MARMARA
UNIVERSITY

Embedded Digital Image Processing

EE4065

Homework 5

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

EE4065 Embedded Digital Image Processing

FSDD Free Spoken Digit Dataset

Hu Moments Hu Invariant Moments

KWS Keyword Spotting

MCU Microcontroller Unit

MFCC Mel-Frequency Cepstral Coefficients

ML Machine Learning

MLP Multilayer Perceptron

UART Universal Asynchronous Receiver Transmitter

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report presents the implementation and results of the 5^{th} homework for the Embedded Digital Image Processing (EE4065) course. In previous homeworks, we focused on image processing and training neural networks on a PC. In this work, we take the next step by deploying trained models onto an embedded system. The primary objective is to implement real-time pattern recognition on the STM32F446RE Microcontroller Unit (MCU) using Universal Asynchronous Receiver Transmitter (UART) communication with a PC.

The tasks in this assignment are based on Sections 12.8 and 12.9 of the course textbook [1]. In the first part, we implement KWS to recognize spoken digits (0-9) from audio signals. We extract Mel-Frequency Cepstral Coefficients (MFCC) features from audio files using the librosa library. In the second part, we implement handwritten digit recognition using Hu Invariant Moments (Hu Moments) as features. A key difference is that the MCU computes the Hu Moments directly from raw image data.

We used the **ST Edge AI Developer Cloud** [2] to convert our TensorFlow models to optimized C code for the STM32. This cloud service analyzes the model, checks if it fits on the target MCU, and generates a complete STM32CubeIDE project. We then modified the generated code to implement a custom UART protocol for real-time inference.

All models were trained on a PC using Python and TensorFlow [3]. We analyzed the performance using accuracy metrics, loss graphs, and confusion matrices. The trained models were then deployed to the MCU for real-time inference testing.

2. PROBLEMS

2.1. Q-1) 12.8 APPLICATION: KEYWORD SPOTTING FROM AUDIO SIGNALS

Theory

KWS is a method to find specific words in audio. In this project, we build a system that can hear spoken digits (0 to 9) and tell us which digit was spoken. We use MFCC to change sound into numbers that a computer can understand [4]. MFCC are good for speech because they copy how human ears work.

The system has two parts:

1. **PC Side:** A Python script gets audio, makes MFCC features, and sends them to the board using UART.
2. **MCU Side:** The STM32F446RE board runs a neural network and sends back the result.

Procedure

Step 1: Training the Model

We use the Free Spoken Digit Dataset (FSDD) dataset which has recordings of spoken digits. We train a Multilayer Perceptron (MLP) model in Python with TensorFlow [3]. The model takes 260 MFCC values (13 coefficients × 20 time frames) as input.

Listing 2.1: Training script for audio model (train_q1.py)

```
import tensorflow as tf
import numpy as np
import librosa
import os
import glob
```

```
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay


DATASET_PATH = "C:/Users/mustrelax/Desktop/EE4065/Homework 5/train/dataset"
SAMPLE_RATE = 8000
DURATION = 1.0
N_MFCC = 13
MAX_FRAMES = 20

def extract_features(file_path):
    # Load audio
    audio, sr = librosa.load(file_path, sr=SAMPLE_RATE)

    # Pad or truncate to specific duration to ensure fixed input size
    max_len = int(SAMPLE_RATE * DURATION)
    if len(audio) < max_len:
        audio = np.pad(audio, (0, max_len - len(audio)))
    else:
        audio = audio[:max_len]

    # Compute MFCC
    mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=N_MFCC)
    if mfccs.shape[1] < MAX_FRAMES:
        mfccs = np.pad(mfccs, ((0, 0), (0, MAX_FRAMES - mfccs.shape[1]))
            )
    else:
        mfccs = mfccs[:, :MAX_FRAMES]

    return mfccs.flatten()

print("Loading Dataset...")
files = glob.glob(os.path.join(DATASET_PATH, "*.wav"))
X = []
y = []

for file_path in files:
    filename = os.path.basename(file_path)
    # Filename format: digit_speaker_index.wav
    label = int(filename.split('_')[0])

    features = extract_features(file_path)
    X.append(features)
    y.append(label)

X = np.array(X)
y = np.array(y)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

print(f"Input Shape: {X_train.shape[1]}")

# --- MODEL DEFINITION ---
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.
        shape[1],)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

