f} {center_y:.6f} {
           norm_width:.6f} {norm_height:.6f}\n")
```

B.2 Training Script Configuration and Logic

Listing 8: Training Configuration and Main Training Function

```
from ultralytics import YOLO
import pandas as pd
from pathlib import Path
import time

# Configuration
HYPERPARAMETER_EXPERIMENTS = [
    {"lr0": 0.001, "batch": 2, "name": "lr0.001_batch2"},
    {"lr0": 0.001, "batch": 4, "name": "lr0.001_batch4"},
    {"lr0": 0.001, "batch": 8, "name": "lr0.001_batch8"},
    {"lr0": 0.01, "batch": 2, "name": "lr0.01_batch2"},
    {"lr0": 0.01, "batch": 4, "name": "lr0.01_batch4"},
    {"lr0": 0.01, "batch": 8, "name": "lr0.01_batch8"},
]

def run_experiment(exp_config, exp_id):
    model = YOLO("yolov8n.pt")
    project_name = f"exp_{exp_id + 1}_{exp_config['name']}"

    # Train model
    results = model.train(
        data="data.yaml",
        epochs=50,
        imgsz=320,
```

```
        batch=exp_config["batch"],
        lr0=exp_config["lr0"],
        device="cpu",
        project=str(RESULTS_DIR),
        name=project_name,
        save=True,
        plots=True,
    )

    # Extract metrics from training results
    if hasattr(results, 'results_dict'):
        metrics = results.results_dict
        map50 = float(metrics.get('metrics/mAP50(B)', 0.0))
        precision = float(metrics.get('metrics/precision(B)', 0.0))
        recall = float(metrics.get('metrics/recall(B)', 0.0))
    elif hasattr(results, 'metrics') and hasattr(results.metrics, '
       box'):
        m = results.metrics.box
        map50 = float(m.map50)
        precision = float(m.mp)
        recall = float(m.mr)
    else:
        # Fallback: run validation explicitly
        val_results = model.val(data="data.yaml", imgsz=320)
        m = val_results.metrics.box
        map50 = float(m.map50)
        precision = float(m.mp)
        recall = float(m.mr)

    return {
        "name": exp_config["name"],
        "learning_rate": exp_config["lr0"],
        "batch_size": exp_config["batch"],
        "mAP50": map50,
        "precision": precision,
        "recall": recall,
        "model_path": str(best_model_path),
    }
```

B.3 Model Evaluation Logic

Listing 9: Test Script Main Evaluation Function

```
from ultralytics import YOLO
import pandas as pd
from pathlib import Path

def extract_metrics(results):
    """Extract metrics from validation results"""
    if hasattr(results, 'metrics') and hasattr(results.metrics, 'box
        '):
        m = results.metrics.box
        return (
            float(m.map50),
            float(m.map),
            float(m.mp),
            float(m.mr),
        )
    return (0.0, 0.0, 0.0, 0.0)

def test_model(model_path):
    model = YOLO(model_path)

    # Create temporary data.yaml pointing val to test set
    temp_yaml = "data_test.yaml"
    with open("data.yaml", "r") as f:
        lines = f.readlines()

    with open(temp_yaml, "w") as f:
        for line in lines:
            if line.strip().startswith("val:"):
                f.write("val: images/test\n")
            else:
                f.write(line)

    try:
        results = model.val(
            data=temp_yaml,
            conf=0.4,
            verbose=True,
        )

        map50, map5095, precision, recall = extract_metrics(results)
```

```
40
41          return {
42              "mAP50": map50,
43              "mAP50_95": map5095,
44              "precision": precision,
45              "recall": recall,
46          }
47      finally:
48          if os.path.exists(temp_yaml):
49              os.remove(temp_yaml)
```

B.4 Inference for Report Figures

Listing 10: Inference Script for Generating Report Figures

