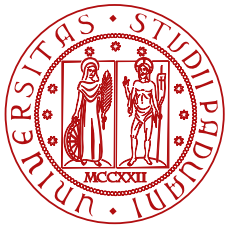


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

MicroFlow: A Rust TinyML Compiler for Neural Network Inference on Embedded Systems

Master Candidate

Matteo Carnelos

Student ID 2006677

Supervisor

Prof. Nicola Bellotto

University of Padova

Academic Year 2022/2023

Graduation Date 03/07/2023

Abstract

MicroFlow is an open-source project that aims to enable the deployment of Neural Networks on embedded systems. In particular, MicroFlow is a TinyML compiler written in Rust and specifically designed for efficiency and robustness, making it suitable for deploying applications in critical environments. To achieve these objectives, MicroFlow employs a compiler-based inference engine approach, coupled with Rust's memory guarantees and features. The software fills the gap left by the majority of the existing solutions, such as TensorFlow Lite for Microcontrollers, Embedded Learning Library, and ARM-NN, which are written in C++ and do not provide the same level of portability, efficiency, and robustness that MicroFlow achieves. MicroFlow demonstrated successful deployment of Neural Networks on highly resource-constrained devices, including bare-metal 8-bit microcontrollers with only 2 kB of RAM. Furthermore, experimental results showed that MicroFlow has been able to use 30% less Flash memory and 21% less RAM with respect to TensorFlow Lite for Microcontrollers when deploying a MobileNet for person detection on an ESP32. MicroFlow has been able to achieve faster inference compared to other state-of-the-art engines on medium-sized networks, such as a TinyConv speech command recognizer, and similar performance on bigger models. Overall, the experimental results proved the efficiency and suitability of MicroFlow for deployment in highly critical environments where resources are limited.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

AI Artificial Intelligence

AOSP Android Open Source Project

BNN Binarized Neural Network

CAGR Compound Annual Growth Rate

CNN Convolutional Neural Network

CPU Central Processing Unit

DNN Deep Neural Network

ELL Embedded Learning Library

FFT Fast Fourier Transform

FNN Feedforward Neural Network

FPU Floating-Point Unit

GPU Graphics Processing Unit

I/O Input/Output

IoT Internet of Things

MCU Micro-Controller Unit

ML Machine Learning

MSE Mean Squared Error

MSRC Microsoft Security Response Center

NN Neural Network

ONNX Open Neural Network eXchange

OS Operating System

RAM Random-Access Memory

RNN Recurrent Neural Network

ReLU Rectified Linear Unit

TFLM TensorFlow Lite for Microcontrollers

TFLite TensorFlow Lite

TF TensorFlow

TPU Tensor Processing Unit

TinyML Tiny Machine Learning

1 Introduction

Tiny Machine Learning (TinyML) is a field of Machine Learning (ML) that involves the deployment of ML models on small and low-power embedded devices. The aim of TinyML is to enable these devices to perform intelligent tasks without relying on cloud-based servers or high-performance computing systems.

The field has gained popularity in recent times due to the increasing demand for smart devices capable of performing intelligent real-time tasks without the need for processing support from the cloud, which is power consuming and involves data security and privacy risks [1]. The increasing emergence of Internet of Things (IoT) devices in houses and industries also made a great contribution to the field, making it possible to achieve even more integrated applications [2].

One of the best-known and most widely used TinyML applications is keyword spotting, also called hotword or wakeword detection [3]. This involves training a Neural Network (NN) to identify a specific sound or phrase, such as "Hey Siri" or "OK Google", that triggers a device to begin listening for user commands. The trained model is then compressed and deployed to edge devices that will perform always-on inference. By locally performing wakeword detection on the device, latency and privacy are greatly improved. This type of application is commonly used in voice assistants and smart speakers developed by companies such as Amazon, Apple, Google, and others. Wakeword detection is not the only TinyML application, others include activity recognition [4], object detection [5], predictive maintenance [6], environmental monitoring [7], and many more.

As of 2022, the global Micro-Controller Unit (MCU) market was valued at USD 25.48 billion and is expected to expand at a Compound Annual Growth Rate (CAGR) of 11.2% from 2023 to 2030 [8]. With such a large market, combined with the rise in popularity of Artificial Intelligence (AI) applications [9], the potential for TinyML-powered devices is particularly high, with a lot of big companies investing in this research field [10].

Moreover, one of the significant advantages of TinyML is its ability to operate on low-power devices, making it ideal for resource-constrained environments. This feature is particularly important in developing countries [11] where access to electricity can be a significant issue. With TinyML, devices can operate on batteries or solar power, enabling access to technology in areas with limited infrastructure.

Another benefit of TinyML is its low cost. Traditional ML models require significant computing power and hardware resources, which can be expensive. In contrast, TinyML models can run on low-cost MCUs, making this technology even more accessible.

One of the major challenges when it comes to developing applications for embedded devices is the limited computational power. TinyML applications often run on MCUs with limited memory and processing capabilities, making it difficult to deploy complex models. Therefore, both the inference engines and the models need to be optimized to run on resource-constrained hardware. Traditional ML models and inference engines may not be suitable for TinyML applications, as they require a significant amount of processing power and memory. Furthermore, the lack of standardization in the TinyML ecosystem can also pose challenges for developers. With the wide range of hardware platforms available, it can be difficult to ensure compatibility and interoperability between different components.

Some solutions have already been developed to overcome these challenges. Some popular TinyML inference engines are: TensorFlow Lite for Microcontrollers (TFLM) [12], Embedded Learning Library (ELL), ARM-NN, Plumerai, uTensor, and Tract. However, although these engines are widely used and well developed, they share some criticalities. In particular, although they are very well optimized, most of them (e.g. TFLM) require a significant amount of memory to run, which can be a challenge on small embedded devices. Moreover, they are specifically designed for 32-bit MCUs and, in some cases (e.g. ARM-NN), ARM Cortex-A processors, which can lead to compatibility and interoperability

issues. Lastly, they are all written in C and C++. Although these languages are the standard for embedded software, they are considered *memory unsafe*. This can be an issue for TinyML applications deployed in critical environments, where memory-related bugs and vulnerabilities are not acceptable. In fact, inference engines have to heavily access memory, so it would be preferable to use memory-safe languages.

To address these challenges and overcome the current limitations, *MicroFlow* has been developed. MicroFlow is a lightweight TinyML inference engine written in Rust, particularly designed for robustness and efficiency. To achieve these goals, state-of-the-art approaches combined with new techniques have been used. In particular, the following are the two major features:

Compiler-based

This approach, used in the currently fastest inference engines (e.g. Plumerai Inference Engine) [13], makes it possible to achieve highly optimized code for the specific hardware of the device. This optimization can result in significant gains in performance and power efficiency, which are critical factors in the development of TinyML applications. Moreover, a compiler-based approach makes it possible to optimize the memory footprint of the engine by using the minimum amount of allocations based on the input model. Memory allocations can be performed statically on the stack instead of dynamically on the heap, improving both performance and memory safety.

Written in Rust

Rust is a systems programming language that focuses on safety, speed, and concurrency. Specifically, Rust top priority is safety. It achieves this by enforcing strict compile-time checks that prevent common programming errors like null pointer dereferencing, buffer overflows, and data races. Moreover, Rust's performance is comparable to that of C and C++. It achieves this by eliminating runtime overheads like garbage collection and runtime type checking. All these features, combined with a rich ecosystem of libraries and tools for embedded development, make Rust an ideal language for developing a TinyML inference engine.

The combination of these factors, along with some other optimizations, made it possible to achieve a robust and memory-efficient TinyML compiler that is able to perform inference on embedded devices.

MicroFlow has been tested on a variety of MCUs. It has been possible to successfully perform accurate inference of a sample NN on Cortex-M, Cortex-A, ESP32, and AVR MCUs. Thanks to the lightweight design and the minimal memory footprint, it has been possible to achieve NN inference also on 8-bit MCUs, like the AVR ATmega328p, used in the popular Arduino Uno board.

In terms of accuracy, MicroFlow has been able to achieve similar performances with respect to other state-of-the-art engines for all the test models. For instance, it achieved a Mean Squared Error (MSE) of 0.0154 on a sine predictor model. In comparison, TFLM achieved a MSE of 0.0157. In terms of memory usage, MicroFlow out-performed TFLM by utilizing up to 60% less Flash memory and Random-Access Memory (RAM) on a ESP32 using the sine predictor model. For bigger models, such as the person detector model, MicroFlow achieved 30% less Flash usage and 21% less RAM usage on the ESP32. For the performance evaluation, MicroFlow has been able to achieve faster inference on small and medium-sized models, such as the sine predictor or the speech command recognizer, while it achieved similar performances with bigger models, such as the person detector.

One of the major limitations of MicroFlow is the number of supported operators. Currently, MicroFlow supports only the most common operators, such as the *FullyConnected* operator, making it difficult to run a large variety of NN models. In contrast, TFLM supports over 50 different operators, including mathematical operations, activation functions, and data manipulation operations. This wide range of supported operators makes it possible to deploy a much broader range of NN models. This limitation is mainly due to the fact that the project is still in its early stages of development, and thus the focus has been on developing the engine rather than adding more operators. However, there are plans to develop support for additional operators in the future, which will further enhance the engine's capabilities and make it more versatile.

In the upcoming chapters, a more detailed description of the project will be provided. In particular, Chapter 2 reviews the current state of the art and provides background information on the topics covered by the project. It also identifies the gaps in the literature that this project seeks to address. Chapter 3 delves into the design process involved in the project, including the principles, choices, and the defined software structure that were followed. Chapter 4 provides a comprehensive overview of the implementation phase of the project. It explores the practical aspects of turning the design into a functional sys-

tem, discussing the tools, technologies, and methodologies used. Subsequently, Chapter 5 focuses on the evaluation of the project. It provides a comprehensive analysis of the system's performance, effectiveness, and efficiency. Finally, Chapter 6 provides a summary and synthesis of the research findings and outcomes presented throughout the thesis. The chapter also explores potential areas for future work and additional study.

