Technische Hochschule Deggendorf
Fakultat Elektrotechnik und Medientechnik
Studiengang Master Elektro- und Informationstechnik

Konzeption, Implementation und Evaluierung von Tiny-Machine-Learning Systemen auf Basis des MAX78002 Mikrocontroller

Design, implementation and evaluation of Tiny-Machine-Learning systems based on the MAX78002 microcontroller

Masterarbeit zur Erlangung des akademischen Grades:
Master of Science (M.Sc.)
an der Technischen Hochschule Deggendorf

Vorgelegt von:
Sertac Engin
Matrikelnummer: 12102945

Prüfungsleitung:
Prof. Dr. Robert Bösnecker

Am: 26.05.2024

Erklärung

Name des Studierenden: Sertac Engin

Name des Betreuenden: Prof. Dr. Robert Bösnecker

Thema der Abschlussarbeit:

Konzeption, Implementation und Evaluierung von Tiny-Machine-Learning Systemen auf Basis des MAX78002 Mikrocontroller

Design, implementation and evaluation of Tiny-Machine- Learning systems based on the MAX78002 microcontroller

1. Ich erkläre hiermit, dass ich die Abschlussarbeit gemäß § 35 Abs. 7 RaPO (Rahmenprüfungsordnung für die Fachhochschulen in Bayern, BayRS 2210-4-1-4-1-WFK) selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

   Deggendorf, 26.05.2024 ..... Datum

   ..... Unterschrift des Studierenden

2. Ich bin damit einverstanden, dass die von mir angefertigte Abschlussarbeit über die Bibliothek der Hochschule einer breiteren Öffentlichkeit zugänglich gemacht wird:

   ◯ Nein

   ◉ Ja, nach Abschluss des Prüfungsverfahrens

   ◯ Ja, nach Ablauf einer Sperrfrist von …Jahren.

   Deggendorf, 26.05.2024 ..... Datum

   ..... Unterschrift des Studierenden

Bei Einverständnis des Verfassenden vom Betreuenden auszufüllen:

Eine Aufnahme eines Exemplars der Abschlussarbeit in den Bestand der Bibliothek und die Ausleihe des Exemplars wird:

◯ Befürwortet

◯ Nicht befürwortet

Deggendorf, ... Datum

..... Unterschrift des Betreuenden

Abstract

Embedded systems are ubiquitous in modern technology, forming the intelligent core of countless devices. Their compact size and efficient processing power make them ideal for applications demanding real-time control and interaction with the physical world. Tiny machine learning refers to the deployment of machine learning algorithms and models on resource-constrained devices, typically microcontrollers or embedded systems with limited computational power, memory, and energy resources. This thesis explores the intersection of embedded systems and machine learning, specifically focusing on robotic arm manipulation.

Initially, the thesis investigates the control of a robotic arm for object manipulation using an Arduino Uno microcontroller. This foundational approach establishes a baseline understanding of real-time control principles and motor actuation for robotic movement.

Following this, the thesis dives into the domain of tiny machine learning (TinyML). By using the MAX78002 microcontroller's capabilities, the research explores the implementation of machine learning models for predicting the robotic arm's behavior. TinyML techniques are employed to develop a lightweight machine learning models that predict the movements of the robotic arm.

This combined approach, utilizing both traditional embedded system control and TinyML, aims to demonstrate the use of machine learning algorithms on resource-limited devices. This thesis aims to help with the analysis of the tiny machine learning systems in terms of design, implementation, and performance, exploring factors such as accuracy, efficiency, and potential limitations.

Keywords: real-time control, TinyML, microcontrollers, robotic arm, object manipulation.

Acknowledgment

First and foremost, I express my gratitude to Prof. Dr. Robert Bösnecker for supervising my thesis and Mr. Andreas Federl for their invaluable guidance, expertise, and support throughout every stage of this research endeavor. Their insights and encouragement have been instrumental in shaping the direction and focus of this work.

Special thanks to Mr. Sergej Lamert for providing access to the robotic arm system used in this research. Their collaboration and assistance have been essential in realizing the experimental setup and conducting experiments effectively.

I would also like to thank Prof. Dr.-Ing. Werner Bogner and other Electrical Engineering and Media Technology faculty members for their advice and support throughout my study.

Last but not least, I am deeply grateful to my family for their unwavering support, understanding, and encouragement. Their belief in my abilities has been a constant source of motivation and inspiration.

Contents

1 Introduction

In recent years, the integration of robotics into various industries has revolutionized manufacturing processes, automation, and even daily tasks. Robotic arms, in particular, have emerged as versatile tools capable of performing a wide range of manipulation tasks with precision and efficiency. As the demand for robotic systems continues to grow, there is a pressing need to develop robust control mechanisms that enable seamless interaction between humans and machines.

This thesis explores the development and implementation of a control system for a robotic arm, focusing on the utilization of embedded systems and tiny machine learning techniques to enhance its functionality and user experience. The documentation presented herein outlines the hardware setup, data measurement capabilities, and control methodologies employed in the construction and operation of the robotic arm.

The hardware setup comprises 6 servo motors controlled by a motor driver interfaced with an Arduino Uno microcontroller. This configuration allows for precise control of the robotic arm's movements within 3d environment. Additionally, a DC-DC Buck Converter provides the necessary power supply to both the Arduino and the motor driver, ensuring uninterrupted operation during task execution. Importantly, the converter also facilitates data measurement, enabling the monitoring of input and output currents, voltages, and temperature levels critical for system performance and safety.

Data measurement plays a crucial role in understanding and optimizing the robotic arm's operation. By capturing real-time operational parameters, such as current consumption and temperature, insights can be gained into the system's behavior and potential areas for improvement.

Furthermore, the control of the robotic arm is facilitated through code, allowing for precise manipulation of individual servo motors. A Python script communicates with the Arduino via a serial interface, enabling users to command specific movements using instructions. This streamlined approach to control enhances the versatility and adaptability of the robotic arm, making it suitable for a wide range of applications.

Through the integration of embedded systems and tiny machine learning techniques, this thesis aims to contribute to the advancement of robotic control methodologies, ultimately enhancing the efficiency, flexibility, and user-friendliness of robotic systems. By combining hardware and software solutions, we endeavor to pave the way for innovative applications in automation, manufacturing, healthcare, and beyond.

2 Robotic Arm Control

Robotic arms have emerged as indispensable tools across various industrial and research domains due to their versatility and precision. In this section, we explain the control mechanism employed in a specific robotic arm setup, delineating the hardware configuration and the software protocols involved in its operation. The discussion encompasses the hardware setup, data measurement process, and the methodology for controlling the robotic arm through code, providing a comprehensive understanding of the system's functionality. [1]

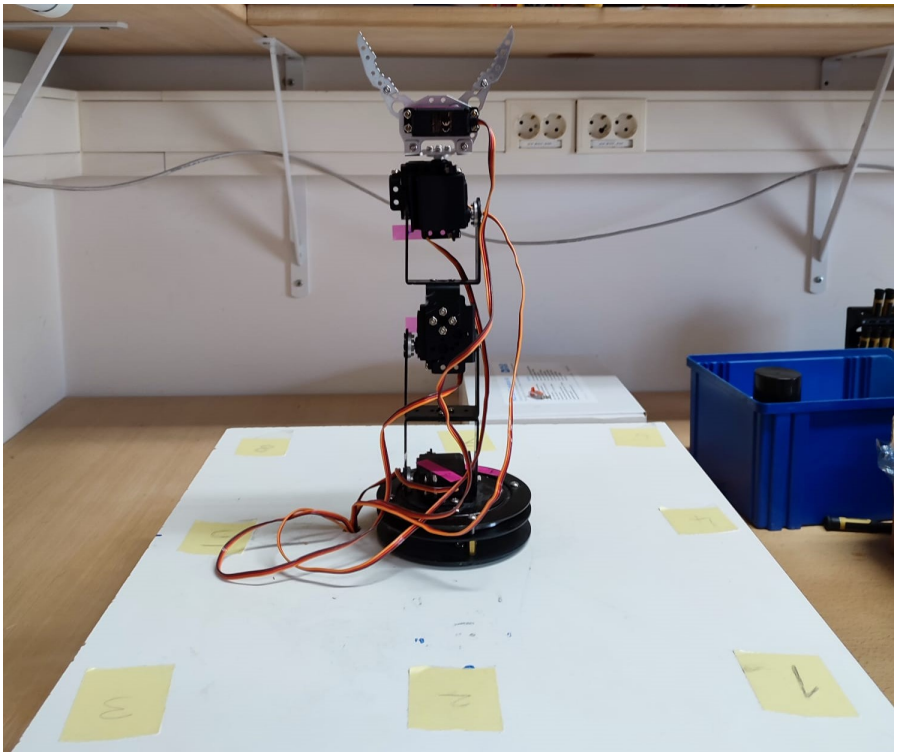

Figure 2.1: The Robotic Arm

Robotic arms represent a pinnacle of modern engineering, embodying intricate systems of mechanical, electrical, and software components to accomplish diverse tasks with efficiency and accuracy. Understanding the intricacies of controlling such systems is imperative for optimizing performance and ensuring seamless integration into applications ranging from manufacturing to medical surgery.

The foundational architecture of the robotic arm encompasses a meticulous arrangement of components aimed at facilitating precise control and efficient operation. At the core of the setup lies the Arduino microcontroller, serving as the central hub for orchestrating the movements of the arm's servo motors. These motors, totaling six in number, are meticulously calibrated to traverse a range of motion from 0 to 180 degrees, enabling intricate manipulations.

Crucially, the servo motors are interfaced with a dedicated motor driver, which acts as an intermediary between the Arduino and the motors themselves. This driver module, mounted on the Arduino Uno, enables the simultaneous control of the 6 servo motors, thereby augmenting the versatility of the robotic arm.

Power management is a critical aspect of the setup, necessitating the utilization of a DC-DC Buck Converter to regulate the voltage supplied to both the motor driver and the Arduino. The converter not only ensures a stable 5V DC voltage output but also facilitates data transmission via an I^2C interface, thereby enabling real-time monitoring of operational parameters such as current, voltage, and temperature.

2.1 Components of the Robotic Arm System

Our robotic arm system consists of 6 servo motors, the motor driver, the Arduino UNO, and the DC-DC Buck Converter. The MAX78002 is also integrated later on.

2.1.1 Servo motors

A servo motor is an electromechanical device that can precisely control the position, velocity, and acceleration of its output shaft. It consists of a motor, a feedback mechanism, and a control system. Servo motors are widely used in various applications, including robotics, automation, aerospace, manufacturing, and more.

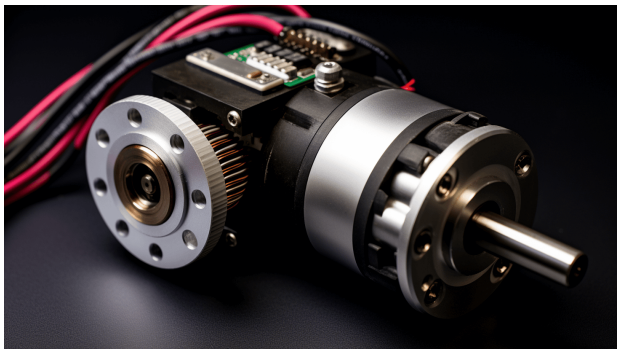

Figure 2.2: Servo Motor

A servo motor consists of 3 components; motor, feedback mechanism, and the control system. The motor in a servo is typically a DC motor, although AC servo motors are also common. DC servo motors are preferred for their simplicity and ease of control. Feedback Mechanism provides feedback to the control system about the position, speed, or torque of the motor shaft. The most common types of feedback devices used in servo motors are encoders and resolvers. Encoders are optical or magnetic devices that provide digital feedback about the motor shaft's position and speed. And resolvers are electromagnetic devices that provide analog feedback about the shaft's position and speed. The control system processes feedback from the feedback mechanism and generates control signals to adjust the motor's operation to maintain the desired position, velocity, or torque.

Working Principle of servo motors are based on closed-loop control systems. The control system compares the actual position, speed, or torque feedback with the desired values and generates an error signal. This error signal is then used to adjust the motor's operation to minimize the error, thus maintaining the desired output.

There are various types of servo motors. DC servo motors use DC power and are widely used in various applications due to their simplicity and ease of control. AC servo motors use AC power and are preferred for high-power applications and where precise control of speed and torque is required. Brushless servo motors use brushless DC motors, offering higher efficiency, lower maintenance, and longer lifespan compared to brushed motors. Linear servo motors produce linear motion directly instead of rotating motion. They are used in applications such as CNC machines, 3D printers, and industrial automation. We use DC servo motors in our application as they provide high precision control, substantial torque even at low speeds, and a feedback loop system that facilitates real-time adjustments, enhancing reliability and performance consistency. [2]

2.1.2 Motor Driver

A pivotal component in the control architecture of the robotic arm is the motor driver, which serves as a linchpin in translating digital commands into physical motion. Integrated seamlessly with the Arduino microcontroller, the motor driver assumes a central role in modulating the operation of the servo motors, thereby enabling precise articulation of the robotic arm.

The motor driver module, intricately interfaced with the Arduino Uno, facilitates bidirectional communication and control between the microcontroller and the servo motors. Through a series of dedicated input pins, the driver module receives command signals from the Arduino, encapsulating instructions regarding the desired position of each servo motor.

