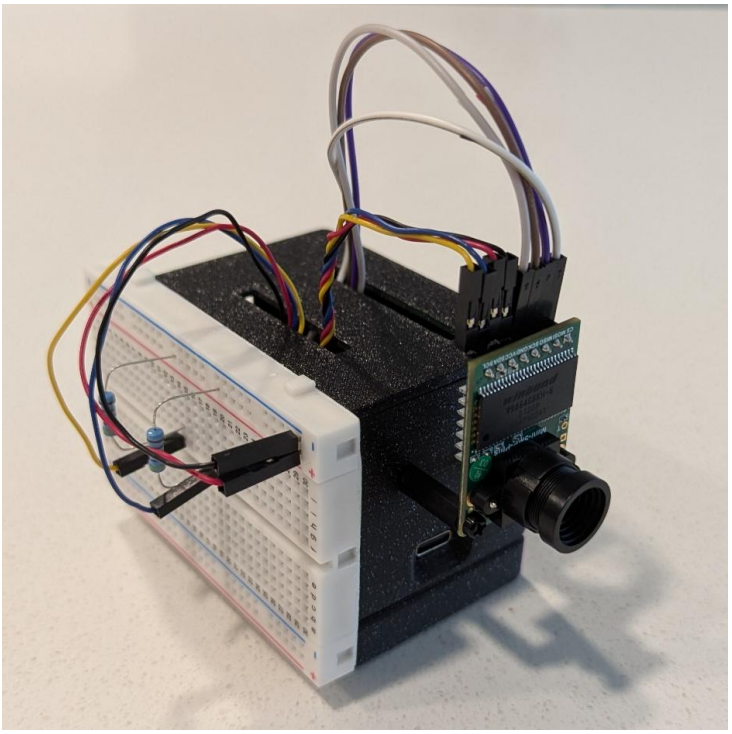
SCALABLE TENSORFLOW MODEL FOR EMBEDDED OBJECT DETECTION

ECEN 5060 Deep Learning Final Project



Barlow, Brady

06 April 2025

Introduction

Object detection is a foundational component of intelligent transportation systems, enabling automated control systems to perceive and respond to dynamic road environments. Advances in machine learning, particularly in TinyML, have made it possible to implement TensorFlow models onto microcontrollers, paving the way for real-time object detection and decision-making in compact, resource-constrained environments.

This project presents the design and deployment of a modular, multi-label road object detector targeting seven critical classes from the COCO dataset: bicycle, car, motorcycle, bus, truck, traffic light, and stop sign. The model is scaled down from a full TensorFlow architecture and converted to TensorFlow Lite Micro for embedded deployment on the SparkFun Thing Plus RP2040. The final quantized model, under 4 MB in size, is integrated with an ArduCAM OV5642 for JPEG image capture and a 128×32 OLED display for real-time output, delivering efficient edge inference without reliance on external compute resources.

Key Contributions

- Developed a compact MobileNetV2-based multi-label classifier tailored for embedded inference.
- Applied per-class threshold tuning and quantization-aware training to preserve accuracy during model compression.
- Delivered an end-to-end TensorFlow to TensorFlow Lite Micro deployment workflow, optimized for low-latency, memory-constrained microcontroller platforms. for microcontroller deployment with a focus on accuracy, latency, memory footprint, and power efficiency.

The remainder of this document is organized as follows:

- **Project Schedule**
- **Dataset Description**
- **Model Architecture and Training Algorithm**
- **Experimental Setup**
- **Results**
- **Summary and Conclusions**
- **References and Appendix**

Project Schedule

Date Range	Milestone	Key Activities
Mar 27 – Apr 3	Proposal and Setup	Finalized project scope, created GitHub repo, reviewed deployment constraints
Apr 4 – Apr 14	Dataset Preparation & Initial Training	Filtered and preprocessed COCO data, ran Stage 1 training
Apr 15 – Apr 22	Model Optimization & Fine-Tuning	Added attention modules, tuned thresholds, trained final model
Apr 23 – Apr 28	Embedded Deployment & Debugging	Exported model, integrated firmware and camera/display drivers
Apr 29 – May 1	Presentation and Demo	Created visuals, ran live demo
May 2 – May 6	Final Report Submission	Compiled report and finalized deliverables

Table 1

Dataset Description

The Common Objects in Context (COCO) 2017 dataset is a large-scale benchmark for object detection, segmentation, and captioning, containing over 118,000 training images and 80 object categories. It features richly annotated scenes with multiple objects per image, varying in size, position, and occlusion. COCO is widely used in academic and industrial research for training and evaluating models in real-world conditions.

The dataset used for this project is a filtered subset of the COCO 2017 dataset, selected to align road-relevant object detection tasks. Seven classes were retained: *bicycle, car, motorcycle, bus, truck, traffic light,* and *stop sign*. These were chosen for their prevalence in real-world driving environments and relevance to autonomous systems. The dataset was loaded via TensorFlow Datasets (TFDS) as shown in Figure 1, then refined through a custom annotation pipeline using Excel spreadsheets to simplify review and correction. To improve class balance, oversampling was applied to underrepresented categories, and image quality filters were enforced to exclude dark, blurry, or skewed samples. Data augmentation played a critical role in improving generalization, introducing variability through brightness shifts, occlusions, geometric transformations, and synthetic mix-ups. The final dataset was resized to 64×64 resolution for training and deployment to balance speed and spatial accuracy. **See Appendix 1**

```
# Road object classes to detect
ROAD_CLASSES = ['bicycle', 'car', 'motorcycle', 'bus', 'truck', 'traffic
light', 'stop sign']
NUM_CLASSES = len(ROAD_CLASSES)

# Create necessary directories
for directory in [EXPORT_DIR, DATA_DIR, EXCEL_DIR, PREDICTION_DIR]:
    os.makedirs(directory, exist_ok=True)

# ——— DATA PREPARATION
------------------------------------------------------------------------
TRAIN_EXCEL_PATH = os.path.join(EXCEL_DIR, 'train.xlsx')
TEST_EXCEL_PATH = os.path.join(EXCEL_DIR, 'test.xlsx')

# Load or create dataset
if not os.path.exists(TRAIN_EXCEL_PATH):
    print("Creating dataset from COCO...")
    dataset, info = tfds.load('coco/2017', split=['train', 'validation'],
shuffle_files=True, with_info=True)
    label_names = info.features['objects']['label'].names
    road_label_ids = [label_names.index(name) for name in ROAD_CLASSES]
```

Figure 1

Model Architecture and Training Algorithm

The model was developed using a custom TensorFlow/Kera's pipeline, built with modularity and deployment efficiency in mind. At its core lies MobileNetV2, selected for its balance of accuracy and computational efficiency, particularly on embedded hardware. To further reduce its memory footprint, the network was configured with an alpha value of 0.35—scaling down the number of filters while retaining essential representational capacity. This adjustment produced a lightweight model architecture well-suited for TensorFlow Lite Micro conversions and RP2040 deployment.

During Stage 1, the base model—pretrained on ImageNet—served as a frozen feature extractor, allowing the upper layers to adapt to the new multi-label classification task. In Stage 2, the final 25 layers of MobileNetV2 were selectively unfrozen to enable fine-tuning with a lower learning rate. The classifier head consists of a global average pooling layer, followed by a dropout layer (rate 0.6), a dense layer with 48 ReLU-activated units, an additional dropout layer (rate 0.4), and a final sigmoid-activated output layer that generates independent class probabilities for multi-label inference.

```python
# ——— MODEL BUILDING ——————————————————————————————————————————

def build_model(trainable_base=False, fine_tuning=False):
    """Build the MobileNetV2-based model"""
    # Load MobileNetV2 with pre-trained weights, using alpha=0.35 for a
smaller model
    base_model = MobileNetV2(
        input_shape=(*IMG_SIZE, 3),
        include_top=False,
        weights='imagenet',
        alpha=0.35  # Lighter model
    )

    # Set base model trainable status
    base_model.trainable = trainable_base

    # If fine-tuning, only make the last few layers trainable
    if fine_tuning:
        for layer in base_model.layers[:-25]:
            layer.trainable = False

    # Build model
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dropout(0.6),  # Prevent overfitting
        layers.Dense(48, activation='relu'),
        layers.Dropout(0.4),  # Additional dropout
        layers.Dense(NUM_CLASSES, activation='sigmoid')
    ])

    # Use in model compilation
    model.compile(
        optimizer=tf.keras.optimizers.Adam(
            INITIAL_LR if not fine_tuning else INITIAL_LR * 0.1,
            clipnorm=1.0
        ),
        loss=FocalLoss(),  # Using our properly serializable FocalLoss
        metrics=[
            tf.keras.metrics.BinaryAccuracy(name='binary_accuracy'),
            tf.keras.metrics.Precision(name='precision'),
            tf.keras.metrics.Recall(name='recall'),
            tf.keras.metrics.AUC(name='auc')
        ]
    )

    return model
```

FIGURE 2

```
# ——— STAGE 1: FROZEN BASE TRAINING ————————————————————————————
print("\n=== Stage 1: Training with Frozen Base Model ===")
stage1_model = build_model(trainable_base=False)

# Callbacks
metrics_callback = MetricsCallback(test_ds, ROAD_CLASSES, df_test)
early_stop = EarlyStopping(
    monitor='val_f1_macro',
    mode='max',
    patience=8,
    restore_best_weights=True,
    verbose=1
)
checkpoint = ModelCheckpoint(
    os.path.join(EXPORT_DIR, 'best_model_stage1.keras'),
    monitor='val_f1_macro',
    mode='max',
    save_best_only=True,
    verbose=1
)
reduce_lr = ReduceLROnPlateau(
    monitor='val_f1_macro',
    mode='max',
    factor=0.5,
    patience=4,
    min_lr=1e-6,
    verbose=1
)

# Train stage 1
history_stage1 = stage1_model.fit(
    train_ds,
    epochs=FROZEN_EPOCHS,
    validation_data=test_ds,
    callbacks=[metrics_callback, early_stop, checkpoint, reduce_lr],
    verbose=1
)
```

FIGURE 3

```
# ——— STAGE 2: FINE TUNING ———————————————————————————————
print("\n=== Stage 2: Fine-tuning Model ===")
# Load best stage 1 model
best_stage1_model = tf.keras.models.load_model(
    os.path.join(EXPORT_DIR, 'best_model_stage1.keras'),
    custom_objects={'FocalLoss': FocalLoss}  # Important: provide custom
objects mapping
)

# Create fine-tuning model
stage2_model = build_model(trainable_base=True, fine_tuning=True)

# Copy weights from stage 1
stage2_model.set_weights(best_stage1_model.get_weights())

# Callbacks for stage 2
metrics_callback_stage2 = MetricsCallback(test_ds, ROAD_CLASSES, df_test)
early_stop_stage2 = EarlyStopping(
    monitor='val_f1_macro',
    mode='max',
    patience=16,
    restore_best_weights=True,
    verbose=1
)
checkpoint_stage2 = ModelCheckpoint(
    os.path.join(EXPORT_DIR, 'best_model_final.keras'),
    monitor='val_f1_macro',
    mode='max',
    save_best_only=True,
    verbose=1
)
reduce_lr_stage2 = ReduceLROnPlateau(
    monitor='val_f1_macro',
    mode='max',
    factor=0.5,
    patience=8,
    min_lr=1e-6,
    verbose=1
)

# Train stage 2
history_stage2 = stage2_model.fit(
    train_ds,
    epochs=FINE_TUNE_EPOCHS,
    validation_data=test_ds,
    callbacks=[metrics_callback_stage2, early_stop_stage2, checkpoint_stage2,
reduce_lr_stage2],
    verbose=1
)
```

FIGURE 4

To address class imbalance and emphasize learning from harder examples, a custom weighted focal loss with γ = 2.5 was employed. This loss function suppressed the contribution of easily classified samples, allowing the model to focus on more challenging cases. Training was conducted using TensorFlow's Adam optimizer with gradient clipping (clip norm = 1.0) to enhance stability during backpropagation. Performance was evaluated using a suite of metrics including binary accuracy, precision, recall, and AUC, providing a comprehensive view of class-wise and overall behavior. In addition, threshold tuning was performed every 5 epochs to optimize the decision boundaries for each class based on F1 score, improving multi-label prediction calibration.

```
# ——— DEFINE CUSTOM LOSS
# ——————————————————————————————————————————————————————————————

# Properly implement focal loss as a subclass for better serialization
@tf.keras.utils.register_keras_serializable(package="custom_losses")
class FocalLoss(tf.keras.losses.Loss):
    """Focal Loss implementation to focus on hard-to-classify examples"""

    def __init__(self, gamma=2.0, alpha=0.25, **kwargs):
        super().__init__(**kwargs)
        self.gamma = gamma
        self.alpha = alpha

    def call(self, y_true, y_pred):
        # Clip prediction values to avoid numerical instability
        epsilon = 1e-7
        y_pred = tf.clip_by_value(y_pred, epsilon, 1 - epsilon)

        # Calculate focal loss
        cross_entropy = -y_true * tf.math.log(y_pred) - (1 - y_true) *
tf.math.log(1 - y_pred)

        # Apply focal weighting
        loss = self.alpha * tf.math.pow(1 - y_pred, self.gamma) *
cross_entropy * y_true + \
               (1 - self.alpha) * tf.math.pow(y_pred, self.gamma) *
cross_entropy * (1 - y_true)

        # Return mean loss
        return tf.reduce_mean(loss)

    def get_config(self):
        config = super().get_config()
        config.update({
            "gamma": self.gamma,
            "alpha": self.alpha,
        })
        return config
```

FIGURE 5

```python
# ——— TRAINING CALLBACK ———————————————————————————————————————
class MetricsCallback(tf.keras.callbacks.Callback):

    def __init__(self, test_ds, classes, df_test):
        super().__init__()
        self.test_ds = test_ds
        self.classes = classes
        self.df_test = df_test
        self.metrics_history = {
            'f1': [], 'precision': [], 'recall': []
        }
        self.thresholds = np.array([0.5] * len(classes))
        # Class weights for threshold optimization
        self.class_weights = np.ones(len(classes))

    def on_epoch_end(self, epoch, logs=None):
        # Get predictions and true labels
        y_true, y_pred = [], []
        for x, y in self.test_ds:
            y_true.append(y.numpy())
            y_pred.append(self.model.predict(x, verbose=0))

        y_true = np.vstack(y_true)
        y_pred = np.vstack(y_pred)

        # Calculate class distribution to identify rare classes
        class_counts = np.sum(y_true, axis=0)
        total_positives = np.sum(class_counts)

        # Update class weights based on distribution
        if total_positives > 0:
            self.class_weights = np.ones(len(self.classes))
            for i, count in enumerate(class_counts):
                if count > 0:
                    # Inversely weight classes by their frequency
                    self.class_weights[i] = total_positives / (count *
len(self.classes))
                    # Cap at 5.0 to prevent extreme values
                    self.class_weights[i] = min(5.0, self.class_weights[i])

        # Optimize thresholds every 5 epochs
        if (epoch + 1) % 5 == 0:
            for i, class_name in enumerate(self.classes):
                best_f1 = 0
                best_thresh = 0.5

                # Use more thresholds for rare classes, fewer for common ones
                if class_counts[i] < 5:
                    # For very rare classes, use more granular thresholds
                    thresholds = np.linspace(0.05, 0.7, 40)
                else:
                    # For common classes, use fewer threshold points
                    thresholds = np.linspace(0.1, 0.8, 20)

                for t in thresholds:
                    preds_bin = (y_pred[:, i] > t).astype(int)
```

```
                    # For rare classes, weight recall higher than precision
                    if class_counts[i] < 5:
                        precision = precision_score(y_true[:, i], preds_bin,
zero_division=0)
                        recall = recall_score(y_true[:, i], preds_bin,
zero_division=0)
                        # Use F2 score to emphasize recall for rare classes
                        if precision > 0 and recall > 0:
                            f_score = (5 * precision * recall) / (4 *
precision + recall)
                        else:
                            f_score = 0
                    else:
                        # Use regular F1 for common classes
                        f_score = f1_score(y_true[:, i], preds_bin,
zero_division=0)

                    if f_score > best_f1:
                        best_f1 = f_score
                        best_thresh = t

                self.thresholds[i] = best_thresh

        # Apply thresholds
        y_pred_bin = (y_pred > self.thresholds).astype(int)

        # Calculate metrics
        f1 = f1_score(y_true, y_pred_bin, average='macro', zero_division=0)
        precision = precision_score(y_true, y_pred_bin, average='macro',
zero_division=0)
        recall = recall_score(y_true, y_pred_bin, average='macro',
zero_division=0)

        # Calculate class-specific F1 scores for tracking
        class_f1 = {}
        for i, class_name in enumerate(self.classes):
            class_f1[class_name] = f1_score(y_true[:, i], y_pred_bin[:, i],
zero_division=0)

        # Store metrics
        self.metrics_history['f1'].append(f1)
        self.metrics_history['precision'].append(precision)
        self.metrics_history['recall'].append(recall)

        # Add to logs for other callbacks
        logs = logs or {}
        logs['val_f1_macro'] = f1

        # Print progress
        print(f"Epoch {epoch + 1}: "
              f"F1={f1:.4f}, Precision={precision:.4f}, Recall={recall:.4f},
"
              f"Thresholds: {np.mean(self.thresholds):.2f}")
        print(f"Class F1 scores: {', '.join([f'{c}={v:.2f}' for c, v in
class_f1.items()])}")
```

```
        # Visualize predictions every 10 epochs
        if (epoch + 1) % 10 == 0 or epoch == 0:
            visualize_predictions(self.model, self.df_test, self.classes,
self.thresholds)

    def on_train_end(self, logs=None):
        # Plot metrics history
        plt.figure(figsize=(10, 6))
        for metric_name, values in self.metrics_history.items():
            plt.plot(range(1, len(values) + 1), values, marker='o',
label=metric_name)

        plt.title('Model Metrics Over Epochs')
        plt.xlabel('Epoch')
        plt.ylabel('Score')
        plt.legend()
        plt.grid(True)
        plt.savefig(os.path.join(EXPORT_DIR, 'metrics_history.png'))
        plt.close()

        # Save thresholds
        np.save(os.path.join(EXPORT_DIR, 'best_thresholds.npy'),
self.thresholds)
        print(f"Saved optimized thresholds to {os.path.join(EXPORT_DIR,
'best_thresholds.npy')}")

        # Generate visual predictions on a small subset (for visualization
only)
        visualize_predictions(self.model, self.df_test, self.classes,
self.thresholds)

        # IMPORTANT: Evaluate on the FULL test dataset, not just a few
samples
        # Get predictions for ALL test data
        y_true, y_pred = [], []
        for x, y in self.test_ds:
            y_true.append(y.numpy())
            y_pred.append(self.model.predict(x, verbose=0))

        y_true = np.vstack(y_true)
        y_pred = np.vstack(y_pred)
        y_pred_bin = (y_pred > self.thresholds).astype(int)

        # Save classification report based on full test set
        report = classification_report(y_true, y_pred_bin,
target_names=self.classes)
        with open(os.path.join(EXPORT_DIR, 'classification_report.txt'), 'w')
as f:
            f.write(report)

        print("\nFinal Classification Report (on full test dataset):")
        print(report)
```

FIGURE 6

Experimental Setup

The model was trained and evaluated using a carefully structured experimental pipeline designed for deployment efficiency and robustness. Data was split into training and testing sets using stratified sampling from a filtered subset of the COCO 2017 dataset. Each image was labeled with a binary vector indicating the presence of seven target road object classes. An Excel-based workflow was used to curate, balance, and review the dataset, enabling efficient annotation and class filtering.