2 Related Work

This chapter provides a comprehensive review of the existing literature related to the research topic of this study. The primary objective of this literature review is to understand the current state of knowledge in the field and identify gaps or limitations in the existing research. The literature review is structured as follows: Firstly, an overview of ML techniques for embedded systems is provided, along with the challenges associated with implementing these techniques on resource-constrained devices. Subsequently, the review will go through the current state of the art in TinyML inference engines. Finally, the major existing solutions will be presented, describing their criticalities and limitations, which the present research aims to address.

2.1 Machine Learning for Embedded Systems

As introduced in Chapter 1, ML for embedded systems, also known as TinyML, is a field of ML that involves the deployment of ML models on small and low-power embedded devices. A ML application typically involves two main phases: training and inference.

The training phase involves creating a model by training it on a large dataset of labeled examples. During training, the model learns to identify patterns and relationships within the data and adjusts its internal parameters to optimize its performance. Once the model has been trained, it is typically saved as a file, which can be used for inference on new data. There are different formats to represent a ML model, some examples are Open Neural Network eXchange

(ONNX), TensorFlow (TF), TensorFlow Lite (TFLite), Coral, etc.

The inference phase involves using the trained model to make predictions on new, unseen data. This phase requires an inference engine, which takes in new data as input and outputs a prediction based on the result of the inference performed on the model. Inference can be done in real-time, which is essential for many applications, such as self-driving cars, voice assistants, and image recognition systems.

Both the training and inference phases of an ML application can be computationally intensive and require significant processing power. Therefore, in ML for embedded systems, the training phase is typically carried out on a host system with access to high-performance computing resources, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs), while the inference phase is typically carried out by an optimized inference engine that runs on the MCU.

2.1.1 Parameters Quantization

Performing the inference of a ML model directly on the MCU might not be enough. The majority of model parameters in ML applications are represented as 32-bit floating point numbers, which demand a considerable amount of memory for storage and consume a substantial amount of Central Processing Unit (CPU) cycles for computation. To overcome this issue, a technique called *quantization* is widely used to reduce the memory and computation requirements of ML models, especially for Deep Neural Networks (DNNs). The technique involves converting the floating-point values of the model parameters to fixed-point values with lower precision, typically 8-bit integers. This reduction in precision allows for significant reductions in memory usage and computation time while maintaining the accuracy of the model within an acceptable range [14].

For example, a simple NN consisting of two layers of 16 nodes each would require 1 kB of memory just to store the weight of each connection. Instead, by applying 8-bit quantization to the network, the memory required to store the weights will be only 256 B. After quantization, each parameter is mapped according to Equation (2.1).

$$r = S(q - Z) \tag{2.1}$$

Where r is the floating-point value, q is the quantized fixed-point value, and S

and Z are the quantization parameters, namely *scale* and *zero point*, respectively. The quantization process takes place before deploying the model to the device, either after training (post-training quantization) or during (quantization-aware training). Quantization parameters are calculated based on a representative sample of input data and embedded in the model. When performing inference, the engine has to use the quantized data together with the quantization parameters to perform efficient operations.

Some of the most popular ML frameworks provide support for quantization. However, the study will focus on TensorFlow Lite, which is a widely used, lightweight version of the TensorFlow framework. TensorFlow Lite provides a set of tools and libraries that enable developers to deploy and run ML models on mobile and embedded devices. In particular, TensorFlow Lite gives the possibility to train and quantize a model, saving it in the TFLite format, which is based on the FlatBuffers serialization and is specifically designed for embedded applications.

2.1.2 Model Representation

As previously mentioned, a trained model is saved as a file with a specific format, depending on the training framework. Despite the different file formats, ML models share the same underlying representation. At their core, ML models can be seen as a directed graph of operators. Each node in the graph represents an operation or a layer of the neural network, and the edges between nodes represent the flow of information through the network. The inputs to the network are usually represented as nodes at the beginning of the graph and the outputs as nodes at the end. An example visualization of a ML model graph can be seen in Figure 2.1. The example shows the graph of a NN composed of three fully connected layers of 16 nodes each, with one input and one output tensor.

To perform inference on this type of graph, the inference engine starts by feeding the input data through the first operator in the graph. The output of this operator is then passed as input to the next operator in the graph, and so on, until the final output of the model is generated.

At each step in the process, the inference engine applies the relevant mathematical operations specified by the operator to the input data, using the quantized parameters learned during the training process. This produces a new output tensor, which is then passed to the next operator in the graph. Operator

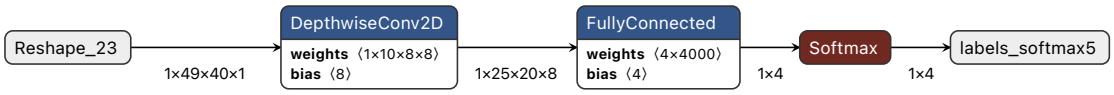


Figure 2.1: Visualization of a ML model graph.

identifiers and correspondent computations are dependent on the framework and must be supported by the inference engine. For example, the TensorFlow inference engine will support the TensorFlow set of operators but not the ONNX one. However, there are some tools to convert models between different representations.

Since TensorFlow Lite has been chosen as the training framework, this study will focus on its set of operators, which is a specialized subset of the TensorFlow set designed specifically for deployment on embedded devices.

2.1.3 Inference Engine Approaches

Typically, there are two main methods used to develop inference engines for embedded systems: the compiler-based approach and the interpreter-based approach. These two approaches differ in their fundamental operation and their impact on system performance.

Interpreter-Based Approach

In the interpreter-based approach, the inference engine functions as an interpreter, dynamically parsing and interpreting the model at runtime. This approach gives better flexibility and shorter compilation times since the model is not parsed and evaluated at compile time. However, this approach has some drawbacks. First of all, interpreting the model at runtime can introduce a significant performance overhead. The interpreter may need to perform many additional operations, such as parsing the model, performing type-checking, and managing memory allocation, each of which can slow down the inference process and make it less efficient. Moreover, performing dynamic allocations at runtime can introduce several risks for the system, such as memory leaks, heap fragmentation, and security vulnerabilities. Having to always load the interpreter has some impact on the memory footprint of the engine too. The interpreter itself can take up a significant amount of memory, regardless of the size of the network, and it cannot be optimized since the network size is not

known before runtime.

Compiler-Based Approach

On the other hand, the compiler-based approach involves translating the model into machine code that can be executed directly by the processor. In this approach, the inference engine parses and evaluates the model at compile time (on a host system), generating optimized code that can be executed quickly and efficiently. Memory management is also handled during the compilation stage, with all the memory allocations done statically. This avoids all the risks related to dynamic memory management and reduces the memory footprint of the engine. Moreover, the compiled model size is proportional to the original size of the model, meaning that inference for small models can be performed on highly constrained devices. Finally, in the compiler-based approach, parts of the model that are not required at runtime (such as operator identifiers, names, and version numbers) can be stripped away, resulting in a smaller binary size. However, the compilation stage is typically time-consuming and resource-intensive, and any changes to the model require recompilation, which can be a disadvantage in the development or tuning phase. This type of inference engine is typically called *TinyML Compiler*.

Overall, the choice between the interpreter-based and compiler-based approaches depends on the specific requirements of the application. The interpreter-based approach provides flexibility and dynamic behavior but can be computationally expensive and less memory-efficient, while the compiler-based approach provides optimized and memory-efficient code but is less dynamic and more resource-intensive at compile time. MicroFlow has been developed using a compiler-based approach to offer the best efficiency.

2.2 Programming Languages for Embedded Systems

There are a variety of programming languages that can be used for programming embedded systems. These languages typically offer low-level control over hardware resources, efficient memory management, and real-time responsiveness. Each language offers unique features and benefits that make it suitable for specific use cases and project requirements. In the upcoming sections, this review will provide an overview of the major challenges and existing solutions

that are most relevant to the study.

2.2.1 Memory Safety

Memory safety is a concept that refers to the protection of a program's memory from errors such as buffer overflows, use-after-free, and dangling pointers. In recent years, memory safety has become a critical concern in software development, particularly in low-level programming languages. The Microsoft Security Response Center (MSRC) reported in 2019 that 70% of all security vulnerabilities were caused by memory safety issues [15]. Similarly, in 2020, a report from Google reported that 70% of all *severe security bugs* in Google Chromium were caused by memory safety problems (see Figure 2.2) [16]. These issues are particularly severe in bare-metal embedded systems, which lack the protection and abstractions provided by an Operating System (OS). In such systems, there is no memory protection or process isolation, making memory safety issues even more critical.

Programming languages can be divided into two categories: memory-safe and memory-unsafe languages. Memory-safe languages provide features such as automatic memory management, safe pointer arithmetic, and bounds checking, which significantly reduce the risk of memory errors. Memory-unsafe languages require programmers to manually manage memory, leaving the responsibility of ensuring memory safety entirely to the programmer. With these languages, programmers must carefully follow memory safety best practices and use static analysis tools to detect potential memory issues. However, some memory-safe languages often rely on mechanisms such as garbage collection, which can introduce additional overhead that can impact performance.

In conclusion, memory-safe languages are a great choice when building robust and reliable software. However, it is important to consider the trade-off between safety and performance when choosing a programming language for embedded systems.