```
1  from ultralytics import YOLO
2  import os
3  import cv2
4  import glob
5
6  MODEL_PATH = None
7  TEST_DIR = "images/test"
8  OUTPUT_DIR = "figures"
9  CONF_THRESHOLD = 0.25
10
11 def run_inference(model_path):
12     model = YOLO(model_path)
13     os.makedirs(OUTPUT_DIR, exist_ok=True)
14     image_paths = []
15     for ext in ("*.jpg", "*.jpeg", "*.png"):
16         image_paths.extend(glob.glob(os.path.join(TEST_DIR, ext)))
17
18     for img_path in sorted(image_paths):
19         results = model(img_path, conf=CONF_THRESHOLD)
20         result = results[0]
21         out_path = os.path.join(OUTPUT_DIR, f"result_{os.path.
               basename(img_path)}")
22         annotated = result.plot()
23         cv2.imwrite(out_path, annotated)
```

B.5 Note on Code Size

The complete YOLO implementation includes:

- Dataset preparation script: 225 lines
- Training script: 201 lines
- Test script: 199 lines
- Inference script: 131 lines
- Data configuration file
- Multiple hyperparameter experiment results

C Question 3: Complete Code Listings

C.1 Python Implementation

Listing 11: Python Implementation for Resizing (`python/q3.py`)

```python
from PIL import Image
import matplotlib.pyplot as plt
from pathlib import Path

# ===== Configuration =====
# Input image settings
INPUT_IMAGE_PATH = "question1_images/reference_taken_from_phone.jpg"
TARGET_SIZE = (160, 120)  # Resize input to this size before processing

# Increased from 1.5x to 2.5x for upsampling and 2/3 to 2/5 for downsampling
# to show more difference. Will not keep this for ESP32 implementation.
# Resizing scale factors
UPSAMPLE_SCALE_NUM = 5  # For 2.5x upsampling: scale = 5/2
UPSAMPLE_SCALE_DEN = 2
DOWNSAMPLE_SCALE_NUM = 2  # For 2/5 downsampling: scale = 2/5
DOWNSAMPLE_SCALE_DEN = 5

# Output settings
```

```
OUTPUT_IMAGE_PATH = "question1_images/q3_comparison.png"
FIGURE_SIZE = (12, 4)  # Figure size in inches (width, height)
FIGURE_DPI = 150  # Resolution for saved figure

# ===== End Configuration =====


def resize_nearest_neighbor(image, scale_num, scale_den):
    """
    ESP32-friendly nearest neighbor resize using integer arithmetic.

    scale = scale_num / scale_den
    Works with RGB images (3 channels).
    """

    input_h = image.height
    input_w = image.width
    pixels = image.load()

    # Compute output dimensions using integer math
    output_h = (input_h * scale_num) // scale_den
    output_w = (input_w * scale_num) // scale_den

    # Create output image
    output = Image.new('RGB', (output_w, output_h))
    output_pixels = output.load()

    # Inverse mapping (nearest neighbor)
    for y in range(output_h):
        for x in range(output_w):
            src_y = (y * scale_den) // scale_num
            src_x = (x * scale_den) // scale_num

            # Clamp to bounds
            if src_y >= input_h:
                src_y = input_h - 1
            if src_x >= input_w:
                src_x = input_w - 1

            # Copy RGB pixel
            output_pixels[x, y] = pixels[src_x, src_y]
```

```
60
61      return output
62
63
64  def visualize(original, upsampled, downsampled, save_path=None,
        figsize=(12, 4), dpi=150):
65      """
66      Visualization for PC validation only.
67      Not part of embedded logic.
68      """
69
70      # Get dimensions for each image
71      orig_w, orig_h = original.size
72      up_w, up_h = upsampled.size
73      down_w, down_h = downsampled.size
74
75      plt.figure(figsize=figsize)
76
77      plt.subplot(1, 3, 1)
78      plt.title(f"Original\n{orig_w}  {orig_h} pixels")
79      plt.imshow(original)
80      plt.axis("off")
81
82      upscale = UPSAMPLE_SCALE_NUM / UPSAMPLE_SCALE_DEN
83      downscale = DOWNSAMPLE_SCALE_NUM / DOWNSAMPLE_SCALE_DEN
84
85      plt.subplot(1, 3, 2)
86      plt.title(f"Upsampled ({upscale}x)\n{up_w}  {up_h} pixels")
87      plt.imshow(upsampled)
88      plt.axis("off")
89
90      plt.subplot(1, 3, 3)
91      plt.title(f"Downsampled ({downscale})\n{down_w}  {down_h} pixels
            ")
92      plt.imshow(downsampled)
93      plt.axis("off")
94
95      plt.tight_layout()
96
97      if save_path:
98          plt.savefig(save_path, dpi=dpi, bbox_inches='tight')
```