Training was conducted using TensorFlow 2.x with the Keras API, leveraging mixed precision to accelerate performance on an NVIDIA RTX 4090 GPU under WSL 2 (Ubuntu). The batch size was set to 32 to optimize GPU usage while maintaining stable gradients. An initial learning rate of 1e-3 was used and adjusted via a ReduceLROnPlateau scheduler during training. The model was compiled using the Adam optimizer with gradient clipping (clipnorm=1.0) for training stability.

To mitigate overfitting, early stopping was applied based on validation F1 score, and dropout layers (0.6 and 0.4) were incorporated into the classifier head. A wide range of data augmentation techniques—brightness and contrast adjustment, Gaussian noise, horizontal/vertical flipping, 90° rotation, random cropping, and mix-up—were used to improve generalization and address class imbalance. A custom build_dataset loader handled file paths, label parsing, augmentation, batching, and prefetching with TensorFlow's tf.data API.

Model performance was evaluated using binary accuracy, macro-averaged precision, recall, AUC, and F1 score. Every 5 epochs, per-class sigmoid thresholds were tuned based on validation performance. After training, temperature scaling was applied to calibrate output probabilities and improve reliability.

For deployment, the system was built around the SparkFun Thing Plus RP2040 microcontroller. Input was provided by an ArduCAM OV5642 (5MP JPEG), and class predictions were displayed on a 128×32 SSD1306 OLED screen via I^2C. A PCA9546 I^2C multiplexer managed peripheral communication see FIGURE 7. The quantized model was converted to a .tflite file and embedded as a C array (model_data.h). JPEG decoding was performed using picojpeg, and inference results were displayed in real time.

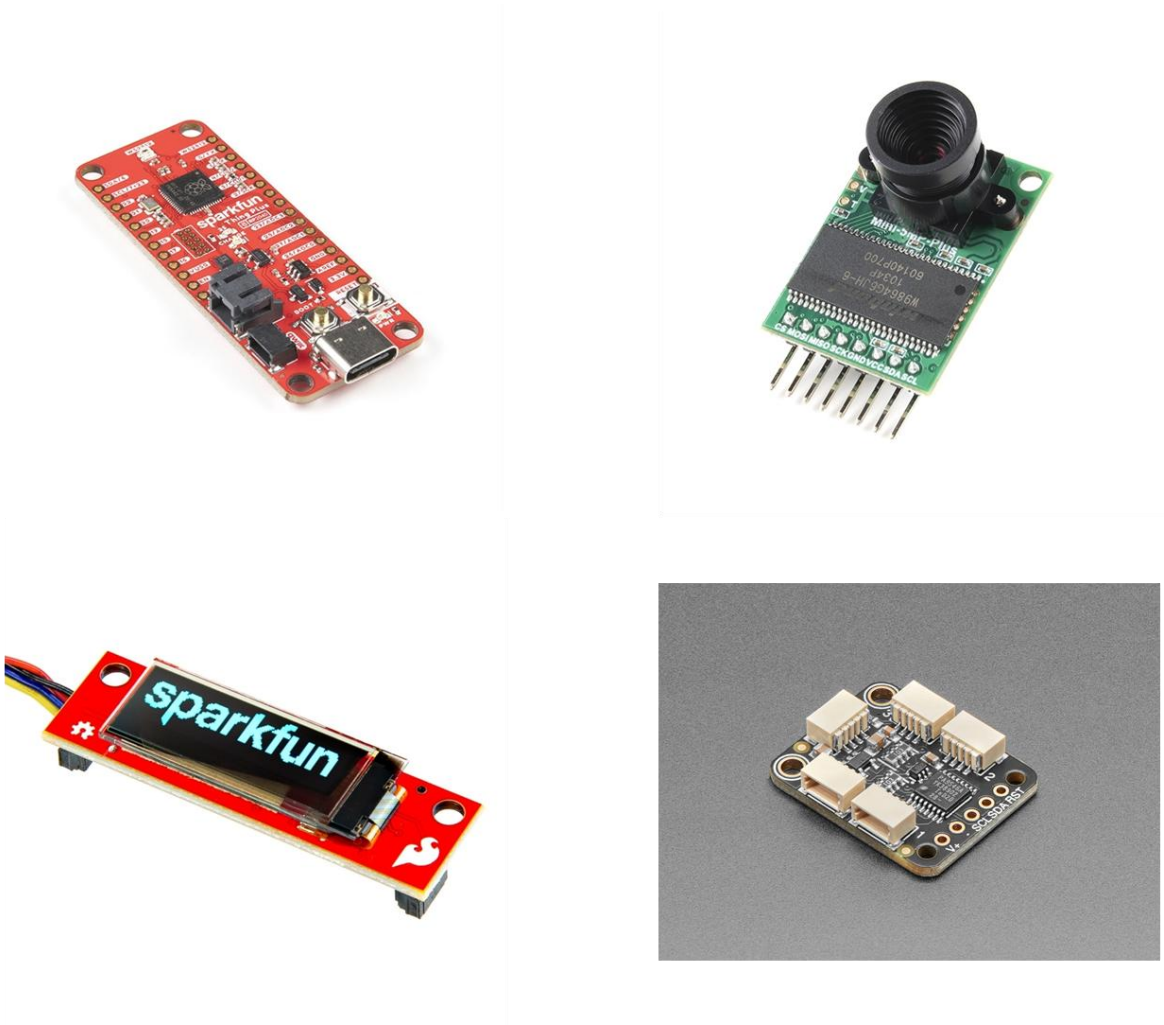


FIGURE 6

Power was supplied by a 1500 mAh LiPo battery, supporting portable operation. The embedded hardware was enclosed in a custom 3D-printed case that underwent multiple iterations for optimal fit, and accessibility.

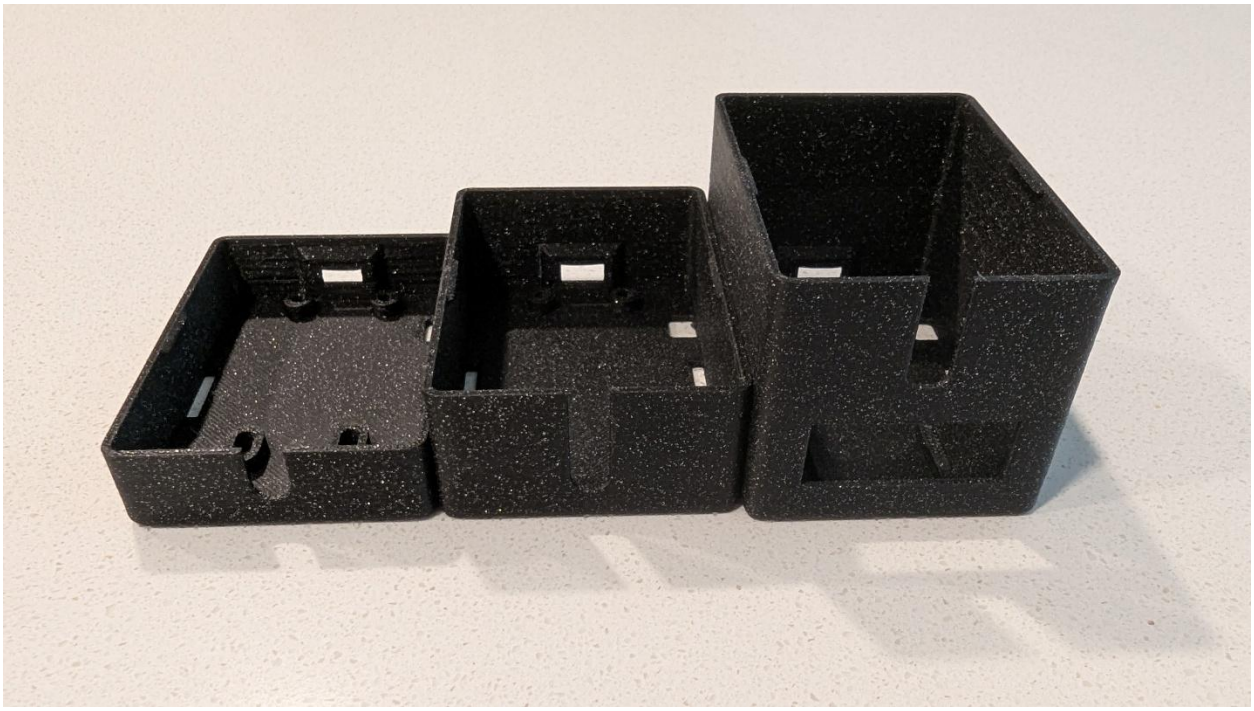


FIGURE 7

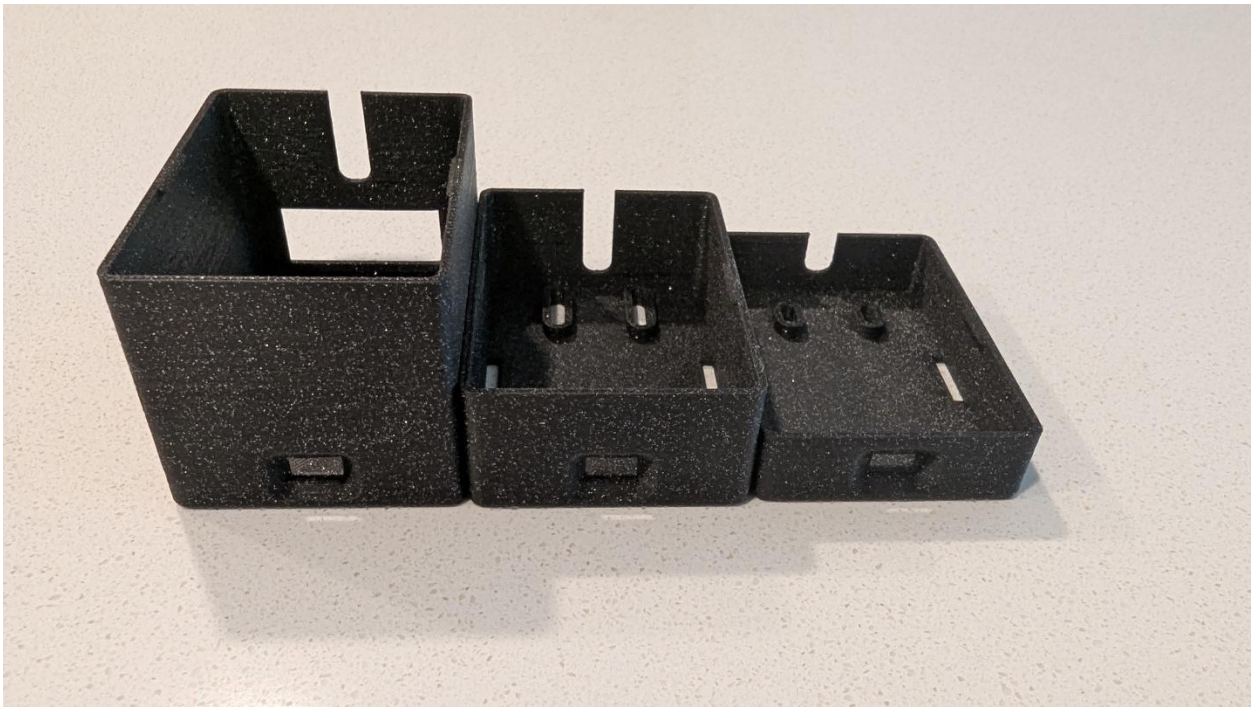
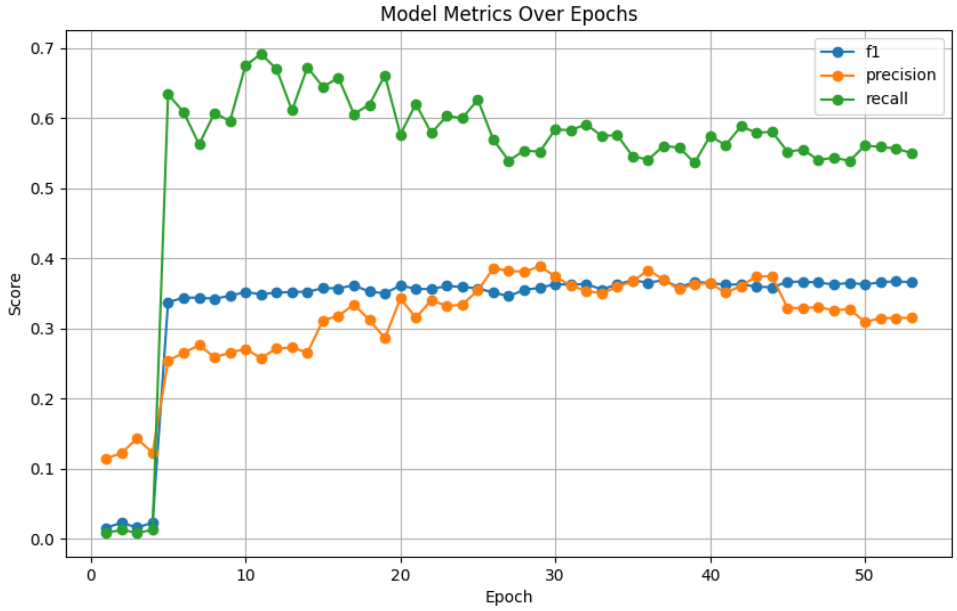


FIGURE 8

Results

The model was trained on 11,321 images and evaluated on 1,760 test samples using a two-stage training strategy. Stage 1 consisted of 16 epochs with the MobileNetV2 base frozen, followed by Stage 2 with 256 fine-tuning epochs on the final 25 layers. Input images were resized to 64×64×3, and training was conducted with a batch size of 32. Post-training, per-class detection thresholds were calibrated to balance sensitivity and precision. These thresholds—ranging from 0.21 to 0.32—were selected to minimize false positives without significantly harming recall



	precision	recall	f1-score	support
bicycle	0.12	0.71	0.2	149
car	0.49	0.72	0.58	534
motorcycle	0.34	0.42	0.37	159
bus	0.47	0.35	0.4	189
truck	0.23	0.74	0.35	249
traffic light	0.24	0.61	0.34	191
stop sign	0.33	0.29	0.31	68
micro avg	0.28	0.61	0.38	1539
macro avg	0.32	0.55	0.37	1539
weighted avg	0.35	0.61	0.42	1539
samples avg	0.15	0.31	0.19	1539

Table 2

FIGURE 10

The quantized .tflite model, approximately 669 KB in size, was evaluated using the Test.py script to simulate real-world deployment conditions and assess post-quantization performance. This evaluation produced a macro-averaged F1 score of 0.366, with a precision of 0.315 and a recall of 0.550. Among the target classes, the highest F1 score was achieved for "car" (0.58), followed by "motorcycle," "bus," and "truck." Lower performance on "bicycle" and "stop sign" was primarily due to their smaller object sizes and limited presence in the training data.

As expected, quantization introduced a slight drop in overall performance; however, results remained within an acceptable margin for edge deployment. The model demonstrated higher recall than precision, indicating a conservative prediction strategy—consistent with the tuned thresholds designed to reduce false positives. These outcomes confirm the model's generalization capability and suitability for deployment in resource-constrained environments.