2.2.2 C

C is the most widely used language for embedded systems development due to its very low-level features. The C language is a good choice for developing embedded systems, especially those that require high performance and real-time responsiveness, as it can be used to write code that can interact directly

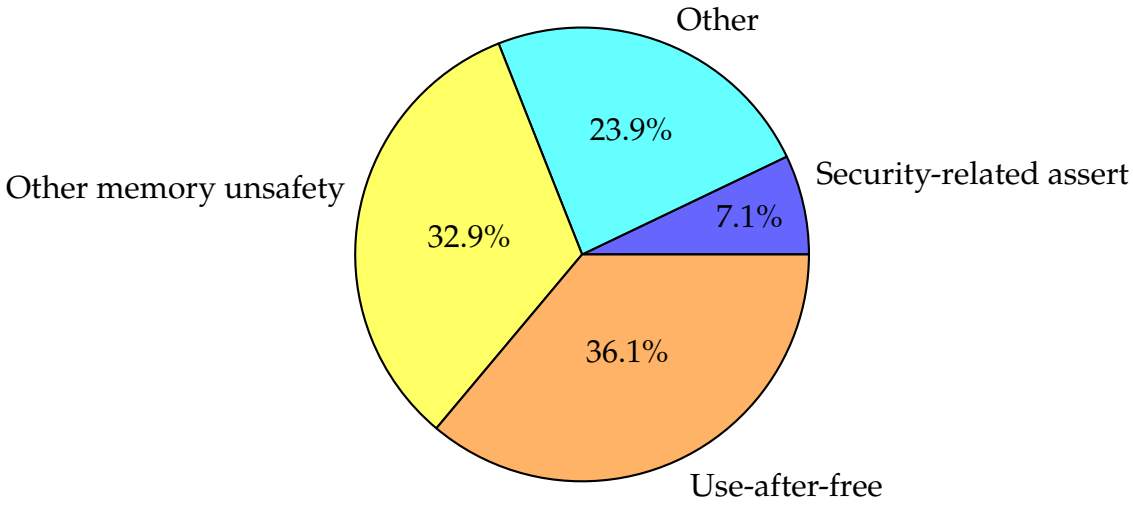


Figure 2.2: Types of high-severity security bugs found in Google Chromium since 2015 (from [16]).

with the hardware. However, the low-level access to memory without any built-in safeguards makes C a memory-unsafe language. In addition, C lacks some of the high-level language features, such as abstraction and encapsulation, which can make it more challenging to develop complex applications. Without these features, C programmers must write more low-level code to achieve the same level of functionality, which can result in code that is harder to read, maintain, and debug. Moreover, due to its longevity, there are many different versions and implementations of the language, as well as a wide range of libraries, tools, compilers, and frameworks available, which can make the C ecosystem quite messy, especially for embedded systems development.

2.2.3 C++

C++ was developed as an extension to the C programming language. It was created with the goal of adding high-level features to the C language, while retaining its efficiency and performance. C++ is also a very popular language for embedded systems development, with a lot of frameworks, libraries, and tools available. However, although the C++ ecosystem has seen significant improvements over the years, it is still lacking in certain areas, such as cross-compiling and package management. Moreover, C++ is still memory unsafe, making it not a great choice for memory-critical applications.

2.2.4 Rust

Rust is a general-purpose programming language introduced by Mozilla in 2010. It is designed to provide the efficiency and control of low-level languages like C and C++, while also prioritizing memory safety and thread safety.

To achieve this, Rust employs an ownership system, which provides a unique approach to memory management. Instead of relying on garbage collection or manual memory management, Rust uses a system of ownership and borrowing to ensure that memory is allocated and deallocated safely and efficiently. In this way, it is possible to write high-performance code without sacrificing safety or stability. This mechanism makes Rust a memory-safe programming language.

Rust also provides a number of other high-level features, including pattern matching, closures, macros, and algebraic data types. Moreover, the Rust toolchain includes a number of tools, including the Rust compiler, the Cargo package manager, and many more, making the Rust ecosystem more streamlined and less fragmented. Additionally, Rust has a central package registry that hosts a growing number of third-party tools and libraries.

When it comes to embedded systems, Rust offers different benefits. The ownership mechanism ensures at compile time that peripherals and Input/Output (I/O) lines are correctly configured and used in a mutually exclusive way. The Rust ecosystem makes it possible to easily develop portable libraries.

Overall, Rust has gained popularity in recent years due to its unique features. It is increasingly being used in a variety of applications, including the Android Open Source Project (AOSP) [17] and the Linux Kernel [18]. For these reasons, the MicroFlow project has been written in Rust to offer robustness without sacrificing efficiency.

2.3 Existing TinyML Inference Engines

There has been a growing interest in TinyML in recent years, with a number of research efforts and practical applications emerging. In the following sections, some of the major solutions developed in the field will be presented, along with their benefits and downsides.

2.3.1 TensorFlow Lite for Microcontrollers

TFLM[1] is a popular inference engine written in C++ and developed by Google. It is built on top of TensorFlow Lite and is designed to be lightweight and efficient. TFLM supports a wide variety of ML models since it supports many of the commonly used operations in ML, such as convolution, pooling, and fully connected layers. Thanks to its popularity, the framework has inspired numerous projects that explore its potential further. One such project is *MicroNets* [19], which focuses on optimizing standard NNs to enable efficient inference using TFLM.

However, TFLM uses an interpreter-based approach, which causes it to require a significant amount of memory to run and be less efficient. Moreover, the framework supports a limited number of architectures, and in particular, only 32-bit MCUs.

2.3.2 Embedded Learning Library

ELL[2] is a library developed by Microsoft and written in C++ designed for deploying ML models on resource-constrained devices. Unlike TFLM, ELL adopted a compiler-based approach, which makes it more efficient and suitable for small embedded devices.

However, ELL is currently limited to a small number of ML algorithms and models. While this may be sufficient for some use cases, it may not be suitable for developers who require more advanced or specialized ML models. As for TFLM, also ELL does not support a wide variety of devices, making it less flexible.

2.3.3 ARM-NN

ARM-NN[3] is an open-source C++ software library designed for accelerating ML models on ARM-based devices. One of the key benefits of ARM-NN is its ability to provide optimized performance for ARM MCUs, which are widely used in embedded systems. The library is designed to work with a variety of

[1] https://www.tensorflow.org/lite/microcontrollers
[2] https://microsoft.github.io/ELL/
[3] https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn

ML frameworks, including TensorFlow, Caffe, and PyTorch, making it a flexible option. However, one of the main issues is that ARM-NN may not be suitable for developers who are not using ARM-based devices. Additionally, while it is designed to work with a variety of ML frameworks, it may not support all the features or functionality of these frameworks.

2.3.4 Plumerai

Plumerai[4] is a startup company that specializes in developing ML tools and platforms for embedded systems. One of Plumerai's key products is the Plumerai inference engine, which combines state-of-the-art techniques, such as Binarized Neural Networks (BNNs) [20], with a compiler-based approach to obtain the currently world's fastest and most efficient inference engine.

However, the Plumerai inference engine is written in C++ and is currently proprietary (i.e. closed source). This limits the ability to understand how the software works, make modifications or improvements, or address potential security vulnerabilities.

2.3.5 uTensor

uTensor[5] is an extremely light-weight ML inference framework built on TensorFlow and optimized for ARM targets. The framework is implemented in C++ and leverages the ARM Compute Library for optimized matrix operations, making it well-suited for deployment only on ARM-based microcontrollers.

While the framework has gained popularity in academic and research settings, it may not be as widely adopted in industry as other ML frameworks. Moreover, it does not offer support for complex ML models or models that require more advanced optimization techniques. In this case, they may be better suited for other frameworks that offer more robust support for larger models and more complex computations.

[4] https://plumerai.com
[5] https://github.com/uTensor/uTensor

2.3.6 Tract

Tract[6] is a self-contained inference engine, written in Rust and developed by Sonos. Unlike other inference engines available in Rust, Tract is self-contained, meaning that it does not contain any bindings or dependencies to external non-Rust components. This makes Tract a memory-safe inference engine. In contrast, many other inference engines currently available in the Rust package registry contain bindings to C or C++ inference engines (like TFLM). This binding voids the guarantees of the Rust language, making this engine memory-unsafe.

One of the main limitations of Tract is that it requires the Rust standard library to run. However, the Rust standard library is not always available on bare-metal systems, such as Cortex-M MCUs, which do not have an OS and have limited processing power and memory. This limitation affects the portability of Tract, making it only suitable for devices with an OS and sufficient processing power and memory.

2.4 Summary

This chapter analyzed the major aspects involved in the development of a TinyML inference engine and presented the most relevant existing solutions for the study. In Table 2.1, it is possible to see a summary of the major features presented, along with the proposed one.

After conducting a thorough survey of existing literature and inference engines, it has been decided to develop a new inference engine written in Rust and featuring a compiler-based approach. This decision has been motivated by the robustness and safety given by the Rust programming language, combined with the efficiency and speed of a compiler-based approach. Additionally, the survey revealed that there is a lack of an existing, widely adopted solution that provides equivalent capabilities. Lastly, the choice of Rust allows for a wide variety of supported MCUs, as demonstrated in Figure 2.3, which is also missing in the literature. In conclusion, the aim of MicroFlow is to fill the current gap in the literature with a robust and efficient new TinyML inference engine. MicroFlow has been designed to be lightweight and highly portable, contributing to the advancement of the field.

[6] https://github.com/sonos/tract

Inference Engine	**Approach**	**Language**	**Minimum supported MCUs**
TFLM	Interpreter-based	C++	Bare-metal 32-bit MCUs
ELL	Compiler-based	C++	Bare-metal 32-bit MCUs
ARM-NN	Interpreter-based	C++	32-bit MCUs with OS
uTensor	Compiler-based	C++	Bare-metal 32-bit MCUs
Tract	Interpreter-based	Rust	32-bit MCUs with OS
MicroFlow	Compiler-based	Rust	Bare-metal 8-bit MCUs

Table 2.1: Summary of the major features among different TinyML inference engines.

