Project Report

Spring 2024

CSE-307L Microprocessor Based System Design Lab

Submitted by:

Ali Asghar(21PWCSE2059)

Shahzad Bangash(21PWCSE1980)

Suleman Shah(21PWCSE1983)

Abu Bakar(21PWCSE2004)

Class Section: **C**

Submitted to:

Engr. Shahzada Fahim Jan

Date:

28th June 2024

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

Handwritten Image Recognition on ESP32

Contents

Chapter 1

Introduction

1.1 Neural Network

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously.[1].

1.1.1 Neural Network - Example

Figure 1.1 shows a simple neural network which has one input, one hidden and one output layer. Input and output layers contain two neurons each while hidden layer contain 3 neurons.

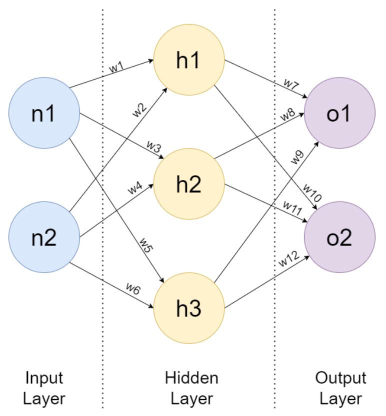


Figure 1.1: Neural Network Example

1.1.2 Why neural networks?

Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are nonlinear and complex. They are widely used in making smart systems nowadays.

1.2 Microcontroller

A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip. Sometimes referred to as an embedded controller or microcontroller unit (MCU), microcontrollers are found in vehicles, robots, office machines, medical devices, mobile radio transceivers, vending machines and home appliances, among other devices. They are essentially simple miniature personal computers (PCs) designed to control small features of a larger component, without a complex front-end operating system (OS)[2].

1.2.1 Example - 8051 Microcontroller

In 1980, Intel introduced a powerful 8051 series of 8 -bit microcontrollers. They are the second generation of 8-bit microcontrollers. The 8051 microcontrollers are used for a variety of applications involving limited calculations and relatively some control strategies. They are used for industrial and commercial control applications, appliances control, instrumentation etc[3]. The 8051 contains Boolean processor, full duplex serial port and power saving circuitry in addition to essential components such as 8-bit CPU, RAM, ROM/EPROM/OTPROM, timer/counter and parallel I/O lines.

1.2.2 Uses of Microcontroller

Microcontrollers (MCUs) are compact integrated circuits designed to govern a specific operation in an embedded system. They integrate a processor, memory, and input/output (I/O) peripherals on a single chip, making them highly efficient for control-oriented applications. Here are some common uses of microcontrollers:

- Consumer Electronics
- Automotive Applications
- Industrial Automation
- Medical Devices
- Communication Systems
- Smart Home and IoT
- Aerospace and Defense
- Agriculture
- Energy Management
- Entertainment and Gaming

1.3 ESP32

ESP32 is the SoC (System on Chip) microcontroller which has gained massive popularity recently. Whether the popularity of ESP32 grew because of the growth of IoT or whether IoT grew because of the introduction of ESP32 is debatable. ESP32 used in this project is **ESP32-WROOM-32**.

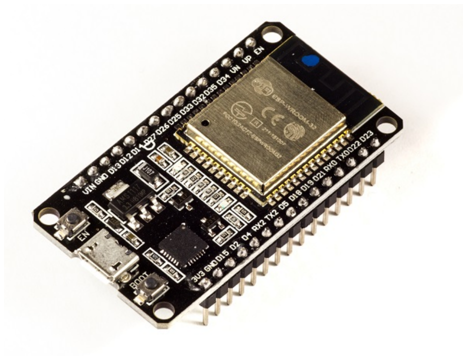


Figure 1.2: ESP32

1.4 Main features of ESP32

From [4], We can see 4 main and important features of ESP32.

1.4.1 Robust Design

ESP32 is capable of functioning reliably in industrial environments, with an operating temperature ranging from –40°C to +125°C. Powered by advanced calibration circuitries, ESP32 can dynamically remove external circuit imperfections and adapt to changes in external conditions.