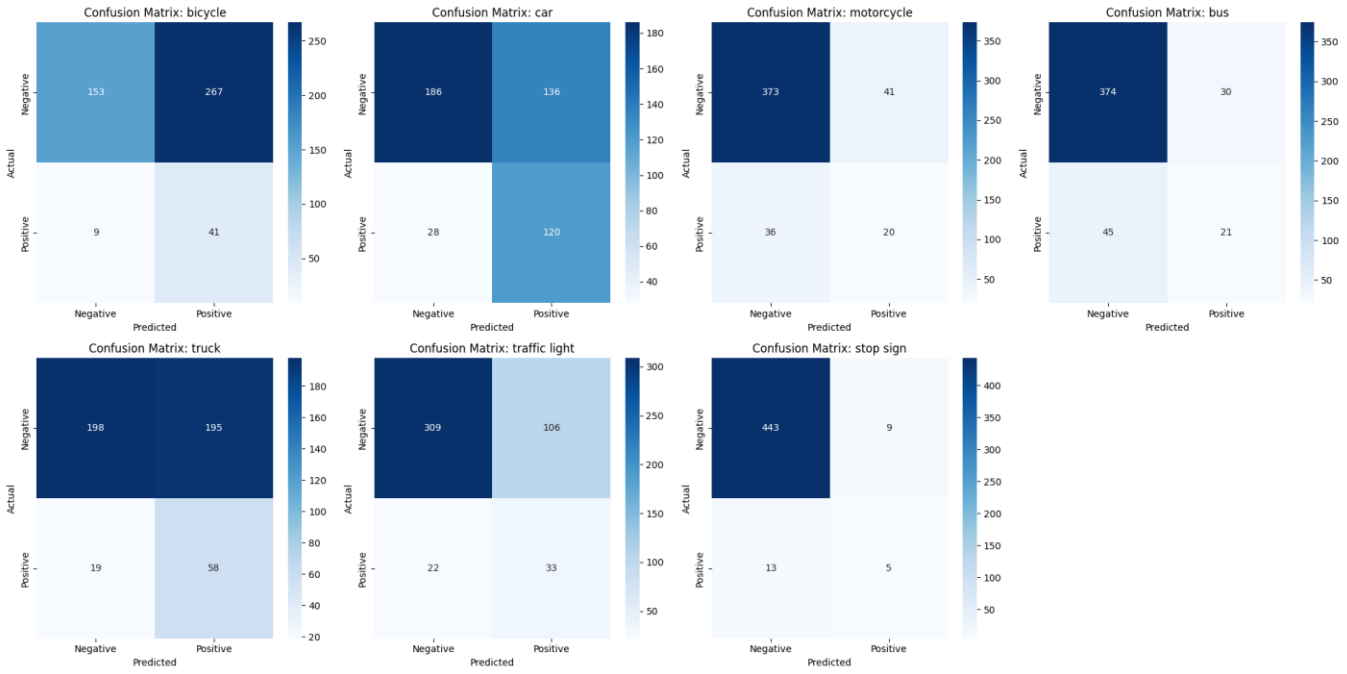


FIGURE 11

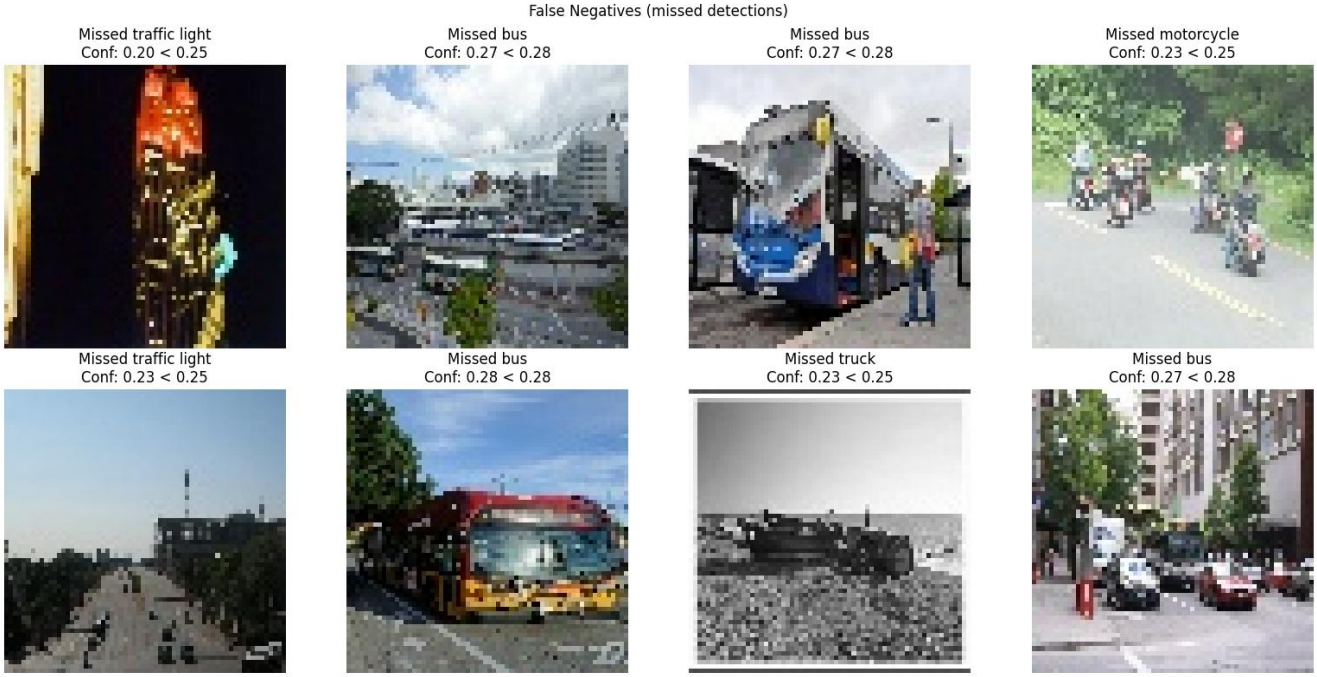


FIGURE 12

FIGURE 13

FIGURE 14

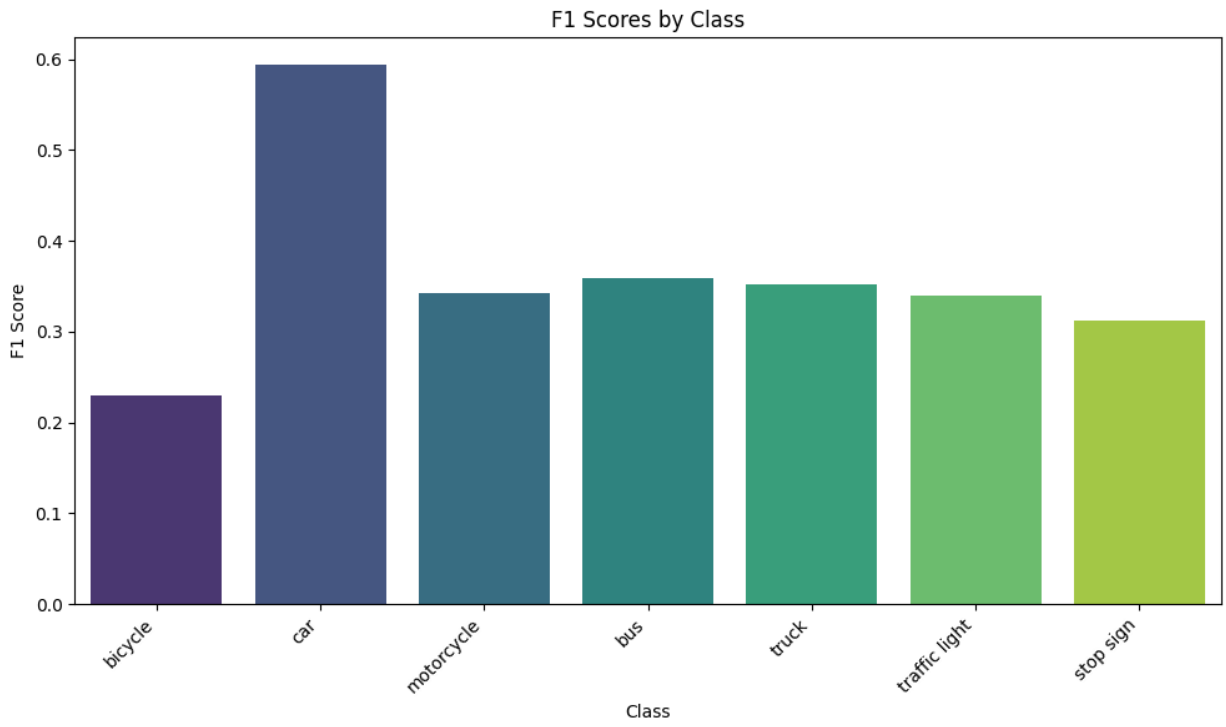


FIGURE 15

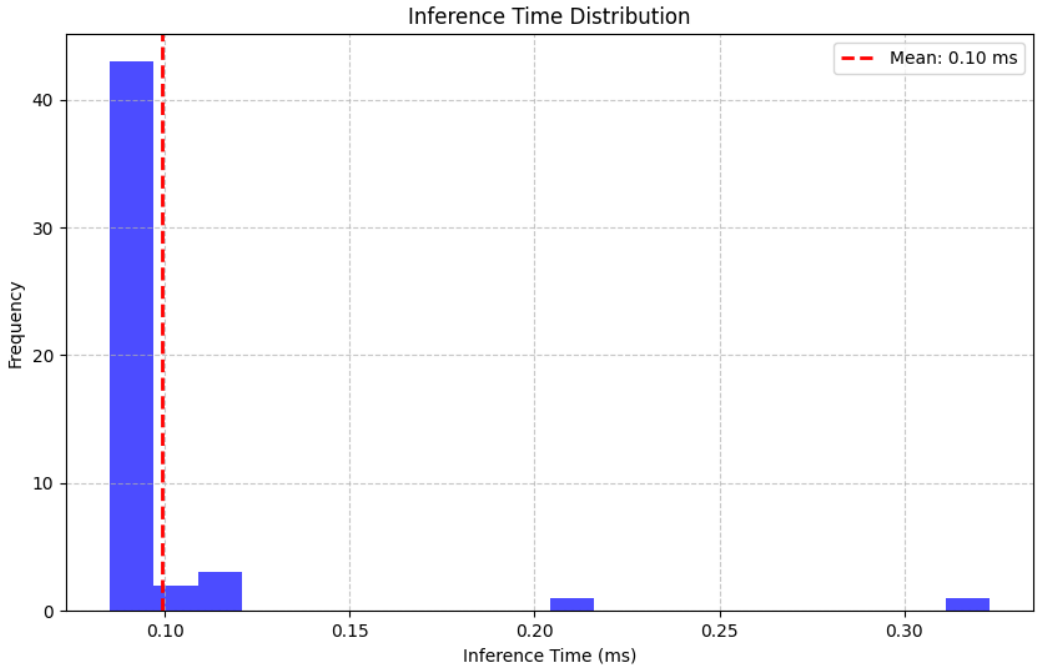


FIGURE 16

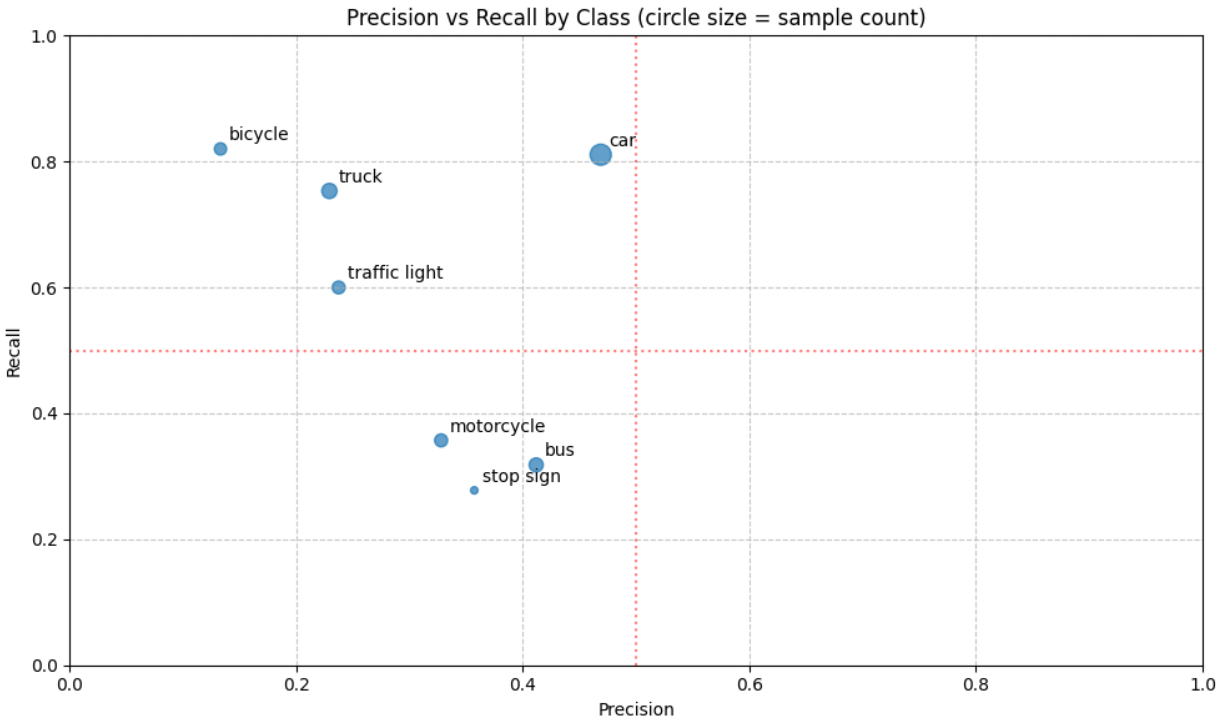


FIGURE 17

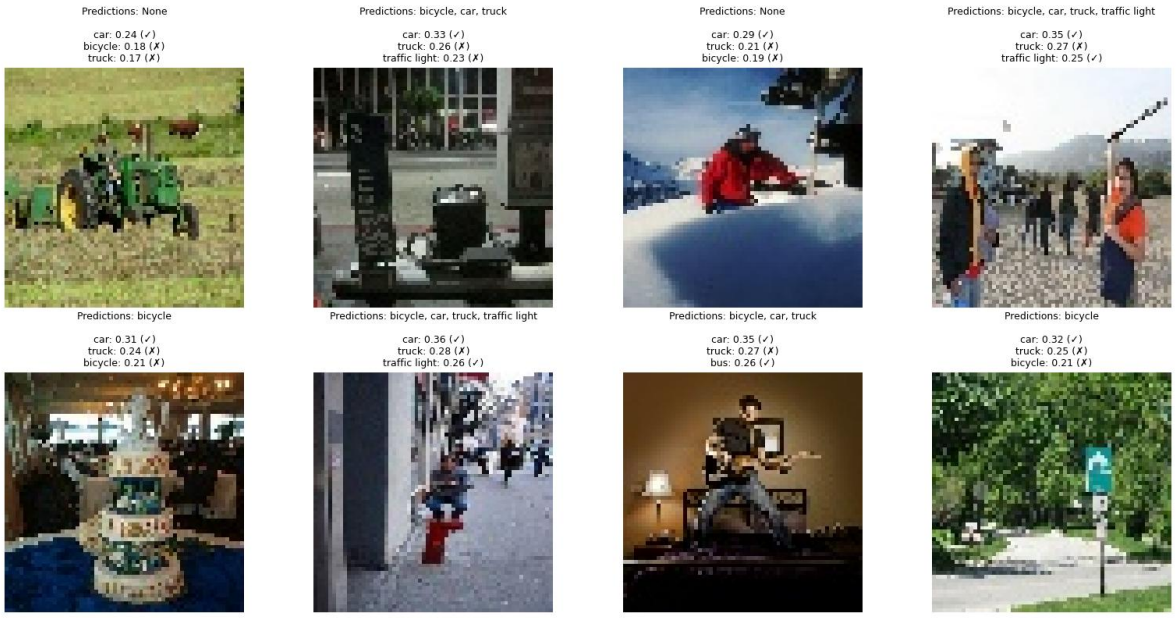


FIGURE 18

Below is a capture of the microcontroller's serial log, along with figures showing its performance while running inference on a computer screen image. During these tests, the model achieved an inference speed of approximately 690 ms per frame. The system operated continuously for 6.5 hours on a 1500 mAh battery, corresponding to an average current draw of roughly 230 mA during inference.

Performance
Starting image capture...
Capture completed successfully in 161 ms
FIFO length: 1640 bytes
First 16 bytes: FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 00
Core 0: Processing image...
Core 1: Inference took 690 ms
--------DETECTION RESULTS--------
Class: BI | Score: 17.97% | Not detected
Class: C | Score: 25.39% | Not detected
Class: M | Score: 13.67% | Not detected
Class: B | Score: 12.11% | Not detected
Class: T | Score: 20.70% | Not detected
Class: TL | Score: 15.62% | Not detected
Class: SS | Score: 19.92% | Not detected
Best detection: C (25%)
Inference time: 690 ms