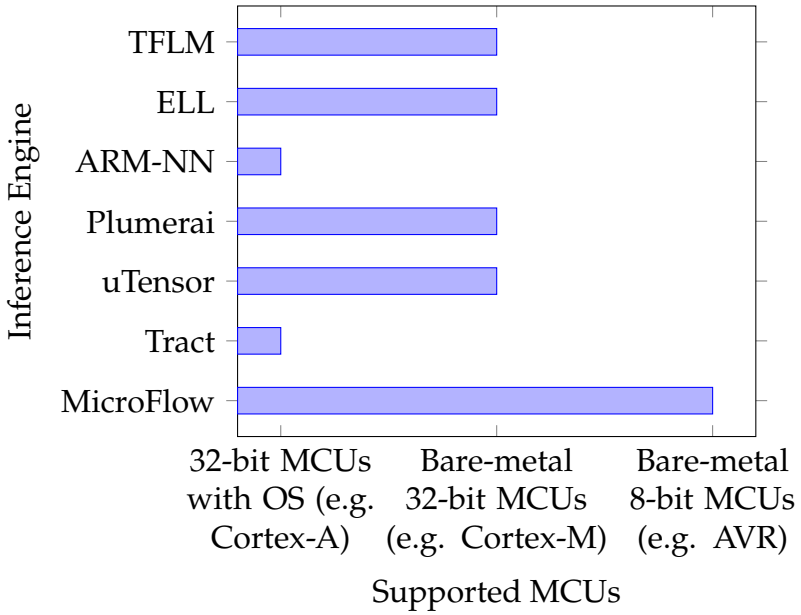


Figure 2.3: Comparison of supported MCUs across inference engines.

3 Design

This chapter covers the underlying concepts, theories, and methodologies that have shaped the design decisions throughout the project. It provides a comprehensive understanding of the design principles employed, highlighting their significance and impact on the final outcome. Furthermore, this chapter will examine how these choices align with established industry standards and best practices.

Finally, this chapter will outline the project's defined structure, which reflects the design choices made in accordance with the established principles. The structure of the project serves as a blueprint that guides the development process, ensuring coherence, organization, and adherence to the established design principles.

3.1 Principles

After conducting an extensive literature survey, as detailed in Chapter 2, the following design choices and principles have been followed and implemented.

3.1.1 Portability

The first design choice that has been made is to maximize the portability of the software. The embedded ecosystem is very fragmented, with a lot of vendor-specific frameworks and architectures. Therefore, it is very difficult to provide a single software package that works efficiently and seamlessly with all

the devices available. This usually results in a lot of work and code adaptations to make the library work on different systems.

When using traditional programming languages, such as C or C++, this challenge is even more accentuated. In fact, these languages are defined by standards but implemented by different compilers, each with its own differences and peculiarities. Therefore, designing portable software using these languages mostly results in having different implementations and providing a build system that builds the code for the target architecture.

However, the Rust programming language provides a more convenient way to build portable software. In fact, the Rust ecosystem is more centralized and managed by the community, making it more suitable for this scenario. Moreover, the language is defined by the compiler, making it possible to have a single, official instance that takes care of building the code for the target architecture. In addition, Rust comes with a built-in official toolchain manager, namely `rustup` [1], and an official package manager, namely `cargo`[2]. With these tools combined, it is possible to build portable software without having to worry about vendor-specific details.

In Rust, each unit of software is shipped inside a so-called *crate*. All the crates are collected and available in the central Rust Crate Registry[3]. In conclusion, Rust offers a compelling solution for achieving portability in software development. By leveraging Rust, it has been possible to write code that is agnostic to the underlying hardware architecture and framework.

3.1.2 Efficiency

One of the major challenges when it comes to embedded systems programming is the limited amount of available resources. For this reason, this principle has been followed to provide resource-efficient software that is able to optimize the memory usage and power consumption of the device.

To achieve this, after the survey conducted and detailed in Chapter 2, it has been decided to adopt a compiler-based approach for the inference engine. By doing so, the software benefits from advanced optimization techniques and static analysis, resulting in improved performance and memory efficiency. Addition-

[1] `https://github.com/rust-lang/rustup`
[2] `https://github.com/rust-lang/cargo`
[3] `https://crates.io`

ally, the inherent characteristics of Rust, such as its low-level control, zero-cost abstractions, and minimal runtime overhead, contribute to the overall efficiency of the software.

3.1.3 Robustness

This principle led to the design choice of having a strongly typed inference engine. In fact, running inference on NN is a very memory-intensive task that requires a lot of matrix manipulation and processing. This can result in a variety of possible memory-related bugs, such as index out of bounds, segmentation faults, stack overflows, and so on. For this reason, it has been decided to heavily rely on generic types offered by Rust to ensure at compile time that all the operations made by the runtime will be memory-safe.

In addition, the utilization of Rust plays a pivotal role in achieving robustness. Rust's design philosophy revolves around the goal of ensuring memory safety, thread safety, and data race prevention without compromising performance. These inherent features of Rust significantly contribute to the robustness of the software.

One key aspect to ensuring maximal robustness is to use external libraries (i.e. crates), which are in turn fully written in Rust. In this way, the whole program execution goes through the strict Rust rules of ownership and borrowing, ensuring a safe execution. In contrast, a lot of existing inference engines built in Rust still rely on bindings to memory-unsafe code, making the whole library memory-unsafe.

In summary, by leveraging Rust's memory safety, ownership system, strong type system, and explicit error handling mechanisms, combined with the usage of libraries fully written in Rust and a strongly typed inference engine, it has been possible to achieve a higher level of robustness.

3.1.4 Scalability

it is essential for an inference engine to be scalable. As explained earlier, NNs are represented as computational graphs consisting of a sequence of operators. The sequence, number, and hierarchy of the operators define the architecture of the model. Therefore, it is crucial to support as many operators as possible to be able to perform inference on different model architectures.

As ML models evolve and become more complex, there is a constant need to introduce new operators or custom operations to enhance their capabilities. A scalable inference engine provides the necessary infrastructure to easily integrate and implement these new operators. Moreover, the scalability of the engine enables modularity and reusability, making it possible to focus only on the specific implementation of new operators rather than dealing with boilerplate code.

3.2 Structure

After considering the principles described in the previous section, the following structure has been defined. The structure served as a fundamental framework that shaped and guided the development process described in Chapter 4. However, during the implementation phase, various adjustments and corrections were made to enhance the project's overall quality and performance.

To achieve portability, the MicroFlow project has been shipped in a Rust crate, letting the Rust tools manage it for all the numerous supported platforms. The software has been split into two components to maximize efficiency: the compiler, which resides on the host machine, and the runtime, which resides on the target MCU. The design goal is to delegate as much as possible to the compiler, resulting in a very lightweight runtime that computes only the strictly necessary computations needed at runtime. Moreover, the memory usage has also been optimized by analyzing the model at compile time and statically allocating the minimal amount of memory needed by the runtime to perform inference. An overview of the software structure can be seen in Figure 3.1, while all the implementation details will be covered in Chapter 4.

Finally, MicroFlow has been designed to offer a high level of scalability, where each new operator derives from a template consisting of two parts: the parser and the kernel. The parser runs statically in the compiler and takes care of preprocessing the model and preparing the weights for the runtime. The kernel runs on the runtime and takes care of the actual computation of the operator, propagating the input to the output. Each operator is isolated, having only the input and output interfaces with the other operators and leaving no memory trace after the execution.

Although MicroFlow has the infrastructure to support a multitude of operators, only a few have been developed for the scope of this research. In particular,

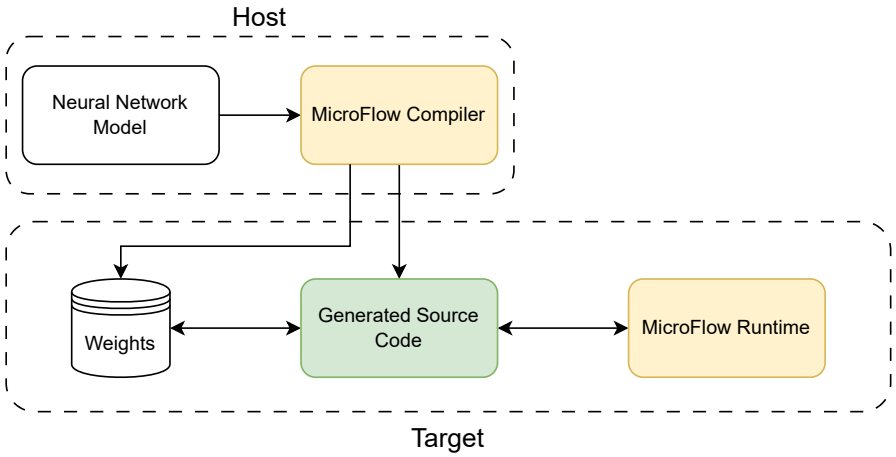


Figure 3.1: Overview of the software structure. The MicroFlow library has been split into two components: the compiler and the runtime.

only the most commonly used operators are currently supported, such as:

- FullyConnected
- Conv2D
- DepthwiseConv2D
- Softmax
- AveragePool2D
- Reshape

With these operators, it is possible to support the vast majority of NNs, such as Feedforward Neural Networks (FNNs) and Convolutional Neural Networks (CNNs). Moreover, the highly scalable structure of the project makes it possible to easily develop support for new operators in the future.

3.3 Usage

The MicroFlow crate is available in the Rust Package Registry (`crates.io`) and it can be easily added to another crate by simply including it in the manifest file (`Cargo.toml`). Once imported, MicroFlow exports the `model` macro, which can be used to annotate a `struct` and bind it to a NN model. The macro takes as input the path to the model in the TFLite format and generates a `predict()` function that, when called, performs inference on the given model.

```rust
use microflow::model;

#[model("models/sine.tflite")]
struct Sine;

fn main() {
    let y_predicted = Sine::predict(1.5);
    println("Predicted sin(1.5): {y_predicted}");
}
```

Code 3.1: Example Rust code for utilizing a sine predictor model with MicroFlow.

The parsing of the model and the generation of the source code for the function are entirely computed during compilation. The `predict()` function is embedded in the source file by the macro expansion, and it is subject to all the operations of the compiler, including memory safety checks and optimizations. An example usage can be seen in Code 3.1.