1.4.2 Ultra-Low Power Consumption

Engineered for mobile devices, wearable electronics and IoT applications, ESP32 achieves ultra-low power consumption with a combination of several types of proprietary software. ESP32 also includes state-of-the-art features, such as fine-grained clock gating, various power modes and dynamic power scaling.

1.4.3 Hybrid Wi-Fi Bluetooth Chip

ESP32 can perform as a complete standalone system or as a slave device to a host MCU, reducing communication stack overhead on the main application processor. ESP32 can interface with other systems to provide Wi-Fi and Bluetooth functionality through its SPI / SDIO or I2C / UART interfaces.

1.4.4 High Level of Integration

ESP32 is highly-integrated with in-built antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power management modules. ESP32 adds priceless functionality and versatility to your applications with minimal Printed Circuit Board (PCB) requirements

1.5 Specifications of ESP32-WROOM-32

The ESP32-WROOM-32 is a powerful, versatile, and cost-effective Wi-Fi and Bluetooth module developed by Espressif. It integrates the ESP32-D0WDQ6 chip and provides various features suitable for a wide range of applications. All the details about ESP32-WROOM-32 can be found in [5]

1.5.1 Key Features

- **Wi-Fi and Bluetooth Connectivity**
  - IEEE 802.11 b/g/n/e/i standards
  - Bluetooth v4.2 BR/EDR and BLE (Bluetooth Low Energy)
- **High Performance**
  - Dual-core 32-bit LX6 microprocessor, up to 240 MHz
  - 520 KB SRAM
  - 448 KB ROM
- **Rich Peripheral Interfaces**
  - 34 programmable GPIOs
  - Multiple analog-to-digital converters (ADCs) and digital-to-analog converters (DACs)
  - SPI, I2C, I2S, UART, PWM, and SDIO
  - Ethernet MAC interface with dedicated DMA and IEEE 1588 Precision Time Protocol support
- **Storage**
  - 4 MB Flash memory
- **Low Power Consumption**
  - Various power modes, including deep sleep, light sleep, and modem sleep
  - Ultra-low power co-processor

1.6 Pinout Diagram ESP32-WROOM-32

Following is the pinout diagram of ESP32-WROOM-32 from citeesp32-pinout.