```
66
67 model.compile(optimizer='adamW', loss='sparse_categorical_crossentropy',
       metrics=['accuracy'])
68
69 print("Training Audio Model...")
70 history = model.fit(X_train, y_train, epochs=200, batch_size=32,
       validation_data=(X_test, y_test))
71
72 # --- TFLITE CONVERSION ---
73 converter = tf.lite.TFLiteConverter.from_keras_model(model)
74 converter.optimizations = [tf.lite.Optimize.DEFAULT]
75 tflite_model = converter.convert()
76
77 with open('audio_model.tflite', 'wb') as f:
78     f.write(tflite_model)
79
80 print("SAVED: audio_model.tflite")
81 print("-" * 30)
82 print(f"#define MFCC_NUM_COEFFS {N_MFCC}")
83 print(f"#define MFCC_NUM_FRAMES {MAX_FRAMES}")
84 print("-" * 30)
85
86
87 plt.figure(figsize=(12, 5))
88
89 plt.subplot(1, 2, 1)
90 plt.plot(history.history['loss'], label='Training Loss')
91 plt.plot(history.history['val_loss'], label='Validation Loss')
92 plt.title('Audio Model Loss')
93 plt.xlabel('Epochs')
94 plt.ylabel('Loss')
95 plt.legend()
96 plt.grid(True)
97
98 plt.subplot(1, 2, 2)
99 plt.plot(history.history['accuracy'], label='Training Accuracy')
100 plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
101 plt.title('Audio Model Accuracy')
102 plt.xlabel('Epochs')
103 plt.ylabel('Accuracy')
104 plt.legend()
105 plt.grid(True)
106
107 plt.tight_layout()
108 plt.show()
109
110 y_pred_probs = model.predict(X_test)
111 y_pred_classes = np.argmax(y_pred_probs, axis=1)
112
113 cm = confusion_matrix(y_test, y_pred_classes)
114
115 disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.
       arange(10))
116 fig, ax = plt.subplots(figsize=(8, 8))
117 disp.plot(cmap=plt.cm.Blues, ax=ax)
118 plt.title("Audio Model Confusion Matrix")
119 plt.show()
```

Step 2: ST Edge AI Developer Cloud

After training, we upload the `audio_model.tflite` file to **ST Edge AI Developer Cloud** [2]. This is a free online tool from STMicroelectronics. It does:

- Checks if the model fits on our MCU
- Makes the model faster for STM32
- Creates a ready-to-use STM32CubeIDE project

We download the project and open it in STM32CubeIDE.

Step 3: STM32 Firmware

We change the `main.c` file to work with UART. The default code is for testing; we need real-time use. We write three helper functions:

AI_Init Function: This function starts the neural network. It sets up memory for the network to use.

Listing 2.2: AI_Init - starts the neural network

```
static int AI_Init(void) {
    stai_return_code ret;

    // Start the network
    ret = stai_network_init((stai_network*)network_context);
    if (ret != STAI_SUCCESS) return -1;

    // Give it memory for calculations
    ret = stai_network_set_activations((stai_network*)network_context,
                                       activation_buffers,
                                       STAI_NETWORK_ACTIVATIONS_NUM);
    if (ret != STAI_SUCCESS) return -1;

    // Tell it where input data is
    ret = stai_network_set_inputs((stai_network*)network_context,
                                  input_ptrs, STAI_NETWORK_IN_NUM);
    if (ret != STAI_SUCCESS) return -1;

    // Tell it where to put output
    ret = stai_network_set_outputs((stai_network*)network_context,
                                   output_ptrs, STAI_NETWORK_OUT_NUM);
    if (ret != STAI_SUCCESS) return -1;

    return 0;
}
```

AI_Run Function: This function runs the neural network one time and waits for the result.

Listing 2.3: AI_Run - runs inference

```
static int AI_Run(void) {
    stai_return_code ret;
    ret = stai_network_run((stai_network*)network_context,
        STAI_MODE_SYNC);
    if (ret != STAI_SUCCESS) return -1;
    return 0;
}
```

FindArgMax Function: This function looks at the 10 output values and finds which one is biggest. The position of the biggest value is the predicted digit.

Listing 2.4: FindArgMax - finds the predicted digit

```
static uint8_t FindArgMax(float *data, int size) {
    uint8_t max_idx = 0;
    float max_val = data[0];

    for (int i = 1; i < size; i++) {
        if (data[i] > max_val) {
            max_val = data[i];
            max_idx = i;
        }
    }
    return max_idx;
}
```

Main Loop: The main loop waits for data from Python. When it gets the sync byte (0xAA), it knows data is coming.

Listing 2.5: Main loop - waits for data and runs inference