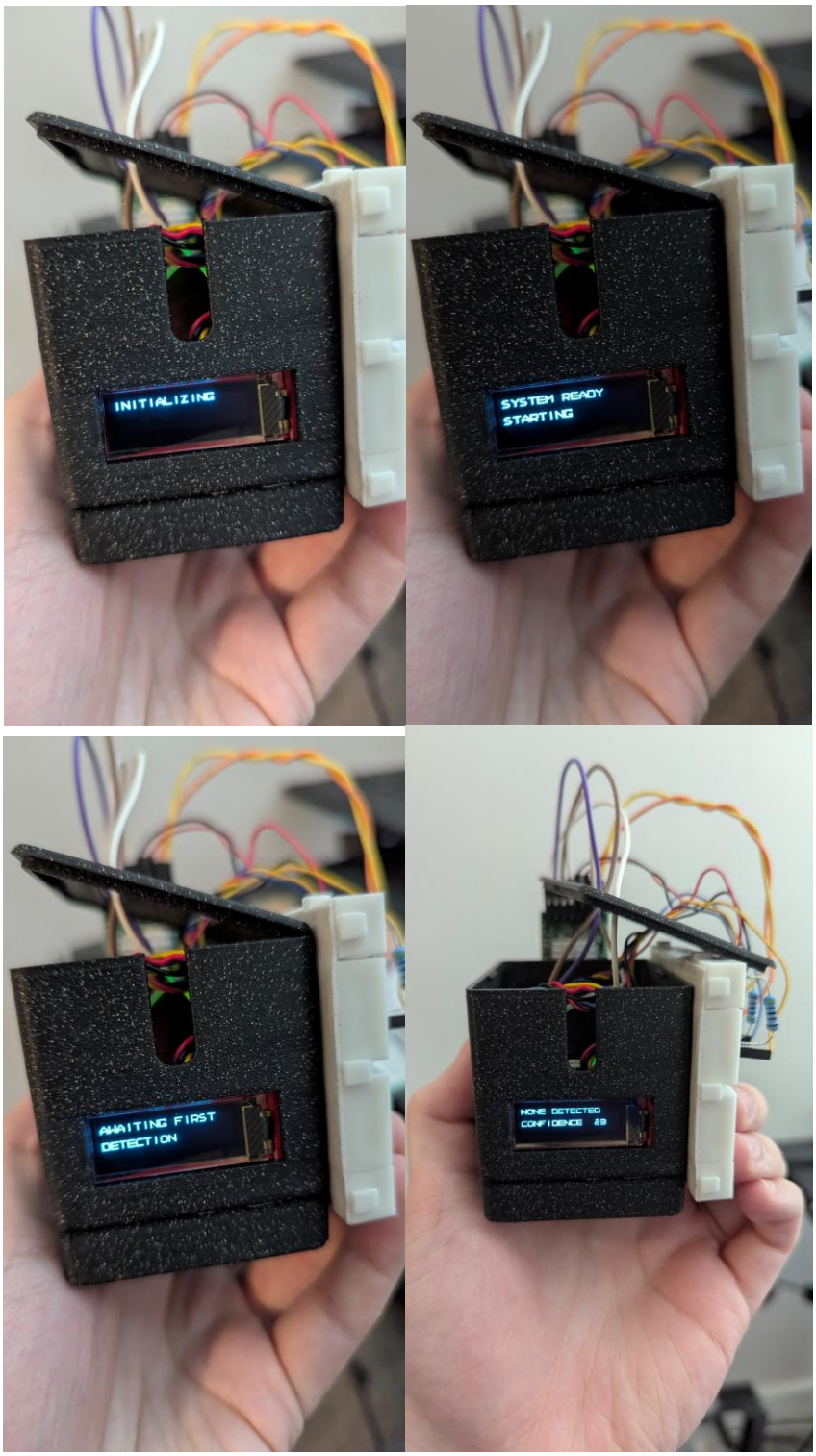

INITIALIZING
SYSTEM READY
STARTING
AWAITING FIRST
DETECTION
NONE DETECTED
CONFIDENCE 23

FIGURE 19

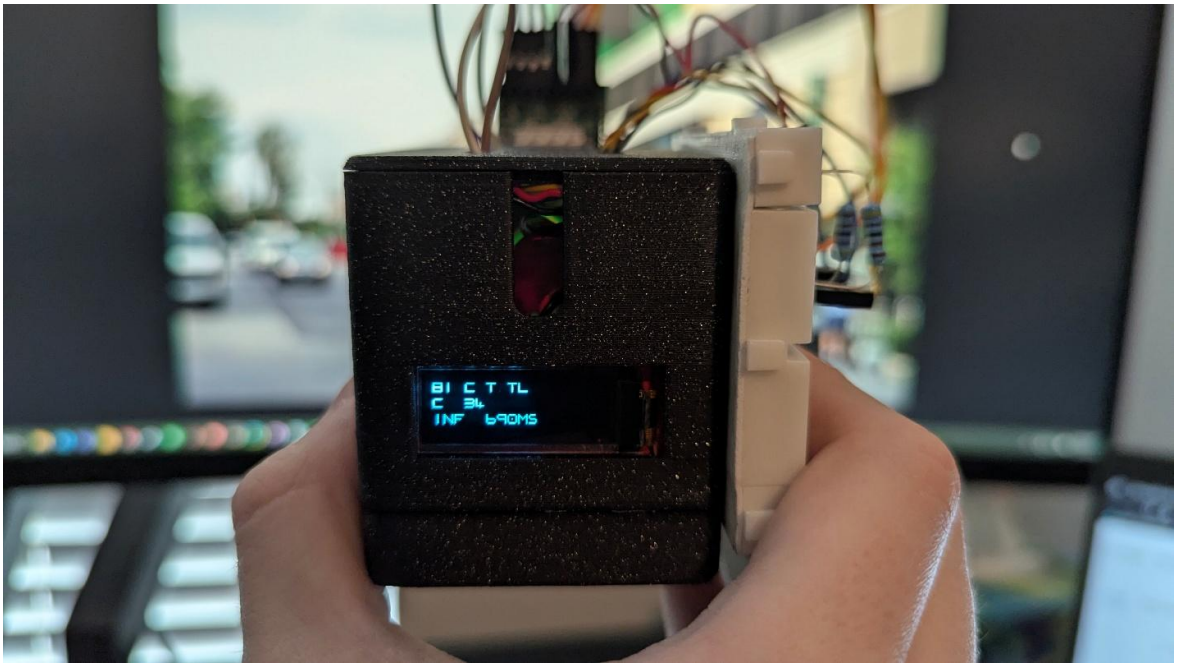


FIGURE 20

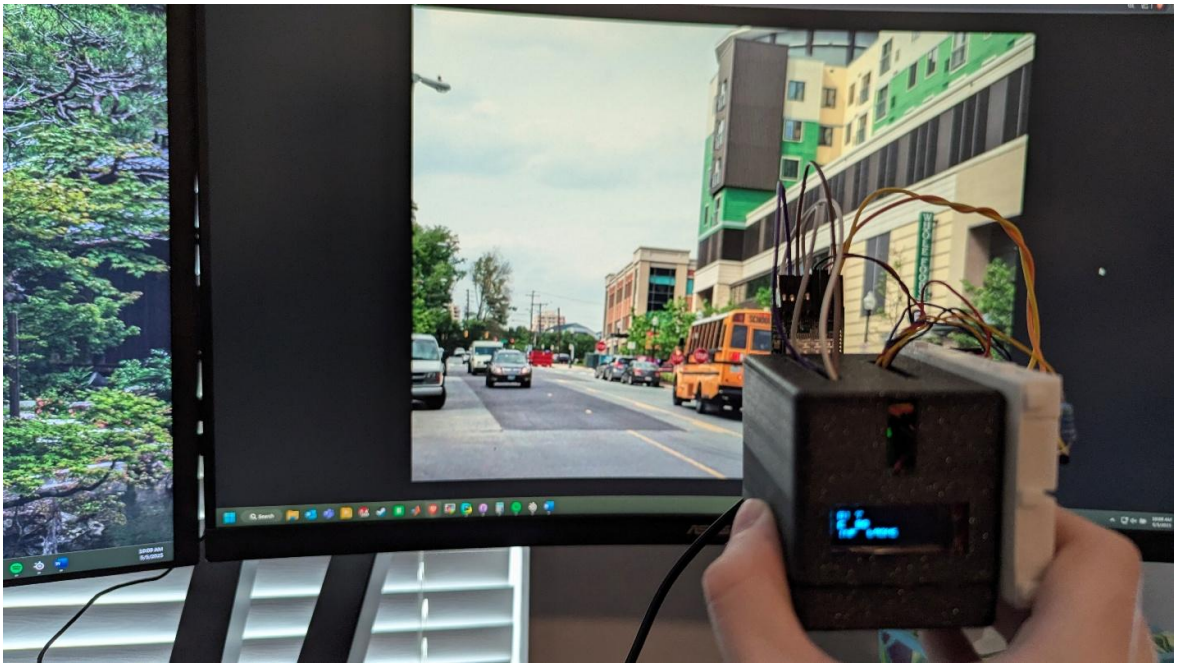


FIGURE 21

Summary and Conclusions

This project successfully developed and deployed a lightweight, multi-label road object detector optimized for microcontroller-based inference. The model was trained on 11,321 images and validated on 1,760 samples using a two-stage training approach: an initial 16-epoch phase with a frozen base, followed by 256 epochs of fine-tuning. With a compact input resolution of 64×64×3 and a batch size of 32, the network was carefully scaled for deployment on the RP2040 platform.

To enhance performance under class imbalance, post-training threshold optimization was applied, resulting in class-specific detection thresholds ranging from 0.21 to 0.32. The final quantized model achieved a macro-averaged F1 score of 0.366, precision of 0.315, and recall of 0.550. The highest performance was observed on the "car" class (F1 = 0.58), while "bicycle" and "stop sign" proved more challenging due to smaller object size and limited representation in the dataset.

Deployment testing on the SparkFun Thing Plus RP2040 demonstrated the full inference pipeline in action—from JPEG image capture with the ArduCAM to class visualization on a 128×32 OLED display. Image capture took approximately 161 ms, with inference completing in 690 ms per frame, confirming the system’s suitability for low-latency, edge-based applications. The complete setup was powered by a 1500 mAh LiPo battery, supporting continuous operation for over 6.5 hours, and housed in a custom 3D-printed enclosure refined across multiple iterations for durability and usability.

REFERENCES

https://github.com/Barlow13/Final-Project-5/tree/main

https://cocodataset.org/#home

https://github.com/tensorflow/tflite-micro

https://github.com/raspberrypi/pico-tflmicro

https://www.sparkfun.com/sparkfun-thing-plus-rp2040.html

https://www.sparkfun.com/arducam-5mp-plus-ov5642-mini-camera-module.html

https://www.sparkfun.com/sparkfun-qwiic-oled-display-0-91-in-128x32-lcd-24606.html

