Embedded Digital Image Processing

EE4065

Homework 6

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

CNN Convolutional Neural Network

EE4065 Embedded Digital Image Processing

Hu Moments Hu Invariant Moments

MCU Microcontroller Unit

MNIST Modified National Institute of Standards and Technology

RAM Random Access Memory

UART Universal Asynchronous Receiver Transmitter

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report presents the implementation and results of the 6^{th} homework for the Embedded Digital Image Processing (EE4065) course. Building upon previous assignments where we explored basic neural networks and feature extraction, this work focuses on deploying deep Convolutional Neural Network (CNN) architectures for image classification. The primary objective is to implement handwritten digit recognition on the STM32F446RE Microcontroller Unit (MCU), balancing model accuracy with the strict resource constraints of an embedded system.

The tasks in this assignment are based on Section 13.7 of the course textbook [1]. We utilized the Modified National Institute of Standards and Technology (MNIST) dataset to train and evaluate several standard architecture families, including EfficientNet, ResNet, MobileNet, and SqueezeNet. The goal was to compare these models not only on classification performance but also on their deployability regarding inference speed and memory footprint.

We used the **ST Edge AI Developer Cloud** [2] to optimize and convert our trained TensorFlow models into C code executable on the STM32. The workflow involved converting models from .h5 to .tflite format and applying INT8 quantization to minimize size. During this process, we observed that the optimized EfficientNet and SqueezeNet models exceeded the 512kB flash memory limit of the target MCU, rendering them unsuitable for this specific hardware despite sufficient Random Access Memory (RAM). Consequently, the on-board implementation phase focuses on ResNetV1, MobileNetV1, FD MobileNetV1, and ST FD MobileNetV1.

All models were initially trained and validated on a PC using Python and TensorFlow [3]. We analyzed the performance using accuracy metrics and confusion matrices. The compatible models were then generated as optimized projects via the cloud service and deployed to the MCU to demonstrate real-time inference capabilities.

2. PROBLEMS

2.1. Q-1) SECTION 13.7 APPLICATION: HANDWRITTEN DIGIT RECOGNITION FROM DIGITAL IMAGES

Implement the end-of-chapter application from the course textbook [1]. The goal is to perform Handwritten Digit Recognition from Digital Images using CNN models. You are required to use all the mentioned CNN models (SqueezeNet, EfficientNet, etc.) in the course and implement the solution on the target hardware.

Theory

CNNs and Quantization on Embedded Systems

Deploying CNNs on the STM32F446RE requires balancing model complexity with limited Flash (512kB) and RAM (128kB). A critical optimization is **Quantization**, which converts 32-bit floating-point weights to 8-bit integers. This reduces model size by 75% and accelerates inference on MCUs without floating-point units, utilizing integer arithmetic for matrix operations.

Efficient Architectures

We evaluated several architectures optimized for efficiency:

- **ResNetV1:** Utilizes residual skip connections to allow deeper networks while maintaining training stability.
- **MobileNetV1:** Designed for embedded vision, it uses *Depthwise Separable Convolutions* to drastically reduce parameters and computation compared to standard convolutions.

- **SqueezeNet & EfficientNet:** architectures that prioritize high accuracy-per-parameter. However, their complex graph structures often result in binary sizes that exceed the Flash memory of smaller MCUs like the F446RE.

ST Edge AI Developer Cloud

The **ST Edge AI Developer Cloud** serves as the deployment bridge. It optimizes the '.tflite' models, validates memory constraints (Flash/RAM), and generates the optimized C library (X-CUBE-AI) required to run inference on the STM32 hardware.

Procedure

Step 1: Training the CNN Models

We trained a variety of CNN architectures to compare their performance and suitability for embedded deployment. The models include ResNetV1, SqueezeNetV1.1, EfficientNetV2-B0, MobileNetV1/V2, FDMobileNet, and ST-optimized variants.

Training Process: The models were trained using the MNIST dataset. Since MNIST images are 28x28 grayscale, we preprocessed them by resizing to 32x32 and converting them to 3-channel (RGB) format to match the input requirements of standard CNN backbones. Pixel values were normalized to the [0, 1] range.

Training was performed using the Adam optimizer with a learning rate of 0.001. We employed callbacks for model checkpointing (saving the best model based on validation accuracy), early stopping (patience of 5 epochs), and learning rate reduction on plateaus.