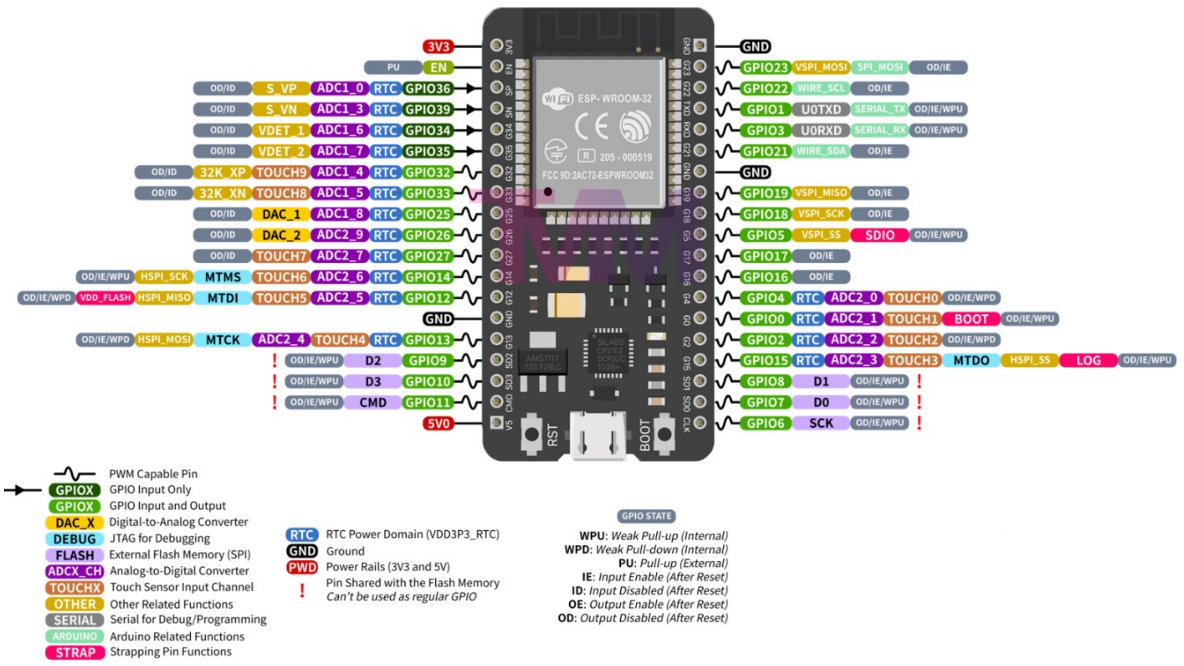


Figure 1.3: Pinout Diagram ESP32-WROOM-32

1.7 Coding an ESP32

To run code on an ESP32, we use one of several development environments, such as the Arduino IDE, Espressif's ESP-IDF, or PlatformIO. Below are the steps which I used to run code on the ESP32 using the Arduino IDE, which is one of the most straightforward methods:

1.7.1 Install the ESP32 Board Manager

- Open the Arduino IDE.
- Go to `File > Preferences`.
- In the *Additional Board Manager URLs* field, add the following URL:

```
https://dl.espressif.com/dl/package_esp32_index.json
```

- Click *OK*.

1.7.2 Install the ESP32 Board

- Go to `Tools > Board > Boards Manager`.
- In the Boards Manager window, search for *ESP32*.
- Click *Install* on the *esp32* entry by Espressif Systems.

1.7.3 Select the ESP32 Board

- Go to `Tools > Board`, and select your specific ESP32 board model (e.g., *ESP32 Dev Module*).

1.7.4 Connect the ESP32 to Your Computer

- Use a USB cable to connect your ESP32 board to your computer.

1.7.5 Select the Port

- Go to `Tools > Port`, and select the port to which your ESP32 is connected.

1.7.6 Upload the code

- After serlecting a port, hit the upload compile button to upload the sketch to ESP32.

1.8 ESP32 for Neural Networks

Microcontrollers are typically small, low-powered computing devices that are embedded within hardware that requires basic computation. By bringing machine learning to tiny microcontrollers, we can boost the intelligence of billions of devices that we use in our lives, including household appliances and Internet of Things devices, without relying on expensive hardware or reliable internet connections, which is often subject to bandwidth and power constraints and results in high latency. This can also help preserve privacy, since no data leaves the device. Imagine smart appliances that can adapt to your daily routine, intelligent industrial sensors that understand the difference between problems and normal operation, and magical toys that can help kids learn in fun and delightful ways[7].

Chapter 2

Neural Network Implementation - Making a Model

2.1 Introduction

Making a Neural Network/Machine Learneing Model involves several steps. Following steps were carried out to make a model in Google Colab which can recognize a handwritten digit.

2.2 Training Data

For training, we have used MNIST dataset. The MNIST (Modified National Institute of Standards and Technology) dataset is a benchmark dataset widely used in the field of machine learning and computer vision, specifically for handwritten digit recognition. It consists of images of handwritten digits from 0 to 9 and serves as a standard for evaluating and comparing the performance of various algorithms and models. It consist of images as shown in figure 2.1.

2.3 Model Training

First step is to train a machine learning model using tools like Jupyter Notebook, Google Colab etc. In this project, we have used Google Colab for training a tensor flow model for handwritten digit recognition as shown in figure 2.2.