https://www.adafruit.com/product/5664

https://blog.tensorflow.org/2021/09/TinyML-Audio-for-everyone.html

https://ai.google.dev/edge/litert

https://github.com/ArduCAM/PICO_SPI_CAM

https://cocodataset.org/#home

Appendix

Appendix 1: Train.py

```
"""
Road Object Detection Training Script
Optimized for RP2040 Deployment with TensorFlow Lite

This script trains a lightweight MobileNetV2-based model to detect common
road objects
and exports an optimized TFLite model suitable for the SparkFun Thing Plus
RP2040.
"""

import os
import random
import logging
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping,
ReduceLROnPlateau
import tensorflow_datasets as tfds
import cv2
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score, precision_score, recall_score,
classification_report

# ——— SUPPRESS LOGGING
————————————————————————————————————————

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
logging.getLogger('tensorflow').setLevel(logging.ERROR)
logging.getLogger('absl').setLevel(logging.ERROR)

# ——— CONFIGURATION
————————————————————————————————————————

BATCH_SIZE = 32
FROZEN_EPOCHS = 16
FINE_TUNE_EPOCHS = 256
INITIAL_LR = 1e-3
IMG_HEIGHT, IMG_WIDTH = 64, 64
IMG_SIZE = (IMG_HEIGHT, IMG_WIDTH)
SEED = 42
NICKNAME = 'RoadLiteMobileNetV2'
EXPORT_DIR = './export'
DATA_DIR = './Data'
```

```
EXCEL_DIR = './excel'
PREDICTION_DIR = './predictions'

# Set seeds for reproducibility
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

# Road object classes to detect
ROAD_CLASSES = ['bicycle', 'car', 'motorcycle', 'bus', 'truck', 'traffic
light', 'stop sign']
NUM_CLASSES = len(ROAD_CLASSES)

# Create necessary directories
for directory in [EXPORT_DIR, DATA_DIR, EXCEL_DIR, PREDICTION_DIR]:
    os.makedirs(directory, exist_ok=True)

# ——— DATA PREPARATION
------------------------------------------------------------

TRAIN_EXCEL_PATH = os.path.join(EXCEL_DIR, 'train.xlsx')
TEST_EXCEL_PATH = os.path.join(EXCEL_DIR, 'test.xlsx')

# Load or create dataset
if not os.path.exists(TRAIN_EXCEL_PATH):
    print("Creating dataset from COCO...")
    dataset, info = tfds.load('coco/2017', split=['train', 'validation'],
shuffle_files=True, with_info=True)
    label_names = info.features['objects']['label'].names
    road_label_ids = [label_names.index(name) for name in ROAD_CLASSES]

    def process_split(split, split_name, max_samples=10000):
        """Extract road objects with better class balance"""
        records = []
        class_counters = {class_name: 0 for class_name in ROAD_CLASSES}
        target_per_class = max_samples // (2 * len(ROAD_CLASSES))  # Target
samples per class

        for i, ex in enumerate(tfds.as_numpy(split)):
            if sum(class_counters.values()) >= max_samples:
                break

            labels = set(ex['objects']['label'].tolist())
            target = [1 if lid in labels else 0 for lid in road_label_ids]

            # Count the classes in this image
            classes_present = [ROAD_CLASSES[j] for j, v in enumerate(target)
if v]

            # Skip if no target classes or if we already have enough of these
```

```
classes
            if not classes_present:
                # Allow some "None" samples but limit them
                if random.random() > 0.2 or sum(class_counters.values()) >
0.7 * max_samples:
                    continue
            else:
                # Check if we already have enough samples of these classes
                if all(class_counters[cls] >= target_per_class for cls in
classes_present):
                    if random.random() > 0.3:  # Still keep some with
probability 0.3
                        continue

                # Update class counters
                for cls in classes_present:
                    class_counters[cls] += 1

            # Basic image quality filtering
            img = ex['image']
            if np.mean(img) < 20 or np.mean(img) > 235:  # Skip very
dark/bright images
                continue

            # Skip images with extreme aspect ratios
            h, w = img.shape[0], img.shape[1]
            if max(h, w) / min(h, w) > 3:
                continue

            # Process and save image
            fname = f"{split_name}_{i:06d}.jpg"
            path = os.path.join(DATA_DIR, fname)
            img = cv2.resize(img, IMG_SIZE)
            cv2.imwrite(path, cv2.cvtColor(img, cv2.COLOR_RGB2BGR))

            records.append({
                'filename': fname,
                'target_bin': ','.join(map(str, target)),
                'target': ','.join(classes_present) or 'None'
            })

            # Print progress periodically
            if len(records) % 500 == 0:
                print(f"Processed {len(records)} records. Class distribution:
{class_counters}")

        print(f"Final class distribution: {class_counters}")
        return pd.DataFrame(records)
```

```
    df_train = process_split(dataset[0], 'train')
    df_test = process_split(dataset[1], 'test')

    # Save datasets
    df_train.to_excel(TRAIN_EXCEL_PATH, index=False)
    df_test.to_excel(TEST_EXCEL_PATH, index=False)

    print(f"Created dataset with {len(df_train)} training and {len(df_test)}
testing samples")
else:
    print("Loading existing dataset...")
    df_train = pd.read_excel(TRAIN_EXCEL_PATH)
    df_test = pd.read_excel(TEST_EXCEL_PATH)
    print(f"Loaded dataset with {len(df_train)} training and {len(df_test)}
testing samples")

# ——— IMAGE AUGMENTATION FUNCTIONS
————————————————————————————————————————

def decode_image(path):
    """Load and normalize an image from path"""
    img = tf.io.read_file(path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, IMG_SIZE)
    return img / 255.0

def augment_image(image):
    """Enhanced augmentation to improve model generalization"""
    # Standard augmentations
    image = tf.image.random_brightness(image, max_delta=0.3)
    image = tf.image.random_contrast(image, lower=0.7, upper=1.3)

    # Add random noise occasionally to improve robustness
    if tf.random.uniform(shape=[], minval=0, maxval=1) < 0.3:
        noise = tf.random.normal(shape=tf.shape(image), mean=0.0,
stddev=0.02)
        image = image + noise

    # More aggressive geometric transformations
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)

    # Random rotation (0, 90, 180, 270 degrees)
    k = tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32)
    image = tf.image.rot90(image, k=k)

    # Random crop and resize
    if tf.random.uniform(shape=[], minval=0, maxval=1) < 0.5:
        scale = tf.random.uniform(shape=[], minval=0.8, maxval=1.0)
        h, w = tf.shape(image)[0], tf.shape(image)[1]
```

```
        new_h = tf.cast(tf.cast(h, tf.float32) * scale, tf.int32)
        new_w = tf.cast(tf.cast(w, tf.float32) * scale, tf.int32)
        offset_h = tf.random.uniform(shape=[], maxval=h - new_h + 1,
dtype=tf.int32)
        offset_w = tf.random.uniform(shape=[], maxval=w - new_w + 1,
dtype=tf.int32)
        image = tf.image.crop_to_bounding_box(image, offset_h, offset_w,
new_h, new_w)
        image = tf.image.resize(image, size=[h, w])

    # Ensure values stay in [0, 1] range
    return tf.clip_by_value(image, 0, 1)

def mixup(images, labels, alpha=0.2):
    """Apply mixup augmentation to a batch of images and labels"""
    batch_size = tf.shape(images)[0]
    indices = tf.random.shuffle(tf.range(batch_size))
    lam = tf.random.uniform(shape=[], minval=0, maxval=alpha)

    mixed_images = lam * images + (1 - lam) * tf.gather(images, indices)
    mixed_labels = lam * labels + (1 - lam) * tf.gather(labels, indices)

    return mixed_images, mixed_labels

# ——— DATASET CREATION
--------------------------------------------------------------------

def build_dataset(df, augment=False, shuffle=True, apply_mixup=False):
    """Build a TensorFlow dataset from DataFrame"""
    paths = [os.path.join(DATA_DIR, f) for f in df['filename']]
    labels = [list(map(int, r.split(','))) for r in df['target_bin']]

    # Create dataset
    ds = tf.data.Dataset.from_tensor_slices((paths, labels))

    # Decode images and convert labels to float
    ds = ds.map(
        lambda x, y: (decode_image(x), tf.cast(y, tf.float32)),
        num_parallel_calls=tf.data.AUTOTUNE
    )

    # Apply augmentation if requested
    if augment:
        ds = ds.map(
            lambda x, y: (augment_image(x), y),
            num_parallel_calls=tf.data.AUTOTUNE
        )

    # Shuffle if requested
    if shuffle:
```

```
        ds = ds.shuffle(1024)

    # Batch dataset
    ds = ds.batch(BATCH_SIZE)

    # Apply mixup if requested
    if apply_mixup:
        ds = ds.map(
            lambda x, y: mixup(x, y, alpha=0.2),
            num_parallel_calls=tf.data.AUTOTUNE
        )

    # Prefetch for performance
    return ds.prefetch(tf.data.AUTOTUNE)

# Create datasets
train_ds = build_dataset(df_train, augment=True, shuffle=True,
apply_mixup=True)
test_ds = build_dataset(df_test, shuffle=False)

# ——— DEFINE CUSTOM LOSS ———————————————————————————————————————
# Properly implement focal loss as a subclass for better serialization
@tf.keras.utils.register_keras_serializable(package="custom_losses")
class FocalLoss(tf.keras.losses.Loss):
    """Focal Loss implementation to focus on hard-to-classify examples"""

    def __init__(self, gamma=2.0, alpha=0.25, **kwargs):
        super().__init__(**kwargs)
        self.gamma = gamma
        self.alpha = alpha

    def call(self, y_true, y_pred):
        # Clip prediction values to avoid numerical instability
        epsilon = 1e-7
        y_pred = tf.clip_by_value(y_pred, epsilon, 1 - epsilon)

        # Calculate focal loss
        cross_entropy = -y_true * tf.math.log(y_pred) - (1 - y_true) *
tf.math.log(1 - y_pred)

        # Apply focal weighting
        loss = self.alpha * tf.math.pow(1 - y_pred, self.gamma) *
cross_entropy * y_true + \
               (1 - self.alpha) * tf.math.pow(y_pred, self.gamma) *
cross_entropy * (1 - y_true)

        # Return mean loss
        return tf.reduce_mean(loss)
```

```
    def get_config(self):
        config = super().get_config()
        config.update({
            "gamma": self.gamma,
            "alpha": self.alpha,
        })
        return config

# ——— MODEL BUILDING ———————————————————————————————————————
def build_model(trainable_base=False, fine_tuning=False):
    """Build the MobileNetV2-based model"""
    # Load MobileNetV2 with pre-trained weights, using alpha=0.35 for a
smaller model
    base_model = MobileNetV2(
        input_shape=(*IMG_SIZE, 3),
        include_top=False,
        weights='imagenet',
        alpha=0.35  # Lighter model
    )

    # Set base model trainable status
    base_model.trainable = trainable_base

    # If fine-tuning, only make the last few layers trainable
    if fine_tuning:
        for layer in base_model.layers[:-25]:
            layer.trainable = False

    # Build model
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dropout(0.6),  # Prevent overfitting
        layers.Dense(48, activation='relu'),
        layers.Dropout(0.4),  # Additional dropout
        layers.Dense(NUM_CLASSES, activation='sigmoid')
    ])

    # Use in model compilation
    model.compile(
        optimizer=tf.keras.optimizers.Adam(
            INITIAL_LR if not fine_tuning else INITIAL_LR * 0.1,
            clipnorm=1.0
        ),
        loss=FocalLoss(),  # Using our properly serializable FocalLoss
        metrics=[
            tf.keras.metrics.BinaryAccuracy(name='binary_accuracy'),
            tf.keras.metrics.Precision(name='precision'),
```

```
            tf.keras.metrics.Recall(name='recall'),
            tf.keras.metrics.AUC(name='auc')
        ]
    )

    return model

# ——— VISUALIZATION UTILITIES
————————————————————————————————————————————————————

def visualize_predictions(model, df, classes, thresholds=None,
num_samples=8):
    """Visualize model predictions on random samples"""
    if thresholds is None:
        thresholds = [0.5] * len(classes)

    samples = df.sample(num_samples)
    imgs = []
    trues = []

    # Prepare images and true labels
    for _, row in samples.iterrows():
        img_path = os.path.join(DATA_DIR, row['filename'])
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, IMG_SIZE)
        imgs.append(img / 255.0)
        trues.append(list(map(int, row['target_bin'].split(','))))

    # Convert to arrays
    imgs = np.array(imgs)
    trues = np.array(trues)

    # Get predictions
    preds = model.predict(imgs, verbose=0)
    preds_bin = (preds > thresholds).astype(int)

    # Create visualization grid
    fig, axes = plt.subplots(2, 4, figsize=(16, 8))
    for i, ax in enumerate(axes.flatten()):
        img = (imgs[i] * 255).astype(np.uint8)
        true_labels = [classes[j] for j, v in enumerate(trues[i]) if v]
        pred_labels = [classes[j] for j, v in enumerate(preds_bin[i]) if v]

        # Top 3 predictions with probabilities
        top_indices = np.argsort(preds[i])[::-1][:3]
        top_preds = [(classes[j], preds[i][j]) for j in top_indices]

        # Display image
        ax.imshow(img)
```

```
        ax.axis('off')
        ax.set_title(
            f"True: {', '.join(true_labels) or 'None'}\n"
            f"Pred: {', '.join(pred_labels) or 'None'}\n"
            f"Top-3: {', '.join([f'{k}({v:.2f})' for k, v in top_preds])}",
            fontsize=8
        )

    plt.tight_layout()
    plt.savefig(os.path.join(PREDICTION_DIR, f"prediction_grid.png"))
    plt.close()
    return preds, trues

# ——— TRAINING CALLBACK
# ——————————————————————————————————————————————————————————————
class MetricsCallback(tf.keras.callbacks.Callback):

    def __init__(self, test_ds, classes, df_test):
        super().__init__()
        self.test_ds = test_ds
        self.classes = classes
        self.df_test = df_test
        self.metrics_history = {
            'f1': [], 'precision': [], 'recall': []
        }
        self.thresholds = np.array([0.5] * len(classes))
        # Class weights for threshold optimization
        self.class_weights = np.ones(len(classes))

    def on_epoch_end(self, epoch, logs=None):
        # Get predictions and true labels
        y_true, y_pred = [], []
        for x, y in self.test_ds:
            y_true.append(y.numpy())
            y_pred.append(self.model.predict(x, verbose=0))

        y_true = np.vstack(y_true)
        y_pred = np.vstack(y_pred)

        # Calculate class distribution to identify rare classes
        class_counts = np.sum(y_true, axis=0)
        total_positives = np.sum(class_counts)

        # Update class weights based on distribution
        if total_positives > 0:
            self.class_weights = np.ones(len(self.classes))
            for i, count in enumerate(class_counts):
                if count > 0:
                    # Inversely weight classes by their frequency
```

```
                    self.class_weights[i] = total_positives / (count *
len(self.classes))
                    # Cap at 5.0 to prevent extreme values
                    self.class_weights[i] = min(5.0, self.class_weights[i])

        # Optimize thresholds every 5 epochs
        if (epoch + 1) % 5 == 0:
            for i, class_name in enumerate(self.classes):
                best_f1 = 0
                best_thresh = 0.5

                # Use more thresholds for rare classes, fewer for common ones
                if class_counts[i] < 5:
                    # For very rare classes, use more granular thresholds
                    thresholds = np.linspace(0.05, 0.7, 40)
                else:
                    # For common classes, use fewer threshold points
                    thresholds = np.linspace(0.1, 0.8, 20)

                for t in thresholds:
                    preds_bin = (y_pred[:, i] > t).astype(int)

                    # For rare classes, weight recall higher than precision
                    if class_counts[i] < 5:
                        precision = precision_score(y_true[:, i], preds_bin,
zero_division=0)
                        recall = recall_score(y_true[:, i], preds_bin,
zero_division=0)
                        # Use F2 score to emphasize recall for rare classes
                        if precision > 0 and recall > 0:
                            f_score = (5 * precision * recall) / (4 *
precision + recall)
                        else:
                            f_score = 0
                    else:
                        # Use regular F1 for common classes
                        f_score = f1_score(y_true[:, i], preds_bin,
zero_division=0)

                    if f_score > best_f1:
                        best_f1 = f_score
                        best_thresh = t

                self.thresholds[i] = best_thresh

        # Apply thresholds
        y_pred_bin = (y_pred > self.thresholds).astype(int)

        # Calculate metrics
```

```
        f1 = f1_score(y_true, y_pred_bin, average='macro', zero_division=0)
        precision = precision_score(y_true, y_pred_bin, average='macro',
zero_division=0)
        recall = recall_score(y_true, y_pred_bin, average='macro',
zero_division=0)

        # Calculate class-specific F1 scores for tracking
        class_f1 = {}
        for i, class_name in enumerate(self.classes):
            class_f1[class_name] = f1_score(y_true[:, i], y_pred_bin[:, i],
zero_division=0)

        # Store metrics
        self.metrics_history['f1'].append(f1)
        self.metrics_history['precision'].append(precision)
        self.metrics_history['recall'].append(recall)

        # Add to logs for other callbacks
        logs = logs or {}
        logs['val_f1_macro'] = f1

        # Print progress
        print(f"Epoch {epoch + 1}: "
              f"F1={f1:.4f}, Precision={precision:.4f}, Recall={recall:.4f},
"
              f"Thresholds: {np.mean(self.thresholds):.2f}")
        print(f"Class F1 scores: {', '.join([f'{c}={v:.2f}' for c, v in
class_f1.items()])}")

        # Visualize predictions every 10 epochs
        if (epoch + 1) % 10 == 0 or epoch == 0:
            visualize_predictions(self.model, self.df_test, self.classes,
self.thresholds)

    def on_train_end(self, logs=None):
        # Plot metrics history
        plt.figure(figsize=(10, 6))
        for metric_name, values in self.metrics_history.items():
            plt.plot(range(1, len(values) + 1), values, marker='o',
label=metric_name)

        plt.title('Model Metrics Over Epochs')
        plt.xlabel('Epoch')
        plt.ylabel('Score')
        plt.legend()
        plt.grid(True)
        plt.savefig(os.path.join(EXPORT_DIR, 'metrics_history.png'))
        plt.close()
```

```
        # Save thresholds
        np.save(os.path.join(EXPORT_DIR, 'best_thresholds.npy'),
self.thresholds)
        print(f"Saved optimized thresholds to {os.path.join(EXPORT_DIR,
'best_thresholds.npy')}")

        # Generate visual predictions on a small subset (for visualization
only)
        visualize_predictions(self.model, self.df_test, self.classes,
self.thresholds)

        # IMPORTANT: Evaluate on the FULL test dataset, not just a few
samples
        # Get predictions for ALL test data
        y_true, y_pred = [], []
        for x, y in self.test_ds:
            y_true.append(y.numpy())
            y_pred.append(self.model.predict(x, verbose=0))

        y_true = np.vstack(y_true)
        y_pred = np.vstack(y_pred)
        y_pred_bin = (y_pred > self.thresholds).astype(int)

        # Save classification report based on full test set
        report = classification_report(y_true, y_pred_bin,
target_names=self.classes)
        with open(os.path.join(EXPORT_DIR, 'classification_report.txt'), 'w')
as f:
            f.write(report)

        print("\nFinal Classification Report (on full test dataset):")
        print(report)

# ─── STAGE 1: FROZEN BASE TRAINING
──────────────────────────────────────────
print("\n=== Stage 1: Training with Frozen Base Model ===")
stage1_model = build_model(trainable_base=False)

# Callbacks
metrics_callback = MetricsCallback(test_ds, ROAD_CLASSES, df_test)
early_stop = EarlyStopping(
    monitor='val_f1_macro',
    mode='max',
    patience=8,
    restore_best_weights=True,
    verbose=1
)
checkpoint = ModelCheckpoint(
    os.path.join(EXPORT_DIR, 'best_model_stage1.keras'),
```

```
    monitor='val_f1_macro',
    mode='max',
    save_best_only=True,
    verbose=1
)
reduce_lr = ReduceLROnPlateau(
    monitor='val_f1_macro',
    mode='max',
    factor=0.5,
    patience=4,
    min_lr=1e-6,
    verbose=1
)

# Train stage 1
history_stage1 = stage1_model.fit(
    train_ds,
    epochs=FROZEN_EPOCHS,
    validation_data=test_ds,
    callbacks=[metrics_callback, early_stop, checkpoint, reduce_lr],
    verbose=1
)

# ——— STAGE 2: FINE TUNING
————————————————————————————————————————————
print("\n=== Stage 2: Fine-tuning Model ===")
# Load best stage 1 model
best_stage1_model = tf.keras.models.load_model(
    os.path.join(EXPORT_DIR, 'best_model_stage1.keras'),
    custom_objects={'FocalLoss': FocalLoss}  # Important: provide custom
objects mapping
)

# Create fine-tuning model
stage2_model = build_model(trainable_base=True, fine_tuning=True)

# Copy weights from stage 1
stage2_model.set_weights(best_stage1_model.get_weights())

# Callbacks for stage 2
metrics_callback_stage2 = MetricsCallback(test_ds, ROAD_CLASSES, df_test)
early_stop_stage2 = EarlyStopping(
    monitor='val_f1_macro',
    mode='max',
    patience=16,
    restore_best_weights=True,
    verbose=1
)
checkpoint_stage2 = ModelCheckpoint(
```

```
    os.path.join(EXPORT_DIR, 'best_model_final.keras'),
    monitor='val_f1_macro',
    mode='max',
    save_best_only=True,
    verbose=1
)
reduce_lr_stage2 = ReduceLROnPlateau(
    monitor='val_f1_macro',
    mode='max',
    factor=0.5,
    patience=8,
    min_lr=1e-6,
    verbose=1
)

# Train stage 2
history_stage2 = stage2_model.fit(
    train_ds,
    epochs=FINE_TUNE_EPOCHS,
    validation_data=test_ds,
    callbacks=[metrics_callback_stage2, early_stop_stage2, checkpoint_stage2,
reduce_lr_stage2],
    verbose=1
)

# ——— EXPORT TO TFLITE
————————————————————————————————————————————————
print("\n=== Converting to TFLite for RP2040 Deployment ===")

# Load the best model
final_model = tf.keras.models.load_model(
    os.path.join(EXPORT_DIR, 'best_model_final.keras'),
    custom_objects={'FocalLoss': FocalLoss}  # Provide custom objects mapping
here too
)

# Summary of the model
final_model.summary()

# Define a representative dataset for quantization
def representative_dataset():
    """Generate better representative dataset for quantization"""
    # Use stratified sampling to ensure all classes are represented
    class_specific_samples = {}

    # Get samples for each class
    for class_idx, class_name in enumerate(ROAD_CLASSES):
        # Find samples with this class
        class_samples = df_train[df_train['target_bin'].apply(
```

```
                lambda x: x.split(',')[class_idx] == '1'
            )]

            # If we have samples for this class, store them
            if len(class_samples) > 0:
                class_specific_samples[class_name] = class_samples

    # Generate calibration data
    for _ in range(50):  # First 50 samples - balanced across classes
        for class_name, samples in class_specific_samples.items():
            if len(samples) > 0:
                row = samples.sample(1).iloc[0]
                img_path = os.path.join(DATA_DIR, row['filename'])
                img = cv2.imread(img_path)
                img = cv2.resize(img, IMG_SIZE)
                img = img.astype(np.float32) / 255.0
                yield [np.expand_dims(img, axis=0)]

    # Add 50 random samples to improve representativeness
    for _ in range(50):
        row = df_train.sample(1).iloc[0]
        img_path = os.path.join(DATA_DIR, row['filename'])
        img = cv2.imread(img_path)
        img = cv2.resize(img, IMG_SIZE)
        img = img.astype(np.float32) / 255.0
        yield [np.expand_dims(img, axis=0)]

# Create TFLite converter
converter = tf.lite.TFLiteConverter.from_keras_model(final_model)

# Apply optimizations for RP2040
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

# Convert model
tflite_model = converter.convert()

# Save TFLite model
tflite_path = os.path.join(EXPORT_DIR, f"{NICKNAME}.tflite")
with open(tflite_path, 'wb') as f:
    f.write(tflite_model)

print(f"TFLite model saved: {tflite_path}")
print(f"TFLite model size: {len(tflite_model) / 1024:.2f} KB")

# Also save a metadata file with class names for inference
```

```
class_names = {i: name for i, name in enumerate(ROAD_CLASSES)}
with open(os.path.join(EXPORT_DIR, "class_names.txt"), 'w') as f:
    for i, name in class_names.items():
        f.write(f"{i}: {name}\n")

# ── GENERATE HEADER FILE FOR RP2040 ──────────────────────────────
with open(tflite_path, 'rb') as f:
    data = f.read()

header_path = os.path.join(EXPORT_DIR, "model_data.h")
with open(header_path, 'w') as f:
    f.write('const unsigned char model_data[] = {\n')
    for i, b in enumerate(data):
        if i % 12 == 0:
            f.write('\n ')
        f.write(f' 0x{b:02x},')
    f.write('\n};\nconst int model_data_len = sizeof(model_data);\n')
print(f"C header file created: {header_path} ({len(data) / 1024:.2f} KB)")

# ── SAVE ADDITIONAL METADATA FILES ──────────────────────────────
# Save model information summary with thresholds included directly in the
text file
model_info_path = os.path.join(EXPORT_DIR, "model_info.txt")
with open(model_info_path, 'w') as f:
    f.write(f"Model Name: {NICKNAME}\n")
    f.write(f"Input Shape: {IMG_HEIGHT}x{IMG_WIDTH}x3\n")
    f.write(f"Classes: {', '.join(ROAD_CLASSES)}\n")
    f.write(f"Model Size: {len(tflite_model) / 1024:.2f} KB\n")

    # Add detailed threshold information
    f.write("\n=== Class Detection Thresholds ===\n")
    for i, class_name in enumerate(ROAD_CLASSES):
        f.write(f"{class_name}:
{float(metrics_callback_stage2.thresholds[i]):.4f}\n")

    f.write("\n=== Training Configuration ===\n")
    f.write(f"Training Dataset: {len(df_train)} samples\n")
    # Note: df_val isn't defined in the script, using test dataset instead
    f.write(f"Test Dataset: {len(df_test)} samples\n")
    f.write(f"Batch Size: {BATCH_SIZE}\n")
    f.write(f"Initial Learning Rate: {INITIAL_LR}\n")
    f.write(f"Frozen Epochs: {FROZEN_EPOCHS}\n")
    f.write(f"Fine-tune Epochs: {FINE_TUNE_EPOCHS}\n")

    f.write("\n=== Thresholds C Array ===\n")
    f.write(f"const float thresholds[NUM_CLASSES] = {{")
    f.write(", ".join([f"{float(metrics_callback_stage2.thresholds[i]):.4f}f"
```

```
for i in range(len(ROAD_CLASSES))]))
    f.write("};\n")

    # Get the final metrics from the metrics_callback
    final_f1 = metrics_callback_stage2.metrics_history['f1'][-1] if
metrics_callback_stage2.metrics_history['f1'] else 0
    final_precision = metrics_callback_stage2.metrics_history['precision'][-
1] if metrics_callback_stage2.metrics_history['precision'] else 0
    final_recall = metrics_callback_stage2.metrics_history['recall'][-1] if
metrics_callback_stage2.metrics_history['recall'] else 0

    f.write("\n=== Final Test Set Performance ===\n")
    f.write(f"F1 Score: {final_f1:.4f}\n")
    f.write(f"Precision: {final_precision:.4f}\n")
    f.write(f"Recall: {final_recall:.4f}\n")

print(f"Model information saved: {model_info_path}")
print("Training complete! The optimized TFLite model is ready for deployment
on the SparkFun Thing Plus RP2040.")
```