4

Implementation

The implementation chapter focuses on translating the theoretical foundations and design principles outlined in the previous chapters into practical realization. This chapter provides a detailed account of how the proposed system has been implemented, including the methodologies, tools, and techniques employed throughout the development process.

In this chapter, a detailed exploration of the software components is conducted, discussing the selected framework, libraries, and patterns. The aim is to provide comprehensive insights into the technical aspects of the implementation process. Additionally, any limitations, challenges, or issues encountered during development will be addressed. During the implementation of the project, the main reference has been *The Rust Programming Language* book [21], which provided guidance and insights into the language's features and best practices. While, for the theory and concepts behind TinyML applications, the *TinyML* book [22] served as a valuable reference. The entire project has been versioned using *Git* and published as an open-source project on *GitHub* [1].

To begin, this chapter will provide an in-depth presentation and description of the project's two primary software components: the compiler and the runtime. Next, the description will go through the implementation of the supported operators, exploring their mathematical properties and the necessary adaptations made to incorporate them into the project. Lastly, this chapter will detail the validation and testing process employed to ensure the reliability and

[1] `https://github.com/matteocarnelos/microflow-rs`

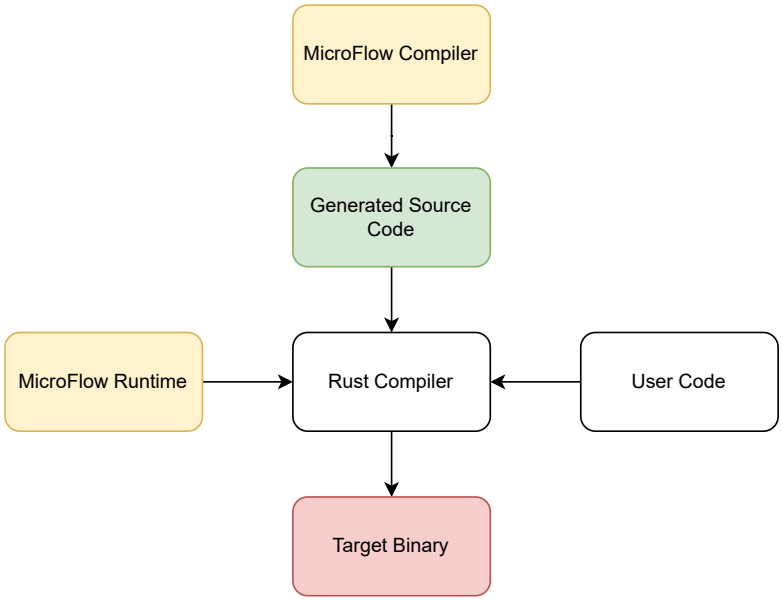


Figure 4.1: Compilation flow of the software. The MicroFlow compiler produces code that will be compiled by the Rust compiler.

effectiveness of the implemented solution.

4.1 Compiler

As described in Chapter 3, the compiler is one of the two main components of the system. The compiler takes care of processing the model and generating the code that does inference on it. The implementation is structured as a sub-crate of the main `microflow` crate, specifically named `microflow-macros`, due to its extensive use of Rust macros.

The compiler runs on the host system, and therefore it has access to the standard library. However, the generated code does not have access to the standard library, but only to the `core` crate and the MicroFlow runtime. The compiler runs as the first component of the building stage. The produced code is then built by the Rust compiler. An overview of the compilation flow can be seen in Figure 4.1.

In the subsequent sections, the description will examine the two primary components of the compiler: the utilization of Rust macros for generating the output code and the parsing process to analyze the model.

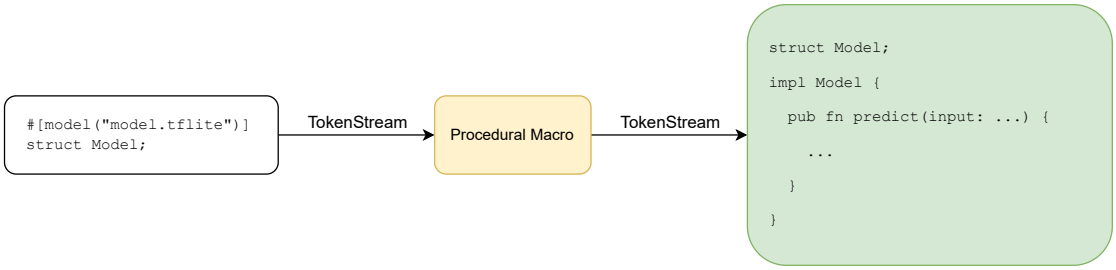


Figure 4.2: Expansion of the macro. The input tokens are expanded by the procedural macro according to the model.

4.1.1 Macros

Rust macros play a vital role in the project. They provide a powerful mechanism for code generation and *metaprogramming*, enabling developers to write code that can dynamically generate and manipulate Rust code itself. In Rust, there are two types of macros: declarative macros and procedural macros. Declarative macros, also known as *macro_rules* macros, allow for pattern matching and substitution within the code. They are defined using the `macro_rules!` keyword and are expanded at compile time. Procedural macros, on the other hand, enable code generation and transformation by implementing custom logic. They are typically defined as separate crates (in the case of this project, they are defined in the `microflow-macros` crate) and are invoked using attributes or function-like syntax. Procedural macros are expanded at the early stage of compilation (see Figure 4.1) and are more powerful than declarative macros.

In this project, it has been decided to have an attribute-like procedural macro named `model`. The macro receives as input a stream of tokens representing the macro invocation (along with the macro parameters) and outputs a stream of tokens representing the generated code. In the case of the `model` macro, it receives as input the struct to which the `predict()` function should be implemented and the path of the NN model as an argument. An example of the code expansion can be seen in Figure 4.2. Overall, macros are the core component of the compiler, and they act as the entrypoint for the entire software. They start the parser and manage the generation of the runtime calls.

4.1.2 Parsing

The parsing phase is in charge of analyzing the input model and generating the output code and memory structures accordingly. The input of the parser

comes from the macro, and it is the path of the model relative to the root of the crate.

As decided by design, MicroFlow accepts as input NN models in the TFLite format. To support other formats, such as ONNX, it is only necessary to expand the parser, providing support for new formats. Under the hood, the TFLite format uses the *FlatBuffers* serialization format. FlatBuffers offers a lightweight and efficient solution for serializing and deserializing structured data. To provide this, FlatBuffers relies on a schema definition, which defines the structure and layout of the data. Therefore, there is not a single parser for FlatBuffers, instead, it depends on the schema. Fortunately, FlatBuffers includes a powerful code generation tool called `flatc`. This compiler takes a FlatBuffers schema as input and automatically generates a parser that can handle serialization and deserialization of FlatBuffers files based on that schema.

Therefore, the MicroFlow parser first invokes the FlatBuffers deserializer generated by compiling the FlatBuffers schema of the TFLite format using `flatc`. Once the model has been deserialized, the parser proceeds to extract the operators that compose the NN, along with all the tensor dimensions, content, and relations. Subsequently, the parser generates an internal representation of the model by constructing a series of operators. Each operator is associated with its respective parameters, such as the input tensor, output tensor, weights, activation function, and other relevant attributes. Additionally, each operator also contains the stream of tokens needed by the macro to generate its runtime call. This is achieved by implementing the method `to_tokens()` of the `ToTokens` trait. For example, the FullyConnected operator in the internal representation will contain the tokens that, once included in the generated source code, will call the `fully_connected()` function in the runtime with all the required arguments.

Overall, the internal representation captures the structure and characteristics of the model, enabling further processing and manipulation for efficient execution of the NN inference. An example execution of the parser can be observed in Figure 4.3. Finally, once the internal representation is built, along with the sequence of operators involved, the parser proceeds to start the pre-processing phase.

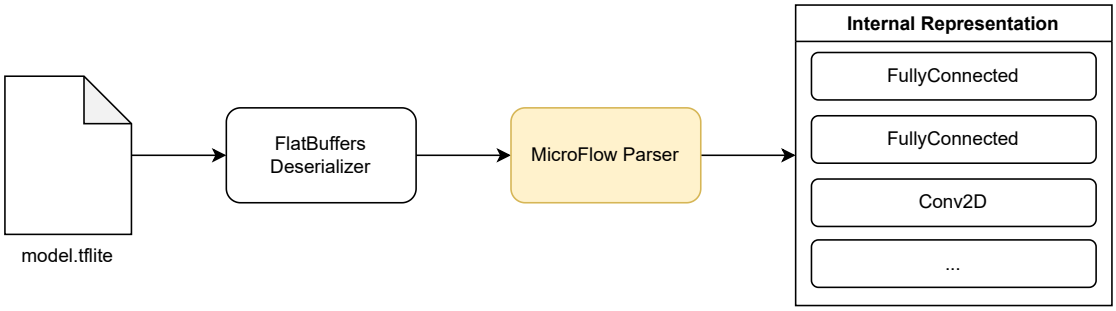


Figure 4.3: Example execution of the parsing phase. The input file is deserialized and parsed to build the internal representation.

4.1.3 Pre-processing

The pre-processing phase of the MicroFlow compiler plays a crucial role in reducing the load at runtime by performing calculations and optimizations on constant values during compile-time. This phase involves invoking the `preprocess()` function that is present in each operator of the internal representation. By offloading constant calculations to the pre-processing phase, the runtime performance is improved as it avoids redundant computations and reduces the computational overhead during inference.

The pre-processable part of each operator is obtained by analyzing the mathematical properties of the operator. In fact, all the components of the operator's formula that are not dependent directly or indirectly on the input can be delegated to this phase. A detailed walkthrough of the extraction of constant values from the operator's transfer functions is carried out in the next sections. Finally, after the pre-processing phase, the computed pre-processed values are stored in custom tensors. These custom tensors are then passed as arguments to the runtime function, ensuring that the pre-computed values are readily available for efficient execution of the inference process.