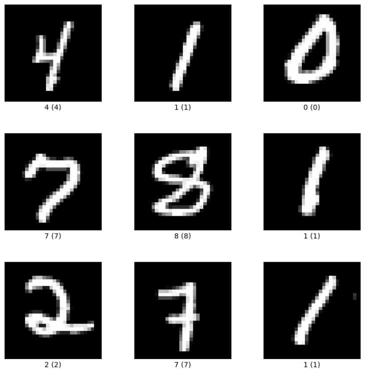
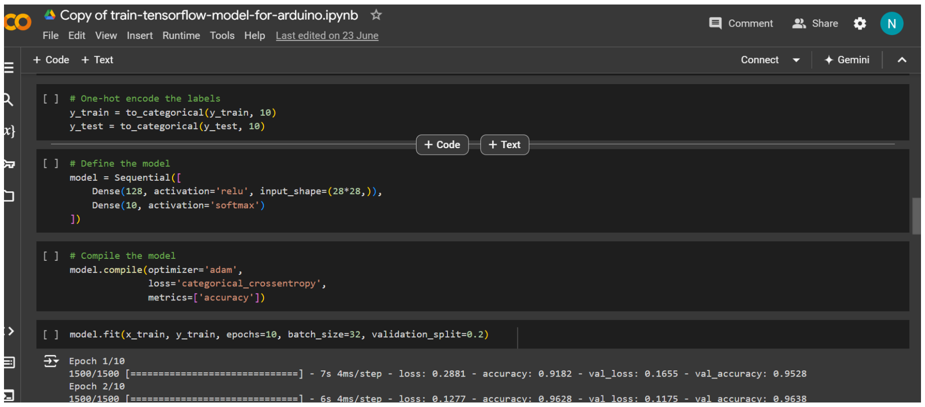


Figure 2.1: MNIST Dataset Sample


Copy of train-tensorflow-model-for-arduino.ipynb
File Edit View Insert Runtime Tools Help Last edited on 23 June
Comment Share
+ Code + Text
Connect
Gemini

```
# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```
# Define the model
model = Sequential([
    Dense(128, activation='relu', input_shape=(28*28,)),
    Dense(10, activation='softmax')
])
```