The motor driver embodies an embedded control algorithm that modulates the magnitude and duration of the supplied voltage pulses, thereby regulating the rotational movement of the servo motors. This dynamic modulation is facilitated through the utilization of Pulse Width Modulation (PWM) signals, wherein the duty cycle of the signal determines the angular position of the motor shaft. [3]

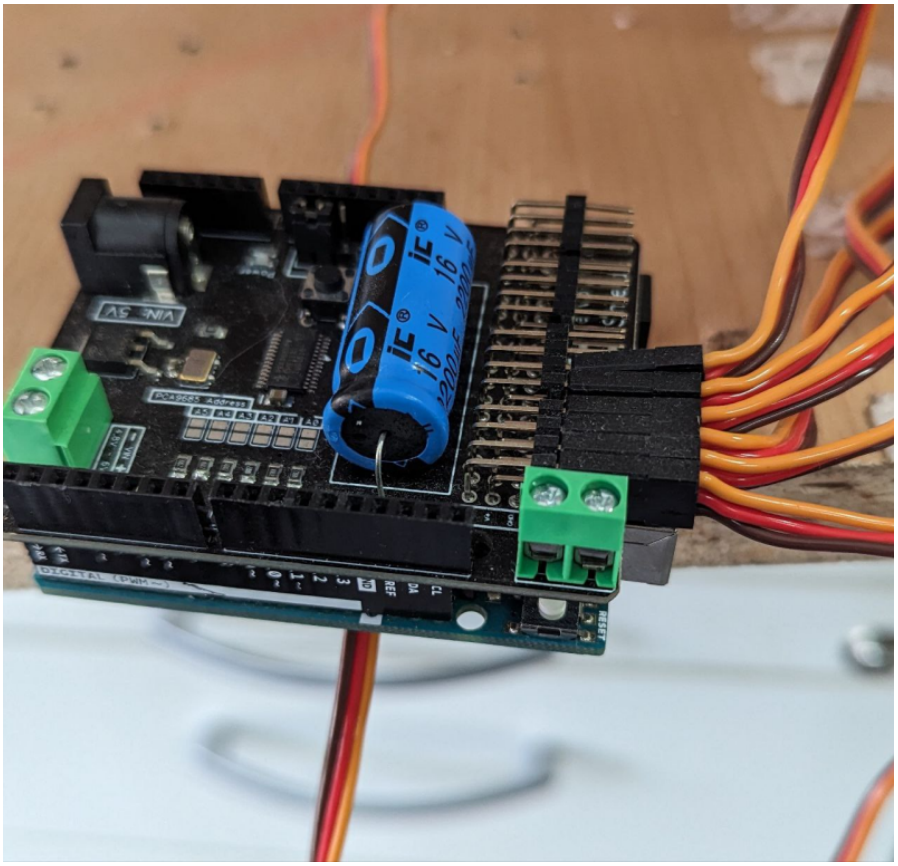

Figure 2.3: Motor Driver mounted on the Arduino UNO

The driver module receives and processes command signals in real-time, dynamically adjusting the PWM signals supplied to each servo motor to achieve the desired trajectory. Moreover, the motor driver augments the versatility of the robotic arm by facilitating the simultaneous control of multiple servo motors, thereby enabling the execution of intricate motion sequences with precision and fluidity. This capability is particularly salient in applications necessitating coordinated motion across multiple degrees of freedom, such as pick-and-place operations or articulated manipulations.

2.1.3 Arduino Uno: Nexus of Control and Intelligence

For robotic arm control, the Arduino Uno emerges as a linchpin, orchestrating a symphony of digital commands and physical movements with finesse and precision. Positioned at the nexus of control and intelligence, the Arduino Uno imbues the robotic arm with the cognitive faculties requisite for executing intricate maneuvers and accomplishing diverse tasks with efficiency and accuracy.

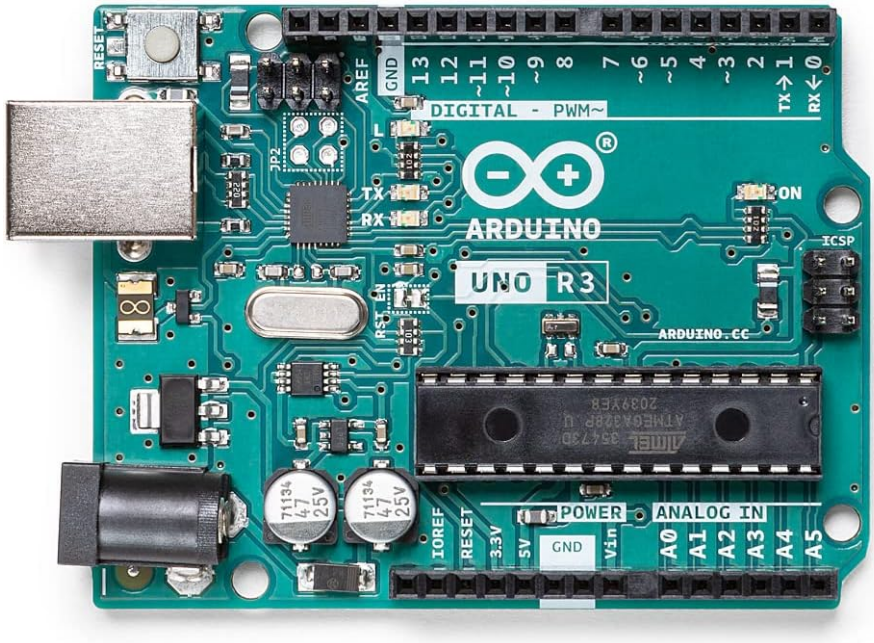

Figure 2.4: The Arduino UNO

At the heart of the robotic arm's control architecture lies the Arduino Uno, serving as the nerve center for interfacing with an array of peripheral devices and sensors. Through its versatile array of digital and analog input/output pins, the Arduino Uno facilitates seamless integration with the motor driver, enabling bidirectional communication and control over the servo motors that animate the robotic arm.

Moreover, the Arduino Uno assumes responsibility for managing power distribution within the robotic arm system, ensuring the stable operation of both the microcontroller itself and the peripheral components. Through meticulous calibration and regulation of voltage levels, the Arduino Uno safeguards against overvoltage or undervoltage conditions, thereby enhancing system robustness and reliability.

One of the hallmark features of the Arduino Uno is its real-time processing capabilities, which endow the robotic arm with the agility and responsiveness requisite for dynamic manipulation tasks. Using its high-speed microcontroller unit (MCU), the Arduino Uno executes control algorithms and motion trajectories with sub-millisecond precision, enabling seamless coordination and synchronization of servo motor movements.

Furthermore, the Arduino Uno facilitates real-time feedback and monitoring of operational parameters, enabling continuous assessment of system performance and preemptive mitigation of potential malfunctions. Through integrated serial communication interfaces such as UART and I2C, the Arduino Uno interfaces with auxiliary sensors and devices, retrieving vital telemetry data pertaining to motor currents, voltages, and temperatures. [4]

2.1.4 DC-DC Buck Converter: Power Regulation and Efficiency Optimization

In the system of robotic arm, the DC-DC Buck Converter emerges as a vital cog, wielding its prowess in power regulation and efficiency optimization to sustain the operational integrity of the system. Positioned as a linchpin in the power management infrastructure, the DC-DC Buck Converter facilitates the seamless integration of diverse components while ensuring a stable and reliable power supply conducive to optimal performance.

Figure 2.5: DC-DC Buck Converter

Central to the functionality of the DC-DC Buck Converter is its ability to regulate voltage levels with precision and efficiency. Operating on the principle of voltage step-down conversion, the converter modulates the 24 V input voltage to 5V for powering the motor driver on the Arduino Uno that controls the robotic. Thus, the DC-DC Buck converter supplies power to the servo motors as well. This conversion ensures compatibility with the 5V requirement

of the motor driver on the Arduino, thereby mitigating the risk of overvoltage conditions that could potentially damage sensitive electronic components.

Furthermore, the DC-DC Buck Converter incorporates an I2C interface, enabling bidirectional communication and data exchange with other devices. This capability empowers us to glean valuable insights into operational parameters such as current, voltage, and temperature, thereby facilitating real-time monitoring.

The DC-DC Buck Converter incorporates robust thermal management mechanisms to mitigate the risk of overheating and thermal degradation. Through the utilization of high-quality components and heat sinks, the converter dissipates excess heat generated during operation, ensuring sustained performance and reliability even under demanding operating conditions. [5]

2.1.5 MAX78002 Microcontroller: Enabling Embedded Intelligence and Machine Learning

In the realm of cutting-edge robotic arm research, the MAX78002 microcontroller, produced by Maxim Integrated, emerges as a beacon of innovation, uniquely poised to revolutionize the landscape of embedded intelligence and machine learning-driven autonomy. Renowned for its exceptional processing capabilities and dedicated neural network accelerator, the MAX78002 microcontroller represents a convergence of hardware and software sophistication, enabling us to deploy sophisticated machine learning algorithms directly on the device with unprecedented efficiency and efficacy.

Figure 2.6: MAX78002

Central to the allure of the MAX78002 microcontroller is its specialized architecture suitable for machine learning applications. Unlike conventional microcontrollers, the MAX78002 incorporates a dedicated neural network accelerator, capable of executing complex inference tasks with remarkable speed and efficiency. This hardware acceleration, coupled with optimized software frameworks, enables the microcontroller to achieve low-latency inference while minimizing computational overhead, thereby facilitating real-time decision-making and response to dynamic stimuli. All these features make the MAX78002 microcontroller the main actor of our thesis.

Moreover, the MAX78002 microcontroller embodies a culture of adaptability and versatility, featuring support for a wide range of machine learning algorithms and model architectures. From convolutional neural networks (CNNs) for image recognition to recurrent neural networks (RNNs) for sequence prediction, the microcontroller empowers us to deploy state-of-the-art machine learning models adapted to the unique requirements of robotic arm systems, thereby unlocking new dimensions of autonomy and intelligence.

The MAX78002 microcontroller's suitability for machine learning applications stems from its robust processing capabilities, dedicated neural network accelerator, and optimized software ecosystem. By offloading intensive computation tasks to the hardware accelerator, the microcontroller achieves unparalleled efficiency and performance in executing machine learning inference, thereby enabling rapid decision-making and response to real-time stimuli.

Furthermore, the MAX78002 microcontroller facilitates seamless integration with auxiliary sensors and actuators, enabling us to use multimodal sensor data for enhanced perception and decision-making. Whether analyzing visual data from cameras, tactile feedback from touch sensors, or inertial data from accelerometers, the microcontroller enables holistic perception of the robotic arm's environment, thereby enhancing operational safety and efficiency.

In conclusion, the MAX78002 microcontroller stands as a testament to the transformative potential of embedded intelligence and machine learning-driven autonomy in the realm of robotic arm research. By harnessing its specialized architecture and optimized software ecosystem, we can deploy sophisticated machine learning models directly on the device, thereby imbuing robotic arms with cognitive faculties essential for navigating dynamic environments and executing complex manipulation tasks with finesse and precision. [6]

2.2 Software for Robotic Arm Control

The orchestration of hardware control and data retrieval is paramount to the realization of sophisticated manipulation tasks and operational insights. Two Arduino sketches play a pivotal role in this endeavor: one dedicated to the real-time control of the robotic arm, and the other tasked with retrieving operational data from the DC-DC Buck Converter, enriching the research endeavor with valuable telemetry insights. This data is used to train a machine learning model that is capable of predicting the robotic arm's movement and object manipulation based on the power consumption.