```
99          print(f"Saved comparison image to: {save_path}")
100         print(f"  Original: {orig_w}  {orig_h} pixels")
101         print(f"  Upsampled: {up_w}  {up_h} pixels")
102         print(f"  Downsampled: {down_w}  {down_h} pixels")
103     else:
104         plt.show()
105
106
107 def main():
108     # Load image relative to this script's folder
109     base_dir = Path(__file__).resolve().parent
110     img_path = base_dir / INPUT_IMAGE_PATH
111
112     print(f"Loading image from: {img_path}")
113     image = Image.open(img_path).convert("RGB")
114
115     # Resize to target size (similar to ESP32 resolution)
116     image = image.resize(TARGET_SIZE, Image.BILINEAR)
117     print(f"Resized to: {TARGET_SIZE}")
118
119     # Upsample
120     upscale = UPSAMPLE_SCALE_NUM / UPSAMPLE_SCALE_DEN
121     print(f"Upsampling ({upscale}x)...")
122     upsampled = resize_nearest_neighbor(image, scale_num=
        UPSAMPLE_SCALE_NUM, scale_den=UPSAMPLE_SCALE_DEN)
123     print(f"Upsampled size: {upsampled.size[0]}x{upsampled.size[1]}"
        )
124
125     # Downsample
126     downscale = DOWNSAMPLE_SCALE_NUM / DOWNSAMPLE_SCALE_DEN
127     print(f"Downsampling ({downscale})...")
128     downsampled = resize_nearest_neighbor(image, scale_num=
        DOWNSAMPLE_SCALE_NUM, scale_den=DOWNSAMPLE_SCALE_DEN)
129     print(f"Downsampled size: {downsampled.size[0]}x{downsampled.
        size[1]}")
130
131     # Save comparison image
132     output_path = base_dir / OUTPUT_IMAGE_PATH
133     visualize(image, upsampled, downsampled, save_path=str(
        output_path), figsize=FIGURE_SIZE, dpi=FIGURE_DPI)
134
```

```
135
136 if __name__ == "__main__":
137     main()
```

C.2 ESP32-CAM Implementation

Listing 12: ESP32 CAM Resizing Code (`esp32_cam_link/esp32_cam_q3/esp32_cam_q3.ino`)

```
1  #include "esp_camera.h"
2  #include "img_converters.h"
3
4  // ===== AI Thinker ESP32-CAM pin map =====
5  #define PWDN_GPIO_NUM     32
6  #define RESET_GPIO_NUM    -1
7  #define XCLK_GPIO_NUM      0
8  #define SIOD_GPIO_NUM     26
9  #define SIOC_GPIO_NUM     27
10 #define Y9_GPIO_NUM       35
11 #define Y8_GPIO_NUM       34
12 #define Y7_GPIO_NUM       39
13 #define Y6_GPIO_NUM       36
14 #define Y5_GPIO_NUM       21
15 #define Y4_GPIO_NUM       19
16 #define Y3_GPIO_NUM       18
17 #define Y2_GPIO_NUM        5
18 #define VSYNC_GPIO_NUM    25
19 #define HREF_GPIO_NUM     23
20 #define PCLK_GPIO_NUM     22
21
22 #define FLASH_GPIO 4
23
24 // Input resolution
25 #define IN_W 160
26 #define IN_H 120
27
28 // ===== SCALE FACTOR CONFIGURATION =====
29 // Set to 0 for ORIGINAL (1/1), 1 for UPSAMPLING (1.5x = 3/2) or 2
      for DOWNSAMPLING (2/3)
30 #define IMAGE_MODE 0  // 0=Original, 1=Upsample, 2=Downsample
31
32 #if IMAGE_MODE == 0
```