```
#define SYNC_BYTE        0xAA
#define INPUT_SIZE       260    // 13 MFCC * 20 frames
#define INPUT_SIZE_BYTES 1040   // 260 * 4 bytes (float32)

STAI_ALIGNED(32) static float input_buffer[INPUT_SIZE];
STAI_ALIGNED(32) static float output_buffer[10];

int main(void) {
    uint8_t sync_byte, result;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();

    // Start neural network
    if (AI_Init() != 0) {
        while (1) { HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
           HAL_Delay(100); }
    }
```

```
    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET); // LED on = ready

    while (1) {
        // Wait for sync byte from PC
        if (HAL_UART_Receive(&huart2, &sync_byte, 1, HAL_MAX_DELAY) == HAL_OK) {
            if (sync_byte == SYNC_BYTE) {
                HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET); // LED off

                // Get MFCC features (1040 bytes)
                if (HAL_UART_Receive(&huart2, (uint8_t*)input_buffer,
                                     INPUT_SIZE_BYTES, 5000) == HAL_OK) {
                    // Run neural network
                    if (AI_Run() == 0) {
                        result = FindArgMax(output_buffer, 10);
                    } else {
                        result = 0xFF; // Error
                    }
                    // Send result back
                    HAL_UART_Transmit(&huart2, &result, 1, 100);
                }
                HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
                    // LED on
            }
        }
    }
}
```

Step 4: Python UART Client

We write a Python script to use the system. It loads audio, makes MFCC, sends via UART, and shows the result with OpenCV.

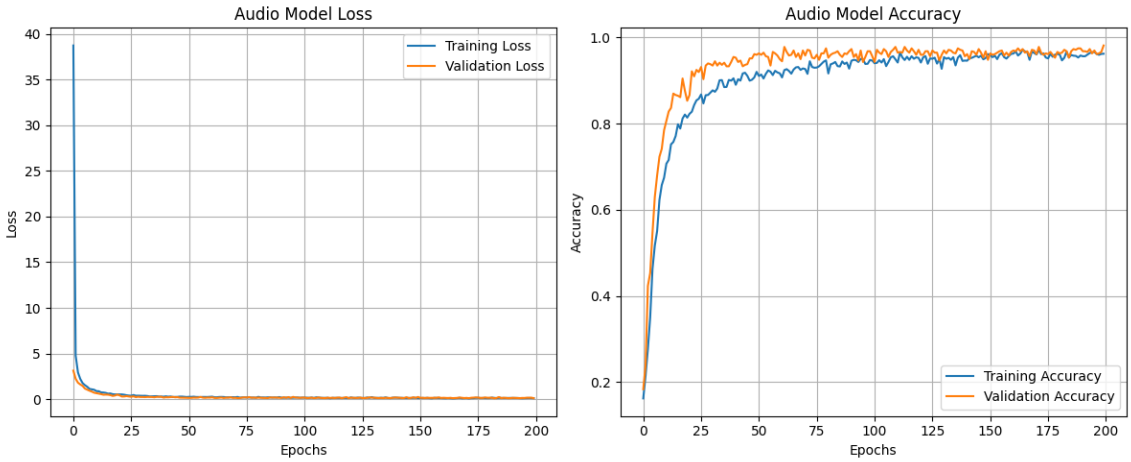
Listing 2.6: Python UART client for audio (kws_uart.py)

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

SAMPLE_RATE = 8000
N_MFCC = 13
MAX_FRAMES = 20
INPUT_SIZE = N_MFCC * MAX_FRAMES
SYNC_BYTE = 0xAA

def extract_features(file_path):
    audio, sr = librosa.load(file_path, sr=SAMPLE_RATE)
    max_len = int(SAMPLE_RATE * 1.0)
    if len(audio) < max_len:
        audio = np.pad(audio, (0, max_len - len(audio)))
    else:
        audio = audio[:max_len]
    mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=N_MFCC)
    if mfccs.shape[1] < MAX_FRAMES:
        mfccs = np.pad(mfccs, ((0, 0), (0, MAX_FRAMES - mfccs.shape[1]))
```