2.2.1 Robotic Arm Control Sketch

The Arduino sketch for robotic arm control embodies a fusion of hardware abstraction and control logic, enabling us to articulate precise motion trajectories and coordinate the movement of servo motors with finesse and accuracy. Using libraries such as "Servo.h" and "Adafruit_PWMServoDriver.h," the sketch facilitates seamless communication with the motor driver module and the servo motors, thereby enabling dynamic adjustment of motor positions in response to external commands.

```
#include <SPI.h>
#include <Servo.h>
#include <Wire.h>
#include "Adafruit_PWMServoDriver.h"
#include "WString.h"

Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();
float deg_to_pulse(int deg)
{
  return deg * 2.5 + 150;
}

float pulse_to_deg(int pulse)
{
  return ((pulse - 150) / (2.5));
}

int angles[6] = {90, 78, 68, 168, 70, 73};
int old_angles[6] = {90, 78, 68, 168, 70, 73};
int angle_pos[6] = {0, 0, 0, 0, 0, 0};

char delim[] = "-";
char delim_big[] = "/";
char* vals[12];
char msg[25];
int x = 0;
int init_size = strlen(msg);
```

Figure 2.7: Arduino Sketch for Robotic Arm Control-1

At its core, the sketch encapsulates a sophisticated control algorithm that interprets incoming serial commands and translates them into corresponding PWM signals for each servo motor. Through the implementation of functions such as ”deg_to_pulse” and ”pulse_to_deg,” the sketch ensures compatibility between angular positions specified by the user and the pulse-width modulation required by the servo motors, thereby facilitating intuitive control over the robotic arm’s articulation.

```
29  void setup() {
30    Serial.begin(115200);
31    Serial.setTimeout(10);
32
33    pwm.begin();
34    pwm.setPWMFreq(50);
35    while (!Serial) {
36    }
37
38    for (int i = 0; i <= 5; i++)
39    {
40      pwm.setPWM(i, 0, deg_to_pulse(old_angles[i]));
41    }
42
43  }
```

Figure 2.8: Arduino Sketch for Robotic Arm Control-2

Furthermore, the sketch incorporates error handling mechanisms and data parsing logic to ensure robustness and reliability in the face of transient communication disruptions or invalid commands. By adhering to strict timing constraints and using efficient data structures, the sketch achieves low-latency response times, thereby enabling real-time interaction and manipulation of the robotic arm in diverse operational scenarios.

```
45  void loop() {
46    if (Serial.available() > 0)
47    {
48      String xstring = Serial.readStringUntil("/");
49      xstring.toCharArray(msg, sizeof(msg));
50      char *ptr2 = strtok(msg, delim_big);
51      char* ptr = strtok(ptr2, delim);
52
53      while (ptr != NULL)
54      {
55        vals[x++] = ptr;
56        ptr = strtok(NULL, delim);
57      }
58
59      for (int i = 0; i < 6; i++)
60      {
61        angles[i] = atoi(vals[i]); // Convert the character to integer, in this case
62      }
63
64      for (int i = 0; i < 6; i++)
65      {
66        int pwm_old = (int)deg_to_pulse(old_angles[i]);
67        int pwm_new = (int)deg_to_pulse(angles[i]);
68        pwm.setPWM(i, 0, pwm_new);
69        old_angles[i] = angles[i];
70      }
71      x = 0;
72    }
73  }
```

Figure 2.9: Arduino Sketch for Robotic Arm Control-3

2.2.2 Data Retrieval Sketch

Complementing the control sketch, the Arduino sketch for data retrieval serves as a conduit for acquiring vital operational insights from the DC-DC Buck Converter, thereby enhancing situational awareness. Using the ”Wire.h” library for inter-device communication, the sketch establishes a seamless connection with the converter, enabling the retrieval of critical telemetry data such as input and output currents, voltages, and temperature.

```
#include <Wire.h>
#include <HardwareSerial.h>

#define DEV_ADDR 0x08
#define I_IN_ADDR 0x1E
#define V_IN_ADDR 0x20
#define I_OUT_ADDR 0x22
#define V_OUT_ADDR 0x24
#define TEMP_ADDR 0x26

uint8_t x1, x2, x3, x4, x5, x6, x7, x8, x9, x10;

void setup() {
  Serial.begin(230400);
  Wire.begin();
}
```

Figure 2.10: Arduino Sketch for Data Retrieval-1

Through a series of I2C transactions, the sketch orchestrates the interrogation of specific memory addresses within the DC-DC Buck Converter, retrieving raw data bytes representing operational parameters. Subsequently, the sketch formats and transmits these data bytes over the serial interface, enabling real-time visualization and analysis by external monitoring systems or data processing modules.

Crucially, the sketch embodies a paradigm of efficiency and reliability, using hardware interrupts and optimized data structures to minimize processing overhead and ensure timely data acquisition. By adhering to stringent timing constraints and error handling protocols, the sketch fosters a culture of robustness and resilience, thereby facilitating uninterrupted data retrieval operations.

In summary, the Arduino sketches for robotic arm control and data retrieval constitute integral components of the research endeavor, empowering us with the tools and capabilities requisite for navigating the complexities of robotic manipulation and operational monitoring. Through their seamless integration into the control architecture of the robotic arm system, these sketches epitomize a synergy of hardware ingenuity and software sophistication, propelling the advancement of robotic arm research and enabling new frontiers of exploration and innovation.

```
18  void loop() {
19    Wire.beginTransmission(DEV_ADDR);
20    Wire.write(I_IN_ADDR);
21    Wire.endTransmission();
22    Wire.requestFrom(DEV_ADDR, 10);
23    if (Wire.available()) {
24      x1 = Wire.read();
25      x2 = Wire.read();
26      x3 = Wire.read();
27      x4 = Wire.read();
28      x5 = Wire.read();
29      x6 = Wire.read();
30      x7 = Wire.read();
31      x8 = Wire.read();
32      x9 = Wire.read();
33      x10 = Wire.read();
34
35      Serial.write(x1);
36      Serial.write(x2);
37      Serial.write(x3);
38      Serial.write(x4);
39      Serial.write(x5);
40      Serial.write(x6);
41      Serial.write(x7);
42      Serial.write(x8);
43      Serial.write(x9);
44      Serial.write(x10);
45      Serial.write("\n");
46    }
47  }
```

Figure 2.11: Arduino Sketch for Data Retrieval-2

2.2.3 Python Script for Robotic Arm Control

The harmony of precise motion trajectories and dynamic articulation plays a pivotal role in enabling sophisticated manipulation tasks and operational efficiency. The Python script employed for robotic arm control serves as a cornerstone in this endeavor, facilitating seamless communication with the hardware interface and enabling the execution of predefined motion sequences with precision and finesse.

```python
import serial
import time
import struct

ser = serial.Serial(port="COM7", baudrate= 115200)
time.sleep(3)

for i in range(1):

    sera = [150, 78, 68, 170, 75, 75]
    val1 = f"{sera[0]}-{sera[1]}-{sera[2]}-{sera[3]}-{sera[4]}-{sera[5]}/"
    ser.write(bytes(val1, encoding ="utf-8"))
    time.sleep(1)

    sera = [150, 58, 68, 170, 75, 75]
    val1 = f"{sera[0]}-{sera[1]}-{sera[2]}-{sera[3]}-{sera[4]}-{sera[5]}/"
    ser.write(bytes(val1, encoding ="utf-8"))
    time.sleep(1)

    sera = [150, 58, 120, 170, 75, 75]
    val1 = f"{sera[0]}-{sera[1]}-{sera[2]}-{sera[3]}-{sera[4]}-{sera[5]}/"
    ser.write(bytes(val1, encoding ="utf-8"))
    time.sleep(1)
```

Figure 2.12: Python Script for Robotic Arm Control

The Python script orchestrates a series of motion sequences through iterative communication with the Arduino microcontroller, using the ”serial” module to establish a bidirectional communication channel over the designated serial port. By encapsulating motion commands within structured data packets, the script ensures robustness and reliability in command transmission, thereby mitigating the risk of data corruption or transmission errors.

Through judicious utilization of the ”time” module, the script enforces precise timing constraints between consecutive motion commands, thereby facilitating smooth and coordinated motion execution. This temporal synchronization is essential for ensuring seamless articulation of the robotic arm and mitigating the risk of mechanical stress or collision during operation.

At the heart of the Python script lies a choreographed sequence of motion commands, meticulously curated to articulate the robotic arm through diverse manipulation tasks and operational scenarios. Using a combination of positional data and interpolation techniques, the script orchestrates smooth and continuous motion trajectories, ensuring fluidity and precision in arm articulation.

Each motion sequence encapsulates a series of joint angle configurations, representing desired positions of the robotic arm’s servo motors. By interfacing with the Arduino microcontroller via serial communication, the script transmits these configurations in real-time, enabling dynamic adjustment of servo motor positions and facilitating seamless trajectory tracking.

2.2.4 Python Script for Data Retrieval

In the pursuit of comprehensive operational insights and telemetry data acquisition for our thesis, the Python script dedicated to data retrieval from the DC-DC Buck Converter stands as a cornerstone, facilitating the seamless extraction and logging of critical operational parameters. Orchestrated through a sophisticated multithreaded design, this script enables real-time monitoring of input and output currents, voltages, and temperature, enriching the research endeavor with invaluable telemetry insights essential for system characterization and performance optimization.

```python
import threading
import serial
import time

class SerialReader(threading.Thread):
    def __init__(self, port, name, baudrate=230400):
        super().__init__()

        self.serial_port = serial.Serial(port, baudrate)
        self.name = name

    def run(self):
        while 1:
            response = self.serial_port.readline()
            if len(response) == 11:
                I_IN = self.get_val(response[0], response[1])
                V_IN = self.get_val(response[2], response[3])
                I_OUT = self.get_val(response[4], response[5])
                V_OUT = self.get_val(response[6], response[7])
                TEMP = self.get_val(response[8], response[9])

                with open(self.name, "a") as t:
                    t.write(f"{I_IN}, {V_IN}, {I_OUT}, {V_OUT}, {TEMP} \n")

#{time.time()} excluded from t.write

    def stop(self):
        self.running = False
        self.serial_port.close()

    def get_val(self, high, low):
        val = (high << 8) + low
        return (val / 100)

serial_reader = SerialReader(port = "COM14", name = "Idle.txt")

serial_reader.start()

```

Figure 2.13: Python Script for Data Retrieval

At the heart of the Python script lies a multithreaded architecture, using the ”threading” module to facilitate concurrent execution of data retrieval and logging operations. Through the instantiation of a dedicated thread, the script establishes a persistent communication channel with the Arduino microcontroller interfacing with the DC-DC Buck Converter, thereby enabling continuous monitoring of operational parameters without impeding the execution of other computational tasks.

Using the ”serial” module, the script establishes bidirectional serial communication with the Arduino microcontroller, enabling the transmission of command requests and reception of telemetry data packets. By encapsulating communication logic within a dedicated thread, the script achieves concurrency and responsiveness, thereby ensuring timely retrieval and logging of operational data with minimal latency.

Upon receiving telemetry data packets from the Arduino microcontroller, the script engages in real-time data processing and parsing, extracting individual operational parameters such as input and output currents, voltages, and temperature. Through meticulous byte manipulation and conversion techniques, the script interprets raw data bytes and computes calibrated parameter values, ensuring accuracy and reliability in telemetry data representation.

Subsequently, the script appends the extracted parameter values to a designated log file, enabling persistent storage and post hoc analysis of operational trends and anomalies. By adhering to established file formatting conventions and timestamping practices, the script facilitates traceability and reproducibility in data logging, thereby enabling us to correlate telemetry data with specific experimental conditions and environmental contexts.

This Python script for data retrieval from the DC-DC Buck Converter exemplifies a convergence of software sophistication and operational insight, enabling us to unlock the full potential of telemetry data. Through its seamless integration into the experimental workflow, this script catalyzes advancements in operational monitoring and data-driven decision-making, thereby propelling the field of robotic arm research towards machine learning. The data that we retrieve here is later used for training a machine learning model to deploy on MAX78002 so that it makes predictions for the robotic arm paths and the object movement.

3 Integrating Machine Learning into the Robotic Arm System

Robotic arm systems have become integral components in various industrial and manufacturing processes due to their versatility and precision in performing complex tasks. One crucial aspect in enhancing the efficiency and effectiveness of robotic arm operations lies in the ability to predict their movement paths and discern their engagement in object manipulation tasks. This thesis was born from this idea. Accurate prediction of these parameters not only optimizes the utilization of robotic arm systems but also enables proactive adjustments to ensure smooth and seamless operations.

In this thesis, we focus on the development and implementation of a machine learning model adapted to predict the path followed by the robotic arm and determine whether it is actively engaged in object manipulation tasks. Our approach uses two key features extracted from the robotic arm's operational data: energy consumption and script completion time. We posit that these features encapsulate essential information regarding the dynamics of robotic arm movements and the presence of object manipulation activities.

The energy consumption of a robotic arm serves as a proxy for discerning its operational state, particularly whether it is involved in lifting and transporting objects or operating in an idle state. Additionally, the length of time taken to execute a script reflects the temporal dynamics of robotic arm movements and can be indicative of the specific path followed during operation. By integrating these features into a machine learning framework, we aim to develop a predictive model capable of accurately anticipating the trajectory of robotic arm movements and identifying instances of object manipulation.

This thesis is motivated by the need to enhance the autonomy and adaptability of robotic arm systems in dynamic operational environments. By enabling predictive capabilities, our proposed approach empowers robotic arm systems to anticipate future actions, optimize resource utilization, and proactively respond to changing task requirements. Furthermore, the insights gained from this thesis have implications for advancing the field of robotics, particularly in domains where precise control and efficient operation are paramount.

3.1 Introduction to Machine Learning

Machine learning is a sub-field of computer science concerned with algorithms that can learn from data without explicit programming. These algorithms can identify patterns and relationships within data, enabling them to make predictions on new, unseen data points. Imagine a child learning to identify different types of animals. By observing pictures or real animals, the child can eventually distinguish between a cat and a dog, even if they haven't seen every single cat or dog breed. Similarly, machine learning algorithms can learn from labeled data with pre-defined categories to classify new, unlabeled data points. [7]

In this thesis, we explore the application of machine learning for classification tasks related to robotic arm movement. Classification models are trained on datasets where each data point represents an instance (e.g., a specific robotic arm movement) and is associated with a corresponding label (e.g., the type of movement or whether an object is being moved). These datasets can be compiled by recording sensor data (like power consumption) while the robotic arm performs various movements under controlled conditions. By analyzing the data, the model learns the subtle differences between, for example, a pick-and-place operation and a simple arm rotation. This allows the model to predict the class label (movement type) for new, unlabeled arm movements it encounters during operation.

Adaptability is one of the biggest advantages of such systems. Machine learning models can adapt to new situations or objects encountered by the robotic arm. As the model is exposed to more data encompassing a wider range of movements and object interactions, it can refine its ability to classify different movements, reducing the need for constant script revisions. This is particularly beneficial in dynamic environments where the robotic arm may need to handle unforeseen situations or objects.

Machine learning reduces the effort required to write programs. By learning from data, machine learning models can automate the classification of various arm movements, freeing up programmers from the need to explicitly define every possible movement scenario. Traditionally, programmers would need to write intricate scripts for each specific movement the robotic arm might perform. Machine learning models can learn the underlying patterns in the data, reducing the need for manual scripting and simplifying the programming process.