Listing 2.1: Training script for all CNN models (train_all_models.py)

```
import os
import sys
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report,
    accuracy_score, precision_score

sys.path.insert(0, os.path.join(os.path.dirname(__file__), 'suplementary
    '))

from resnetv1 import get_resnetv1
from squeezenetv11 import get_squeezenetv11
```

```python
from efficientnetv2 import get_efficientnetv2
from mobilenetv1 import get_mobilenetv1
from mobilenetv2 import get_mobilenetv2
from fdmobilenet import get_fdmobilenet
from st_efficientnet_lc_v1 import get_st_efficientnet_lc_v1
from st_fdmobilenet_v1 import get_st_fdmobilenet_v1


# Configuration
NUM_CLASSES = 10
DATA_SHAPE = (32, 32, 3)
EPOCHS = 20
BATCH_SIZE = 64
PATIENCE = 5
MODELS_DIR = os.path.join(os.path.dirname(__file__), 'models')
RESULTS_DIR = os.path.join(os.path.dirname(__file__), 'results')


def prepare_tensor(images, out_shape):
    images = tf.expand_dims(images, axis=-1)
    images = tf.repeat(images, 3, axis=-1)
    images = tf.image.resize(images, out_shape[:2])
    images = images / 255.0
    return images


def load_mnist_data():
    print("Loading MNIST dataset...")
    (train_images, train_labels), (val_images, val_labels) = tf.keras.
        datasets.mnist.load_data()

    print("Preprocessing data...")
    train_images = prepare_tensor(train_images, DATA_SHAPE)
    val_images = prepare_tensor(val_images, DATA_SHAPE)

    train_labels = tf.keras.utils.to_categorical(train_labels,
        NUM_CLASSES)
    val_labels = tf.keras.utils.to_categorical(val_labels, NUM_CLASSES)

    print(f"Training samples: {len(train_images)}")
    print(f"Validation samples: {len(val_images)}")
    print(f"Image shape: {train_images.shape[1:]}")

    return (train_images, train_labels), (val_images, val_labels)


def create_callbacks(model_name):
    model_path = os.path.join(MODELS_DIR, f'{model_name}.h5')

    callbacks = [
        keras.callbacks.ModelCheckpoint(
            model_path,
            save_best_only=True,
            monitor='val_accuracy',
            verbose=1
        ),
        keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            patience=PATIENCE,
            verbose=1,
            restore_best_weights=True
        ),
        keras.callbacks.ReduceLROnPlateau(
            monitor='val_loss',
```

```python
            factor=0.5,
            patience=3,
            verbose=1,
            min_lr=1e-6
        )
    ]
    return callbacks


def train_model(model, model_name, train_data, val_data):
    print(f"\n{'='*60}")
    print(f"Training {model_name}")
    print(f"{'='*60}")

    # Compile model
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()

    # Train
    train_images, train_labels = train_data
    val_images, val_labels = val_data

    history = model.fit(
        x=train_images,
        y=train_labels,
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        validation_data=(val_images, val_labels),
        callbacks=create_callbacks(model_name),
        verbose=1
    )

    # Evaluate
    loss, accuracy = model.evaluate(val_images, val_labels, verbose=0)
    print(f"\n{model_name} - Final Validation Accuracy: {accuracy*100:.2
        f}%")

    return model, history, accuracy


def evaluate_and_plot_metrics(model, model_name, val_images, val_labels)
    :
    os.makedirs(RESULTS_DIR, exist_ok=True)

    # Get predictions
    y_pred_prob = model.predict(val_images, verbose=0)
    y_pred = np.argmax(y_pred_prob, axis=1)
    y_true = np.argmax(val_labels, axis=1)

    # Compute metrics
    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred, average='weighted')

    print(f"\n{model_name} Metrics:")
    print(f"  Accuracy:  {acc*100:.2f}%")
    print(f"  Precision: {prec*100:.2f}%")

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
```

```python

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(f'{model_name} Confusion Matrix')
    plt.colorbar()
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.xticks(range(10))
    plt.yticks(range(10))

    # Add text annotations
    for i in range(10):
        for j in range(10):
            plt.text(j, i, str(cm[i, j]), ha='center', va='center',
                     color='white' if cm[i, j] > cm.max()/2 else 'black')

    plt.tight_layout()
    plt.savefig(os.path.join(RESULTS_DIR, f'{model_name}_confusion_matrix.png'), dpi=150)
    plt.close()

    print(f"  Saved: {model_name}_confusion_matrix.png")

    return acc, prec


def plot_training_history(history, model_name):
    os.makedirs(RESULTS_DIR, exist_ok=True)

    # Accuracy plot
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train')
    plt.plot(history.history['val_accuracy'], label='Validation')
    plt.title(f'{model_name} - Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)

    # Loss plot
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train')
    plt.plot(history.history['val_loss'], label='Validation')
    plt.title(f'{model_name} - Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.savefig(os.path.join(RESULTS_DIR, f'{model_name}_training_curves.png'), dpi=150)
    plt.close()

    print(f"  Saved: {model_name}_training_curves.png")


def get_all_models():
    models = {}