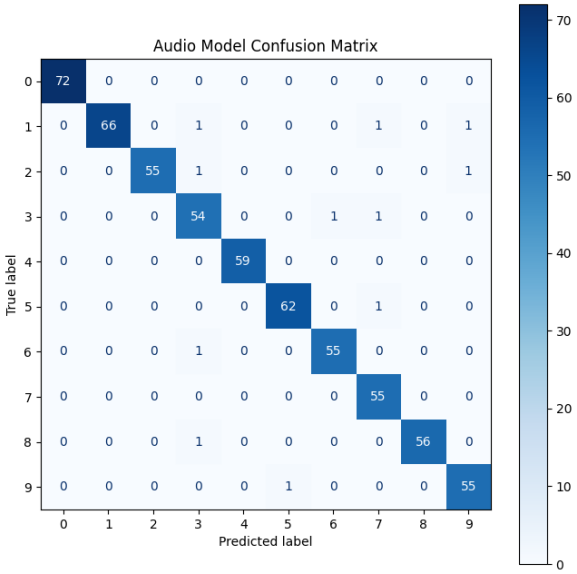
```
            )
    else:
        mfccs = mfccs[:, :MAX_FRAMES]
    return mfccs.flatten().astype(np.float32)

def send_features_and_receive_result(features, port='COM9'):
    with serial.Serial(port, 115200, timeout=5) as ser:
        ser.write(bytes([SYNC_BYTE]))
        ser.write(features.astype('<f4').tobytes())
        result = ser.read(1)
        return result[0] if len(result) == 1 else -1

def display_digit(digit):
    img = np.zeros((600, 800, 3), dtype=np.uint8)
    cv2.putText(img, str(digit), (280, 400), cv2.FONT_HERSHEY_SIMPLEX,
       15, (0,255,0), 30)
    cv2.putText(img, "Recognized Digit", (200, 60), cv2.
       FONT_HERSHEY_SIMPLEX, 1.5, (255,255,255), 3)
    cv2.imshow('Keyword Spotting Result', img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('wav_file')
    parser.add_argument('--port', default='COM9')
    args = parser.parse_args()

    features = extract_features(args.wav_file)
    digit = send_features_and_receive_result(features, args.port)
    if 0 <= digit <= 9:
        display_digit(digit)

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

Results

Figure 2.1 shows our training results. The model learns well.



(a) Training Loss and Accuracy



(b) Confusion Matrix

Figure 2.1: Training results for the KWS model.

We tested the system with real audio files. Figure 2.2 shows the inference result on our computer.

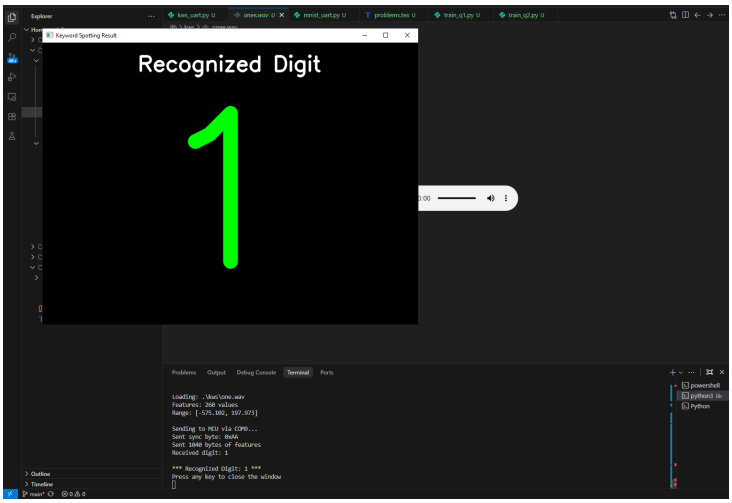


Figure 2.2: OpenCV window showing recognized digit from audio file.

Figure 2.3 shows the STM32CubeIDE debug window with the output buffer values.

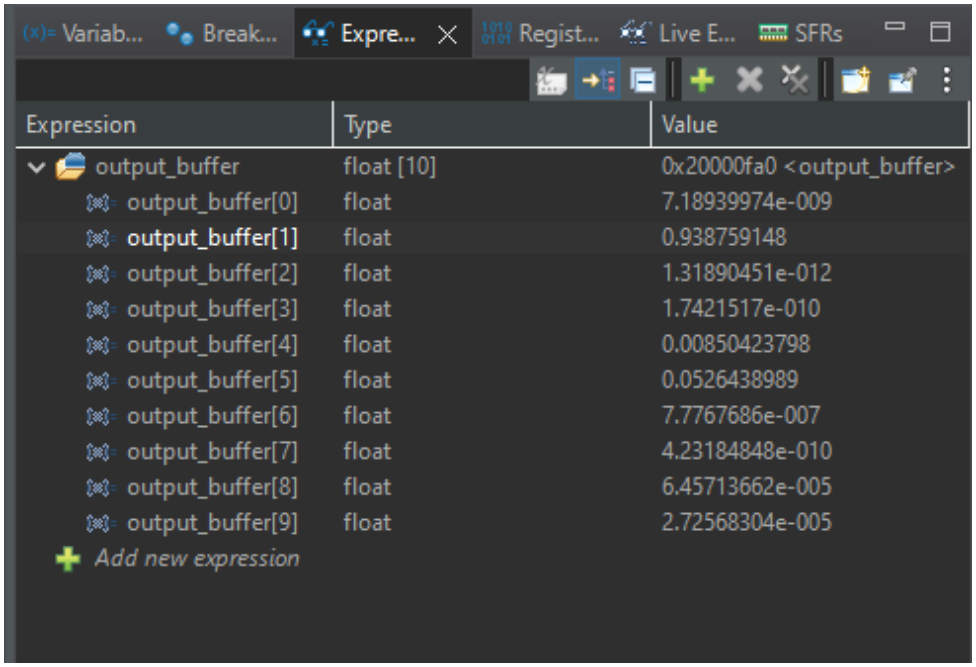


Figure 2.3: STM32CubeIDE Expressions window showing network output values.

2.2. Q-2) 12.9 APPLICATION: HANDWRITTEN DIGIT RECOGNITION

Theory

In this part, we make a system that can read handwritten digits from images. We use **Hu Moments** [5] as features. Hu Moments are 7 special numbers from an image. They do not change when you move, rotate, or resize the shape. This makes them good for digit recognition.

The main difference from Q1: The MCU calculates Hu Moments from the raw image. Python only sends the picture.

Procedure

Step 1: Training the Model

We use the MNIST dataset [6]. We get Hu Moments from each image using OpenCV [7].

Listing 2.7: Training script for MNIST model (train_q2.py)