Real-time classification is crucial in tiny machine learning. With proper implementation, machine learning models can potentially classify movements in real-time, enabling the robotic arm to react dynamically to its environment or unexpected situations. This allows for more flexible and responsive robotic manipulation. Imagine a robotic arm tasked with assembling a product. If a component is slightly out of place, the model can detect the unexpected movement based on power consumption and allow the arm to adjust its motion accordingly, leading to more robust and adaptable robotic operations.

3.2 Tiny Machine Learning for Robotic Arm Movement Classification

We endeavor to enhance the efficiency and intelligence of robotic arm systems through the implementation of tiny machine learning techniques in this thesis. Specifically, the focus lies in the classification of robotic arm movements, discerning between various paths traversed and identifying whether the arm is engaged in the manipulation of objects. Using data pertaining to energy consumption and script completion times, a machine learning model is trained to make real-time decisions regarding the robotic arm behavior. The MAX78002 microcontroller is explored as a suitable platform for executing machine learning tasks owing to its compact size, low power consumption, and inherent capabilities for deploying neural networks. The deployment process entails optimizing the trained model for execution on the MAX78002, ensuring efficient utilization of its resources while maintaining high classification accuracy. This thesis elucidates the principles of tiny machine learning, highlights the suitability of MAX78002 for such tasks, elucidates the deployment methodology, and anticipates the performance of the model on the MAX78002 microcontroller.

Robotic arm systems are integral components of various industrial and domestic applications, facilitating tasks ranging from manufacturing to assistance in daily chores. The efficacy of these systems is contingent upon their ability to adapt to dynamic environments and perform tasks with precision and efficiency. Incorporating machine learning techniques within robotic systems presents an avenue for enhancing their intelligence, enabling autonomous decision-making and task optimization. However, traditional machine learning approaches may pose challenges in terms of resource constraints and real-time execution on embedded platforms. In response, the concept of tiny machine learning has emerged, emphasizing lightweight algorithms adapted for deployment on low-power microcontrollers. We investigate the feasibility of employing tiny machine learning for the classification of robotic arm movements, with a focus on the MAX78002 microcontroller as a viable platform for deployment.

Tiny machine learning encompasses the development and deployment of lightweight machine learning models optimized for execution on resource-constrained devices such as microcontrollers. Unlike conventional machine learning algorithms which often require significant computational resources, tiny machine learning algorithms are characterized by their efficiency in terms of memory footprint, computational complexity, and power consumption. These algorithms are specifically designed to operate within the constraints imposed by embedded systems, enabling the implementation of intelligent functionalities in devices with limited resources. [8]

The MAX78002 microcontroller, developed by Maxim Integrated, represents a compelling platform for executing machine learning tasks in embedded systems. Its notable features include ultra-low power consumption, compact form factor, and built-in hardware acceleration for neural network operations. Equipped with an Arm Cortex-M4F processor and dedicated hardware for matrix multiplication and activation functions, the MAX78002 is adept at exe-

cuting inference tasks with minimal latency and power overhead. Furthermore, its integrated peripherals and interfaces facilitate seamless integration with various sensors and actuators commonly employed in robotic systems, making it an ideal choice for implementing intelligent functionalities in such applications.

The deployment of machine learning models on the MAX78002 entails several key steps to ensure optimal performance and resource utilization. Firstly, the trained model is quantized and optimized to reduce its memory footprint and computational complexity while preserving classification accuracy. In our case we load the model parameters into a .pth file Next, the model is converted into a format compatible with the MAX78002's hardware accelerator. Subsequently, the optimized model is integrated into the firmware of the microcontroller. Finally, real-time inference is performed on incoming data, enabling the classification of robotic arm movements directly on the embedded platform.

The performance of the deployed machine learning model on the MAX78002 is contingent upon several factors, including classification accuracy, inference latency, and power consumption. Through empirical evaluation, the efficacy of the model in accurately classifying robotic arm movements can be assessed under varying conditions, including different paths traversed and object manipulation scenarios. Additionally, benchmarking experiments can be conducted to quantify the inference latency and power consumption of the deployed model, providing insights into its real-world applicability and efficiency. Overall, the MAX78002's capabilities for executing machine learning tasks in resource-constrained environments are expected to enable intelligent decision-making within robotic arm systems, thereby enhancing their versatility and efficiency.

In this thesis we are aiming to explore the potential of tiny machine learning for enhancing the intelligence of robotic arm systems, particularly in the context of movement classification and object manipulation. Having the MAX78002 microcontroller as a platform for deployment facilitates real-time inference and decision-making while mitigating the challenges posed by resource constraints. By optimizing machine learning models for execution on embedded platforms, it becomes feasible to imbue robotic systems with autonomous capabilities, paving the way for enhanced efficiency and adaptability in various applications.

3.3 Create a Machine Learning Model

Creating a machine learning model is a crucial step in developing intelligent systems, combining both theory and practice in artificial intelligence. In this thesis, we focus on classifying robotic arm movements and determining the status of object manipulation. The training phase is vital as it sets the stage for the model's ability to make decisions and predictions.

The main goal of training a model is to enable it to recognize complex patterns in the data, allowing it to make accurate predictions on new, unseen data. This ability depends on how well the model can identify important features from the input data and generalize to new situations. Therefore, training involves optimizing the model's parameters by repeatedly adjusting them to improve its predictions using labeled training examples.

Training a machine learning model involves a series of algorithmic and computational steps to adjust the model's internal parameters and minimize the difference between its predictions and the actual outcomes. This process is often aided by optimization algorithms like gradient descent, which help find the best parameter settings to reduce errors.

The success of model training also heavily relies on the quality and representativeness of the training data. Careful preprocessing and data augmentation are necessary to address issues such as data imbalance, noise, or outliers, which can affect the model's performance. Feature engineering, which involves selecting and extracting the most relevant features from the raw data, is also crucial for improving the model's accuracy.

Overall, the training phase is a complex process that brings together theoretical knowledge, computational techniques, and practical experiments. By carefully orchestrating these elements, the training process aims to equip the machine learning model with the intelligence needed to understand subtle details in the data and make informed decisions regarding robotic arm movements and object manipulation.

To deploy this trained model on the MAX78002 platform, several adaptations are required. The MAX78002 is optimized for executing neural networks efficiently in environments with limited resources.

3.3.1 Acquire and Import the Data

The acquisition and importation of data represent foundational steps in the construction of a robust machine learning model adapted to the task of classifying robotic arm movements and discerning object manipulation status. In the pursuit of empirical insights and informative features conducive to model training, the acquisition process entails the procurement of pertinent data sources capturing key facets of robotic arm behavior. The acquired data encompassing power consumption measurements and script execution durations serve as pivotal predictors in the classification endeavor in our thesis.

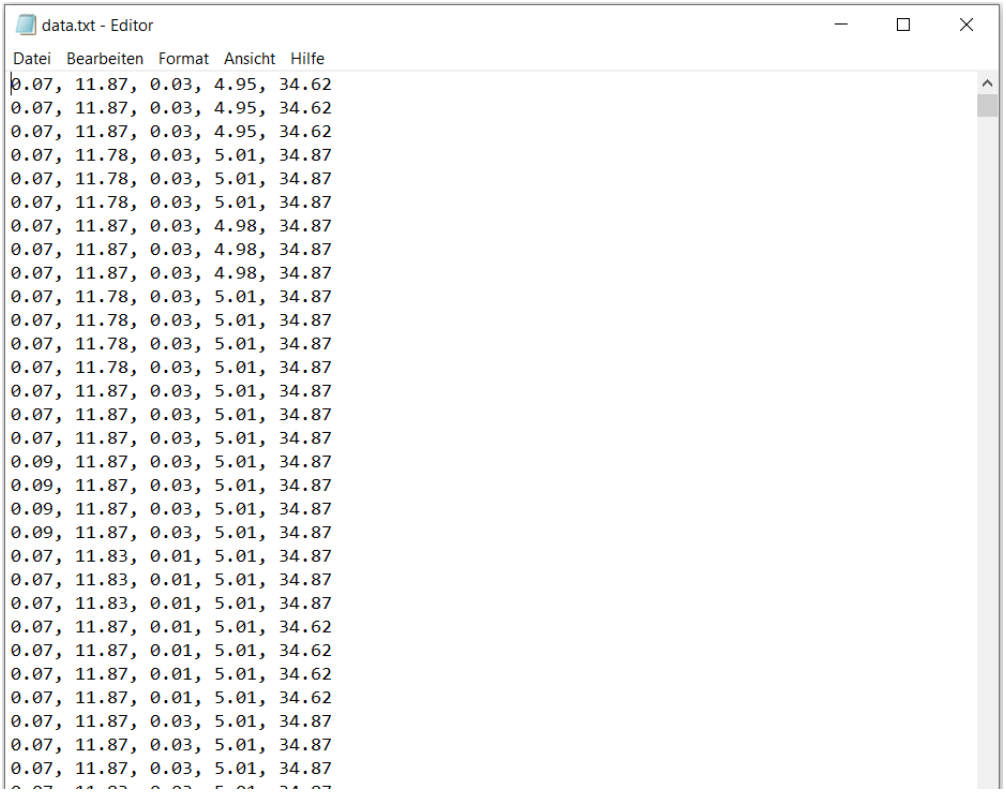

Figure 3.1: Acquired Data

The acquisition of power consumption data, elucidating the energy dynamics inherent in robotic arm operations, constitutes a fundamental pillar in the predictive modeling framework. To this end, a DC-DC Buck converter is used as a versatile instrumentation tool capable of monitoring and quantifying the electrical power drawn by the robotic arm system during operational phases. Integrated within the experimental setup, an Arduino microcontroller serves as an intermediary interface, facilitating the seamless transmission of power consumption readings from the DC-DC Buck converter to a host computing platform for subsequent analysis and model training.

In tandem with power consumption measurements, the temporal dimension of robotic arm operations is encapsulated through the acquisition of script execution durations, furnishing invaluable insights into the temporal dynamics governing arm movement trajectories. Utilizing the same DC-DC Buck converter as the instrumentation conduit, temporal data pertaining to the length of time taken for each script execution is captured with meticulous precision. This temporal dimension serves as a pivotal predictor variable in the classification task, aiding in the discernment of distinct movement patterns and trajectories traversed by the robotic arm system.

Following the acquisition phase, the acquired datasets comprising power consumption measurements and script execution durations are meticulously imported into the computational environment for subsequent processing and model training endeavors. Using established protocols and data interchange formats, such as CSV (Comma-Separated Values) or JSON (JavaScript Object Notation), the acquired datasets are seamlessly integrated into the machine learning pipeline, ensuring compatibility with prevalent data manipulation and analysis frameworks. This importation process lays the groundwork for subsequent data preprocessing and feature engineering tasks, heralding the commencement of the model training journey. [9]

In summary, the acquisition and importation of data encompass a meticulously orchestrated endeavor, epitomizing the foundational stage in the construction of a robust machine learning model geared towards classifying robotic arm movements and discerning object manipulation status. Through the judicious integration of power consumption measurements and script execution durations, imbued with meticulous precision and instrumentation, the acquired datasets serve as the cornerstone for subsequent model training and predictive modeling endeavors.

3.3.2 Prepare the Data

In the realm of real-world data acquisition, imperfections and inconsistencies often permeate the acquired datasets, potentially impeding the efficacy of subsequent model training endeavors. The imperative task of data cleaning, therefore, assumes paramount significance, constituting a pivotal pre-processing stage aimed at rectifying discrepancies and ensuring the integrity of the training data. Within the context of our robotic arm movement classification task, the data cleaning process is meticulously adapted to address specific nuances inherent in the acquired power consumption and script execution duration datasets.

In the context of our system, meticulous attention is directed towards ensuring the completeness of power consumption data for each temporal instance. Duplicates always have to be removed because deleting duplicates in a dataset before training a machine learning model is crucial for several reasons. Firstly, it helps reduce bias by preventing the model from overfitting to duplicated samples, which could skew its predictions. Secondly, removing duplicates promotes improved generalization by ensuring the model is trained on a more diverse set of examples, enhancing its ability to perform well on unseen data. Additionally, this process leads to more efficient resource utilization, as training on duplicates doesn't offer any new information but only consumes computational resources unnecessarily. Moreover, it contributes to data quality by identifying and rectifying issues that may arise from duplicate entries. Lastly, removing duplicates aids in fairness by mitigating biases that may arise from disproportionate representation of certain groups or categories in the dataset, ultimately resulting in fairer model outcomes. [10]

3.3.3 Build the Machine Learning Model

The creation of a machine learning model adapted to the classification of robotic arm movements and object manipulation status embodies a seminal endeavor, predicated upon the judicious orchestration of algorithmic architectures and computational methodologies. This section delineates the systematic construction of the machine learning model, elucidating the sequential steps encompassing data preprocessing, model architecture definition, training, evaluation, and inference. Deploying this model on a microcontroller like the MAX78002 is advantageous due to its efficient handling of complex tasks in constrained environments, and the support for PyTorch, as TensorFlow/Keras support is deprecated. [11]

The initial phase of model creation requires meticulous data preprocessing endeavors, aimed at harmonizing, standardizing, and augmenting the acquired datasets to facilitate their assimilation into the subsequent model training pipeline. Using the versatile functionalities offered by the pandas library, the acquired data, encapsulating power consumption measurements and script execution durations, is imported and organized into structured dataframes. Subsequent data manipulation tasks encompass the extraction of feature variables ('Power Consumption', 'Time') and target variables ('Object', 'Path'), laying the groundwork for subsequent model training endeavors.

```python
import pandas as pd

# Load the data with specified delimiter
data = pd.read_csv('Data.csv', delimiter=';')

# Split the data into features (X) and target variables (y)
X = data[['Power Consumption', 'Time']].values
y_object = data['Object'].astype(int).values  # Only 'Object' column for binary classification
y_path = pd.get_dummies(data['Path']).values  # Only 'Path' column for multi-class classification
```

Figure 3.2: Data Preprocessing

In this code, we load the data from a CSV file and split it into features and labels. This step is crucial as it structures the raw data into a format suitable for model training, ensuring the input data ('Power Consumption' and 'Time') and the labels ('Object' and 'Path') are clearly defined.