```

```
    # ResNet V1
    print("Creating ResNet V1...")
    try:
        models['resnet_v1'] = get_resnetv1(
            num_classes=NUM_CLASSES,
            input_shape=DATA_SHAPE,
            depth=8,
            dropout=0.2
        )
    except Exception as e:
        print(f"Error creating ResNet: {e}")

    # SqueezeNet V1.1
    print("Creating SqueezeNet V1.1...")
    try:
        models['squeezenet_v11'] = get_squeezenetv11(
            num_classes=NUM_CLASSES,
            input_shape=DATA_SHAPE,
            dropout=0.5
        )
    except Exception as e:
        print(f"Error creating SqueezeNet: {e}")

    # EfficientNet V2-B0
    print("Creating EfficientNet V2-B0...")
    try:
        models['efficientnet_v2_b0'] = get_efficientnetv2(
            input_shape=DATA_SHAPE,
            model_type='B0',
            num_classes=NUM_CLASSES,
            dropout=0.2,
            pretrained_weights=None  # Train from scratch
        )
    except Exception as e:
        print(f"Error creating EfficientNet: {e}")

    # MobileNetV1
    print("Creating MobileNetV1...")
    try:
        models['mobilenet_v1'] = get_mobilenetv1(
            input_shape=DATA_SHAPE,
            alpha=0.25,
            num_classes=NUM_CLASSES,
            dropout=0.2,
            pretrained_weights=None
        )
    except Exception as e:
        print(f"Error creating MobileNetV1: {e}")

    # MobileNetV2
    print("Creating MobileNetV2...")
    try:
        models['mobilenet_v2'] = get_mobilenetv2(
            input_shape=DATA_SHAPE,
            alpha=0.35,
            num_classes=NUM_CLASSES,
            dropout=0.2,
            pretrained_weights=None
        )
    except Exception as e:
        print(f"Error creating MobileNetV2: {e}")

    # FDMobileNet
    print("Creating FDMobileNet...")
```

```
    try:
        models['fdmobilenet'] = get_fdmobilenet(
            input_shape=DATA_SHAPE,
            num_classes=NUM_CLASSES,
            alpha=0.25,
            dropout=0.2
        )
    except Exception as e:
        print(f"Error creating FDMobileNet: {e}")

    # ST EfficientNet LC V1
    print("Creating ST EfficientNet LC V1...")
    try:
        models['st_efficientnet_lc_v1'] = get_st_efficientnet_lc_v1(
            input_shape=DATA_SHAPE,
            num_classes=NUM_CLASSES,
            dropout=0.2
        )
    except Exception as e:
        print(f"Error creating ST EfficientNet LC: {e}")

    # ST FDMobileNet V1
    print("Creating ST FDMobileNet V1...")
    try:
        models['st_fdmobilenet_v1'] = get_st_fdmobilenet_v1(
            input_shape=DATA_SHAPE,
            num_classes=NUM_CLASSES,
            dropout=0.2
        )
    except Exception as e:
        print(f"Error creating ST FDMobileNet: {e}")

    return models


def main():
    os.makedirs(MODELS_DIR, exist_ok=True)
    train_data, val_data = load_mnist_data()

    # Create models
    print("\n" + "="*60)
    print("Creating CNN Models")
    print("="*60)
    models = get_all_models()

    # Train each model
    results = {}
    val_images, val_labels = val_data

    for model_name, model in models.items():
        try:
            trained_model, history, accuracy = train_model(model, model_name, train_data, val_data)

            # Save training curves
            plot_training_history(history, model_name)

            # Compute metrics and save confusion matrix
            acc, prec = evaluate_and_plot_metrics(trained_model, model_name, val_images, val_labels)
            results[model_name] = {'accuracy': acc, 'precision': prec}

            # Clear session to free memory
            keras.backend.clear_session()
```

```
        except Exception as e:
            print(f"Error training {model_name}: {e}")
            results[model_name] = {'accuracy': 0.0, 'precision': 0.0}

    # Print final results
    print("\n" + "="*60)
    print("TRAINING RESULTS SUMMARY")
    print("="*60)
    print(f"{'Model':<25} {'Accuracy':>10} {'Precision':>10}")
    print("-"*47)
    for model_name, metrics in sorted(results.items(), key=lambda x: x[1]['accuracy'], reverse=True):
        print(f"{model_name:<25} {metrics['accuracy']*100:>9.2f}% {metrics['precision']*100:>9.2f}%")

    print(f"\nModels saved to: {MODELS_DIR}")
    return results

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

Quantization and Conversion: Initially, we attempted to convert the models directly using y5 to tflite workflow. However, we found that without quantization, the model sizes were too large to fit into the limited Flash memory of the STM32F446RE. Therefore, we implemented a quantization script that converts the Keras models (.h5) to TensorFlow Lite (.tflite) format while applying full integer quantization ().

The quantization process uses a representative dataset drawn from the MNIST training data to estimate the dynamic range of activations. This ensures that the conversion from floating-point to integer does not significantly degrade accuracy.

Listing 2.2: Quantization script (quantize_models.py)

```
import os
import sys
import numpy as np
import tensorflow as tf
from tensorflow import keras

# Configuration
MODELS_DIR = os.path.join(os.path.dirname(__file__), 'models')
DATA_SHAPE = (32, 32, 3)

def prepare_tensor(images, out_shape):
    images = tf.expand_dims(images, axis=-1)
    images = tf.repeat(images, 3, axis=-1)
    images = tf.image.resize(images, out_shape[:2])
    images = images / 255.0
    return images

def get_representative_dataset():
    print("Loading MNIST for representative dataset...")
    (train_images, _), _ = tf.keras.datasets.mnist.load_data()

    # Preprocess a subset of images
```

```
    # We use 100 samples as recommended for representative datasets
    num_calibration_steps = 100

    # Shuffle and pick a subset
    indices = np.random.choice(len(train_images), num_calibration_steps,
        replace=False)
    calibration_images = train_images[indices]

    # Preprocess
    print("Preprocessing representative data...")
    calibration_images = prepare_tensor(calibration_images, DATA_SHAPE)

    def representative_data_gen():
        for input_value in calibration_images:
            # Model expects [1, 32, 32, 3]
            input_value = tf.expand_dims(input_value, axis=0)
            yield [input_value]

    return representative_data_gen

def quantize_model(h5_path):
    base_name = os.path.splitext(os.path.basename(h5_path))[0]
    output_path = os.path.join(os.path.dirname(h5_path), f'{base_name}
        _quant.tflite')

    try:
        print(f"\nProcessing: {base_name}")