4.2 Runtime

The MicroFlow runtime is the second component of the project. As the name suggests, the runtime contains all the implementation of the operators and, more generally, everything that is executed on the target MCU. Therefore, a key difference between the compiler and the runtime is that the latter cannot rely on the standard library. In fact, on bare-metal MCUs, there is no OS and

therefore the software has to rely sorely on the core library, which contains the most essential structures and components to write Rust programs. The core library is present in every Rust program, and it replaces the standard library when the `#[no_std]` attribute is present at the top of the library file, namely the `lib.rs` file. The standard library is a superset of the core library. For this reason, a *no_std* program can run on *std* platforms without any change in the code. This means that the MicroFlow runtime can seamlessly run on platforms with an OS.

The goal of the runtime is to be as efficient as possible, both in terms of performance and memory management. The runtime functions are called by the code generated by the compiler. However, some static checks are also performed at runtime to ensure reliable execution.

The runtime receives the sequence of operators to execute along with the tensors to use. Therefore, it is not needed to evaluate the model at runtime, unlike other inference engines that do. For example, the TFLM inference engine, which is interpreter-based, consists only of the runtime, represented by the interpreter. This causes overhead due to the fact that all the operations carried out by the compiler need to be carried out at runtime by the interpreter instead.

Another responsibility of the runtime is to manage memory allocation. However, since the model is fully analyzed prior to execution, the memory needed for performing inference is statically defined. As a result, the runtime has advance knowledge of the exact amount of memory needed and the specific locations where tensors should be stored. This enables the runtime to allocate memory resources with precision, optimizing memory usage and minimizing overhead. In summary, this section delves into the implementation details of the MicroFlow runtime, covering various aspects such as the choice and usage of external libraries and the leverage on generic programming.

4.2.1 External Libraries

The runtime heavily relies on the use of external libraries to perform matrix operations and manipulations. However, the choice of the library plays a pivotal role in ensuring memory safety and compatibility with no_std environments. In fact, the chosen library has to be independent from the standard library and fully written in Rust to ensure its memory safety. Additionally, the project needs a library that can be used with static memory allocation and generics.

While there are various linear algebra Rust crates available, it is worth nothing that the majority of them prioritize flexibility over raw performance. One such example is the popular `ndarray` crate, which offers a versatile multi-dimensional array representation along with a comprehensive set of linear algebra operations. However, due to the inherent trade-off between flexibility and performance, these crates may not always offer the same level of efficiency as specialized linear algebra libraries. For this scenario, there are alternative Rust crates available that focus specifically on optimizing linear algebra operations. These crates often utilize specialized algorithms, data structures, and numerical optimizations to achieve higher computational efficiency.

After a careful search for the optimal crate, it has been decided to use the highly specialized `nalgebra` [2] crate. `nalgebra` is a powerful linear algebra library, fully written in Rust (and thus memory-safe), that provides a comprehensive set of tools and structures for mathematical operations involving vectors, matrices, and other geometric entities. It is designed to be efficient, generic, and easy to use.

One of the key features of `nalgebra` is its support for both fixed-size and dynamically-sized matrices and vectors. This possibility makes the crate usable for the project, as the fixed-size matrices and vectors can be extensively used. One additional noteworthy aspect of `nalgebra` is its strong emphasis on generics, which plays a significant role in this project. This powerful use of generics enhances the versatility and adaptability of the runtime, making it suitable for a wide range of models.

4.2.2 Generics

Generic programming is a fundamental concept in Rust that allows the creation of highly versatile and reusable code. It enables the definition of functions, structs, and traits that can work with multiple data types, providing a high level of flexibility and abstraction.

In the context of the MicroFlow project, generic programming plays a crucial role in achieving the project's goals. By leveraging generics, the MicroFlow runtime can provide a generic interface for working with various NN models, allowing users to seamlessly integrate different model architectures and quan-

[2] `https://nalgebra.org`

tization types without sacrificing performance or safety.

Additionally to conventional generics, Rust offers since version 1.51 another type of generic, const generics. Const generics are generic arguments that range over constant values rather than types or lifetimes. This allows, for instance, types or functions to be parameterized by integers. As for the other types of generics, const generics are also evaluated and expanded at compile time.

The MicroFlow runtime is heavily built on top of generic types and const generics. The first makes it possible to provide a single definition of the runtime function that works for different types, while the second allows the runtime to work for generic sizes of input and output tensors without having to evaluate the tensor dimensions at runtime. Moreover, the types and the const generics can be restricted and correlated with each other to ensure correctness at compile time.

The example reported in Code 4.1 shows how generics are used in the MicroFlow runtime to abstract the kernel from the type and have compile-time guarantees over the tensors dimensions. In particular, the example shows a FullyConnected kernel that takes as arguments the input tensor of type `T` and dimension $m \times n$, the weights tensor of type `T` and dimension $n \times p$, and the biases tensor of type `T` and dimension $p \times 1$. The kernel returns as output a tensor of type `T` and dimension $m \times p$. In addition, the type parameter `T` is constrained by the `Quantized` trait, which ensures that the type is either a signed or unsigned integer. The implementation of the operator is omitted in the example for the sake of brevity. However, for a comprehensive view of the operator and other kernels developed for this project, please refer to Appendix A, where the full Rust implementation is provided.

In this way, the `fully_connected` function works with arguments that reflect the generic parameters. An incorrect data type (as, for example, a floating point type) or an incorrect size of any of the tensors (as, for example, an input tensor of size $m \times n + 1$) will result in a compilation error. All these checks are delegated to the Rust compiler and do not take any computing resources at runtime.

4.3 Memory Management

This section covers all the aspects related to the management of the program's memory. In fact, the management of the memory is a critical aspect of the project, as it can make a difference between being able to perform inference on a highly

```rust
fn fully_connected<
    T: Quantized,
    const M: usize,
    const N: usize,
    const P: usize,
>(
    input: Tensor2D<T, M, N>,
    weights: &Tensor2D<T, N, P>,
    biases: &Tensor2D<T, P, 1>,
) -> Tensor2D<T, M, P> {
    // Implementation...
}
```

Code 4.1: Example definition of an operator kernel that uses both type and const generics. The correctness of the data is completely ensured at compile time without the need for any runtime checks.

resource-constrained MCU or not. Not only that, efficient memory utilization is essential to ensure optimal performance, minimize memory footprint, and avoid issues like memory leaks and crashes. In this section, the techniques and strategies employed in the project are explored.

4.3.1 Ownership

To understand how memory is managed in MicroFlow, it is better to first understand how memory is managed in Rust. The ownership concept is one of the fundamental pillars of Rust's memory management system. It provides a unique approach to managing memory and resource allocation while ensuring memory safety and preventing common pitfalls such as data races and memory leaks.

In Rust, every value has a single owner at any given time. Ownership represents the authority and responsibility for managing the memory used by a value. When a value is created, its owner is responsible for allocating and releasing the memory associated with it.

The key aspect associated with ownership is the concept of ownership transfer. When a value is assigned to another variable or passed as an argument to a function, ownership is transferred from the source to the destination. This ensures that there is always a clear and well-defined owner for each value, avoiding ambiguities or conflicts.

Ownership also enables Rust to enforce the borrowing rules. These rules prevent data races by ensuring that multiple references to a value can either all

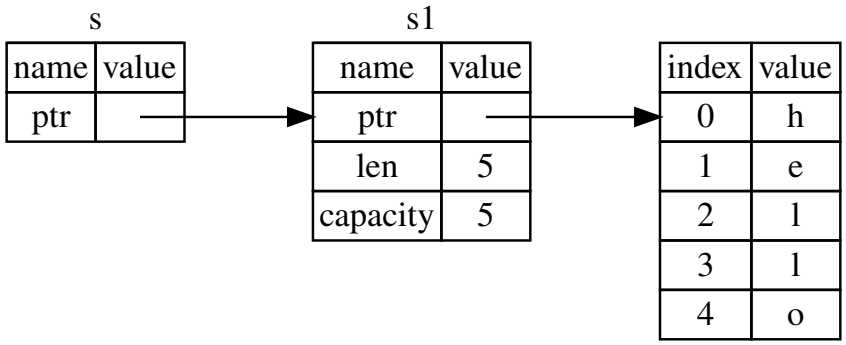


Figure 4.4: Memory diagram of an example borrow. The variable `s` represents an immutable borrow pointing at the `String` struct `s1`.

be immutable or a single mutable reference exists. This eliminates the possibility of concurrent modifications or unsynchronized access to data. A diagram representing the memory structure of a borrowing example can be seen in Figure 4.4.

When an owner goes out of scope, Rust automatically frees the associated memory by invoking the value's destructor. This deterministic and automatic memory deallocation eliminates the need for manual memory management or explicit deallocation calls. It helps prevent memory leaks and ensures that resources are released promptly.

To work with data without transferring ownership, Rust provides borrowing. Borrowing allows temporary and limited access to a value without taking ownership. Borrowing is controlled through references, which can be either immutable references (`&T`) or mutable references (`&mut T`). References enable safe and controlled sharing of data between different parts of the program.

The ownership model in Rust helps ensure memory safety by statically enforcing these ownership and borrowing rules at compile time. The Rust compiler analyzes the code and ensures that all ownership transfers, borrows, and references adhere to the rules. If the rules are violated, the compiler will raise errors, preventing potentially unsafe operations.