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Split the data into training and testing sets
X_train, X_test, y_object_train, y_object_test, y_path_train, y_path_test = train_test_split(
    X, y_object, y_path, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Figure 3.3: Data Splitting and Standardization

Following data importation, the dataset is divided into training and testing subsets. Standardization is subsequently applied to the feature variables to ensure uniform scaling, a prerequisite for optimal model performance.

Here, we divide the data such that 80% is allocated for training and 20% for testing. Standardizing the data ensures that each feature contributes equally to the model's training process, thereby enhancing model accuracy.

To enable the use of the data with PyTorch, it is converted to tensors. DataLoaders are then created to facilitate efficient batch processing during the training phase.

```python
import torch
from torch.utils.data import DataLoader, TensorDataset

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_object_train_tensor = torch.tensor(y_object_train, dtype=torch.float32)
y_object_test_tensor = torch.tensor(y_object_test, dtype=torch.float32)
y_path_train_tensor = torch.tensor(y_path_train, dtype=torch.float32)
y_path_test_tensor = torch.tensor(y_path_test, dtype=torch.float32)

# Create DataLoader
train_dataset_object = TensorDataset(X_train_tensor, y_object_train_tensor)
test_dataset_object = TensorDataset(X_test_tensor, y_object_test_tensor)
train_loader_object = DataLoader(train_dataset_object, batch_size=32, shuffle=True)
test_loader_object = DataLoader(test_dataset_object, batch_size=32, shuffle=False)

train_dataset_path = TensorDataset(X_train_tensor, y_path_train_tensor)
test_dataset_path = TensorDataset(X_test_tensor, y_path_test_tensor)
train_loader_path = DataLoader(train_dataset_path, batch_size=32, shuffle=True)
test_loader_path = DataLoader(test_dataset_path, batch_size=32, shuffle=False)
```

Figure 3.4: Conversion to Tensors and DataLoader Creation

These steps convert the data into a format compatible with PyTorch (tensors) and set up DataLoaders to manage data batching. This is vital for handling large datasets efficiently during training.

Central to the model creation process is the definition of algorithmic architectures capable of discerning intricate patterns and relationships latent within the input data, thereby facilitating accurate predictions pertaining to robotic arm behavior. Here, we define two neural network models: one for object prediction (binary classification) and another for path prediction (multi-class classification).

```python
import torch.nn as nn

# Define the neural network model for object prediction
class ObjectPredictionModel(nn.Module):
    def __init__(self):
        super(ObjectPredictionModel, self).__init__()
        self.fc1 = nn.Linear(2, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x
```

Figure 3.5: Model Architecture Definition for the Object

Each neural network consists of several layers. The object prediction model ends with a sigmoid activation function to handle binary classification, while the path prediction model uses a softmax activation function for multi-class classification.

```python
# Define the neural network model for path prediction
class PathPredictionModel(nn.Module):
    def __init__(self):
        super(PathPredictionModel, self).__init__()
        self.fc1 = nn.Linear(2, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 4)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.softmax(self.fc3(x))
        return x
```

Figure 3.6: Model Architecture Definition for the Path

With the neural network architectures delineated, the model training process commences, characterized by the iterative adjustment of model parameters to minimize the disparity between predicted and actual outcomes. Using the Adam optimizer and appropriate loss functions adapted to the respective classification tasks, the neural network models are compiled, thereby configuring the optimization strategies and evaluation metrics governing the training process.

```python
import torch.optim as optim

# Instantiate models
model_object = ObjectPredictionModel()
model_path = PathPredictionModel()

# Define loss and optimizer
criterion_object = nn.BCELoss()
optimizer_object = optim.Adam(model_object.parameters(), lr=0.001)

criterion_path = nn.CrossEntropyLoss()
optimizer_path = optim.Adam(model_path.parameters(), lr=0.001)

# Train the model for object prediction
num_epochs = 50
for epoch in range(num_epochs):
    model_object.train()
    for X_batch, y_batch in train_loader_object:
        optimizer_object.zero_grad()
        outputs = model_object(X_batch).squeeze()
        loss = criterion_object(outputs, y_batch)
        loss.backward()
        optimizer_object.step()
```

Figure 3.7: Model Training

The training loop runs for 50 epochs. During each epoch, the model parameters are updated to minimize the loss, enhancing the model's accuracy over time.

Upon convergence of the model training process, the trained neural network models are subjected to rigorous evaluation, aimed at quantifying their predictive efficacy and generalization performance on unseen data instances.

```python
from sklearn.metrics import accuracy_score

# Evaluate the model for object prediction
model_object.eval()
with torch.no_grad():
    outputs = model_object(X_test_tensor).squeeze()
    loss_object = criterion_object(outputs, y_object_test_tensor)
    accuracy_object = accuracy_score(y_object_test, (outputs > 0.5).int().numpy())

print("Object Prediction Test Loss:", loss_object.item())
print("Object Prediction Test Accuracy:", accuracy_object)

# Train the model for path prediction
for epoch in range(num_epochs):
    model_path.train()
    for X_batch, y_batch in train_loader_path:
        optimizer_path.zero_grad()
        outputs = model_path(X_batch)
        loss = criterion_path(outputs, y_batch)
        loss.backward()
        optimizer_path.step()

# Evaluate the model for path prediction
model_path.eval()
with torch.no_grad():
    outputs = model_path(X_test_tensor)
    loss_path = criterion_path(outputs, y_path_test_tensor)
    accuracy_path = accuracy_score(y_path_test.argmax(axis=1), outputs.argmax(axis=1).numpy())

print("Path Prediction Test Loss:", loss_path.item())
print("Path Prediction Test Accuracy:", accuracy_path)
```

Figure 3.8: Model Evaluation

This code evaluates the models on test data, calculating the loss and accuracy to determine how well the models perform on unseen data.

Subsequently, the trained models are deployed for real-world inference tasks, enabling the prediction of object manipulation status and robotic arm movement paths based on novel input data instances.

```python
# Make predictions
new_data = pd.DataFrame({'Power Consumption': [15119], 'Time': [26.64]})
new_data_scaled = scaler.transform(new_data)
new_data_tensor = torch.tensor(new_data_scaled, dtype=torch.float32)

# Object prediction
with torch.no_grad():
    object_prediction = model_object(new_data_tensor).item()
print("Object Prediction:", int(object_prediction > 0.5))

# Path prediction
with torch.no_grad():
    path_prediction_probabilities = model_path(new_data_tensor)
    path_prediction = [23, 25, 27, 28][path_prediction_probabilities.argmax().item()]
print("Path Prediction:", path_prediction)
```

Figure 3.9: Real-World Inference

In this scenario, we demonstrate the practical application of the trained models to predict outcomes for new input data. This step illustrates the utility of the model in real-world scenarios.

3.3.4 Decision Boundaries Analysis

In the realm of machine learning, the concept of decision boundaries serves as a cornerstone for comprehending how models classify data points. Decision boundaries are essentially the borders or surfaces that demarcate the separation between different classes in a feature space. These boundaries guide the model in determining which class a new data point belongs to based on its features. [12]

Decision boundaries offer a visual and analytical representation of how a machine learning model partitions the feature space. By examining these boundaries, researchers and practitioners can gain insights into the underlying logic of the model's decision-making process. For instance, linear boundaries indicate that the model relies on linear relationships between features and classes, while non-linear boundaries suggest more complex interactions.

Analyzing decision boundaries aids in evaluating the performance and robustness of machine learning models. By visualizing the separation between classes, one can identify areas of overlap or ambiguity where the model may struggle to make accurate predictions. This insight is invaluable for refining the model architecture, fine-tuning hyperparameters, or augmenting the dataset to improve overall performance.

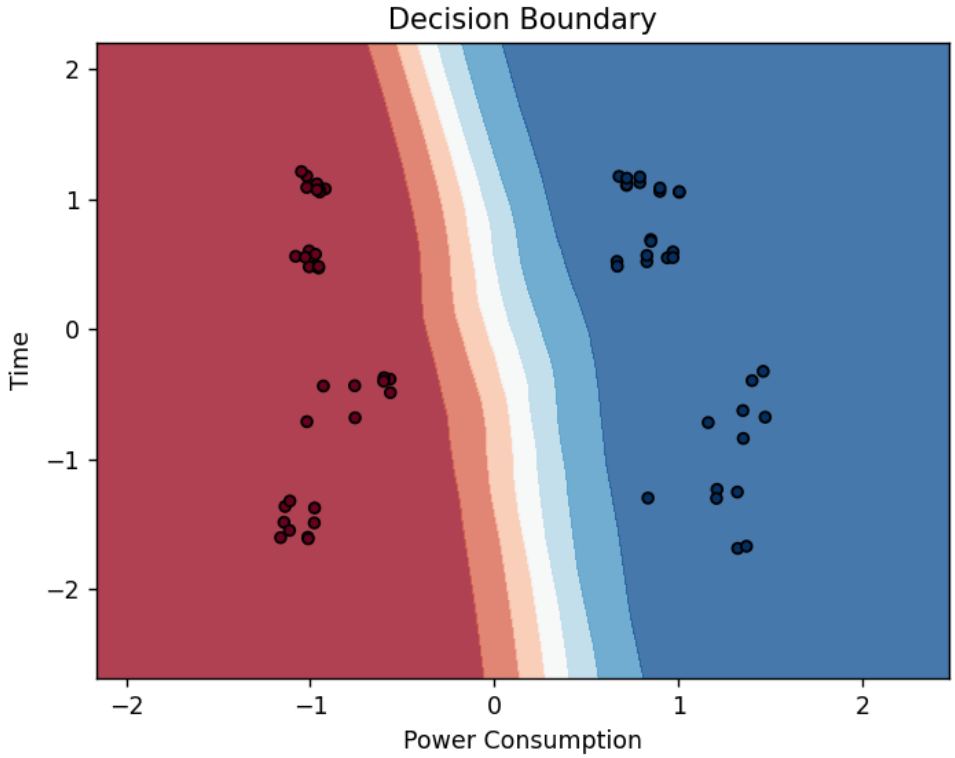

Figure 3.10: Decision Boundary for our Models

Decision boundaries provide a glimpse into the intrinsic structure of the data. They reveal patterns, clusters, and relationships within the feature space that influence classification outcomes. By examining the shape, smoothness, and complexity of these boundaries, researchers can discern the inherent characteristics of the dataset, thereby guiding feature selection, engineering, and preprocessing efforts.

Our object prediction model utilizes a binary classification task to predict whether an object belongs to a certain category. The decision boundary in this case is delineating between two classes: object present (1) and object absent (0). Visualizing the decision boundary in a 2D space defined by 'Power Consumption' and 'Time' features reveals how the model separates instances with and without the object. The boundary's shape and position illustrate the model's decision-making process, showing which regions of the feature space correspond to each class.

For the multi-class classification task of predicting the path, the decision boundaries represent the separation between different path categories. The model's decision boundary is more complex compared to the binary classification case, as it must distinguish between multiple classes. Visualizing the decision boundary in a reduced-dimensional space (e.g., using PCA or t-SNE) helps in comprehending how the model partitions the feature space into distinct paths. Understanding the decision boundaries aids in discerning the model's ability to differentiate between various path options and identifying potential areas of ambiguity or misclassification.

In conclusion, decision boundaries analysis serves as a powerful tool for unraveling the intricacies of machine learning models and their interaction with data. By delving into the geometry of these boundaries, researchers can glean profound insights into model behavior, dataset characteristics, and avenues for improvement. Incorporating decision boundaries analysis into the research workflow not only enhances the interpretability and performance of machine learning models but also fosters a deeper understanding of the underlying phenomena driving predictive outcomes.

3.3.5 Decision Trees for Interpretable Machine Learning Models

In the realm of machine learning, the interpretability of models is often as crucial as their predictive performance. Decision trees, a fundamental concept in machine learning, offer a transparent and interpretable approach to understanding the decision-making processes of models. In this section, we delve explore the utility and significance of decision trees within the context of our machine learning models for object and path prediction. [13]

Decision trees are hierarchical structures that mimic human decision-making processes by recursively partitioning the feature space based on the values of input features. This inherent interpretability makes decision trees invaluable for tasks where understanding the rationale behind predictions is paramount.

Decision trees consist of nodes, branches, and leaf nodes. Each node represents a feature, each branch denotes a decision based on that feature, and each leaf node represents a class label or a value. By traversing the tree from the root to the leaf nodes, one can decipher the series of decisions that lead to a particular prediction.