Appendix 2: Test.py

```
"""
Road Object Detection Testing Script
For TFLite model evaluation on RP2040 deployment
"""

import os
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
import random
from collections import defaultdict
import pandas as pd

# ——— CONFIGURATION ———————————————————————————————————————
EXPORT_DIR = "../TensorFlow/export"
TFLITE_MODEL_PATH = os.path.join(EXPORT_DIR, "RoadLiteMobileNetV2.tflite")
DATA_DIR = "../TensorFlow/Data"
PREDICTION_DIR = "./predictions"
IMG_HEIGHT, IMG_WIDTH = 64, 64
IMG_SIZE = (IMG_HEIGHT, IMG_WIDTH)
EXCEL_TEST_PATH = "../TensorFlow/excel/test.xlsx"  # Path to test dataset
info

# Load the class names from the export directory
CLASS_NAMES = []
with open(os.path.join(EXPORT_DIR, "class_names.txt"), 'r') as f:
    for line in f:
        parts = line.strip().split(': ', 1)
        if len(parts) == 2:
            CLASS_NAMES.append(parts[1])
NUM_CLASSES = len(CLASS_NAMES)

# Create output directory
os.makedirs(PREDICTION_DIR, exist_ok = True)

# ——— LOAD TFLITE INTERPRETER ———————————————————————————————
interpreter = tf.lite.Interpreter(model_path = TFLITE_MODEL_PATH)
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

input_shape = input_details[0]['shape']
input_dtype = input_details[0]['dtype']

print(f"TFLite model loaded and ready for inference")
print(f"Input shape: {input_shape}")
print(f"Input dtype: {input_dtype}")
print(f"Detecting {NUM_CLASSES} classes: {', '.join(CLASS_NAMES)}")
```

```
# ── LOAD THRESHOLDS ──────────────────────────────────────────────
thresholds_path = os.path.join(EXPORT_DIR, "best_thresholds.npy")
if os.path.exists(thresholds_path):
    thresholds = np.load(thresholds_path)
    print(f"Loaded optimized thresholds from {thresholds_path}:
{thresholds}")
else:
    thresholds = np.array([0.5] * NUM_CLASSES)
    print("Using default thresholds of 0.5")


# ── PREPROCESS FUNCTION ──────────────────────────────────────────
def preprocess_image(img_path):
    """Preprocess image for model inference with proper handling for
quantized models"""
    img = cv2.imread(img_path)
    if img is None:
        raise ValueError(f"Failed to load image: {img_path}")

    img = cv2.resize(img, IMG_SIZE)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Handle different input types based on model quantization
    if input_dtype == np.int8:
        # For quantized models, need to properly quantize input
        scale, zero_point = input_details[0]['quantization']
        img = img.astype(np.float32)
        img = img / 255.0  # Normalize to [0,1]
        img = img / scale + zero_point  # Apply quantization params
        img = np.clip(img, -128, 127).astype(np.int8)  # Clip and convert to
int8
    elif input_dtype == np.uint8:
        # For uint8 quantized models
        img = img.astype(np.uint8)
    else:
        # For float models
        img = img.astype(np.float32) / 255.0

    img = np.expand_dims(img, axis = 0)  # Add batch dimension
    return img


# ── INFERENCE FUNCTION ───────────────────────────────────────────
def run_inference(img_path):
    """Run inference on an image and return raw outputs and thresholded
predictions"""
    img = preprocess_image(img_path)

    # Set input tensor
    interpreter.set_tensor(input_details[0]['index'], img)

    # Run inference
    interpreter.invoke()
```

```
    # Get output
    output = interpreter.get_tensor(output_details[0]['index'])[0]

    # If output is quantized, dequantize it
    if output_details[0]['dtype'] == np.int8 or output_details[0]['dtype'] ==
np.uint8:
        # Get quantization parameters
        scale, zero_point = output_details[0]['quantization']
        output = (output.astype(np.float32) - zero_point) * scale

    # Ensure outputs are in range [0,1] for confidence scores (apply sigmoid
if needed)
    if np.any(output < 0) or np.any(output > 1):
        output = 1 / (1 + np.exp(-output))  # Apply sigmoid if outputs aren't
already sigmoids

    # Apply thresholds for binary predictions
    preds = (output > thresholds).astype(int)

    return output, preds


# ——— EVALUATE ON TEST DATASET
————————————————————————————————————————
def evaluate_test_dataset():
    """Evaluate model on test dataset with labeled ground truth"""
    # Load test dataset info if available
    if not os.path.exists(EXCEL_TEST_PATH):
        print(f"⚠ Test dataset info not found at {EXCEL_TEST_PATH}")
        return None, None, None, None

    try:
        df_test = pd.read_excel(EXCEL_TEST_PATH)
        print(f"Loaded test dataset with {len(df_test)} samples")
    except Exception as e:
        print(f"Error loading test dataset: {e}")
        return None, None, None, None

    # Prepare for evaluation
    y_true = []
    y_pred = []
    y_scores = []
    filenames = []

    # Use a subset for faster evaluation if dataset is large
    max_eval_samples = 500
    if len(df_test) > max_eval_samples:
        df_test = df_test.sample(max_eval_samples, random_state = 42)
        print(f"Using {max_eval_samples} random samples for evaluation")

    # Process each test sample
    for idx, row in df_test.iterrows():
        try:
            filename = row['filename']
            filenames.append(filename)
```

```
            img_path = os.path.join(DATA_DIR, filename)

            # Get ground truth
            true_labels = list(map(int, row['target_bin'].split(',')))
            y_true.append(true_labels)

            # Run inference
            scores, preds = run_inference(img_path)
            y_scores.append(scores)
            y_pred.append(preds)

            # Progress indicator
            if (idx + 1) % 50 == 0:
                print(f"Processed {idx + 1}/{len(df_test)} test samples")

        except Exception as e:
            print(f"Error processing {row['filename']}: {e}")

    # Convert to numpy arrays
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
    y_scores = np.array(y_scores)

    return y_true, y_pred, y_scores, filenames


# ——— CALCULATE METRICS ———————————————————————————————————————
def calculate_metrics(y_true, y_pred):
    """Calculate and print evaluation metrics"""
    if y_true is None or y_pred is None:
        return

    # Print full classification report
    report = classification_report(
        y_true, y_pred,
        target_names = CLASS_NAMES,
        zero_division = 0,
        output_dict = False
    )
    print("\n===== Classification Report =====")
    print(report)

    # Also get report as dictionary for visualization
    report_dict = classification_report(
        y_true, y_pred,
        target_names = CLASS_NAMES,
        zero_division = 0,
        output_dict = True
    )

    # Save the report to a file
    with open(os.path.join(PREDICTION_DIR, "evaluation_report.txt"), 'w') as
f:
        f.write("===== Classification Report =====\n")
        f.write(report)
```

```
    return report_dict


# ——— VISUALIZE METRICS ————————————————————————————————————————
def visualize_metrics(report_dict):
    """Create visualizations of model performance metrics"""
    if report_dict is None:
        return

    # Extract class metrics
    class_metrics = {}
    for cls in CLASS_NAMES:
        if cls in report_dict:
            class_metrics[cls] = {
                'precision':report_dict[cls]['precision'],
                'recall':report_dict[cls]['recall'],
                'f1-score':report_dict[cls]['f1-score'],
                'support':report_dict[cls]['support']
            }

    # Create metrics dataframe
    metrics_df = pd.DataFrame(class_metrics).T
    metrics_df = metrics_df.reset_index().rename(columns = {'index':'class'})

    # 1. Bar chart of F1 scores
    plt.figure(figsize = (10, 6))
    ax = sns.barplot(x = 'class', y = 'f1-score', data = metrics_df, palette
= 'viridis')
    ax.set_title('F1 Scores by Class')
    ax.set_xlabel('Class')
    ax.set_ylabel('F1 Score')
    plt.xticks(rotation = 45, ha = 'right')
    plt.tight_layout()
    plt.savefig(os.path.join(PREDICTION_DIR, "f1_scores.png"))

    # 2. Precision vs Recall plot
    plt.figure(figsize = (10, 6))
    plt.scatter(metrics_df['precision'], metrics_df['recall'], s =
metrics_df['support'], alpha = 0.7)
    for i, cls in enumerate(metrics_df['class']):
        plt.annotate(cls,
                     (metrics_df['precision'][i], metrics_df['recall'][i]),
                     xytext = (5, 5),
                     textcoords = 'offset points')
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.xlabel('Precision')
    plt.ylabel('Recall')
    plt.title('Precision vs Recall by Class (circle size = sample count)')
    plt.grid(True, linestyle = '--', alpha = 0.7)
    # Add reference lines
    plt.axhline(y = 0.5, color = 'r', linestyle = ':', alpha = 0.5)
    plt.axvline(x = 0.5, color = 'r', linestyle = ':', alpha = 0.5)
    plt.tight_layout()
    plt.savefig(os.path.join(PREDICTION_DIR, "precision_recall.png"))
```

```
    # 3. Confusion matrices (one-vs-rest for each class)
    fig, axes = plt.subplots(2, 4, figsize = (20, 10))
    axes = axes.flatten()

    for i, cls in enumerate(CLASS_NAMES):
        if i < len(axes):
            ax = axes[i]
            tp = report_dict[cls]['support'] * report_dict[cls]['recall']
            fn = report_dict[cls]['support'] - tp
            fp = (tp / report_dict[cls]['precision']) - tp if
report_dict[cls]['precision'] > 0 else 0
            tn = report_dict['samples avg']['support'] - tp - fn - fp

            cm = np.array([[tn, fp], [fn, tp]])
            sns.heatmap(cm, annot = True, fmt = '.0f', cmap = 'Blues', ax =
ax)
            ax.set_title(f'Confusion Matrix: {cls}')
            ax.set_xlabel('Predicted')
            ax.set_ylabel('Actual')
            ax.set_xticklabels(['Negative', 'Positive'])
            ax.set_yticklabels(['Negative', 'Positive'])

    # Hide any unused subplots
    for j in range(i + 1, len(axes)):
        axes[j].axis('off')

    plt.tight_layout()
    plt.savefig(os.path.join(PREDICTION_DIR, "confusion_matrices.png"))


# ——— ANALYZE ERRORS
--------------------------------------------------------------------

def analyze_errors(y_true, y_pred, y_scores, filenames):
    """Analyze and visualize prediction errors and edge cases"""
    if y_true is None or y_pred is None:
        return

    # Calculate error types for each sample
    errors = defaultdict(list)

    for i, (true, pred, scores, fname) in enumerate(zip(y_true, y_pred,
y_scores, filenames)):
        # Check for false positives and false negatives
        for cls_idx, (t, p) in enumerate(zip(true, pred)):
            if t == 1 and p == 0:  # False negative
                errors['false_negatives'].append((fname, cls_idx,
scores[cls_idx]))
            elif t == 0 and p == 1:  # False positive
                errors['false_positives'].append((fname, cls_idx,
scores[cls_idx]))

        # Check for samples with low confidence (high entropy in predictions)
        if np.max(scores) < 0.6:
            errors['low_confidence'].append((fname, np.argmax(scores),
np.max(scores)))

        # Check for samples with multiple high confidence predictions
```

```
        if sum(scores > 0.7) > 1:
            errors['multiple_detections'].append((fname, np.where(scores >
0.7)[0].tolist(),
                                                  scores[scores >
0.7].tolist()))

    # Print error statistics
    print("\n===== Error Analysis =====")
    print(f"False Negatives: {len(errors['false_negatives'])}")
    print(f"False Positives: {len(errors['false_positives'])}")
    print(f"Low Confidence: {len(errors['low_confidence'])}")
    print(f"Multiple Detections: {len(errors['multiple_detections'])}")

    # Visualize examples of each error type
    error_types = {
        'false_negatives':'False Negatives (missed detections)',
        'false_positives':'False Positives (incorrect detections)',
        'low_confidence':'Low Confidence Predictions',
        'multiple_detections':'Multiple High-Confidence Detections'
    }

    for error_type, title in error_types.items():
        if len(errors[error_type]) == 0:
            continue

        # Sample up to 8 examples of this error type
        samples = random.sample(errors[error_type], min(8,
len(errors[error_type])))

        # Create figure
        fig, axes = plt.subplots(2, 4, figsize = (16, 8))
        axes = axes.flatten()

        for i, sample in enumerate(samples):
            if i < len(axes):
                fname = sample[0]
                img_path = os.path.join(DATA_DIR, fname)

                # Load and display image
                try:
                    img = cv2.imread(img_path)
                    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                    img = cv2.resize(img, IMG_SIZE)

                    axes[i].imshow(img)
                    axes[i].axis('off')

                    # Add appropriate caption based on error type
                    if error_type == 'false_negatives':
                        cls_idx = sample[1]
                        conf = sample[2]
                        axes[i].set_title(
                            f"Missed {CLASS_NAMES[cls_idx]}\nConf: {conf:.2f}
< {thresholds[cls_idx]:.2f}")
                    elif error_type == 'false_positives':
                        cls_idx = sample[1]
                        conf = sample[2]
```

```
                        axes[i].set_title(f"False
{CLASS_NAMES[cls_idx]}\nConf: {conf:.2f} > {thresholds[cls_idx]:.2f}")
                    elif error_type == 'low_confidence':
                        cls_idx = sample[1]
                        conf = sample[2]
                        axes[i].set_title(f"Low conf:
{CLASS_NAMES[cls_idx]}\nConf: {conf:.2f}")
                    elif error_type == 'multiple_detections':
                        class_indices = sample[1]
                        confs = sample[2]
                        class_names = [CLASS_NAMES[idx] for idx in
class_indices]
                        axes[i].set_title("\n".join([f"{cls}: {conf:.2f}" for
cls, conf in zip(class_names, confs)]))

                except Exception as e:
                    print(f"Error displaying {fname}: {e}")
                    axes[i].text(0.5, 0.5, f"Error: {e}", ha = 'center', va =
'center')
                    axes[i].axis('off')

        # Hide any unused subplots
        for j in range(i + 1, len(axes)):
            axes[j].axis('off')

        plt.suptitle(title)
        plt.tight_layout()
        plt.savefig(os.path.join(PREDICTION_DIR, f"error_{error_type}.png"))


# ——— TEST RANDOM IMAGES
-------------------------------------------------------------
def test_random_images(num_samples = 8):
    """Test and visualize model on random images"""
    image_files = [f for f in os.listdir(DATA_DIR) if
f.lower().endswith((".jpg", ".jpeg", ".png"))]

    if not image_files:
        print(f"No image files found in {DATA_DIR}")
        return

    # Sample randomly
    sampled_files = np.random.choice(image_files, size = min(num_samples,
len(image_files)), replace = False)

    # Create figure with appropriate number of rows and columns
    cols = min(4, num_samples)
    rows = (num_samples + cols - 1) // cols
    fig, axes = plt.subplots(rows, cols, figsize = (cols * 4, rows * 4))

    # Handle single axis case
    if num_samples == 1:
        axes = np.array([axes])
    axes = axes.flatten()

    for i, fname in enumerate(sampled_files):
        img_path = os.path.join(DATA_DIR, fname)
```

```
        try:
            # Load original image for display
            raw_img = cv2.imread(img_path)
            raw_img = cv2.cvtColor(raw_img, cv2.COLOR_BGR2RGB)
            raw_img = cv2.resize(raw_img, IMG_SIZE)

            # Run inference
            output, preds = run_inference(img_path)

            # Get predicted class names
            pred_labels = [CLASS_NAMES[j] for j in range(NUM_CLASSES) if
preds[j]]
            if not pred_labels:
                pred_labels = ["None"]

            # Get top 3 predictions with scores
            top_indices = np.argsort(output)[::-1][:3]
            top_scores = [(CLASS_NAMES[j], output[j]) for j in top_indices]

            # Format label string
            label_str = "\n".join([f"{name}: {score:.2f} ({'✓' if score >
thresholds[j] else '✗'})"
                                   for j, (name, score) in
enumerate(top_scores)])

            # Display image and predictions
            axes[i].imshow(raw_img)
            axes[i].axis("off")
            axes[i].set_title(
                f"Predictions: {', '.join(pred_labels)}\n\n{label_str}",
                fontsize = 9
            )

        except Exception as e:
            print(f"Error processing {fname}: {e}")
            axes[i].text(0.5, 0.5, f"Error: {e}", ha = 'center', va =
'center')
            axes[i].axis("off")

    # Hide unused subplots
    for j in range(i + 1, len(axes)):
        axes[j].axis("off")

    plt.tight_layout()
    plt.savefig(os.path.join(PREDICTION_DIR, "test_predictions.png"))
    plt.close()


# ——— INFERENCE SPEED TEST
-------------------------------------------------------------------
def test_inference_speed(num_runs = 50):
    """Test inference speed over multiple runs"""
    image_files = [f for f in os.listdir(DATA_DIR) if
f.lower().endswith((".jpg", ".jpeg", ".png"))]
```

```
    if not image_files or len(image_files) < 5:
        print(f"Not enough image files found in {DATA_DIR} for speed test")
        return

    # Sample some files for testing
    sampled_files = np.random.choice(image_files, size = min(5,
len(image_files)), replace = False)

    # Preload images to avoid I/O overhead in timing
    preloaded_images = {}
    for fname in sampled_files:
        img_path = os.path.join(DATA_DIR, fname)
        img = preprocess_image(img_path)
        preloaded_images[fname] = img

    # Run inference multiple times and measure
    inference_times = []

    for _ in range(num_runs):
        # Select random image
        fname = np.random.choice(list(preloaded_images.keys()))
        img = preloaded_images[fname]

        # Time inference
        start_time = cv2.getTickCount()

        # Set input tensor
        interpreter.set_tensor(input_details[0]['index'], img)
        # Run inference
        interpreter.invoke()
        # Get output
        output = interpreter.get_tensor(output_details[0]['index'])[0]

        end_time = cv2.getTickCount()
        inference_time = (end_time - start_time) / cv2.getTickFrequency() *
1000  # ms
        inference_times.append(inference_time)

    # Calculate statistics
    avg_time = np.mean(inference_times)
    std_time = np.std(inference_times)
    min_time = np.min(inference_times)
    max_time = np.max(inference_times)

    print("\n===== Inference Speed Test =====")
    print(f"Average inference time: {avg_time:.2f} ms ({1000 / avg_time:.2f}
FPS)")
    print(f"Standard deviation: {std_time:.2f} ms")
    print(f"Min/Max time: {min_time:.2f}/{max_time:.2f} ms")

    # Plot histogram of inference times
    plt.figure(figsize = (10, 6))
    plt.hist(inference_times, bins = 20, alpha = 0.7, color = 'blue')
    plt.axvline(avg_time, color = 'red', linestyle = 'dashed', linewidth = 2,
label = f'Mean: {avg_time:.2f} ms')
    plt.xlabel('Inference Time (ms)')
    plt.ylabel('Frequency')
```

```
    plt.title('Inference Time Distribution')
    plt.legend()
    plt.grid(True, linestyle = '--', alpha = 0.7)
    plt.savefig(os.path.join(PREDICTION_DIR, "inference_speed.png"))
    plt.close()

    # Save results to file
    with open(os.path.join(PREDICTION_DIR, "inference_speed.txt"), 'w') as f:
        f.write("===== Inference Speed Test =====\n")
        f.write(f"Average inference time: {avg_time:.2f} ms ({1000 /
avg_time:.2f} FPS)\n")
        f.write(f"Standard deviation: {std_time:.2f} ms\n")
        f.write(f"Min/Max time: {min_time:.2f}/{max_time:.2f} ms\n")


# ── ANALYZE MODEL SIZE ──────────────────────────────────────────
def analyze_model_size():
    """Analyze and report model size information"""
    model_size = os.path.getsize(TFLITE_MODEL_PATH)
    model_size_kb = model_size / 1024
    model_size_mb = model_size_kb / 1024

    print("\n===== Model Size Analysis =====")
    print(f"Model file: {TFLITE_MODEL_PATH}")
    print(f"Model size: {model_size_kb:.2f} KB ({model_size_mb:.2f} MB)")

    # RP2040 has limited flash (typically 2MB) and RAM (264KB), so provide
context
    print(f"Percentage of typical RP2040 flash (2MB): {model_size_mb /
2:.2f}%")
    print(f"Percentage of typical RP2040 RAM (264KB): {model_size_kb /
264:.2f}%")

    # Save results to file
    with open(os.path.join(PREDICTION_DIR, "model_size.txt"), 'w') as f:
        f.write("===== Model Size Analysis =====\n")
        f.write(f"Model file: {TFLITE_MODEL_PATH}\n")
        f.write(f"Model size: {model_size_kb:.2f} KB ({model_size_mb:.2f}
MB)\n")
        f.write(f"Percentage of typical RP2040 flash (2MB): {model_size_mb /
2:.2f}%\n")
        f.write(f"Percentage of typical RP2040 RAM (264KB): {model_size_kb /
264:.2f}%\n")


# ── MAIN ────────────────────────────────────────────────────────
def main():
      # Test on random images
    print("\n1. Testing on random images...")
    test_random_images(num_samples = 8)

    # Evaluate on test dataset
    print("\n2. Evaluating on test dataset...")
    y_true, y_pred, y_scores, filenames = evaluate_test_dataset()
```

```
    # Calculate and visualize metrics
    if y_true is not None:
        print("\n3. Calculating metrics...")
        report_dict = calculate_metrics(y_true, y_pred)

        print("\n4. Visualizing performance metrics...")
        visualize_metrics(report_dict)

        print("\n5. Analyzing errors...")
        analyze_errors(y_true, y_pred, y_scores, filenames)

    # Test inference speed
    print("\n6. Testing inference speed...")
    test_inference_speed()

    # Analyze model size
    print("\n7. Analyzing model size...")
    analyze_model_size()

    print(f"\nEvaluation complete. All results saved to {PREDICTION_DIR}/")


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

Appendix 3: PICO Code