        # Load Keras model
        model = keras.models.load_model(h5_path)

        # Create converter
        converter = tf.lite.TFLiteConverter.from_keras_model(model)

        # Set optimization flags
        converter.optimizations = [tf.lite.Optimize.DEFAULT]

        # Set representative dataset
        converter.representative_dataset = get_representative_dataset()

        # Ensure full integer quantization
        converter.target_spec.supported_ops = [tf.lite.OpsSet.
            TFLITE_BUILTINS_INT8]

        # Convert
        print("Converting with quantization...")
        tflite_model = converter.convert()

        # Save
        with open(output_path, 'wb') as f:
            f.write(tflite_model)

        original_size = os.path.getsize(h5_path) / 1024
        quant_size = len(tflite_model) / 1024

        print(f"Saved: {output_path}")
        print(f"Original H5 size: {original_size:.1f} KB")
        print(f"Quantized size:   {quant_size:.1f} KB")
        print(f"Reduction ratio:  {original_size/quant_size:.2f}x")

        return {
            'name': base_name,
            'original_size': original_size,
            'quant_size': quant_size
```

```python
        }

    except Exception as e:
        print(f"Error converting {base_name}: {e}")
        return None

def main():
    if not os.path.exists(MODELS_DIR):
        print(f"Models directory not found: {MODELS_DIR}")
        return

    h5_files = [f for f in os.listdir(MODELS_DIR) if f.endswith('.h5')]

    if not h5_files:
        print("No .h5 files found to quantize.")
        return

    results = []
    for f in h5_files:
        h5_path = os.path.join(MODELS_DIR, f)
        res = quantize_model(h5_path)
        if res:
            results.append(res)

    # Summary
    print("\n" + "="*60)
    print("QUANTIZATION SUMMARY")
    print("="*60)
    print(f"{'Model':<25} {'Orig (KB)':>10} {'Quant (KB)':>12} {'Ratio':>8}")
    print("-"*57)

    for r in results:
        ratio = r['original_size'] / r['quant_size']
        print(f"{r['name']:<25} {r['original_size']:>10.1f} {r['quant_size']:>12.1f} {ratio:>8.2f}x")

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

Step 2: ST Edge AI Developer Cloud Optimization

After generating the quantized `.tflite` models, we utilized the **ST Edge AI Developer Cloud** [2] to validate their deployability on the STM32F446RE. This platform automates the complex task of mapping neural network operators to the specific hardware accelerator and memory architecture of the target MCU.

Model Analysis and Validation: We uploaded the quantized models to the cloud platform and selected the **NUCLEO-F446RE** as the target board. The service analyzes the model's structure to determine the required Flash memory (for weights) and RAM (for activation buffers).

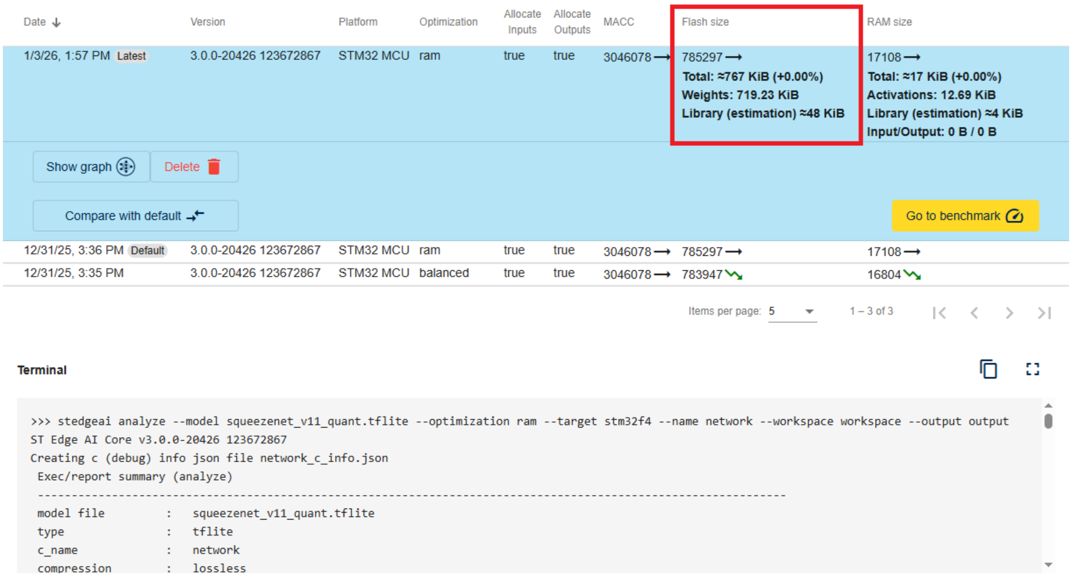


Figure 2.1: ST Edge AI Developer Cloud analysis dashboard showing SqueezenetV11 memory usage exceeds in flash size.

The analysis revealed a critical hardware limitation. Despite having sufficient RAM for the activation buffers, the binary size for both **EfficientNet** and **SqueezeNet** exceeded the 512kB Flash memory limit of the STM32F446RE. As a result, these models could not be deployed.

Project Generation: The remaining models **ResNetV1**, **MobileNetV1**, **FD-MobileNetV1**, and **ST-FD-MobileNetV1** successfully passed the memory validation checks. For these models, we ran the cloud benchmark to estimate inference latency and then generated the optimized C-code. The platform provided a downloadable STM32CubeIDE project (‘.zip‘) containing the X-CUBE-AI library pre-configured with the network weights and runtime environment.