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2881 - accuracy: 0.9182 - val_loss: 0.1655 - val_accuracy: 0.9528
Epoch 2/10
1500/1500 [==============================] - 6s 4ms/step - loss: 0.1277 - accuracy: 0.9628 - val_loss: 0.1175 - val_accuracy: 0.9638
```

Figure 2.2: Model Training in Google Colab

2.4 Structure of Model

The model consists of two dense (fully connected) layers. The first layer has 128 hidden units, and the second layer has 10 output units corresponding to 10 possible classes. The network architecture is as follows:

- Input layer: 784 units (28x28 pixels)
- Dense layer 1: 128 units
- ReLU activation
- Dense layer 2: 10 units
- Softmax activation

2.5 Softmax and ReLU Activation Functions

2.5.1 ReLU (Rectified Linear Unit)

The Rectified Linear Unit (ReLU) is one of the most widely used activation functions in deep learning models, particularly in convolutional neural networks. The ReLU function is defined as follows:

$$\text{ReLU}(x) = \max(0, x)$$

This function outputs the input directly if it is positive; otherwise, it outputs zero. The key advantages of ReLU are:

- It helps to mitigate the vanishing gradient problem, as it provides a constant gradient for positive inputs.
- It introduces non-linearity into the model, enabling the neural network to learn complex patterns.
- It is computationally efficient, as it requires only a simple thresholding operation.

However, ReLU has a few drawbacks:

- The "dying ReLU" problem, where neurons can sometimes get stuck in the inactive region (outputting zero) and stop learning.

2.5.2 Softmax

The Softmax function is commonly used in the output layer of a neural network for multi-class classification problems. It converts the raw output scores (logits) of the network into a probability distribution over the predicted output classes. The Softmax function is defined as:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

where z_i is the input score (logit) for class i, and N is the number of classes. The key properties of the Softmax function are:

- The output values are in the range (0, 1), and they sum to 1, forming a valid probability distribution.
- It amplifies the differences between the input scores, making the highest score more dominant, which helps in clear classification decisions.

The Softmax function is particularly useful in the context of categorical cross-entropy loss, which is commonly used for training classification models. Graphs of such functions are show in figure 2.3.

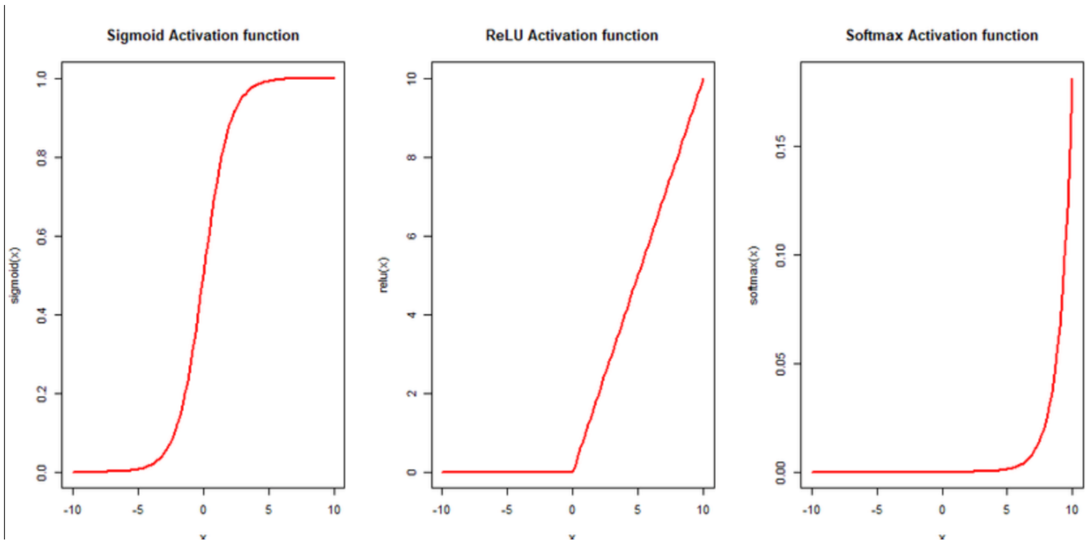
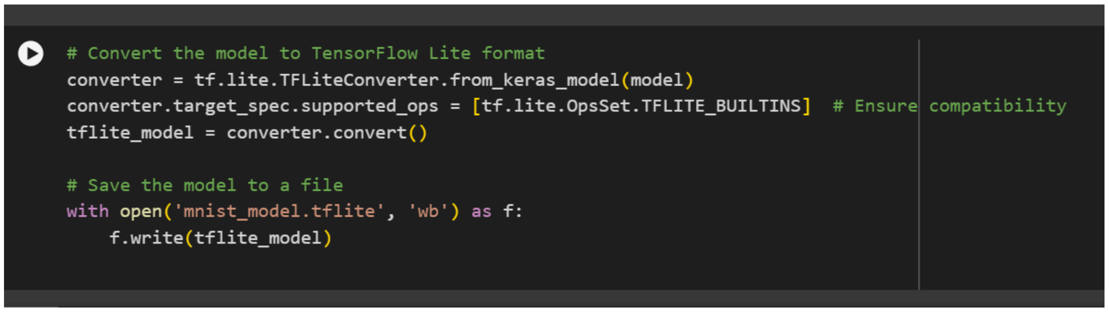


Figure 2.3: Graph of ReLU, Softmax and Sigmoid Functions

2.6 Converting TensorFlow Model to TensorFlow lite Model

In this section, a TensorFlow(TF) model is converted to TensorFlow Lite(TFLITE) model. Following code section as shown in figure 2.4 describe the conversion process. The output files is saved by the name **mnist_model.tflite**. As microcontroller has limited computational power and memory, the TensorFlow model must be converted to its light version, so that it can run safely on a microcontroller like ESP32.



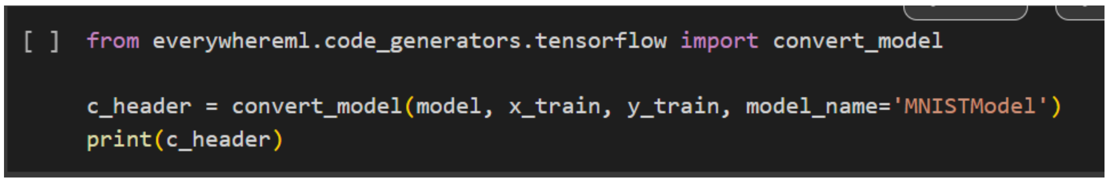
```
# Convert the model to TensorFlow Lite format
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]  # Ensure compatibility
tflite_model = converter.convert()