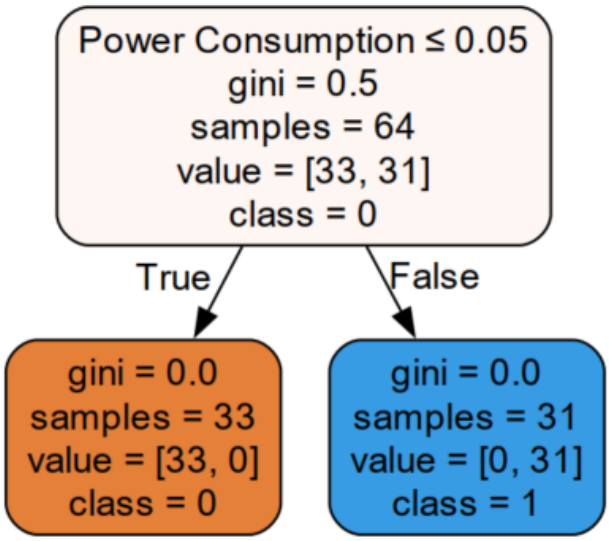

Figure 3.11: Decision Tree for the Object

In the object prediction model, the decision tree embodies a series of hierarchical decisions based on the features of power consumption and time. At each node of the tree, a specific feature is evaluated, and a decision is made based on its value. By following the branches of the tree from the root to the leaf nodes, we uncover the sequence of conditions leading to the classification of objects. The simplicity and transparency of the decision tree empower stakeholders to comprehend the reasoning behind each prediction, thereby instilling confidence in the model's outputs.

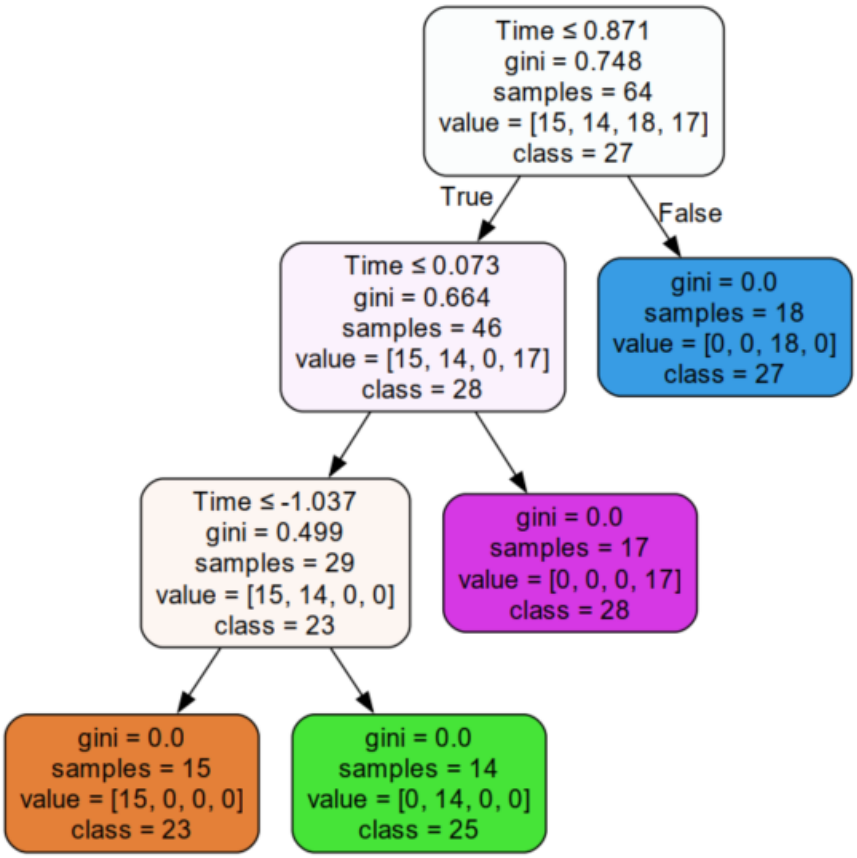

Figure 3.12: Decision Tree for the Path

Similarly, the decision tree in the path prediction model offers a roadmap for understanding the selection of paths based on input features. The tree branches represent decisions made at different levels of feature space, guiding the model towards the most appropriate path designation. Through visualization of the decision tree, we gain insights into the hierarchical nature of path selection, discerning the pivotal features that influence the model's choices. This transparency facilitates model interpretation and enables stakeholders to grasp the underlying logic governing path predictions.

In conclusion, decision trees serve as indispensable tools for unraveling the decision-making processes embedded within our machine learning models. Through meticulous exploration and visualization of decision trees, we gain invaluable insights into the features and decision boundaries shaping our models' behavior. Embracing decision trees as integral components of our machine learning pipelines not only enhances interpretability but also fosters trust and transparency, paving the way for the responsible deployment of AI systems.

3.4 Deploy the Machine Learning Model on the MAX78002

The MAX78002 is a powerful microcontroller developed by Maxim Integrated, designed specifically for edge AI applications. It integrates an efficient AI accelerator, optimized for running deep learning models with minimal power consumption, making it ideal for battery-powered and energy-constrained environments. The MAX78002 stands out for its ability to perform complex AI tasks in real-time, which is crucial for applications like robotics, where rapid decision-making is essential. [11]

The MAX78002 is designed to operate with extremely low power consumption, which is essential for extending battery life in portable and remote devices. Another advantage of this microcontroller is that the AI accelerator in the MAX78002 allows for real-time inference, meeting the stringent latency requirements of applications such as robotics, where immediate responses are crucial.

Deploying a machine learning model on the MAX78002 involves additional complexities compared to a PC due to resource constraints, energy efficiency requirements, and the need for real-time processing. We have to follow the instructions from the AI8X training repository to set up the environment and install the necessary tools. Otherwise we will get an error during the compilation of some files that are necessary to compile the project.

First of all, we have to use Python 3.8.11 or a later 3.8.x version. We also need to use PyTorch 1.8 as TensorFlow and Keras are not supported anymore. The optimal way to work on deployment process is to use Ubuntu Linux 20.04 LTS or 22.04 LTS. We can also use a Windows 10 version 21H2 or newer but this is not recommended due to some compatibility issues and slightly degraded performance. While we can run the AI8X software in a virtual machine (VM) with Ubuntu Linux 20.04 or 22.04, setting up GPU passthrough for CUDA hardware acceleration can be complex and is not always possible. Some Nvidia GPUs support vGPU software, which allows sharing of GPU resources among VMs, but this can be costly and is not covered by standard documentation. [11]

3.4.1 Install the AI8X Training Package

First, we need to set up the environment and install the necessary tools.

```
# Clone the AI8X training repository
git clone https://github.com/analogdevicesinc/ai8x-training
cd ai8x-training

# Create a virtual environment
python -m venv ai8x-env
source ai8x-env/bin/activate

# Install the required packages
pip install -r requirements.txt
```

Figure 3.13: Install the Training Package

With these commands, we clone the repository from GitHub to our local machine. change the current working directory to ai8x-training and create a virtual environment using Python's 'venv' module. A virtual environment is an isolated environment that allows us to manage dependencies for a specific project without interfering with other projects. The environment is named 'ai8x-env' here. Then, the source command activates the virtual environment that was just created. When a virtual environment is activated, it changes the shell's environment variables to point to the isolated Python environment. This ensures that any packages installed or scripts run will use the Python interpreter and libraries from the virtual environment. And lastly, 'pip install -r requirements.txt' command uses pip, the Python package installer, to install all the dependencies listed in the requirements.txt file. The requirements.txt file typically contains a list of all the Python packages needed for the project, including their specific versions. [11]

3.4.2 Serializing the Models' State Dictionary

Serialization is the process of converting an object into a format that can be easily saved to disk and later reconstructed. In PyTorch, this involves converting the model parameters (weights and biases) into a dictionary format that can be saved as a file.

State dictionary is Python dictionary object that maps each layer to its corresponding parameters (weights and biases). In PyTorch, state_dict() is a method that returns this dictionary for a model. In PyTorch, saving the state dictionary and then loading it back is a common practice for saving and reusing models, particularly for the purposes of checkpointing, inference, and model sharing. Checkpointing is saving the model's state periodically during training so that training can be resumed from that point in case of interruptions. And, inference is saving the trained model so it can be used later to make predictions on new data without retraining. Model sharing is the distribution of trained models to other developers or deploying them into production environments. [11]

```
torch.save(model_object.state_dict(), 'object_prediction_model.pth')
torch.save(model_path.state_dict(), 'path_prediction_model.pth')
```

Figure 3.14: Create .pth Files

Both lines of code are saving the parameters (weights and biases) of two different PyTorch models to separate files. These files can later be used to load the models back with the same parameters, allowing for model persistence and reuse without needing to retrain them.

3.4.3 Create YAML Configuration Files

After creating the .pth files for the quantized models, the next step is creating YAML configuration files that specify the architecture and parameters of the quantized models. These YAML files provide important information needed for inference on the MAX78002 platform.

We need to make sure that the architecture specified in the YAML configuration files matches the architecture of the quantized models. We also have to ensure that that the configuration parameters (such as layer types, input/output sizes, activation functions) accurately represent the quantized models. And lastly, The architecture has to be compatible with the Eclipse MaximSDK (The Integrated Development Environment for the MAX78002). [11]

```yaml
---
# CHW (big data) configuration for MNIST

arch: ai85net5
dataset: MNIST

# Define layer parameters in order of the layer sequence
layers:
  - pad: 1
    activate: ReLU
    out_offset: 0x2000
    processors: 0x0000000000000001
    data_format: CHW
    op: conv2d
  - max_pool: 2
    pool_stride: 2
    pad: 2
    activate: ReLU
    out_offset: 0
    processors: 0xfffffffffffffff0
    op: conv2d
```

Figure 3.15: An Example of a .yaml File

In this example, arch specifies the architecture or model being used and dataset indicates the dataset being used for training/testing. Each element in the layers list represents a layer in the neural network. Layers are defined sequentially according to their order in the network. Each layer definition consists of parameters specific to that layer. The parameters include operations like convolution (conv2d), activation functions (ReLU), and pooling (max_pool, avg_pool), among others. Additional parameters such as padding (pad), stride for pooling (pool_stride), output offset (out_offset), and processor configuration (processors) are also specified. The data_format parameter specifies the data format, which appears to be in the Channel-Height-Width (CHW) format commonly used in deep learning frameworks like PyTorch.

The network architecture consists of several convolutional layers (conv2d) followed by activation functions (ReLU) and pooling layers (max_pool, avg_pool). Each convolutional layer is configured with specific padding, activation function, output offset, and processor configuration. The last layer is a fully connected layer with parameters for flattening the input, output width, output offset, and processor configuration. The processors parameter specifies the processor configuration for each layer. It indicates how computations are distributed across multiple cores.

Overall, this YAML file provides a detailed configuration for a CNN architecture adapted for image classification tasks on the MNIST dataset, including specifications for layer operations, activations, and processor configurations. For each new project, a unique .yaml file has to be used depending on several factors, including the architecture, the model, hardware configuration, dataset, data preprocessing, and model deployment. [11]

3.4.4 Generate Demo

We can use the ai8xize.py script to generate a demo for the MAX78002.

```
python ai8xize.py --verbose --test-dir robotic_arm_demo --prefix robotic_arm_model
--checkpoint-file robotic_arm_model.pth.tar --config-file robotic_arm_model.yaml
--device MAX78000 --compact-data --softmax
```

Figure 3.16: Generate Demo

- -verbose: This flag enables verbose output, providing more detailed information during the execution of the script.

- -test-dir: This flag specifies the directory where the generated demo will be placed. In this case, it's the robotic_arm_demo directory.

- -prefix: This flag sets the name prefix for the generated demo.

- -checkpoint-file: This flag specifies the path to the checkpoint file, which contains the trained model parameters. The script will use this checkpoint file to load the trained model.

- -config-file: This flag specifies the path to the configuration file, which contains the configuration details of the neural network model used for the demo.

- -device: This flag specifies the target device for the demo, which is MAX78002 for our research.

- -compact-data: This flag indicates that the demo should use compact data representations to optimize memory usage or data transfer.

- -softmax: This flag indicates that software Softmax functions will be added to the generated code.

After this command is successfully executed we get a demo file with all the files of our project in it. Then, we can open the project on Eclipse MaximSDK and we can flash it to the MAX78002 after modifying it if necessary.

3.4.5 Eclipse MaximSDK

Eclipse MaximSDK is an integrated development environment (IDE) specifically designed for developing applications targeting Analog Devices' MAX78000 family of microcontrollers. It consists of peripheral drivers, board support packages (BSPs), libraries, examples, and toolchains (Arm GCC, RISC-V GCC, Make, OpenOCD). It supports C and C++ and provides a comprehensive set of tools and features designed for the unique requirements of embedded system development, including support for code editing, debugging, and project management. [11]

Eclipse MaximSDK is a software development environment built on top of the Eclipse IDE platform. It offers a user-friendly interface and a rich set of tools for developing and deploying applications on Analog Devices' MAX78000 family of microcontrollers. We can download it from the manufacturer's website. After we run Eclipse MaximSDK we can open our projects.