```
import numpy as np
import cv2
import tensorflow as tf

(train_images, train_labels), (test_images, test_labels) = \
    tf.keras.datasets.mnist.load_data()

def extract_hu_moments(images):
    hu_features = np.empty((len(images), 7))
    for i, img in enumerate(images):
        moments = cv2.moments(img, binaryImage=True)
        hu = cv2.HuMoments(moments).reshape(7)
        hu_features[i] = hu
    return hu_features

train_hu = extract_hu_moments(train_images)
test_hu = extract_hu_moments(test_images)

# Normalize features
features_mean = np.mean(train_hu, axis=0)
features_std = np.std(train_hu, axis=0)
features_std[features_std == 0] = 1.0
train_hu = (train_hu - features_mean) / features_std
test_hu = (test_hu - features_mean) / features_std

model = tf.keras.Sequential([
    tf.keras.layers.Dense(100, input_shape=[7], activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adamW', loss='sparse_categorical_crossentropy',
```

```
    metrics=['accuracy'])
model.fit(train_hu, train_labels, epochs=200, batch_size=32,
    validation_split=0.1)

# Print values for STM32
print("float features_mean[7] = {", ", ".join(f"{x:.6f}f" for x in
    features_mean), "};")
print("float features_std[7] = {", ", ".join(f"{x:.6f}f" for x in
    features_std), "};")

converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
with open('mnist_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Step 2: ST Edge AI Developer Cloud

Same as Q1: We upload `mnist_model.tflite` to ST Edge AI Developer Cloud [2] and download the STM32CubeIDE project.

Step 3: STM32 Firmware with Hu Moments

The MCU does more work here. It calculates Hu Moments from the image pixels. We write four helper functions:

ComputeHuMoments Function: This is the most complex function. It calculates the 7 Hu Moments from a 28x28 image. The steps are:

1. Calculate raw moments (m00, m10, m01, etc.)
2. Find the center of the image (xc, yc)
3. Calculate central moments (mu20, mu11, etc.)
4. Calculate normalized central moments (nu20, nu11, etc.)
5. Calculate the 7 Hu invariants