```
/**
 * Pico-RoadDetector: Optimized Dual-Core Implementation
 *
 * This application uses both RP2040 cores to efficiently run TensorFlow Lite
 * machine learning models for road object detection:
 * - Core 0: I/O operations (camera, display, user interface)
 * - Core 1: TensorFlow Lite inference and image processing
 */

#include "pico/stdlib.h"
#include "pico/multicore.h"
#include "pico/util/queue.h"
#include "hardware/i2c.h"
#include "hardware/spi.h"
#include "hardware/gpio.h"
#include "hardware/sync.h"
#include "pico/time.h"
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include "picojpeg.h"
#include "jpeg_decoder.h"

// Project headers
#include "hardware_config.h"
#include "ov5642_regs.h"
#include "ssd1306.h"
#include "pca9546.h"
#include "class_names.h"
#include "model_data.h"

// TensorFlow Lite includes
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"

// Debug mode
#define DEBUG 1

// Model dimensions
#define MODEL_WIDTH 64
#define MODEL_HEIGHT 64

// Spinlock for shared memory access
spin_lock_t *memory_lock;
#define MEMORY_LOCK_ID 0

// Queue for inter-core commands
queue_t core0_to_core1_queue;
queue_t core1_to_core0_queue;

// Commands for inter-core communication
```

```
enum CoreCommand {
    CMD_PROCESS_IMAGE = 1,
    CMD_INFERENCE_COMPLETE = 2,
    CMD_ERROR = 3
};

// Shared data structure for inference results
struct InferenceResult {
    float scores[7];
    uint8_t predictions[7];
    uint32_t inference_time_ms;
    bool valid;
};

// Pre-allocated buffers to reduce memory fragmentation
// Buffer for camera capture - shared between cores
static uint8_t g_capture_buffer[32768] __attribute__((aligned(8)));
static uint32_t g_capture_size = 0;

// Buffer for image processing - used by core 1
static uint8_t g_process_buffer[MODEL_WIDTH * MODEL_HEIGHT * 3]
__attribute__((aligned(8)));

// Results shared between cores
static volatile InferenceResult g_inference_result
__attribute__((aligned(8)));

// Optimized thresholds from first version based on evaluation
const float g_improved_thresholds[7] = {
    0.2105, // bicycle
    0.3211, // car
    0.2474, // motorcycle
    0.2842, // bus
    0.2474, // truck
    0.2474, // traffic light
    0.2474  // stop sign
};

// Global variables for hardware
ArduCAM myCAM(OV5642, PIN_CS);

// Forward declarations
bool setup_hardware();
bool setup_camera();
bool setup_display();
bool capture_image_to_buffer(uint8_t *buffer, size_t buffer_size, uint32_t
*captured_size);
void debug_print(const char *msg);
void display_message(const char *line1, const char *line2, const char
*line3);

// TensorFlow Lite globals for Core 1
namespace {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter *error_reporter = &micro_error_reporter;
    const tflite::Model *model = nullptr;
    tflite::MicroInterpreter *interpreter = nullptr;
```

```
    TfLiteTensor *input = nullptr;
    TfLiteTensor *output = nullptr;
    constexpr int kTensorArenaSize = 128 * 1024;
    alignas(16) uint8_t tensor_arena[kTensorArenaSize];
}

// Helper function for debug output
void debug_print(const char *msg) {
#if DEBUG
    printf("%s\n", msg);
#endif
}

// Display a message on the OLED screen with up to three rows of text
void display_message(const char *line1, const char *line2 = nullptr, const
char *line3 = nullptr) {
    // Select OLED on multiplexer
    int result = pca9546_select(I2C_PORT, MUX_PORT_OLED);
    if (result != PCA9546_SUCCESS) {
        printf("Failed to select OLED on multiplexer: %s\n",
               pca9546_error_string(result));
        return;
    }

    ssd1306_clear();

    // Display first line
    if (line1) {
        ssd1306_draw_string(0, 0, line1, 1);
    }

    // Display second line
    if (line2) {
        ssd1306_draw_string(0, 16, line2, 1);
    }

    // Display third line
    if (line3) {
        ssd1306_draw_string(0, 32, line3, 1);
    }

    ssd1306_show();
}

// Initialize hardware components
bool setup_hardware() {
    // Initialize stdio
    stdio_init_all();
    sleep_ms(1000); // Give USB time to initialize

    // Set CPU clock to maximum
    set_sys_clock_khz(133000, true);

    debug_print("Initializing hardware...");

    // Initialize I2C
    i2c_init(I2C_PORT, I2C_FREQ);
```

```
    gpio_set_function(PIN_SDA, GPIO_FUNC_I2C);
    gpio_set_function(PIN_SCL, GPIO_FUNC_I2C);
    gpio_pull_up(PIN_SDA);
    gpio_pull_up(PIN_SCL);

    // Initialize SPI
    spi_init(SPI_PORT, ARDUCAM_SPI_FREQ);
    gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);

    // Initialize CS pin
    gpio_init(PIN_CS);
    gpio_set_dir(PIN_CS, GPIO_OUT);
    gpio_put(PIN_CS, 1);

    printf("I2C and SPI initialized\n");

    // Initialize I2C multiplexer
    int mux_result = pca9546_init(I2C_PORT, PCA9546_ADDR);
    if (mux_result != PCA9546_SUCCESS) {
        printf("Failed to initialize PCA9546 multiplexer: %s\n",
               pca9546_error_string(mux_result));
        return false;
    }

    printf("PCA9546 multiplexer initialized\n");

    // Initialize spinlock for memory protection
    memory_lock = spin_lock_init(MEMORY_LOCK_ID);

    // Initialize queues for inter-core communication
    queue_init(&core0_to_core1_queue, sizeof(uint32_t), 4);
    queue_init(&core1_to_core0_queue, sizeof(uint32_t), 4);

    debug_print("Hardware initialized");
    return true;
}

// Initialize OLED display
bool setup_display() {
    debug_print("Setting up OLED display...");

    // Select OLED on multiplexer
    int result = pca9546_select(I2C_PORT, MUX_PORT_OLED);
    if (result != PCA9546_SUCCESS) {
        printf("Failed to select OLED on multiplexer: %s\n",
               pca9546_error_string(result));
        return false;
    }

    // Initialize display
    ssd1306_init(I2C_PORT, SSD1306_ADDR);
    ssd1306_clear();
    ssd1306_draw_string(0, 0, "INITIALIZING...", 1);
    ssd1306_show();
```

```
    debug_print("OLED display ready");
    return true;
}

// Verify correct SPI communication with ArduCAM controller
bool verify_arducam_spi() {
    printf("Verifying ArduCAM SPI communication...\n");

    // Test write/read to a test register
    myCAM.write_reg(ARDUCHIP_TEST1, 0x55);
    sleep_ms(5);
    uint8_t read_val = myCAM.read_reg(ARDUCHIP_TEST1);
    printf("Write 0x55, read: 0x%02X\n", read_val);

    if (read_val != 0x55) {
        printf("SPI communication test failed! Expected 0x55, got 0x%02X\n",
read_val);
        return false;
    }

    printf("ArduCAM SPI communication verified\n");
    return true;
}

// Verify camera sensor is responding
bool verify_camera_sensor() {
    // Select ArduCAM channel on multiplexer
    int result = pca9546_select(I2C_PORT, MUX_PORT_ARDUCAM);
    if (result != PCA9546_SUCCESS) {
        printf("Failed to select ArduCAM channel on multiplexer: %s\n",
               pca9546_error_string(result));
        return false;
    }

    // Software reset the sensor
    myCAM.wrSensorReg16_8(0x3008, 0x80);
    sleep_ms(100);

    // Clear reset flag
    myCAM.wrSensorReg16_8(0x3008, 0x00);
    sleep_ms(100);

    // Check camera ID
    uint8_t vid, pid;
    myCAM.wrSensorReg16_8(0xFF, 0x01);
    myCAM.rdSensorReg16_8(OV5642_CHIPID_HIGH, &vid);
    myCAM.rdSensorReg16_8(OV5642_CHIPID_LOW, &pid);

    printf("Camera ID check - VID: 0x%02X, PID: 0x%02X\n", vid, pid);

    if (vid != 0x56 || pid != 0x42) {
        printf("Camera ID mismatch! Expected VID=0x56, PID=0x42\n");
        return false;
    }

    printf("OV5642 camera sensor verified\n");
    return true;
```

```
}

// Test camera capture to verify functionality
bool test_camera_capture() {
    printf("Testing camera capture...\n");

    // Force JPEG mode for test capture
    myCAM.wrSensorReg16_8(0x4300, 0x18);
    sleep_ms(10);

    // Reset FIFO before capture
    myCAM.flush_fifo();
    myCAM.clear_fifo_flag();
    sleep_ms(50);

    // Start capture
    myCAM.start_capture();
    printf("Test capture started\n");

    // Wait for capture with timeout
    bool capture_done = false;
    uint32_t start_time = to_ms_since_boot(get_absolute_time());

    while (!capture_done) {
        if (myCAM.get_bit(ARDUCHIP_TRIG, CAP_DONE_MASK)) {
            capture_done = true;
            printf("Test capture completed in %d ms\n",
                   (int)(to_ms_since_boot(get_absolute_time()) -
start_time));
            break;
        }

        if (to_ms_since_boot(get_absolute_time()) - start_time > 3000) {
            printf("Test capture timed out after 3 seconds\n");
            return false;
        }

        sleep_ms(10);
    }

    // Read FIFO length
    uint32_t length = myCAM.read_fifo_length();
    printf("Test image size: %lu bytes\n", length);

    // Check if image size is reasonable
    if (length < 1000 || length > 200000) {
        printf("Test image size is unreasonable: %lu bytes\n", length);
        return false;
    }

    // Allocate buffer for the JPEG header to check validity
    const size_t HEADER_SIZE = 32;
    uint8_t header[HEADER_SIZE];

    // Read image header data from FIFO
    myCAM.CS_LOW();
    myCAM.set_fifo_burst();
```

```
    // Read JPEG header to confirm it's valid
    for (int i = 0; i < HEADER_SIZE && i < length; i++) {
        spi_read_blocking(SPI_PORT, 0, &header[i], 1);
    }

    myCAM.CS_HIGH();

    // Print first bytes for debugging
    printf("JPEG header: ");
    for (int i = 0; i < 16; i++) {
        printf("%02X ", header[i]);
    }
    printf("\n");

    // Check for JPEG signature (0xFF 0xD8)
    bool valid_jpeg = (header[0] == 0xFF && header[1] == 0xD8);

    if (!valid_jpeg) {
        printf("Invalid JPEG header\n");
        return false;
    }

    printf("Valid JPEG data captured - Camera test PASSED\n");

    // Clear FIFO flag to complete test
    myCAM.clear_fifo_flag();
    return true;
}

// Initialize ArduCAM camera
bool setup_camera() {
    debug_print("Setting up ArduCAM...");

    // Initialize ArduCAM
    myCAM.Arducam_init();
    sleep_ms(100);

    // Select ArduCAM channel on multiplexer
    int result = pca9546_select(I2C_PORT, MUX_PORT_ARDUCAM);
    if (result != PCA9546_SUCCESS) {
        printf("Failed to select ArduCAM channel on multiplexer: %s\n",
               pca9546_error_string(result));
        return false;
    }

    // Reset hardware
    myCAM.write_reg(0x07, 0x80);
    sleep_ms(100);
    myCAM.write_reg(0x07, 0x00);
    sleep_ms(100);

    // Verify SPI communication
    if (!verify_arducam_spi()) {
        printf("SPI communication verification failed\n");
        return false;
    }
```

```
    // Verify camera sensor
    if (!verify_camera_sensor()) {
        printf("Camera sensor verification failed\n");
        return false;
    }

    // Initialize camera mode and settings
    myCAM.set_format(JPEG);
    myCAM.InitCAM();
    sleep_ms(50);

    // Configure timing
    myCAM.write_reg(ARDUCHIP_TIM, VSYNC_LEVEL_MASK);
    sleep_ms(50);

    // Set lower resolution for memory savings
    myCAM.OV5642_set_JPEG_size(OV5642_64x64);
    sleep_ms(100);

    myCAM.OV5642_set_Mirror_Flip(FLIP);
    sleep_ms(200);

    // Reset FIFO
    myCAM.clear_fifo_flag();
    myCAM.write_reg(ARDUCHIP_FRAMES, 0x00);
    sleep_ms(100);

    // Set critical JPEG registers
    myCAM.wrSensorReg16_8(0x4300, 0x18); // Format control - YUV422 + JPEG
    myCAM.wrSensorReg16_8(0x3818, 0xA8); // Timing control
    myCAM.wrSensorReg16_8(0x3621, 0x10); // Array control
    myCAM.wrSensorReg16_8(0x3801, 0xB0); // Timing HS
    myCAM.wrSensorReg16_8(0x4407, 0x04); // Compression quantization
    sleep_ms(100);

    // Perform test capture to verify all is working
    if (!test_camera_capture()) {
        printf("Camera capture test failed\n");
        return false;
    }

    debug_print("ArduCAM camera initialized successfully");
    return true;
}

// Capture image from camera and store in buffer
// Optimized to reduce processing time
bool capture_image_to_buffer(uint8_t *buffer, size_t buffer_size, uint32_t
*captured_size) {
    // Select ArduCAM on multiplexer
    int result = pca9546_select(I2C_PORT, MUX_PORT_ARDUCAM);
    if (result != PCA9546_SUCCESS) {
        printf("Failed to select ArduCAM on multiplexer for capture: %s\n",
               pca9546_error_string(result));
        return false;
    }
```

```
    // Force critical JPEG registers before capture
    myCAM.wrSensorReg16_8(0x4300, 0x18); // Format control - YUV422 + JPEG
    myCAM.wrSensorReg16_8(0x501F, 0x00); // ISP output format

    // Reset FIFO
    myCAM.flush_fifo();
    sleep_ms(5);
    myCAM.clear_fifo_flag();
    sleep_ms(5);

    // Start capture
    printf("Starting image capture...\n");
    absolute_time_t start_time = get_absolute_time();
    myCAM.start_capture();

    // Wait for capture with timeout
    bool capture_timeout = false;
    absolute_time_t timeout = make_timeout_time_ms(1000);  // 1-second
timeout

    while (!myCAM.get_bit(ARDUCHIP_TRIG, CAP_DONE_MASK)) {
        if (absolute_time_diff_us(get_absolute_time(), timeout) <= 0) {
            capture_timeout = true;
            break;
        }
        sleep_us(100);  // Short sleep to be responsive but not waste CPU
    }

    if (capture_timeout) {
        printf("Error: Camera capture timeout\n");
        return false;
    }

    int elapsed_ms = absolute_time_diff_us(start_time, get_absolute_time()) /
1000;
    printf("Capture completed successfully in %d ms\n", elapsed_ms);