# Save the model to a file
with open('mnist_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Figure 2.4: Converting TensorFlow Model to TensorFlow lite Model

2.7 Converting TensorFlow lite Model to C++ Array

In order to use our tflite model in ESP32, it should be made compatible with ESP32 code environment. ESP32's architecture/compiler runs on C++ code, i-e it accepts C++ code as input when we are programming it. Hence, using **everywhereml** library, we converted the tflite model to a C++ array and then saved it in a header file as shown in figure 2.5.



```
[ ] from everywhereml.code_generators.tensorflow import convert_model

    c_header = convert_model(model, x_train, y_train, model_name='MNISTModel')
    print(c_header)
```

Figure 2.5: Converting TensorFlow lite Model to C++ Array

2.8 Output model generated in form of C++ code

```
#pragma once

#ifdef __has_attribute
#define HAVE_ATTRIBUTE(x) __has_attribute(x)
#else
#define HAVE_ATTRIBUTE(x) 0
#endif
#if HAVE_ATTRIBUTE(aligned) || (defined(__GNUC__) && !defined(__clang__))
#define DATA_ALIGN_ATTRIBUTE __attribute__((aligned(4)))
#else
#define DATA_ALIGN_ATTRIBUTE
#endif

// automatically configure network
#define TF_NUM_INPUTS 784
#define TF_NUM_OUTPUTS 10
#define TF_NUM_OPS 2
#define TF_OP_FULLYCONNECTED
#define TF_OP_SOFTMAX


// sample data
float x0[784] = {..}
float x1[784] = {..}
float x2[784] = {..}

/** model size = 408768 bytes **/
const unsigned char MNISTModel[] DATA_ALIGN_ATTRIBUTE = {....};
```

Chapter 3

Neural Network Implementation - Deploying tflite Model on ESP32

3.1 Introduction

Using Arduino IDE, Now we can deploy our model in the form of C++ array (in .h file) to our ESP32. We can test the accuracy and performance of our model based on the test images.

3.2 Arduino Sketch Code

```
#include "model_softmax.h" //Contains the model C++ code
#include "mnist_image_array.h" //contains the input image array
#include <tflm_esp32.h> //Library Files
#include <eloquent_tinyml.h> //Library
#define ARENA_SIZE 10000

Eloquent::TF::Sequential<TF_NUM_OPS, ARENA_SIZE> tf;

void setup() {
    Serial.begin(115200);
    delay(3000);
    Serial.println("__TENSORFLOW MNISTModel");

    while (!tf.begin(MNISTModel).isOk())
        Serial.println(tf.exception.toString());
}
```

```
void loop() {

    // classify sample from
    if (!tf.predict(image).isOk()) {
        Serial.println(tf.exception.toString());
        return;
    }
    Serial.print("Predicted class ");
    Serial.print(tf.classification);
    Serial.println("");
    Serial.print("It takes ");
    Serial.print(tf.benchmark.microseconds());
    Serial.println("us for a single prediction");
    Serial.println("");
    delay(1000);
}
```

3.3 Libraries Used

3.3.1 TensorFlowLite_ESP32

This library runs TensorFlow machine learning models on microcontrollers, allowing us to build AI/ML applications powered by deep learning and neural networks[8].

3.3.2 EloquentTinyML

This Arduino library is here to simplify the deployment of Tensorflow Lite for Microcontrollers models to Arduino boards using the Arduino IDE[9].