Listing 2.8: ComputeHuMoments - calculates 7 Hu Moments from image

```
static void ComputeHuMoments(uint8_t *img, float *hu) {
    double m00=0, m10=0, m01=0, m20=0, m11=0, m02=0;
    double m30=0, m21=0, m12=0, m03=0;
    int x, y;

    // Step 1: Calculate raw moments
    for (y = 0; y < IMG_SIZE; y++) {
        for (x = 0; x < IMG_SIZE; x++) {
```

```
            // binaryImage=True: pixel > 0 becomes 1, else 0
            double p = (img[y * IMG_SIZE + x] > 0) ? 1.0 : 0.0;
            m00 += p;
            m10 += x * p;
            m01 += y * p;
            m20 += x * x * p;
            m11 += x * y * p;
            m02 += y * y * p;
            m30 += x * x * x * p;
            m21 += x * x * y * p;
            m12 += x * y * y * p;
            m03 += y * y * y * p;
        }
    }

    if (m00 < 1e-10) { for (int i=0; i<7; i++) hu[i]=0; return; }

    // Step 2: Find center
    double xc = m10 / m00, yc = m01 / m00;

    // Step 3: Central moments
    double mu20 = m20 - xc * m10;
    double mu11 = m11 - xc * m01;
    double mu02 = m02 - yc * m01;
    double mu30 = m30 - 3*xc*m20 + 2*xc*xc*m10;
    double mu21 = m21 - 2*xc*m11 - yc*m20 + 2*xc*xc*m01;
    double mu12 = m12 - 2*yc*m11 - xc*m02 + 2*yc*yc*m10;
    double mu03 = m03 - 3*yc*m02 + 2*yc*yc*m01;

    // Step 4: Normalized central moments
    double n2 = m00 * m00;
    double n3 = n2 * sqrt(m00);
    double nu20 = mu20/n2, nu11 = mu11/n2, nu02 = mu02/n2;
    double nu30 = mu30/n3, nu21 = mu21/n3, nu12 = mu12/n3, nu03 = mu03/
        n3;

    // Step 5: Hu invariants
    double t0 = nu30 + nu12, t1 = nu21 + nu03;
    hu[0] = (float)(nu20 + nu02);
    hu[1] = (float)((nu20-nu02)*(nu20-nu02) + 4*nu11*nu11);
    hu[2] = (float)((nu30-3*nu12)*(nu30-3*nu12) + (3*nu21-nu03)*(3*nu21-
        nu03));
    hu[3] = (float)(t0*t0 + t1*t1);
    hu[4] = (float)((nu30-3*nu12)*t0*(t0*t0-3*t1*t1) + (3*nu21-nu03)*t1
        *(3*t0*t0-t1*t1));
    hu[5] = (float)((nu20-nu02)*(t0*t0-t1*t1) + 4*nu11*t0*t1);
    hu[6] = (float)((3*nu21-nu03)*t0*(t0*t0-3*t1*t1) - (nu30-3*nu12)*t1
        *(3*t0*t0-t1*t1));
}
```

Normalize Function: This function makes the Hu Moments have zero mean and unit variance. We use the mean and std values from training.

Listing 2.9: Normalize - scales features to match training data

```
static const float feat_mean[7] = {0.334226f, 0.044768f, 0.008187f,
                                   0.001415f, 0.000007f, 0.000156f,
                                      -0.000006f};
static const float feat_std[7] = {0.084046f, 0.063766f, 0.014061f,
                                  0.002583f, 0.000082f, 0.000721f,
                                     0.000063f};
```

```
5
6  static void Normalize(float *f) {
7      for (int i = 0; i < 7; i++) {
8          float s = (feat_std[i] < 1e-10f) ? 1.0f : feat_std[i];
9          f[i] = (f[i] - feat_mean[i]) / s;
10     }
11 }
```

AI_Init and ArgMax Functions: Same as Q1 - they start the network and find the biggest output value.

Main Loop: The main loop waits for image data (784 bytes), calculates features, runs the network, and sends back the result.

Listing 2.10: Main loop - receives image and runs full pipeline

```
1  #define SYNC_BYTE 0xBB
2  #define IMG_SIZE 28
3  #define IMG_BYTES 784  // 28 * 28
4
5  static uint8_t img_buf[IMG_BYTES];
6  static float in_buf[7];
7  static float out_buf[10];
8
9  int main(void) {
10     uint8_t sync, res;
11
12     HAL_Init();
13     SystemClock_Config();
14     MX_GPIO_Init();
15     MX_USART2_UART_Init();
16
17     if (AI_Init() != 0) {
18         while (1) { HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
               HAL_Delay(100); }
19     }
20     HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
21
22     while (1) {
23         if (HAL_UART_Receive(&huart2, &sync, 1, HAL_MAX_DELAY) == HAL_OK
24             && sync == SYNC_BYTE) {
25             HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
26
27             // Get image (784 bytes)
28             if (HAL_UART_Receive(&huart2, img_buf, IMG_BYTES, 5000) ==
                   HAL_OK) {
29                 // Calculate Hu Moments on MCU
30                 ComputeHuMoments(img_buf, in_buf);
31                 // Normalize to match training
32                 Normalize(in_buf);
33                 // Run neural network
34                 if (stai_network_run((stai_network*)net_ctx,
                       STAI_MODE_SYNC) == STAI_SUCCESS) {
35                     res = ArgMax(out_buf, 10);
36                 } else {
37                     res = 0xFF;
38                 }
39                 HAL_UART_Transmit(&huart2, &res, 1, 100);
```

```
            }
            HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
        }
    }
}
```

Step 4: Python UART Client

The Python side is simple. It loads the image, makes it 28x28, and sends it to the MCU.