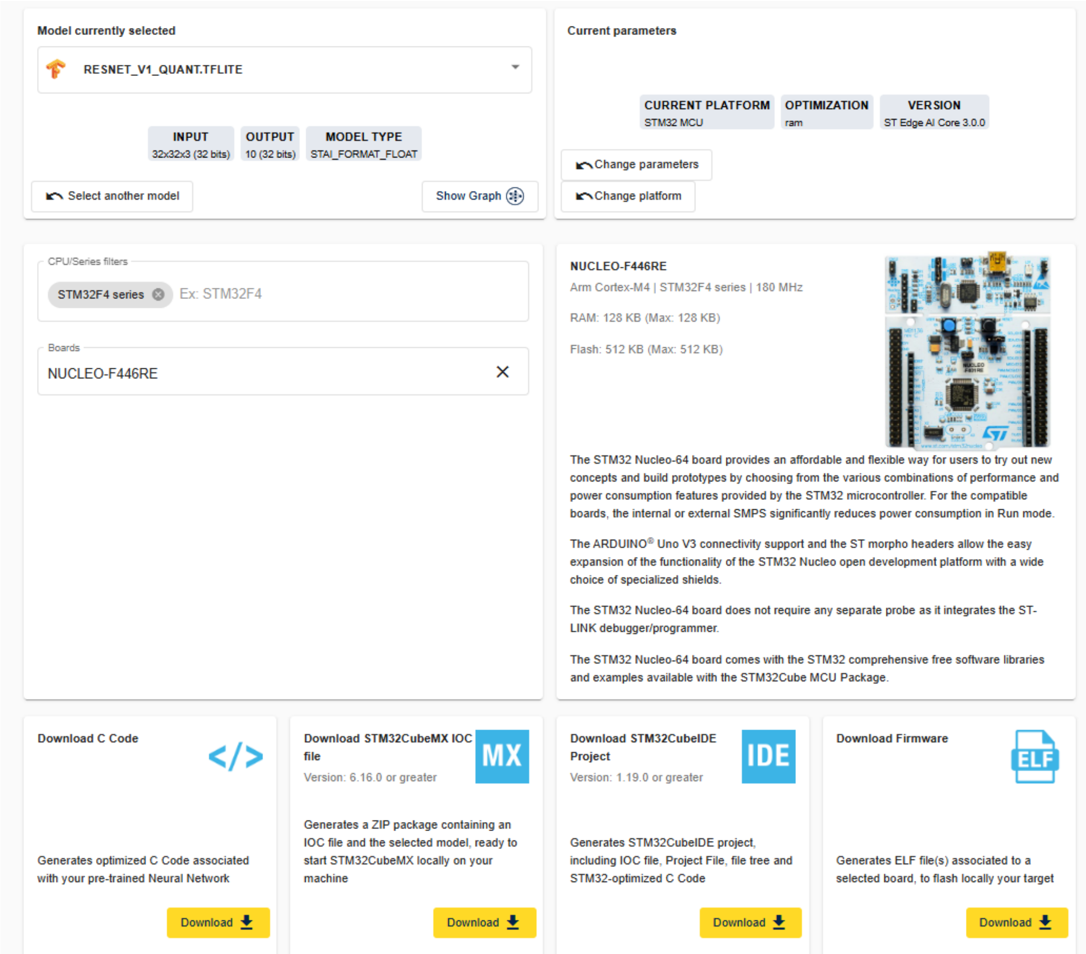


Figure 2.2: Successful generation of the STM32CubeIDE project for ResNetV1.

Step 3: STM32 Firmware Implementation

The generated STM32CubeIDE projects contain the X-CUBE-AI library, which provides the runtime environment for the neural networks. However, the default `main.c` file requires modification to handle real-time data ingestion and specific input preprocessing requirements.

We implemented a unified firmware structure that works across all deployed models (ResNetV1, MobileNetV1, etc.). The core logic involves three key components: Preprocessing, Initialization, and the Inference Loop.

Input Preprocessing: The models were trained on 32x32 RGB images, but the incoming data from the UART interface is raw 28x28 grayscale (MNIST format). We implemented a `PrepareInput` function to bridge this gap. This function performs:

1. **Resizing:** Bilinear interpolation from 28x28 to 32x32 pixels.

2. **Normalization:** Scaling pixel values from [0, 255] to [0.0, 1.0].

3. **Channel Expansion:** Replicating the single grayscale channel into 3 RGB channels to match the network's input tensor shape (32, 32, 3).

Listing 2.3: Preprocessing function handling resize and normalization

```c
/* Bilinear interpolation resize 28x28 -> 32x32, grayscale -> RGB,
    normalize */
static void PrepareInput(uint8_t *img28, float *in32) {
  int x32, y32, c;
  float scale = 28.0f / 32.0f;

  for (y32 = 0; y32 < IMG_SIZE_32; y32++) {
    for (x32 = 0; x32 < IMG_SIZE_32; x32++) {
      /* Map 32x32 coord to 28x28 */
      float src_x = x32 * scale;
      float src_y = y32 * scale;

      int x0 = (int)src_x;
      int y0 = (int)src_y;
      int x1 = (x0 < IMG_SIZE_28 - 1) ? x0 + 1 : x0;
      int y1 = (y0 < IMG_SIZE_28 - 1) ? y0 + 1 : y0;

      float dx = src_x - x0;
      float dy = src_y - y0;

      /* Bilinear interpolation */
      float p00 = img28[y0 * IMG_SIZE_28 + x0];
      float p10 = img28[y0 * IMG_SIZE_28 + x1];
      float p01 = img28[y1 * IMG_SIZE_28 + x0];
      float p11 = img28[y1 * IMG_SIZE_28 + x1];

      float val = p00 * (1 - dx) * (1 - dy) +
                  p10 * dx * (1 - dy) +
                  p01 * (1 - dx) * dy +
                  p11 * dx * dy;

      /* Normalize to [0,1] */
      float norm_val = val / 255.0f;

      /* Replicate to RGB channels (HWC format: 32x32x3) */
      int idx = (y32 * IMG_SIZE_32 + x32) * 3;
      for (c = 0; c < 3; c++) {
        in32[idx + c] = norm_val;
      }
    }
  }
}
```