3.3.3 tflm_esp32

This Arduino library implement the operations for the functions present in tflite model. These operations could be Softmax,Sigmoid etc[10].

Chapter 4

Testing, Verification and Results

4.1 Introduction

This chapter describes the testing of the neural network on the ESP32. To test the model on ESP32, I have used a python script to generate an array from given 28x28 grayscale handwritten digit image.

4.2 Image to Array Converter

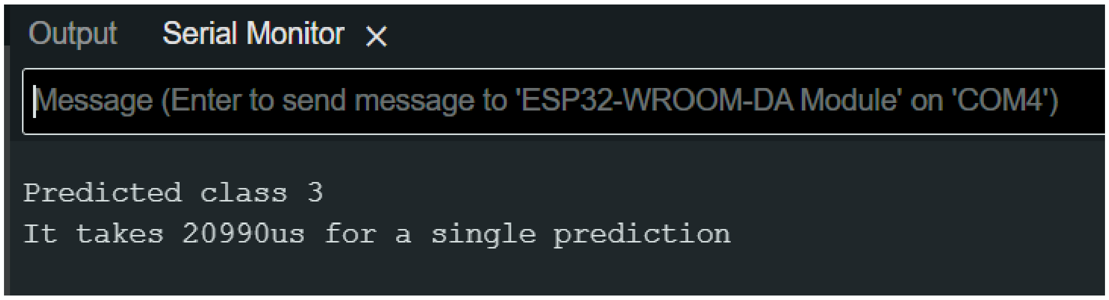
Following script convert the given input image to an array(using numpy and PIL libraries) and save it in a header file.

```
def image_to_c_array(image_path, output_path):
    # Open the image file
    img = Image.open(image_path).convert('L')  # Convert to grayscale
    # Resize image to 28x28 if it's not already
    img = img.resize((28, 28))
    # Convert image to numpy array and normalize pixel values
    img_array = np.array(img) / 255.0
    # Flatten the array
    flat_array = img_array.flatten()
    # Convert to C array format
    c_array_str = ',␣'.join(map(str, flat_array))
    c_array = f"float␣image[784]␣=␣{{␣{c_array_str}␣}};"
    # Write to output file
    with open(output_path, 'w') as file:
        file.write(c_array)

    print(f"C␣array␣saved␣to␣{output_path}")
```

4.3 Output of Model in Arduino IDE Serial Monitor

Figure 4.1 shows the output of our model in Arduino IDE Serial Monitor for the input image shown in 4.2.


Output
Serial Monitor
Message (Enter to send message to 'ESP32-WROOM-DA Module' on 'COM4')

```
Predicted class 3
It takes 20990us for a single prediction
```

Figure 4.1: Output for Handwritten Image of 3

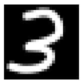


Figure 4.2: Input Image for 3

4.3.1 Model Performance on ESP32

The time took by the microcontroller(ESP32) to make decision is calculated using the benchmark function. According to figure 4.1, it is approximately **20.99ms**.

Chapter 5

Conclusion

This report demonstrated the implementation of a neural network on an ESP32. The MNIST Based Handwritten Digit Recognition Model was succesfully deployed on ESP32. We can further increase the performance of this model by optimizing the layers and functions inside the neural network. Further, the output can be displayed on an output display connected with the microcontroller.

References

[1] https://aws.amazon.com/what-is/neural-network

[2] https://www.techtarget.com/iotagenda/definition/

[3] https://www.javatpoint.com/microcontroller

[4] https://www.espressif.com/en/products/socs/esp32

[5] https://www.espressif.com/sites/default/files/documentation

[6] https://www.teachmemicro.com/esp32-pinout-diagram-wroom-32/

[7] https://www.tensorflow.org/lite/microcontrollers

[8] https://www.arduino.cc/reference/en/libraries/tensorflowlite$_{e}sp32/$

[9] https://github.com/eloquentarduino/EloquentTinyML

[10] https://github.com/eloquentarduino/tflm$_{e}sp32/tree/main/src/tensorflow/lite$