In MicroFlow, these concepts are used to ensure memory efficiency. First of all, since all the tensor dimensions are known at compile time, the entire execution of the runtime does not require any dynamic allocation on the heap. This results in the best possible memory utilization since everything is allocated on the stack and freed after use. By doing so, problems such as memory fragmen-

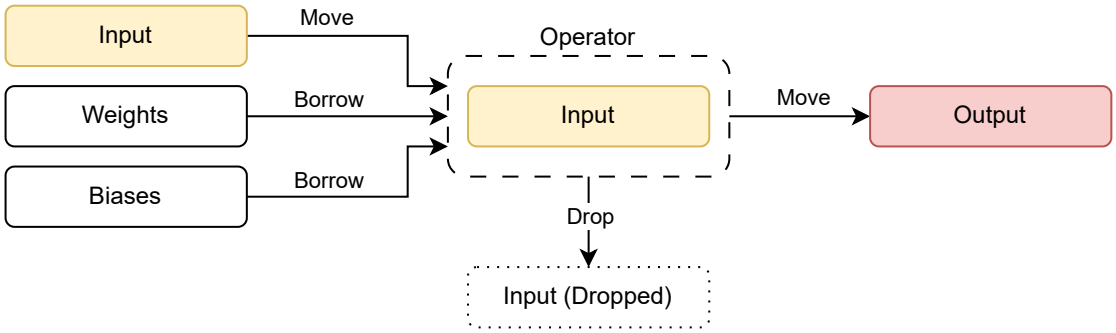


Figure 4.5: Illustration of ownership propagation during the execution of an operator. The input tensor is transferred to the operator, which assumes ownership and will release it after execution.

tation and dangling pointers are avoided. Additionally, the code becomes more portable and easy to use since the user does not have to provide a global heap allocator or a memory arena. With MicroFlow, the needed memory is statically defined and allocated on the stack.

In MicroFlow, the transfer of responsibilities occurs according to the following mechanism. Each operator takes ownership of the input tensor but immutably borrows the others. The operator then moves the output tensor to the next operator's input, which in turn will take ownership of it. This mechanism ensures that the lifetime of the input tensor is bound to the operator, making it drop the tensor once all the values have been propagated to the output. This means that, at any point in time, only the current working operator is using the minimal amount of memory possible. However, for the other tensors, such as the weights and the biases, transferring ownership is not needed as they are constant values and therefore will never be dropped. Instead, since they are used only for reading values, they can be efficiently accessed by a borrow (i.e. an immutable reference). An example diagram of this mechanism can be seen in Figure 4.5.

In terms of code, this mechanism is simply expressed by the signature of the operator's function. As it can be seen in Code 4.1, the input argument is not prefixed by anything, meaning that it will be moved. The other tensors are instead prefixed by the borrowing operator (`&`).

4.3.2 Paging

With the ownership mechanism described in the previous section, the entire layer is loaded into RAM during computation. This approach offers a balance between memory usage and performance, ensuring quick and efficient access to data when needed. However, certain MCUs have limited RAM size, which can pose a challenge when attempting to load an entire layer into memory simultaneously.

For instance, the ATmega328 MCU found in Arduino Uno boards has a limited flash size of 32 kB and only 2 kB of RAM. When attempting to perform inference on a NN with a dense layer comprising 32 fully connected neurons using this MCU, it quickly becomes apparent that there is not enough RAM available. In fact, the memory required for computing this layer alone amounts to approximately 5.184 kB. This calculation takes into account factors such as weights (32×32), 32-bit signed integer accumulators ($4 \times 32 \times 32$), and vectors containing biases, input, and output (3×32). As a result, although the flash memory can hold the entire NN, the stack overflows.

To address this issue, a solution has been implemented by loading only parts of the layer into RAM at a time, referred to as *pages*. Rather than loading the entire layer into memory, only the necessary portions are loaded and processed sequentially. This approach allows for efficient memory utilization and ensures that the MCU's limited RAM is not overwhelmed. In Figure 4.6 it is possible to see the amount of neurons, weights, and biases stored in RAM for an example layer.

In the example provided, dividing the layer into 32 pages results in a significantly reduced RAM usage of only 162 B. However, it is important to note that relaxing the space constraint by loading pages into memory would inevitably lead to increased execution time. Therefore, the decision to use paging or load the entire layer into memory depends on the specific context and application requirements. In situations where memory resources are limited and slower inference times are acceptable, the paging approach can be a viable solution. On the other hand, if memory constraints are less stringent and faster inference times are crucial, loading the entire layer into memory may be the preferred option.

In summary, this mechanism makes it possible to run inference on highly-constrained devices. However, it is essential to carefully consider the trade-off

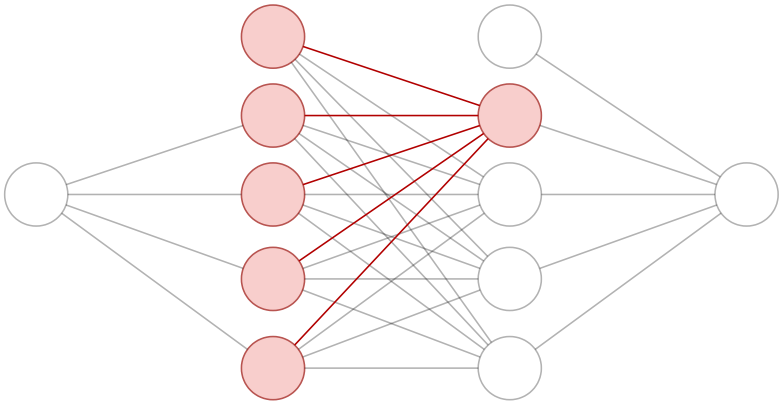


Figure 4.6: Example of paging in a simplified fully connected layer. The page, highlighted in red, contains 4 inputs, 4 weights, 1 bias, and 1 output, representing the memory requirements.

between memory usage and inference speed to determine the most suitable approach for achieving the desired performance and resource utilization balance.

4.3.3 Stack Overflow Protection

While Rust is widely known for its emphasis on memory safety, it is important to note that bare-metal Rust programs may not be memory-safe in the presence of stack overflows. For example, on ARM Cortex-M architectures, the default memory layout in RAM is structured as shown in Figure 4.7. As it can be noticed, the function call stack, also known as the *stack*, grows downwards on function calls and when local variables are created. If the stack grows too large (as shown on the right side of Figure 4.7) it collides with the `.bss + .data` region, which contains all the program's static variables. This collision results in the static variables being overwritten with unrelated data, causing an undefined behavior.

The solution to this problem is to change the memory layout of the program and place the stack below the `.bss + .data` region. With the flipped memory layout (shown in Figure 4.8) the stack cannot collide with the static variables. Instead, it will collide with the boundary of the physical RAM memory region. In the ARM Cortex-M architecture, trying to read or write past the boundaries of the physical RAM memory region produces a *hardware exception*, resulting in a Rust `HardFault` exception.

As explained earlier, MicroFlow heavily relies on stack usage. Therefore, this modified memory layout is important to guarantee the best reliability and

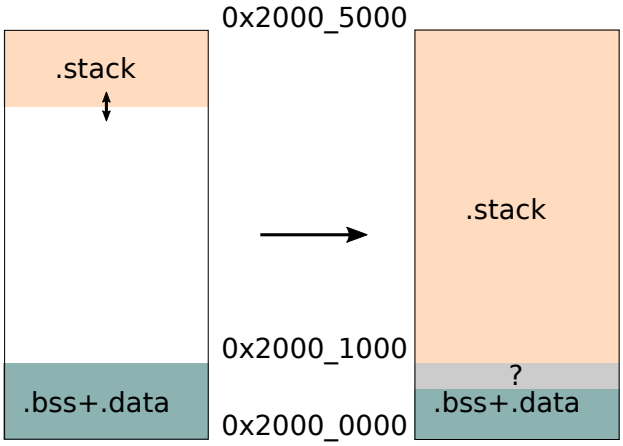


Figure 4.7: Example of stack overflow resulting in an undefined behavior on ARM Cortex-M architectures.

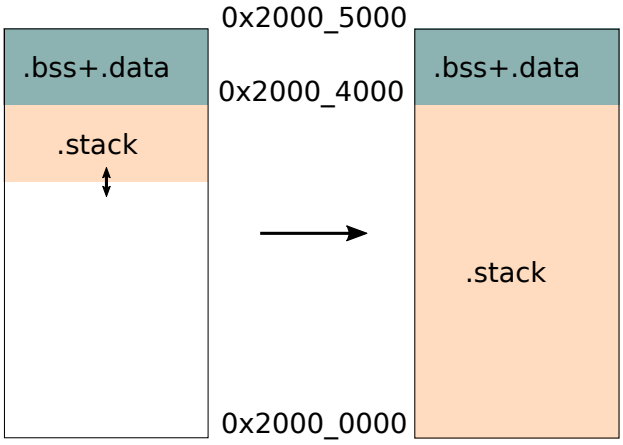


Figure 4.8: Example of flipped memory layout. The stack overflow will not result in undefined behavior but in a hardware exception.

robustness. Flipping the memory layout is not a trivial task. The start address of the stack is not the beginning of the RAM region anymore, and the beginning of the `.bss + .data` region is not the end of the RAM. Instead, the size of the `.bss + .data` region needs to be calculated during the linking process, and the start of the stack is set accordingly.

Fortunately, the `flip-link` [3] crate does exactly this. In particular, `flip-link` is a linker that can be used as a replacement for the default Rust linker (`rust-lld`) and takes care of flipping the memory layout. Currently, the crate works only for the Cortex-M architecture, but more platforms are planned to be supported in the future. Overall, by simply flipping the memory layout of the RAM, it has been possible to add zero-cost stack overflow protection to the software.

4.4 Operators

This section focuses on exploring the implementation and functionality of the operators. Operators are mathematical functions that transform input data into meaningful output. They encompass a wide range of mathematical operations, activation functions, and data manipulation techniques that are essential for the successful execution of a NN model. The design principles, algorithms, and optimizations employed in implementing these operators will be discussed, highlighting their significance in achieving accurate and efficient inference.

Operators are the building blocks of NNs. Model architectures are often described as the sequence of operators employed, represented as a computational graph. There are a multitude of operators available, with new ones being added to the literature over time. However, for the scope of this project, it has been decided to focus only on a specific set of operators, and in particular the set of most used operators for FNNs and CNNs. NN operators possess several key properties that contribute to their functionality and effectiveness. These properties include:

- Independence: Each operator functions independently, taking input tensors and producing output tensors. These tensors represent the data flowing through the network and carry the information for each layer's computation.