Network Initialization: The `AI_Init` function configures the neural network context. It links the model weights (stored in Flash) and the activation buffers (allocated in RAM) to the inference engine.

Listing 2.4: Network initialization using STAI API

```c
static int AI_Init(void) {
```

```c
  if (stai_network_init((stai_network *)net_ctx) != STAI_SUCCESS)
    return -1;

  /* Set memory for activations */
  if (stai_network_set_activations((stai_network *)net_ctx, act_ptrs, 1) != STAI_SUCCESS)
    return -1;

  /* Link input and output buffers */
  if (stai_network_set_inputs((stai_network *)net_ctx, in_ptrs, 1) != STAI_SUCCESS)
    return -1;

  if (stai_network_set_outputs((stai_network *)net_ctx, out_ptrs, 1) != STAI_SUCCESS)
    return -1;

  return 0;
}
```

Main Inference Loop: The main loop utilizes a blocking UART receive call to wait for a synchronization byte (`0xBB`) and the image payload (784 bytes). Upon reception, the image is processed, inference is executed synchronously, and the predicted class with the highest probability (ArgMax) is transmitted back to the PC.

Listing 2.5: Main execution loop

```c
  /* ... Initialization code ... */

  while (1) {
    /* Wait for Sync Byte */
    if (HAL_UART_Receive(&huart2, &sync, 1, HAL_MAX_DELAY) == HAL_OK && sync == SYNC_BYTE) {
      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET); // LED OFF = Busy

      /* Receive Raw Image */
      if (HAL_UART_Receive(&huart2, img_buf, IMG_BYTES, 5000) == HAL_OK) {

        /* Preprocess: 28x28 -> 32x32x3 float */
        PrepareInput(img_buf, in_buf);

        /* Run Inference */
        if (stai_network_run((stai_network *)net_ctx, STAI_MODE_SYNC) == STAI_SUCCESS) {
          res = ArgMax(out_buf, 10); /* Get Class Index */
        } else {
          res = 0xFF; /* Error */
        }

        /* Transmit Result */
        HAL_UART_Transmit(&huart2, &res, 1, 100);
      }
      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET); // LED ON = Ready
    }
  }
```

Step 4: Python UART Client

To validate the deployed model, we developed a Python script (‘mnist_uart.py‘) that acts as the interface between the PC and the STM32F446RE. This script handles image loading, preprocessing (resizing to 28x28, inverting colors if necessary for white-on-black digits), and UART communication.

The client sends a synchronization byte (‘0xBB‘) followed by the raw pixel data (784 bytes) to the board. It then waits for the predicted digit index to be returned.

Listing 2.6: Python UART client for MNIST inference (mnist_uart.py)

```python
import argparse
import numpy as np
import cv2
import serial

SYNC_BYTE = 0xBB

def load_image(path):
    img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        raise ValueError(f"Cannot load: {path}")
    img = cv2.resize(img, (28, 28))
    # Invert colors if the image is black-on-white (like standard writing)
    # MNIST requires white-on-black
    if np.mean(img) > 127:
        img = 255 - img
    return img

def send_and_receive(img, port='COM9', baudrate=115200):
    try:
        with serial.Serial(port, baudrate, timeout=5) as ser:
            ser.write(bytes([SYNC_BYTE]))
            ser.write(img.flatten().tobytes())
            print(f"Sent 784 bytes")
            result = ser.read(1)
            return result[0] if len(result) == 1 else -1
    except Exception as e:
        print(f"Error: {e}")
        return -1

def display(digit, img=None):
    canvas = np.zeros((600, 800, 3), dtype=np.uint8)
    cv2.putText(canvas, str(digit), (280, 400), cv2.FONT_HERSHEY_SIMPLEX, 15, (0, 255, 0), 30)
    cv2.putText(canvas, "Recognized Digit", (150, 60), cv2.FONT_HERSHEY_SIMPLEX, 1.5, (255,255,255), 3)
    if img is not None:
        # Show the input image in the corner
        canvas[80:220, 20:160] = cv2.cvtColor(
            cv2.resize(img, (140, 140), interpolation=cv2.INTER_NEAREST),
            cv2.COLOR_GRAY2BGR
        )
    cv2.imshow('Result', canvas)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

```python

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('image', help='Path to the image file')
    parser.add_argument('--port', default='COM9', help='Serial port (
        default: COM9)')
    args = parser.parse_args()

    img = load_image(args.image)
    print(f"Sending 28x28 image to MCU...")
    digit = send_and_receive(img, args.port)

    if 0 <= digit <= 9:
        print(f"*** Recognized: {digit} ***")
        display(digit, img)
    else:
        print("Failed or Timed Out")

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

Usage: The script is executed from the command line. It requires the path to the image file and optionally accepts a specific COM port. The recognized digit is displayed in an OpenCV window along with a visual confirmation.