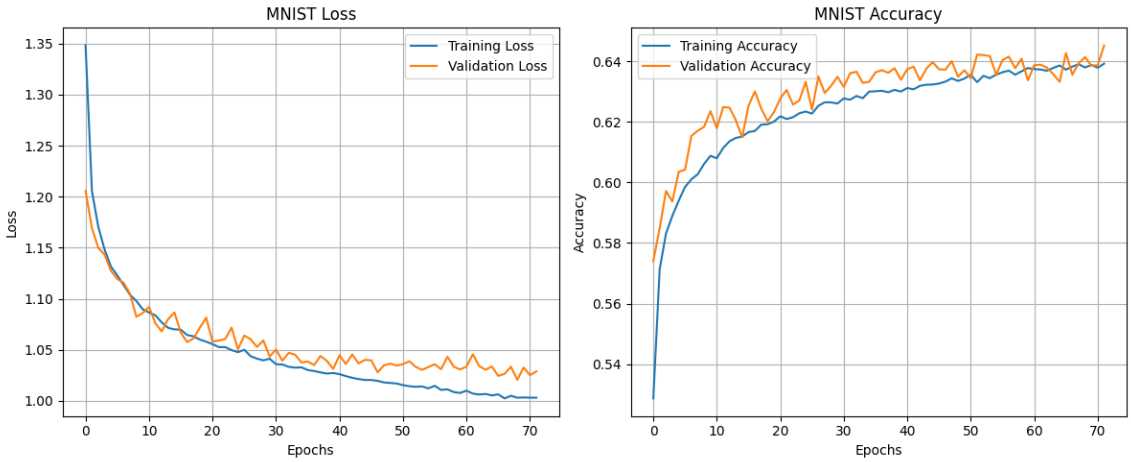
Listing 2.11: Python UART client for MNIST (mnist_uart.py)

```
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))
    # MNIST uses black background, so invert if needed
    if np.mean(img) > 127:
        img = 255 - img
    return img

def send_and_receive(img, port='COM9'):
    with serial.Serial(port, 115200, timeout=5) as ser:
        ser.write(bytes([SYNC_BYTE]))
        ser.write(img.flatten().tobytes())
        result = ser.read(1)
        return result[0] if len(result) == 1 else -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 (MCU)", (150, 60), cv2.
        FONT_HERSHEY_SIMPLEX, 1.5, (255,255,255), 3)
    if img is not None:
        big = cv2.resize(img, (140, 140), interpolation=cv2.
            INTER_NEAREST)
        canvas[80:220, 20:160] = cv2.cvtColor(big, cv2.COLOR_GRAY2BGR)
    cv2.imshow('Result', canvas)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('image')
    parser.add_argument('--port', default='COM9')
    args = parser.parse_args()

    img = load_image(args.image)
    digit = send_and_receive(img, args.port)
    if 0 <= digit <= 9:
        display(digit, img)

```

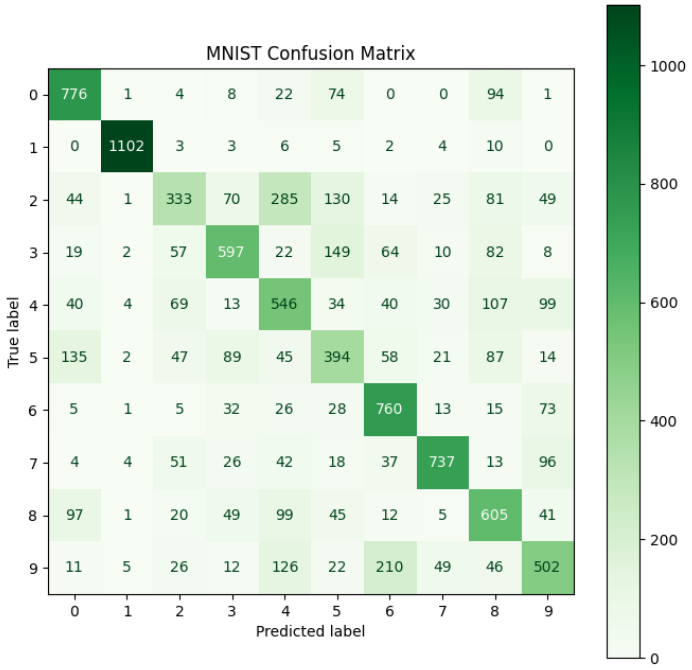
```
47 if __name__ == '__main__':
48     main()
```