After we open our project it will look like this:

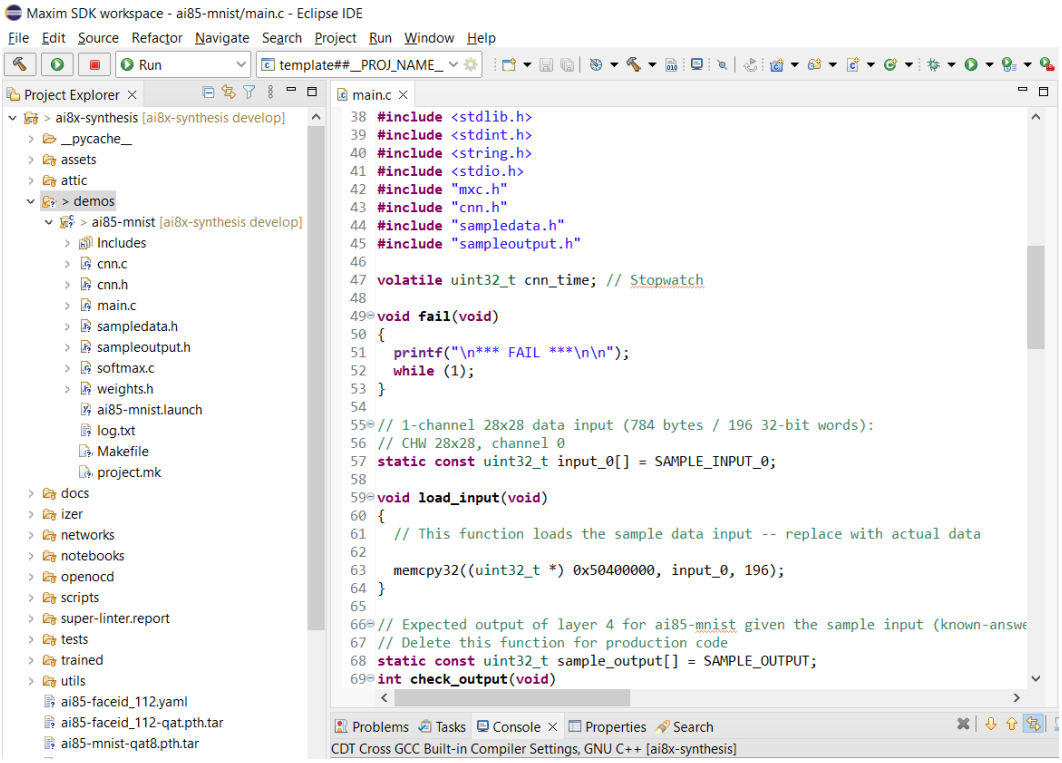

Figure 3.17: Eclipse MaximSDK

Then, we can flash our software onto the MAX78002 when it is ready.

3.4.6 Flash onto the MAX78002

Flashing software onto a microcontroller is the process of transferring and permanently storing firmware or a program onto the non-volatile memory of the microcontroller. This procedure is critical for embedding software into hardware systems, enabling the microcontroller to execute specific tasks as defined by the program code. The process involves several key steps and components. The first step involves writing the program code in a high-level language such as C or C++. This code is then compiled into machine code, generating a binary file or a hex file. These files are in a format suitable for the microcontroller's architecture. [11]

To facilitate the transfer of the binary file to the microcontroller, a programming interface is used. This can be a dedicated hardware programmer or an in-circuit programming (ICP) tool. Common interfaces include JTAG (Joint Test Action Group), SPI (Serial Peripheral Interface), and UART (Universal Asynchronous Receiver-Transmitter). During the flashing process, specific communication protocols are employed to ensure the accurate transmission of data. These protocols are designed to handle the handshaking, error checking, and data integrity verification required for reliable programming.

Microcontrollers typically contain embedded flash memory, which is a type of non-volatile memory that retains data without power. Flash memory is divided into blocks or sectors, which can be individually erased and programmed. This attribute is essential for updating firmware without needing to replace the entire microcontroller.

Flashing a project onto the MAX78002 microcontroller involves several steps, including preparing the development environment, compiling the code, and using the appropriate tools to upload the compiled firmware to the MAX78002. After we import our project we have to make sure that our project is configured correctly for the MAX78002. This includes setting the correct microcontroller model in the project settings and ensuring all paths and libraries are correctly referenced.

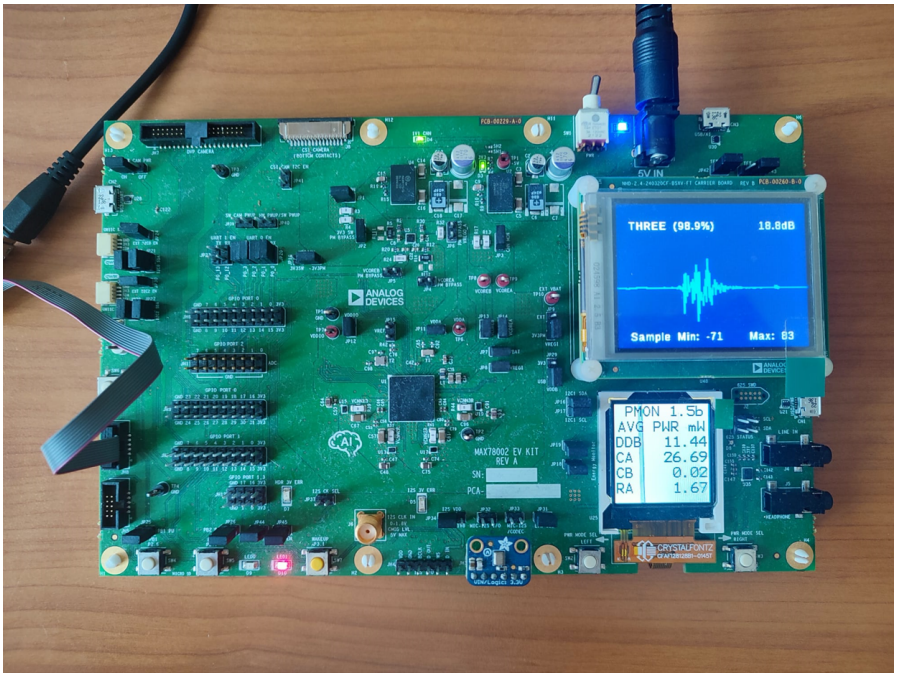

Figure 3.18: MAX78002

First, we need to power up the MAX78002. Then, we connect the SWD pins of the MAX78002 to one of the USB ports of our computer. If we see on the screen that the Keyword Spotting Demo program is running it means the microcontroller works as expected. We can test it further by pressing PB1 button. This is a program to test the MAX78002. 2 seconds after we press PB1, we can say one of the keywords such as a number from 1 to 9. It will detect what we are saying and display the number on the screen if it works well.

After we see that the MAX78002 works as we would expect we can compile our project. We use the build function in Eclipse MaximSDK to compile our project. This will generate the binary file that will be flashed onto the MAX78002. We have to watch the errors if there are any. If any errors occur, we have to solve them before proceeding.

3.5 Tiny Machine Learning for Face Recognition

In this thesis, our goal is to evaluate the applications and performance of the MAX78002. Face recognition is a good example of for this. This application consists of 2 CNN models. These are FaceDetection and FaceID CNN models. Both of these models run sequentially on the MAX78002. The face detection model detects faces in the captured image and extracts a rectangular sub-image containing only one face. The face Identification model identifies a person from their facial images by generating the embedding for a given face image. The dot product layer outputs the dot product representing the similarity between the embedding from the given image and embeddings in the database.

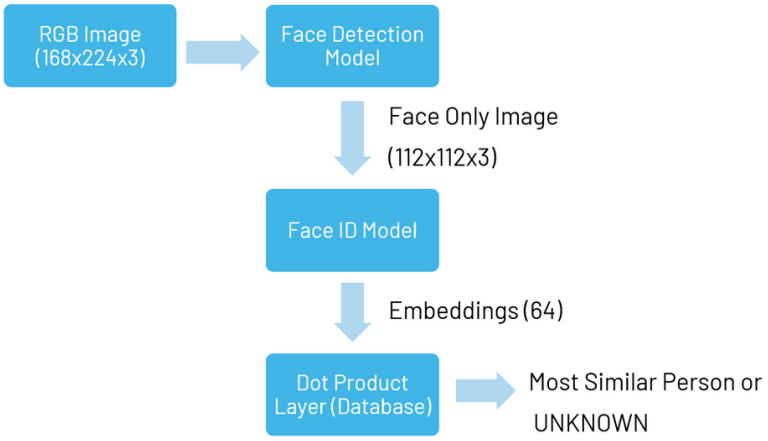

Figure 3.19: Facial Recognition System Sequential Diagram

The FaceID CNN model generates a 64-length embedding for a given image, whose distance to whole embeddings stored for each subject is calculated. The image is identified as either one of these subjects or 'Unknown' depending on the embedding distances. The FaceID CNN model is trained with the VGGFace-2 dataset using MTCNN and FaceNet models for embedding generation.

On Windows, we load the firmware image with OpenOCD in a MinGW shell.

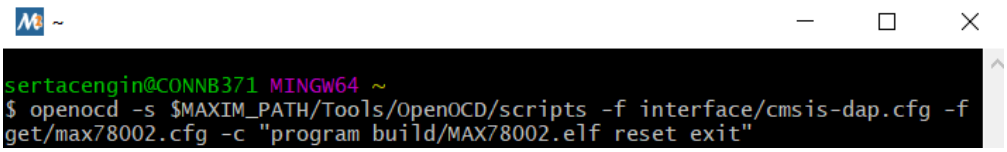

Figure 3.20: Load firmware image to MAX78002 EVKIT

Then, we can add new pictures to the database. We can either use the MAX78002 EVKIT, or the python script.

On the MAX78002 EVKIT, the Mobilefacenet_112 CNN model is used to generate the database. First, we program the demo firmware to the MAX78002. Then, we press the record button and write the name of the person. The name cannot contain more than 6 characters. Then, we press OK proceed. Then, we navigate to 'db' folder and create a folder for each person in our database and copy each person's photos into these folders. It is highly recommended to have at least 5 pictures for each person. There must be no other person in the images.

We run the pyhon script 'gen_db.sh' to generate 'db'. The script updates embeddings.h file under 'include' folder using the images under 'db' folder. Then, we can rebuild the project and load the firmware with the commands 'cd$ make clean' and then 'make'. For precise results, even small amount of rotations should be considered. The person must face directly into the camera. The following sample output can be observed in Tera Term.

```
MAX78002 Facial Recognition Demo

Camera I2C slave address is 42
Camera Product ID is 5643
Camera Manufacture ID is 0A35

Loop time: 50ms
Loop time: 49ms
Loop time: 51ms
Face detected!
Identifying face...
Recognized face: Sertac
Loop time: 102ms
Loop time: 48ms
Loop time: 50ms
Recording new face...
New face recorded successfully.
Loop time: 200ms
```

Figure 3.21: A Sample Output of Facial Recognition Project

4 The Evaluation of the MAX78002

In the scope of this thesis, we build a tiny machine learning model based on a robotic arm system and implement a face recognition model to the MAX78002. In this chapter we evaluate the MAX78002 based on various aspects.

Evaluating a microcontroller requires a comprehensive approach that examines suitability for specific applications, its capabilities, and performance.

4.1 Range of Applications of the MAX78002

The MAX78002 is a microcontroller specifically designed for artificial intelligence at the edge. Its architecture, which includes a dual-core Arm Cortex-M4 processor and a neural network accelerator, makes it highly suitable for a range of applications, particularly those requiring low-power AI processing. It is ideal for image recognition, facial recognition, object detection, and voice detection tasks. These are the most common use cases for the MAX78002. Applications in security and consumer electronics such surveillance systems and smart cameras can benefit significantly. [11]

In the realm of machine learning, especially for edge and tiny machine learning applications, the security of data and the integrity of systems are paramount. The MAX78002 microcontroller provides a distinct advantage in this regard due to its capability to operate independently of internet connectivity. This attribute not only enhances the security of the applications but also aligns with best practices in data privacy and protection.

While powerful for its size, the MAX78002 cannot compete with high-performance GPUs or CPUs in terms of raw processing power. Despite its strengths, the MAX78002 is not suitable for all types of applications, particularly those with the requirements of high-performance computing and extensive memory requirements.

The MAX78002 development ecosystem is still evolving. While tools like Analog Devices' ai8x-synthesis tool exist for converting trained models to C code, developers might encounter problems with converting their own machine learning model into a format compatible with the MAX78002. The most problematic part of this conversion is to create .pth and .yaml files. These are the files that are used by ai8x-synthesis tool to create a demo do deploy on the MAX78002. We save our model parameters in a .pth file and create a .yaml file with the configurations in it. The purpose of the .yaml file is to describe the model in a hardware-centric manner. So, the demo is created depending on the configuration in the .yaml file and then it is compatible with the MAX78002. But this part requires extensive knowledge not only about the machine

learning and the models but also about the hardware and the architecture itself. When we consider in the first place that we cannot use any library other than PyTorch and all the other limitations, we are very limited in terms of building our machine learning model. Because at the end, our model has to be suitable for being converted into a demo compatible with the MAX78002.

The MAX78002 and its surrounding tools are under continuous development. While this can lead to improvements and new features over time, it also means developers might encounter occasional bugs or compatibility issues as the ecosystem matures. That is why most developers use the MAX78002 only for object and voice detection tasks for now.

4.2 Performance Evaluation of the MAX78002

The MAX78002 is a very good microcontroller in ultra-low-power execution of neural networks, making it ideal for battery-powered edge devices. However, evaluating its performance requires a nuanced approach, considering both its strengths and limitations.

The MAX78002's on-chip CNN accelerator boasts impressive power efficiency while performing neural network computations. This allows for real-time inference at the edge of the IoT network without draining battery life excessively. The device offers various power modes, allowing developers to find a balance between processing power and power consumption. This enables optimization for specific tasks, prioritizing either speed or ultra-low-power operation depending on the application requirements.