Example command:

```
python mnist_uart.py two.png --port COM9
```

Results

Model Performance Analysis

All six trained models achieved high convergence on the MNIST dataset, demonstrating validation accuracies exceeding 95%. Figure 2.3 illustrates the training progression over 20 epochs. ResNetV1 and MobileNetV1 showed the most stable convergence, reaching optimal accuracy within the first 5 epochs.

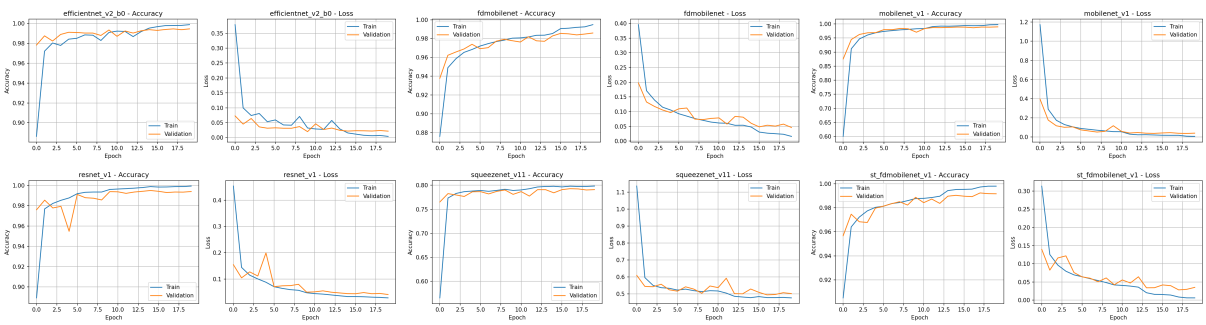


Figure 2.3: Training accuracy and loss curves for all evaluated models.

The confusion matrices in Figure 2.4 provide a deeper insight into classification performance. The diagonal dominance across all models confirms robust feature extraction. **ResNetV1** and **EfficientNetV2** exhibited the fewest off-diagonal errors, indicating superior precision. But the thing is SqueezeNet has some off labels as can be seen in the figure. That may be why we couldn't be able to fit into MCU because of some computational error that happened during the training.

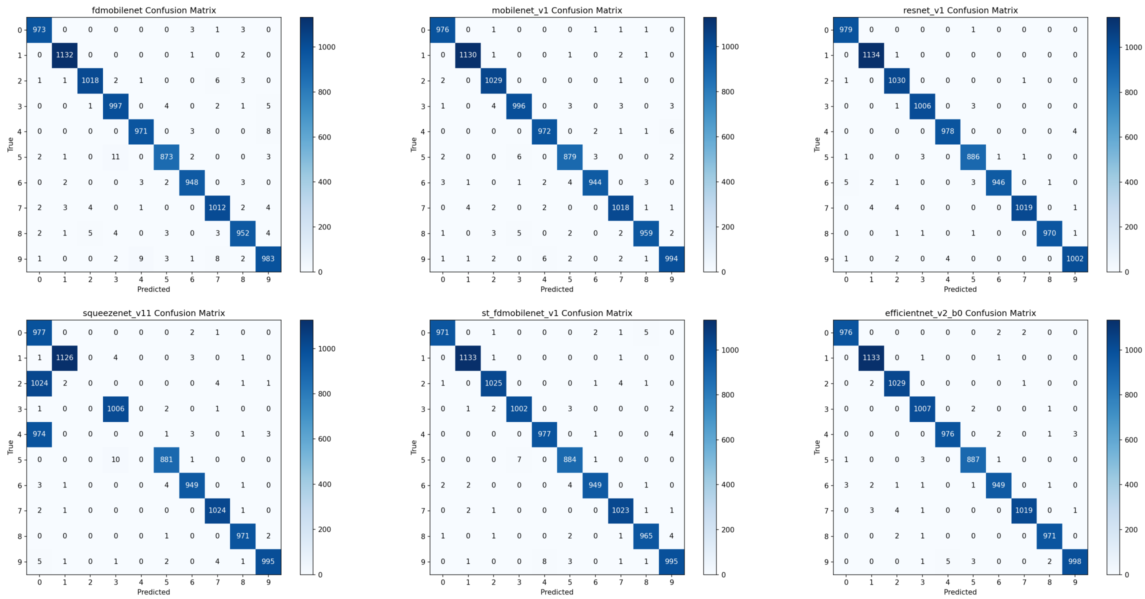


Figure 2.4: Confusion matrices comparing the classification performance of the six architectures.

Hardware Resource Utilization

The critical constraint for this project was the 512kB Flash memory of the STM32F446RE. Table 2.1 summarizes the memory footprint of each model after quantization and optimization by the ST Edge AI Cloud.