```
33    // Original: 1/1 -> scale_num=1, scale_den=1
34    #define SCALE_NUM 1
35    #define SCALE_DEN 1
36  #elif IMAGE_MODE == 1
37    // Upsampling: 1.5x -> scale_num=3, scale_den=2
38    #define SCALE_NUM 3
39    #define SCALE_DEN 2
40  #else
41    // Downsampling: 2/3 -> scale_num=2, scale_den=3
42    #define SCALE_NUM 2
43    #define SCALE_DEN 3
44  #endif
45
46  #define OUT_W ((IN_W * SCALE_NUM) / SCALE_DEN)
47  #define OUT_H ((IN_H * SCALE_NUM) / SCALE_DEN)
48
49  static uint16_t resized[OUT_W * OUT_H];
50  static uint8_t *jpeg_buf = NULL;
51  static size_t jpeg_len = 0;
52
53  void setup() {
54    Serial.begin(921600);
55    delay(2000);
56
57    #if IMAGE_MODE == 0
58      Serial.println("ESP32 CAM: ORIGINAL mode (1/1 - no resizing)");
59      Serial.printf("Output: %dx%d\n", OUT_W, OUT_H);
60    #elif IMAGE_MODE == 1
61      Serial.println("ESP32 CAM: UPSAMPLING mode (1.5x = 3/2)");
62      Serial.printf("Input: %dx%d -> Output: %dx%d\n", IN_W, IN_H,
          OUT_W, OUT_H);
63    #else
64      Serial.println("ESP32 CAM: DOWNSAMPLING mode (2/3)");
65      Serial.printf("Input: %dx%d -> Output: %dx%d\n", IN_W, IN_H,
          OUT_W, OUT_H);
66    #endif
67
68    pinMode(FLASH_GPIO, OUTPUT);
69    digitalWrite(FLASH_GPIO, LOW);
70
71    camera_config_t config;
```

```
72    config.ledc_channel = LEDC_CHANNEL_0;
73    config.ledc_timer   = LEDC_TIMER_0;
74    config.pin_d0 = Y2_GPIO_NUM;
75    config.pin_d1 = Y3_GPIO_NUM;
76    config.pin_d2 = Y4_GPIO_NUM;
77    config.pin_d3 = Y5_GPIO_NUM;
78    config.pin_d4 = Y6_GPIO_NUM;
79    config.pin_d5 = Y7_GPIO_NUM;
80    config.pin_d6 = Y8_GPIO_NUM;
81    config.pin_d7 = Y9_GPIO_NUM;
82    config.pin_xclk = XCLK_GPIO_NUM;
83    config.pin_pclk = PCLK_GPIO_NUM;
84    config.pin_vsync = VSYNC_GPIO_NUM;
85    config.pin_href = HREF_GPIO_NUM;
86    config.pin_sscb_sda = SIOD_GPIO_NUM;
87    config.pin_sscb_scl = SIOC_GPIO_NUM;
88    config.pin_pwdn = PWDN_GPIO_NUM;
89    config.pin_reset = RESET_GPIO_NUM;
90
91    config.xclk_freq_hz = 20000000;
92    config.pixel_format = PIXFORMAT_RGB565;
93    config.frame_size   = FRAMESIZE_QQVGA;  // 160x120
94    config.fb_count     = 1;
95
96    esp_camera_init(&config);
97  }
98
99  // Nearest neighbor resize (RGB565 color)
100 // Works for both upsampling and downsampling using integer
      arithmetic
101 // Uses inverse mapping: for each output pixel, find nearest input
      pixel
102 void resize_nearest(uint8_t *in) {
103   uint16_t *in_rgb565 = (uint16_t *)in;
104   for (int y = 0; y < OUT_H; y++) {
105     for (int x = 0; x < OUT_W; x++) {
106       // Inverse mapping: calculate source pixel position
107       int src_y = (y * SCALE_DEN) / SCALE_NUM;
108       int src_x = (x * SCALE_DEN) / SCALE_NUM;
109
110       // Clamp to bounds
```