The MAX78002 delivers high-speed performance for machine learning applications. The hardware accelerator allows for rapid inference, processing neural network models significantly faster than software-based implementations. This speed is critical for real-time applications such as object detection and voice recognition. The on-device processing capability ensures low latency, as data does not need to be transmitted to a remote server for processing. This is particularly important for applications requiring immediate response, such as autonomous systems and interactive user interfaces. [6]

Despite its strengths, the MAX78002 has limitations that could impact its performance. The microcontroller has limited on-chip memory, which is not suitable for large machine learning models. While it supports efficient model execution, very large or complex models may need to be quantized to fit within the available memory. The neural network accelerator is optimized for certain types of models, such as CNNs. More complex architectures or models requiring extensive dynamic operations may not achieve the same level of performance and may need to be simplified or adapted. Integrating the MAX78002 into existing systems may require significant development effort, particularly in adapting models to the microcontroller's architecture and optimizing code for performance and memory usage. While the MAX78002 is supported by a comprehensive development ecosystem, including the Eclipse MaximSDK and various tools for model conversion and deployment, developers may face a learning curve when transitioning from more familiar platforms or tools.

Overall, the MAX78002 microcontroller stands out as a powerful and efficient solution for edge-based machine learning applications. Its dual-core Arm Cortex-M4 processor and dedicated neural network accelerator provide a significant boost in performance, enabling fast and efficient execution of machine learning models. While it excels in speed, low latency, and power efficiency, developers must consider its memory constraints, model compatibility, and integration challenges. Overall, the MAX78002 offers a great combination of performance and efficiency, making it ideal for a wide range of applications, from real-time processing in IoT devices to battery-powered wearable technology.

5 Results

The purpose of this thesis is to design, implement, and evaluate tiny machine learning applications using the MAX78002 microcontroller. The focus is on deploying machine learning models for a robotic arm system, specifically predicting whether the robotic arm is moving an object based on power consumption and predicting the path the robotic arm follows based on time period. The research was divided into several key phases: model development, model deployment, and performance evaluation on the MAX78002.

The initial phase of this thesis is to develop a robotic arm environment so that we can work on the data that we get from it. We were able to move the objects with the robotic arm precisely and the data acquisition from the system was ensured by Arduino UNOs. Then this data is used to train our machine learning models.

Then, we develop 2 machine learning models. Our first model is object movement prediction model. This model was designed to predict whether the robotic arm is moving an object based on the power consumed by the robotic arm that has 6 servo motors. The model was trained using a dataset containing features related to power consumption and corresponding labels indicating whether an object was being moved. Our second model is path prediction model. This model aimed to predict the path followed by the robotic arm based on the time period of the movement. The dataset included features related to the time taken for various movements and labels indicating the specific path followed by the robotic arm. Both models were developed using Python and PyTorch because there are specific requirements by the software that we use to created a demo to deploy on the MAX78002. We used neural network architectures optimized for binary and multi-class classification tasks respectively. The models demonstrated high accuracy and performance during the testing phase on a standard PC environment, indicating their potential effectiveness for the intended application. We can see the output of our machine learning models in the following image.

```
\Deploy into MAX78002>python "machine learning model.py"
Object Prediction Test Loss: 0.021723084151744843
Object Prediction Test Accuracy: 1.0
Path Prediction Test Loss: 0.9075037837028503
Path Prediction Test Accuracy: 1.0
C:\Users\sertacengin\AppData\Local\Programs\Python\Python38\lib\site-packages\sklear
n\base.py:458: UserWarning: X has feature names, but StandardScaler was fitted witho
ut feature names
  warnings.warn(
Object Prediction: 0
Path Prediction: 27
```

Figure 5.1: The Ouput of the Machine Learning Models

5.1 Challenges in Model Deployment on MAX78002

Model conversion is the most crucial part of the deployment. The MAX78002 requires machine learning models to be converted into a specific format compatible with its architecture. This involves generating .yaml configuration files and .pth checkpoint files. These files are essential for converting the trained models into a format that can be deployed on the MAX78002 using tools provided by the manufacturer. These files have to be created by the developers. Even though .pth files are created with automated python scripts, there is still more to take care of. The configuration in the .yaml file tells the ai8xize.py script how to create a demo with the model parameters in the .pth file. Our model parameters may not be suitable with our hardware architecture. Depending on the errors we get during the creation of the demos, we modify the the python script that creates the .pth files. We still have to make sure that the .yaml file configuration is compatible with our hardware architecture and the model parameters.

For this thesis, we encountered difficulties in generating the necessary .yaml and .pth files from the developed models. Despite extensive efforts to adapt the models and utilize various conversion tools, the process was not successful. This challenge underscores the complexity of deploying custom machine learning models on specialized hardware like the MAX78002.

5.2 Deployment of Face Recognition Model on MAX78002

To demonstrate the capabilities of the MAX78002, a pre-trained face recognition model was deployed as an example. This process involved using a pre-trained face recognition model that was compatible with the MAX78002 requirements. This model was selected due to its availability and compatibility with the existing tools and frameworks.

The model was successfully converted into the required format using the provided tools (ai8xize.py), and a demo was generated. This demo included various files necessary for deployment, such as the main.c file, which were then imported into the Eclipse MaximSDK for further development and testing.

The face recognition model was successfully deployed and executed on the MAX78002, demonstrating the microcontroller's capability to handle complex machine learning tasks. This process validated the potential of the MAX78002 for real-world tiny machine learning applications with a face recognition machine learning model.

In conclusion, the research successfully developed machine learning models for predicting robotic arm movements but faced significant challenges in deploying these models on the MAX78002 due to issues with model conversion. However, the successful deployment of a face recognition model demonstrated the potential of the MAX78002 for tiny machine learning applications. The performance evaluation underscored the microcontroller's capabilities in terms of speed, efficiency, and suitability for real-time, power-sensitive applications, while also highlighting areas that require further optimization and development for broader applicability.

6 Discussion

In this chapter, we discuss the results, compare our findings with the existing literature, and the contribution of our work to the advancement of knowledge in the field of tiny machine learning.

6.1 Interpretation of Results

The research aimed to develop and evaluate tiny machine learning applications on the MAX78002 microcontroller, focusing on predicting the movements of a robotic arm. Two machine learning models were developed to predict whether the robotic arm is moving an object based on power consumption and the path of the robotic arm based on the time period. These models demonstrated high accuracy during testing on a PC environment, confirming their effectiveness in the intended application.

However, the deployment of these models on the MAX78002 faced significant challenges, particularly related to the conversion of model formats (.yaml and .pth files). Despite these problems, a face recognition model was successfully deployed on the MAX78002, providing valuable insights into the microcontroller's capabilities and limitations.

The successful deployment of the face recognition model indicated the MAX78002's potential for handling complex machine learning tasks efficiently. The high inference speed and low latency observed underscore its suitability for real-time applications. The power efficiency of the MAX78002 further validates its use in resource-constrained environments, making it an attractive option for embedded systems and IoT applications.

6.2 Comparison with Existing Literature

The findings of this research align with previous studies that highlight the capabilities of the MAX78002 in executing machine learning tasks efficiently. Prior research has demonstrated the effectiveness of the MAX78002 in various applications, such as gesture recognition and voice command detection, where low power consumption and high processing speed are crucial .

However, the challenges encountered in model deployment suggest that the MAX78002 might not be as straightforward to use as other microcontrollers that support more seamless model conversion and deployment processes. This discrepancy highlights an area where the MAX78002's toolchain could be improved to enhance user experience and broaden its applicability.

6.3 Implications and Significance

The broader implications of this research are significant for the field of tiny machine learning. The MAX78002's ability to execute complex models with high efficiency suggests its potential for a wide range of applications, including real-time monitoring systems, predictive maintenance, and various IoT applications. The successful demonstration with a face recognition model indicates that other complex tasks, such as object detection and environmental sensing, could also be effectively managed by the MAX78002.

This research contributes to the advancement of knowledge by highlighting both the strengths and limitations of the MAX78002. It emphasizes the need for improved model deployment tools and more user-friendly interfaces to facilitate the adoption of the MAX78002 in diverse applications. The insights gained from this study can guide future developments in tiny machine learning and microcontroller technology.

6.4 Limitations and Future Work

While this research provides valuable insights, it is not without limitations. The primary limitation was the challenge in converting the developed models into a format compatible with the MAX78002. This methodological constraint impacted the ability to fully evaluate the models on the target hardware. Additionally, the focus was on specific machine learning tasks related to a robotic arm, which may limit the generalizability of the findings to other applications.

Future work should focus on addressing these limitations by developing more robust and user-friendly tools for model conversion and deployment on the MAX78002. Further research is needed to explore the full range of applications for the MAX78002, including its performance with other machine learning models and tasks. Investigating the integration of the MAX78002 with other sensors and systems could also provide a more comprehensive evaluation of its capabilities and limitations.

In conclusion, this research highlights the potential of the MAX78002 for tiny machine learning applications, particularly in scenarios where power efficiency and real-time processing are critical. The successful deployment of a face recognition model demonstrates the microcontroller's capabilities, while the challenges encountered underscore the need for improved tools and processes for model deployment. This study contributes to the field by providing a detailed evaluation of the MAX78002, offering insights that can guide future research and development in tiny machine learning.

7 Conclusion

This research embarked on the journey of designing, implementing, and evaluating tiny machine learning applications using the MAX78002 microcontroller. The primary objective was to assess the feasibility and performance of deploying machine learning models on a resource-constrained device like the MAX78002, which is known for its low power consumption and high efficiency in executing neural networks.

7.1 Interpretation and Implications

This research shows that the MAX78002 is great for running machine learning models when saving battery life and making super-fast decisions are important. We were able to deploy a face recognition model on the MAX78002, which proves it can be used for such applications.

Getting our own models to work on the MAX78002 was not as straightforward as with other devices. It seems the tools for converting and using models on this microcontroller need improvement to be smoother and easier to use. Making these tools better is key to getting more people to use the MAX78002 for all sorts of applications. It is crucial to have a large online community for further advancement of knowledge in any field.

7.2 Contribution to the Field of Tiny Machine Learning

This research contributes to the field of tiny machine learning by providing a comprehensive evaluation of the MAX78002 microcontroller. It highlights both the strengths and limitations of the device, offering valuable insights for future developments. The study underscores the importance of developing better tools for model deployment and suggests that with these improvements, the MAX78002 can be an even more powerful asset in the realm of tiny machine learning.

Moreover, this research opens the doors for future investigations into the full range of applications for the MAX78002. Robotic arms are being used more and more everyday in various industries. The idea of bringing robotics and machine learning systems that predicts robotics arms' behaviors together is a great opportunity to create more efficient systems.

7.3 Future Directions

Future research should focus on developing and refining tools for easier model conversion and deployment on the MAX78002. This includes creating more user-friendly interfaces and ensuring compatibility with a wider range of machine learning frameworks. Additionally, further studies should explore the integration of the MAX78002 with various sensors and systems to fully harness its capabilities in diverse applications.

Expanding the scope of applications and conducting more extensive performance evaluations will provide a deeper understanding of the MAX78002's potential and limitations. Such efforts will be instrumental in advancing the field of tiny machine learning and ensuring the effective use of microcontrollers in real-world applications.

In conclusion, this research has demonstrated the potential and challenges of deploying tiny machine learning applications on the MAX78002 microcontroller. While the device shows great promise in terms of power efficiency and processing speed, the deployment process requires significant improvement to be more accessible and user-friendly. The findings of this study contribute to the advancement of knowledge in the field and provide a foundation for future research and development. By addressing the these challenges and use of the MAX78002, the field of tiny machine learning can continue to evolve, offering innovative solutions for a wide range of applications.

Bibliography

[1] "Reference to a website." [Online]. Available: https://www.universal-robots.com/in/blog/robotic-arm/

[2] "Reference to a website." [Online]. Available: https://www.electrical4u.com/what-is-servo-motor/

[3] "Reference to a website." [Online]. Available: https://www.instructables.com/Running-DC-Motor-With-Arduino-Using-L298N-Motor-Dr/

[4] "Reference to a website." [Online]. Available: https://www.arduino.cc/en/Guide/ArduinoUno

[5] "Reference to a website." [Online]. Available: https://www.analog.com/en/resources/technical-articles/dc-to-dc-buck-converter-tutorial.html

[6] "Reference to a website." [Online]. Available: https://www.analog.com/media/en/technical-documentation/user-guides/max78002-user-guide.pdf

[7] A. C. Müller and S. Guido, "Introduction to machine learning with python: A guide for data scientists."

[8] P. Ing, "Reference to a website." [Online]. Available: https://cms.tinyml.org/wp-content/uploads/industry-news/tinyML_Talks-_Peter_Ing_210924.pdf

[9] "Reference to a website." [Online]. Available: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-inputs-outputs-pipeline?view=azureml-api-2&tabs=cli

[10] S. Anunaya, "Reference to a website." [Online]. Available: https://www.analyticsvidhya.com/blog/2021/08/data-preprocessing-in-data-mining-a-hands-on-guide/

[11] "Reference to a website." [Online]. Available: https://github.com/analogdevicesinc/ai8x-training?tab=readme-ov-file

[12] C. M. Bishop, "Pattern recognition and machine learning."

[13] T. H. Gareth James, Daniela Witten and R. Tibshirani, "Introduction to statistical learning."