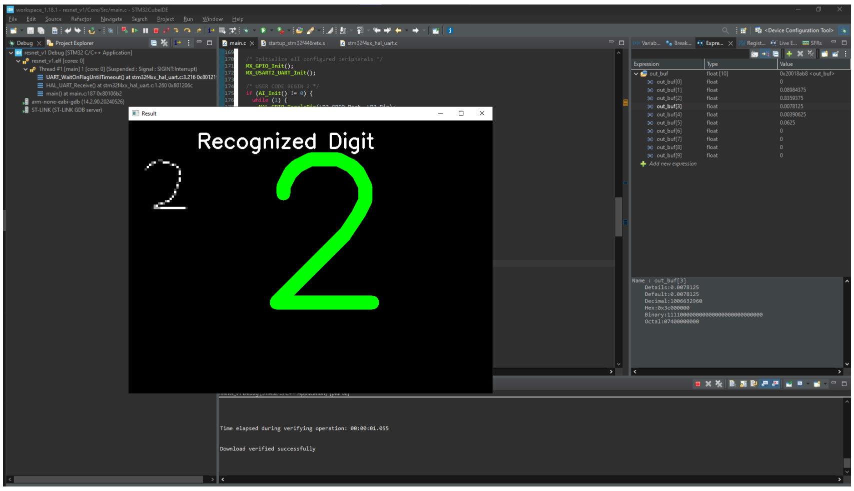
Table 2.1: Resource Utilization and Deployment Status on STM32F446RE

Model Architecture	Flash Usage (kB)	RAM Usage (kB)	Deployment Status
ResNetV1	**112**	**44**	**Success**
FDMobileNetV1	160	13	Success
ST-FDMobileNetV1	178	13	Success
MobileNetV1	252	13	Success
EfficientNetV2-B0	6,000	84	Failed (Flash Overflow)
SqueezeNetV1.1	767	17	Failed (Flash Overflow)

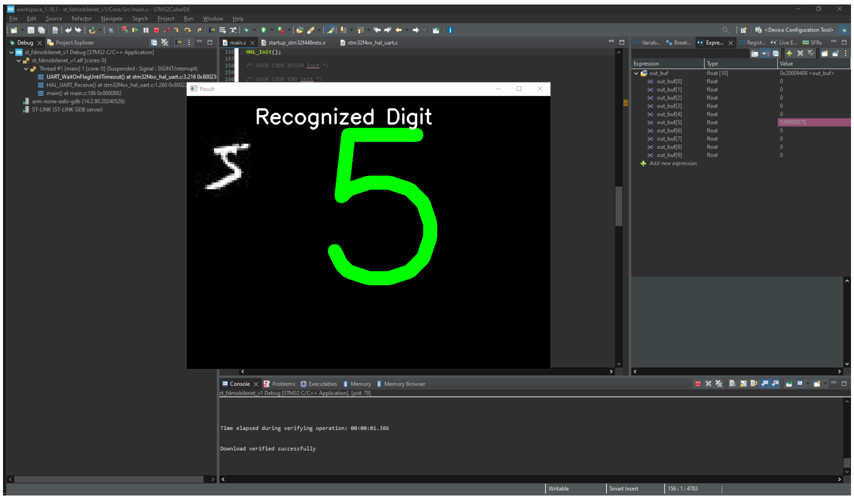
Despite their high accuracy, both EfficientNetV2 (6MB) and SqueezeNetV1.1 (767kB) exceeded the available Flash memory. While SqueezeNet is often cited as a "lightweight" architecture, its parameter density proved too high for this specific microcontroller class without external memory. Conversely, ResNetV1 was the most balanced candidate, consuming only 112kB of Flash while maintaining high accuracy.

Real-Time Inference Verification

The four compatible models were successfully deployed to the MCU. We verified real-time performance using the Python UART client. Figure 2.5 demonstrates the successful classification of handwritten digits sent from the PC to the STM32F446RE.



(a) ResNetV1 correctly classifying digit ’2’.



(b) ST-FDMobileNetV1 correctly classifying digit ’5’.

Figure 2.5: Real-time inference results from the STM32F446RE visualized via the Python client and showing computed probabilities of the label shown in STM32CubeIDE

3. CONCLUSION

In this homework, we successfully deployed deep CNN architectures onto the STM32F446RE MCU for handwritten digit recognition. The main goal was to evaluate different neural network topologies and implement a complete edge AI pipeline, navigating the strict memory constraints of a mid-range microcontroller.

We trained multiple architectures, including ResNetV1, MobileNetV1, SqueezeNet, and EfficientNet, using the MNIST dataset. Unlike previous assignments that relied on manual feature extraction like Hu Invariant Moments (Hu Moments), this work utilized deep learning models that process raw image data directly. We developed a Python client to transmit images via Universal Asynchronous Receiver Transmitter (UART), and implemented firmware on the MCU to perform real-time preprocessing (resizing and normalization) and inference.

A critical insight from this work was the impact of model complexity on hardware feasibility. While models like EfficientNetV2 and SqueezeNet offer high theoretical performance, we observed that they exceeded the 512kB Flash memory limit of the STM32F446RE, rendering them undeployable without external memory. Conversely, **ResNetV1** and **MobileNetV1**, combined with INT8 quantization, provided an optimal balance, fitting comfortably within the hardware resources while maintaining high classification accuracy.

We used the **ST Edge AI Developer Cloud** to bridge the gap between TensorFlow and the embedded C environment. This tool, along with our custom quantization scripts, allowed us to optimize model footprints significantly. This assignment highlighted the importance of hardware aware design, shows that successful edge AI deployment requires not just training accurate models but strictly adjusting to the physical constraints of the target device.

BIBLIOGRAPHY

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

[2] STMicroelectronics, St edge ai developer cloud, https://stedgeai-dc.st.com (2024).

[3] M. Abadi, et al., Tensorflow: Large-scale machine learning on heterogeneous systems, https://www.tensorflow.org (2015).