```
      if (src_y >= IN_H) src_y = IN_H - 1;
      if (src_x >= IN_W) src_x = IN_W - 1;

      // Copy pixel from source to destination
      resized[y * OUT_W + x] =
        in_rgb565[src_y * IN_W + src_x];
    }
  }
}

void loop() {
  digitalWrite(FLASH_GPIO, HIGH);
  delay(20);

  camera_fb_t *fb = esp_camera_fb_get();
  digitalWrite(FLASH_GPIO, LOW);
  if (!fb) return;

  #if IMAGE_MODE == 0
    // Original mode: copy directly without resizing
    uint16_t *in_rgb565 = (uint16_t *)fb->buf;
    for (int i = 0; i < OUT_W * OUT_H; i++) {
      resized[i] = in_rgb565[i];
    }
  #else
    // Upsampling or downsampling: use resize function
    resize_nearest(fb->buf);
  #endif

  // Encode resized image as JPEG
  // Debug: print actual dimensions being encoded
  Serial.printf("Encoding image: %dx%d pixels\n", OUT_W, OUT_H);

  bool ok = fmt2jpg(
      (uint8_t *)resized,
      OUT_W * OUT_H * 2,
      OUT_W,
      OUT_H,
      PIXFORMAT_RGB565,
      80,
      &jpeg_buf,
```

```
152         &jpeg_len
153     );
154 
155     if (ok) {
156       Serial.printf("JPEG encoded: %d bytes\n", jpeg_len);
157       uint8_t sync[2] = {0xAA, 0x55};
158       Serial.write(sync, 2);
159       Serial.write((uint8_t *)&jpeg_len, 4);
160       Serial.write(jpeg_buf, jpeg_len);
161       free(jpeg_buf);
162     } else {
163       Serial.println("JPEG encoding failed!");
164     }
165 
166     esp_camera_fb_return(fb);
167     delay(1000);
168 }
```

C.3 Python Connector Script

Listing 13: Python Receiver Script (`esp32_cam_link/esp32_cam_q3/receive.py`)

```
1  import serial
2  import struct
3  
4  PORT = "COM4"
5  BAUD = 921600
6  
7  # Simple receiver for ESP32 CAM resized images (upsampled or
       downsampled)
8  # Just receives JPEG images and saves them
9  
10 ser = serial.Serial(
11     PORT,
12     BAUD,
13     timeout=5,
14     dsrdtr=False,
15     rtscts=False
16 )
17 
18 ser.setDTR(False)
```

```
ser.setRTS(False)

img_count = 0
print("Waiting for frames from ESP32 CAM...")

while True:
    # Find sync word (0xAA 0x55)
    if ser.read(1) != b'\xAA':
        continue
    if ser.read(1) != b'\x55':
        continue

    # Read image size (4 bytes, little-endian)
    size_bytes = ser.read(4)
    if len(size_bytes) != 4:
        continue

    size = struct.unpack("<I", size_bytes)[0]

    # Read image data
    data = ser.read(size)
    if len(data) != size:
        print("Incomplete frame")
        continue

    img_count += 1
    filename = f"resized.jpg"
    with open(filename, "wb") as f:
        f.write(data)

    print(f"Saved {filename} ({size} bytes) - Frame #{img_count}")
```

References

[1] Espressif Systems. *ESP32-CAM*. Available: https://www.espressif.com/en/products/devkits/esp32-cam/overview

[2] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). *You Only Look Once: Unified, Real-Time Object Detection*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

[3] Ultralytics. *YOLOv8 Documentation*. Available: https://docs.ultralytics.com/

[4] Google. *TensorFlow Lite*. Available: https://www.tensorflow.org/lite

[5] Bradski, G. (2000). *The OpenCV Library*. Dr. Dobb's Journal of Software Tools.