    // Read captured data size
    uint32_t length = myCAM.read_fifo_length();
    printf("FIFO length: %lu bytes\n", length);

    if (length > buffer_size || length < 20) {
        printf("Error: Image size invalid or too large for buffer: %lu
bytes\n", length);
        return false;
    }

    // Store captured size
    *captured_size = length;

    // Read image data from FIFO into buffer
    myCAM.CS_LOW();
    myCAM.set_fifo_burst();

    // Read data using DMA-optimized chunks for better performance
    const size_t CHUNK_SIZE = 1024;
```

```
    for (uint32_t i = 0; i < length; i += CHUNK_SIZE) {
        size_t bytes_to_read = (i + CHUNK_SIZE > length) ? (length - i) :
CHUNK_SIZE;
        spi_read_blocking(SPI_PORT, 0, &buffer[i], bytes_to_read);
    }

    myCAM.CS_HIGH();

    // Print first few bytes for debugging
    printf("First 16 bytes: ");
    for (int i = 0; i < 16 && i < length; i++) {
        printf("%02X ", buffer[i]);
    }
    printf("\n");

    // Verify JPEG header
    if (buffer[0] != 0xFF || buffer[1] != 0xD8) {
        printf("Error: Invalid JPEG header\n");
        return false;
    }

    // Reset FIFO flag
    myCAM.clear_fifo_flag();

    return true;
}

// Process image for inference - runs on Core 1
bool process_image_for_inference(const uint8_t *raw_buffer, uint32_t
raw_size, uint8_t *output_buffer) {
    // Decode JPEG to RGB at reduced resolution directly
    if (!jpeg_decode_to_model_input(raw_buffer, raw_size,
                                    output_buffer,
                                    MODEL_WIDTH, MODEL_HEIGHT)) {
        printf("Core 1: Error - JPEG decoding failed\n");
        return false;
    }

    return true;
}

// Fill TFLite input tensor with preprocessed image data
bool fill_input_tensor(const uint8_t *image_data) {
    if (!input) {
        printf("Error: Input tensor not initialized\n");
        return false;
    }

    // Handle different tensor types
    if (input->type == kTfLiteInt8) {
        // For int8 quantized model input
        int8_t *input_data = input->data.int8;
        float scale = 0.003922f; // 1/255 from training
        int zero_point = -128;   // From training

        for (int y = 0; y < MODEL_HEIGHT; y++) {
            for (int x = 0; x < MODEL_WIDTH; x++) {
```

```
                // Get pixel value from our decoded grayscale image
                uint8_t pixel = image_data[y * MODEL_WIDTH + x];

                // Convert to INT8 range (-128 to 127) using training
parameters
                int8_t quantized = (int8_t)(pixel - 128);

                // For RGB model inputs with 3 channels
                if (input->dims->data[3] == 3) {
                    int dst_idx = (y * MODEL_WIDTH + x) * 3;
                    input_data[dst_idx + 0] = quantized; // R
                    input_data[dst_idx + 1] = quantized; // G
                    input_data[dst_idx + 2] = quantized; // B
                }
                // For single channel model inputs
                else if (input->dims->data[3] == 1) {
                    input_data[y * MODEL_WIDTH + x] = quantized;
                }
            }
        }
    }
    else if (input->type == kTfLiteUInt8) {
        // For quantized model input (uint8)
        uint8_t *input_data = input->data.uint8;

        for (int y = 0; y < MODEL_HEIGHT; y++) {
            for (int x = 0; x < MODEL_WIDTH; x++) {
                // Get pixel value
                uint8_t pixel = image_data[y * MODEL_WIDTH + x];

                // For RGB model inputs with 3 channels
                if (input->dims->data[3] == 3) {
                    int dst_idx = (y * MODEL_WIDTH + x) * 3;
                    input_data[dst_idx + 0] = pixel; // R
                    input_data[dst_idx + 1] = pixel; // G
                    input_data[dst_idx + 2] = pixel; // B
                }
                // For single channel model inputs
                else if (input->dims->data[3] == 1) {
                    input_data[y * MODEL_WIDTH + x] = pixel;
                }
            }
        }
    }
    else if (input->type == kTfLiteFloat32) {
        // For floating point model input (float32)
        float *input_data = input->data.f;

        for (int y = 0; y < MODEL_HEIGHT; y++) {
            for (int x = 0; x < MODEL_WIDTH; x++) {
                // Get pixel value and normalize to 0-1 range
                float pixel = image_data[y * MODEL_WIDTH + x] / 255.0f;

                // For RGB model inputs with 3 channels
                if (input->dims->data[3] == 3) {
                    int dst_idx = (y * MODEL_WIDTH + x) * 3;
                    input_data[dst_idx + 0] = pixel; // R
```

```
                    input_data[dst_idx + 1] = pixel; // G
                    input_data[dst_idx + 2] = pixel; // B
                }
                // For single channel model inputs
                else if (input->dims->data[3] == 1) {
                    input_data[y * MODEL_WIDTH + x] = pixel;
                }
            }
        }
    }
    else {
        printf("Error: Unsupported input tensor type: %d\n", input->type);
        return false;
    }

    return true;
}

// Format and display detection results using three lines
// Enhanced to improve serial monitor and OLED display
void display_results(const InferenceResult *result) {
    if (!result || !result->valid) {
        display_message("DETECTION ERROR", "Invalid result", "");
        return;
    }

    // Keep track of highest confidence class for display
    int best_idx = 0;
    float best_score = result->scores[0];

    // Find the highest confidence class
    for (int i = 1; i < 7; i++) {
        if (result->scores[i] > best_score) {
            best_score = result->scores[i];
            best_idx = i;
        }
    }

    // Convert highest confidence to percentage
    int confidence_pct = (int)(best_score * 100);

    // Print individual class predictions to serial monitor
    printf("--------DETECTION RESULTS--------\n");
    for (int i = 0; i < 7; i++) {
        printf("Class: %-13s | Score: %5.2f%% | %s\n",
               class_names[i],
               result->scores[i] * 100.0f,
               result->predictions[i] ? "DETECTED" : "Not detected");
    }
    printf("Best detection: %s (%d%%)\n", class_names[best_idx],
confidence_pct);
    printf("Inference time: %ld ms\n", result->inference_time_ms);
    printf("-------------------------------\n");

    // Build a string with detected classes (for line 1)
    char detected_classes[32] = ""; // Buffer for detected classes
    int detected_count = 0;
```

```
    for (int i = 0; i < 7; i++) {
        if (result->predictions[i]) {
            // If not the first detection, add comma separator
            if (detected_count > 0 && strlen(detected_classes) < 28) {
                strcat(detected_classes, ",");
            }

            // Add class name if there's enough space
            if (strlen(detected_classes) + strlen(class_names[i]) < 29) {
                strcat(detected_classes, class_names[i]);
                detected_count++;
            }
        }
    }

    // If nothing detected, show "NONE"
    if (detected_count == 0) {
        strcpy(detected_classes, "NONE DETECTED");
    }

    // Prepare second line - best detection with confidence
    char best_detection[17];
    if (detected_count > 0) {
        snprintf(best_detection, sizeof(best_detection), "%s: %d%%",
                class_names[best_idx], confidence_pct);
    } else {
        snprintf(best_detection, sizeof(best_detection), "CONFIDENCE: %d%%",
                confidence_pct);
    }

    // Prepare third line with timing info
    char timing_info[17];
    snprintf(timing_info, sizeof(timing_info), "INF: %ldms", result-
>inference_time_ms);

    // Select OLED on multiplexer before displaying
    pca9546_select(I2C_PORT, MUX_PORT_OLED);

    // Display results on OLED
    ssd1306_clear();

    // Display first line (detected classes)
    ssd1306_draw_string(0, 0, detected_classes, 1);

    // Display second line (best detection with confidence)
    ssd1306_draw_string(0, 12, best_detection, 1);

    // Display third line (timing info)
    ssd1306_draw_string(0, 24, timing_info, 1);

    // Update display
    ssd1306_show();
}

// Core 1 entry function - handles all ML inference
void core1_entry() {
```

```
    printf("Core 1: Starting TensorFlow Lite initialization...\n");

    // Initialize TensorFlow Lite
    model = tflite::GetModel(model_data);
    if (!model) {
        printf("Core 1: ERROR - Failed to get TFLite model\n");
        uint32_t error_cmd = CMD_ERROR;
        queue_try_add(&core1_to_core0_queue, &error_cmd);
        return;
    }

    // Create resolver with all required operations
    static tflite::MicroMutableOpResolver<16> resolver;
    resolver.AddConv2D();
    resolver.AddDepthwiseConv2D();
    resolver.AddFullyConnected();
    resolver.AddReshape();
    resolver.AddSoftmax();
    resolver.AddAdd();
    resolver.AddMul();
    resolver.AddAveragePool2D();
    resolver.AddMaxPool2D();
    resolver.AddMean();
    resolver.AddQuantize();
    resolver.AddDequantize();
    resolver.AddPad();
    resolver.AddConcatenation();
    resolver.AddRelu6();
    resolver.AddLogistic();

    // Create interpreter
    static tflite::MicroInterpreter static_interpreter(
        model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
    interpreter = &static_interpreter;

    // Allocate tensors
    TfLiteStatus allocate_status = interpreter->AllocateTensors();
    if (allocate_status != kTfLiteOk) {
        printf("Core 1: ERROR - Failed to allocate tensors: %d\n",
allocate_status);
        uint32_t error_cmd = CMD_ERROR;
        queue_try_add(&core1_to_core0_queue, &error_cmd);
        return;
    }

    // Get input and output tensors
    input = interpreter->input(0);
    output = interpreter->output(0);

    printf("Core 1: TensorFlow Lite initialized successfully\n");
    printf("Core 1: Input tensor type: %d, dims: %d x %d x %d x %d\n",
           input->type, input->dims->data[0], input->dims->data[1],
           input->dims->data[2], input->dims->data[3]);
    printf("Core 1: Output tensor type: %d, dims: %d x %d\n",
           output->type, output->dims->data[0], output->dims->data[1]);

    // Signal that initialization is complete
```

```
    uint32_t init_complete = CMD_INFERENCE_COMPLETE;
    queue_try_add(&core1_to_core0_queue, &init_complete);

    // Main processing loop
    uint32_t command;

    while (true) {
        // Wait for a command from Core 0
        if (queue_try_remove(&core0_to_core1_queue, &command)) {
            if (command == CMD_PROCESS_IMAGE) {
                // Get a lock on the shared memory
                uint32_t save = spin_lock_blocking(memory_lock);

                // Process the captured image
                bool process_success = process_image_for_inference(
                    g_capture_buffer, g_capture_size, g_process_buffer);

                // Release the lock
                spin_unlock(memory_lock, save);

                if (!process_success) {
                    printf("Core 1: Image processing failed\n");
                    uint32_t error_cmd = CMD_ERROR;
                    queue_try_add(&core1_to_core0_queue, &error_cmd);
                    continue;
                }

                // Fill input tensor with processed image
                fill_input_tensor(g_process_buffer);

                // Run inference with timing
                absolute_time_t inference_start = get_absolute_time();

                TfLiteStatus invoke_status = interpreter->Invoke();

                uint32_t inference_time_ms = absolute_time_diff_us(
                    inference_start, get_absolute_time()) / 1000;

                printf("Core 1: Inference took %ld ms\n", inference_time_ms);

                if (invoke_status != kTfLiteOk) {
                    printf("Core 1: Inference failed with status: %d\n",
invoke_status);
                    uint32_t error_cmd = CMD_ERROR;
                    queue_try_add(&core1_to_core0_queue, &error_cmd);
                    continue;
                }

                // Process results and store in shared memory
                save = spin_lock_blocking(memory_lock);

                // Get lock on shared inference result
                InferenceResult* result =
(InferenceResult*)&g_inference_result;
                result->inference_time_ms = inference_time_ms;

                // Process results based on output tensor type
```

```
                if (output->type == kTfLiteUInt8) {
                    uint8_t *results = output->data.uint8;

                    for (int i = 0; i < 7; i++) {
                        float score = results[i] / 255.0f;  // Normalize to
[0,1]
                        result->scores[i] = score;
                        // Apply thresholds
                        result->predictions[i] = (score >
g_improved_thresholds[i]) ? 1 : 0;
                    }
                }
                else if (output->type == kTfLiteInt8) {
                    // For int8 output, we need to dequantize manually
                    int8_t *int8_results = output->data.int8;
                    float scale = output->params.scale;
                    int zero_point = output->params.zero_point;

                    for (int i = 0; i < 7; i++) {
                        // Properly dequantize
                        float score = scale * (int8_results[i] - zero_point);
                        // Clip to 0-1 range
                        score = score < 0.0f ? 0.0f : (score > 1.0f ? 1.0f :
score);
                        result->scores[i] = score;
                        // Apply thresholds
                        result->predictions[i] = (score >
g_improved_thresholds[i]) ? 1 : 0;
                    }
                }
                else if (output->type == kTfLiteFloat32) {
                    float *float_results = output->data.f;

                    for (int i = 0; i < 7; i++) {
                        float score = float_results[i];
                        // Apply sigmoid if needed
                        if (score < 0.0f || score > 1.0f) {
                            score = 1.0f / (1.0f + expf(-score));
                        }
                        result->scores[i] = score;
                        // Apply thresholds
                        result->predictions[i] = (score >
g_improved_thresholds[i]) ? 1 : 0;
                    }
                }

                result->valid = true;
                spin_unlock(memory_lock, save);

                // Signal that inference is complete
                uint32_t complete_cmd = CMD_INFERENCE_COMPLETE;
                queue_try_add(&core1_to_core0_queue, &complete_cmd);
            }
        }

        // Small sleep to avoid tight loop
        sleep_us(100);
```

```
    }
}

// Main function - runs on Core 0
int main() {
    // Set inference result as invalid initially
    g_inference_result.valid = false;

    // Initialize all components
    if (!setup_hardware()) {
        printf("Hardware setup failed\n");
        while (true)
            sleep_ms(1000); // Halt
    }

    if (!setup_display()) {
        printf("Display setup failed\n");
        while (true)
            sleep_ms(1000); // Halt
    }

    if (!setup_camera()) {
        printf("Camera setup failed\n");
        while (true)
            sleep_ms(1000); // Halt
    }

    // Display status
    display_message("LOADING MODEL", "Please wait...", nullptr);

    // Launch Core 1 for ML processing
    multicore_launch_core1(core1_entry);

    // Wait for Core 1 to initialize TFLite
    printf("Core 0: Waiting for TensorFlow Lite initialization...\n");

    uint32_t response;
    bool init_success = false;
    absolute_time_t timeout = make_timeout_time_ms(10000);  // 10 second
timeout

    while (!init_success) {
        if (queue_try_remove(&core1_to_core0_queue, &response)) {
            if (response == CMD_INFERENCE_COMPLETE) {
                init_success = true;
            }
            else if (response == CMD_ERROR) {
                printf("Core 0: Error during TensorFlow initialization\n");
                display_message("MODEL ERROR", "Restart device", nullptr);
                while (true)
                    sleep_ms(1000); // Halt
            }
        }

        // Check for timeout
        if (absolute_time_diff_us(get_absolute_time(), timeout) <= 0) {
            printf("Core 0: Timeout waiting for TensorFlow
```

```
initialization\n");
            display_message("INIT TIMEOUT", "Restart device", nullptr);
            while (true)
                sleep_ms(1000); // Halt
        }

        sleep_ms(10);
    }

    // Show ready message
    display_message("SYSTEM READY", "Starting...", nullptr);
    sleep_ms(1000);

    // Initialize with a first result message
    display_message("AWAITING FIRST", "DETECTION", nullptr);

    // Flag to track if we need to update display
    bool display_needs_update = false;

    // Main processing loop
    while (true) {
        // Don't update display to "CAPTURING..." - keep previous results
visible
        // Instead just log to console
        printf("Core 0: Capturing new image...\n");

        // Capture image
        uint32_t capture_size = 0;
        bool capture_success = capture_image_to_buffer(
            g_capture_buffer, sizeof(g_capture_buffer), &capture_size);

        if (!capture_success) {
            printf("Core 0: Image capture failed\n");
            // Only update display in case of error
            display_message("CAPTURE ERROR", "Retrying...", nullptr);
            sleep_ms(1000);
            continue;
        }

        // Set the capture size in the shared memory
        uint32_t save = spin_lock_blocking(memory_lock);
        g_capture_size = capture_size;
        spin_unlock(memory_lock, save);

        // Don't update display to "PROCESSING..." - keep previous results
visible
        // Instead just log to console
        printf("Core 0: Processing image...\n");

        // Signal Core 1 to process the image
        uint32_t process_cmd = CMD_PROCESS_IMAGE;
        queue_try_add(&core0_to_core1_queue, &process_cmd);

        // Wait for Core 1 to complete processing
        bool processing_complete = false;
        bool processing_error = false;
        absolute_time_t timeout = make_timeout_time_ms(5000);  // 5 second
```

```
timeout

        while (!processing_complete && !processing_error) {
            // Check for response from Core 1
            uint32_t response;
            if (queue_try_remove(&core1_to_core0_queue, &response)) {
                if (response == CMD_INFERENCE_COMPLETE) {
                    processing_complete = true;
                    display_needs_update = true;  // Mark that we need to
update display
                }
                else if (response == CMD_ERROR) {
                    processing_error = true;
                }
            }

            // Check for timeout
            if (absolute_time_diff_us(get_absolute_time(), timeout) <= 0) {
                printf("Core 0: Timeout waiting for inference\n");
                display_message("INFERENCE TIMEOUT", "Retrying...", nullptr);
                processing_error = true;
            }

            sleep_ms(10);
        }

        if (processing_error) {
            sleep_ms(1000);
            continue;
        }

        // Only update the display when we have new results
        if (display_needs_update) {
            // Display results
            save = spin_lock_blocking(memory_lock);
            InferenceResult local_result;
            memcpy(&local_result, (void*)&g_inference_result,
sizeof(InferenceResult));
            spin_unlock(memory_lock, save);

            display_results(&local_result);
            display_needs_update = false;  // Reset flag
        }

        // Small delay before next capture to avoid CPU hogging
        // but not too long to keep detection responsive
        sleep_ms(100);
    }

    return 0;
}
```