Results

Figure 2.4 shows our results. The model gets about 64% accuracy. This is lower than Q1 because Hu Moments only have 7 numbers. They cannot show all the small details of different digits.



(a) Training Loss and Accuracy



(b) Confusion Matrix

Figure 2.4: Training results for the MNIST Hu Moments model.

We tested the system with handwritten digit images. Figure 2.5 shows the inference result.

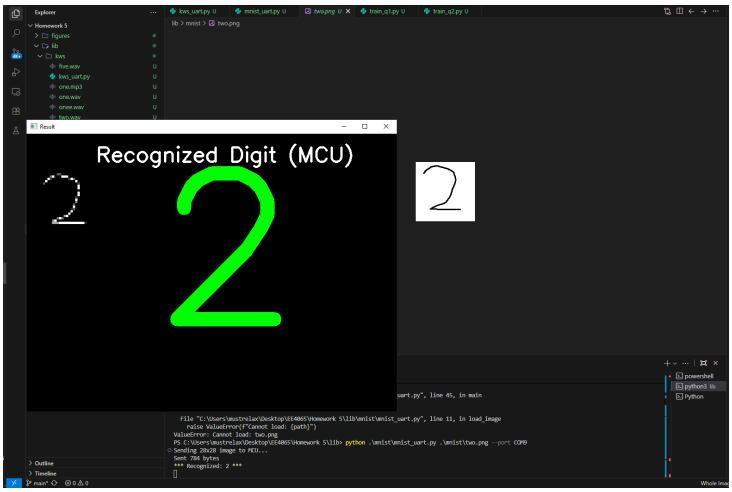


Figure 2.5: OpenCV window showing recognized handwritten digit.

Figure 2.6 shows the STM32CubeIDE debug window with Hu Moments values and network output.

Expression	Type	Value
out_buf	float [10]	0x20000774 <out_buf>
out_buf[0]	float	0
out_buf[1]	float	0
out_buf[2]	float	1
out_buf[3]	float	0
out_buf[4]	float	0
out_buf[5]	float	0
out_buf[6]	float	0
out_buf[7]	float	0
out_buf[8]	float	0
out_buf[9]	float	0
Add new expression		

Figure 2.6: STM32CubeIDE Expressions window showing Hu Moments (in_buf) and output values.

3. CONCLUSION

In this homework, we successfully deployed AI models onto the STM32F446RE MCU for real-time pattern recognition. The main goal was to understand the complete pipeline from model training on a PC to inference on an embedded system.

For the KWS task, we trained a model using MFCC features extracted from the FSDD dataset. The Python client extracts features from audio files and sends them to the MCU via UART. The MCU runs the neural network inference and returns the predicted digit. This approach achieved high accuracy in recognizing spoken digits.

For the handwritten digit recognition task, we used Hu Moments as features. Unlike the audio task, the MCU performs the feature extraction directly on the device. The Python client sends the raw 28x28 image, and the MCU computes the 7 Hu Moments, normalizes them, and runs inference. While the accuracy (around 64%) is lower than modern approaches, this demonstrates that complex image processing can be performed on resource-limited embedded systems.

A key learning from this assignment was using the **ST Edge AI Developer Cloud** to convert TensorFlow models to optimized C code. This tool simplified the deployment process significantly. We also learned how to implement custom UART protocols for data transfer between a PC and MCU.

Overall, this assignment provided practical experience in embedded Machine Learning (ML), bridging the gap between high-level model training and low-level hardware deployment. These skills form a strong foundation for developing real-world edge AI applications.

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).

[4] S. Davis, P. Mermelstein, Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences, IEEE transactions on acoustics, speech, and signal processing 28 (4) (1980) 357–366.

[5] M.-K. Hu, Visual pattern recognition by moment invariants, IRE transactions on information theory 8 (2) (1962) 179–187.

[6] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.

[7] G. Bradski, The OpenCV Library, http://opencv.org, accessed: October 30, 2025 (2000).