[3] `https://crates.io/crates/flip-link`

- Learnable Parameters: Operators often have parameters that are learned during the training process. These parameters, such as weights and biases, are adjusted through algorithms such as backpropagation.
- Activation Functions: Operators apply activation functions to introduce non-linearity into the network. Activation functions, such as Rectified Linear Unit (ReLU) or sigmoid, help in modeling complex relationships and enable the network to learn and generalize better.
- Forward and Backward Propagation: Operators facilitate both forward propagation, where data flows through the network from input to output, and backward propagation, where gradients are computed and used to update the parameters during training.
- Differentiability: NN operators are typically differentiable, meaning that their gradients can be computed. This property is crucial for backpropagation, as it allows the network to learn through gradient-based optimization algorithms.
- Non-local Operations: Some operators involve non-local operations, such as convolutional operations, which consider the spatial or temporal relationships between neighboring data points. These operations help capture local patterns and structures in the input data.
- Scalability: Operators should be designed to handle varying input sizes and be scalable to work with larger networks and datasets. This allows for flexibility and adaptability in training and deploying NNs. This scalable nature of operators also contributes to the project's overall scalability.

In addition to these properties, for TinyML applications and models, operators need to be quantized (as explained in Chapter 2). This means that instead of leveraging floating point operations, like conventional operators do, quantized operators have to sorely rely on integer operations while still providing enough accuracy. Therefore, before implementing an operator in the MicroFlow framework, it has to be quantized, meaning that its transfer function has to be converted to an integer-only function. To achieve this, a transformation of the operator's formula has been carried out for each operator.

Once the operator has been quantized, its implementation in the framework follows the design described in Chapter 3. The operator is split into two components: the parser, which runs on the compiler, and the kernel, which runs on the runtime. The goal of the parser is to prepare the kernel by preparing the input, output, and intermediate tensors and pre-processing the constant values. The goal of the kernel is to propagate the input to the output in the most efficient way possible. An overview of the operator's components can be observed in Figure 4.9.

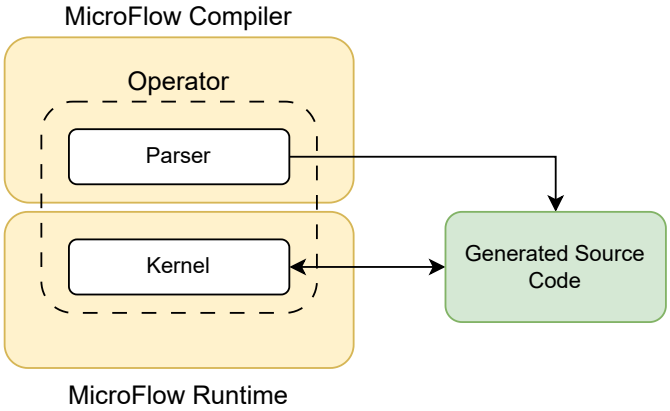


Figure 4.9: Components of an operator. The parser resides in the compiler and contributes to the generated code, the kernel resides in the runtime and contributes to the inference.

In the following sections, the analysis will go through the key operators supported by the system, highlighting the quantization and implementation procedures. Finally, the implemented activation functions will be explored.

4.4.1 FullyConnected

The FullyConnected operator, also known as the dense or linear operator, is a key building block in neural networks. It is responsible for mapping inputs from one layer to outputs in the next layer, connecting every input element to every output element. In this operator, each input element is multiplied by a corresponding weight and summed with other weighted inputs and biases. The resulting sum is then passed through an activation function to introduce non-linearity, producing the final output values.

The FullyConnected operator is characterized by its weight matrix, which represents the learned parameters of the network. Each row in the weight matrix corresponds to the weights associated with a specific output neuron, and each column corresponds to the weights connecting a specific input neuron to all the output neurons. This allows the operator to learn complex relationships between input and output features.

The FullyConnected operator is widely used in various neural network architectures, including FNNs, CNNs, and Recurrent Neural Networks (RNNs). It provides a high level of flexibility and expressive power, enabling the network to learn and represent complex patterns and dependencies in the data. Fully-

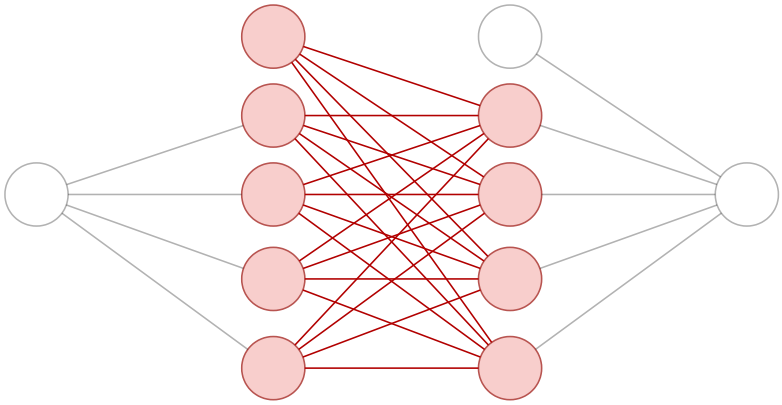


Figure 4.10: Diagram showing a typical representation of a FullyConnected layer. The neurons and connections highlighted in red represent the input, output, biases, and weights of the layer.

Connected layers are typically represented by a set of fully connected neurons, as shown in Figure 4.10.

In the following sections, the quantized version of the operator will be obtained, along with the pre-processing and implementation overview. The full implementation code of this and the other operators is reported in Appendix A.

Quantization

To obtain the quantized version of the operator, it is necessary to integrate the quantization process into its formula. This involves applying quantization to the input data, weights, and intermediate results within the operator's computations.

Given $X \in \mathbb{R}^{m \times n}$, $W \in \mathbb{R}^{n \times p}$, and $b \in \mathbb{R}^{p}$, representing respectively the input, weights, and biases of the operator, the output $Y \in \mathbb{R}^{m \times p}$ is obtained as:

$$Y_{i,j} = b_j + \sum_{k=1}^{n} X_{i,k} W_{k,j} \tag{4.1}$$

By applying the dequantization formula reported in Equation (2.1), the fol-

lowing result is obtained:

$$
\begin{aligned}
&Y_{i,j} \\
&= s_b(b_{q,j} - z_b) + \sum_{k=1}^{n} s_X(X_{q,i,k} - z_X)s_W(W_{q,k,j} - z_W) \\
&= s_b(b_{q,j} - z_b) + s_X s_W \sum_{k=1}^{n}(X_{q,i,k} - z_X)(W_{q,k,j} - z_W) \\
&= s_b(b_{q,j} - z_b) + s_X s_W \left[\left(\sum_{k=1}^{n} X_{q,i,k} W_{q,k,j} \right) - \left(z_W \sum_{k=1}^{n} X_{q,i,k} \right) \right. \\
&\qquad \left. - \left(z_X \sum_{k=1}^{n} W_{q,k,j} \right) + n z_X z_W \right] \\
&= s_Y(Y_{q,i,j} - z_Y)
\end{aligned}
\tag{4.2}
$$

Where X_q, W_q, b_q, and Y_q are the quantized versions of X, W, b, and Y, respectively, s_X, s_W, s_b, and s_Y are the scales for X, W, b, and Y, respectively, and z_X, z_W, z_b, and z_Y are the zero points for X, W, b, and Y, respectively.

Therefore:

$$
\begin{aligned}
Y_{q,i,j} = z_Y + \frac{s_b}{s_Y}(b_{q,j} - z_b) + \frac{s_X s_W}{s_Y} \left[\left(\sum_{k=1}^{n} X_{q,i,k} W_{q,k,j} \right) - \left(z_W \sum_{k=1}^{n} X_{q,i,k} \right) \right. \\
\left. - \left(z_X \sum_{k=1}^{n} W_{q,k,j} \right) + n z_X z_W \right]
\end{aligned}
\tag{4.3}
$$

In summary, by using Equation (4.3), it is possible to perform an integer-only quantized version of the FullyConnected operator. The computation of the operator can be further optimized by pre-computing constant values at compile time.

Pre-processing

When examining Equation (4.3), it is possible to note that the following terms are constant during inference and therefore can be pre-computed offline by the compiler:

- $z_Y + \frac{s_b}{s_Y}(b_{q,j} - z_b)$

- $\frac{s_X s_W}{s_Y}$
- $z_X \sum_{k=1}^{n} W_{q,k,j}$
- $n z_X z_W$

In conclusion, Equation (4.3) will be implemented in the operator's kernel, while the constant terms will be computed in the operator parsing phase. By doing so, it is possible to achieve better optimizations and less overhead at runtime.

Implementation

The FullyConnected operator has been implemented as described in the previous section. The parser of the operator is in charge of pre-computing the constant terms listed above, while the kernel receives the variable tensors along with the constants and propagates them to the output tensor.

4.4.2 Conv2D

The Conv2D operator, short for *Convolutional 2D* operator, is a fundamental building block in CNNs used for image and signal processing tasks. It performs a convolution operation on an input tensor using a set of learnable filters or kernels.

The Conv2D operator applies a sliding window or filter over the input tensor, performing element-wise multiplications between the filter and the corresponding input region. The results of these multiplications are summed to produce a single output value, which represents the activation of a specific feature at a certain spatial location. This process is repeated for all possible locations, resulting in an output tensor with spatial dimensions reduced based on the filter size and stride.

Conv2D operators are commonly used for tasks such as image recognition, object detection, and image segmentation. They capture local patterns and spatial relationships in the input data, allowing the NN to learn hierarchical representations and extract meaningful features.

One key aspect of the Conv2D operator is the padding. In this context, padding refers to the technique of adding extra elements or values to the input tensor before performing the convolutional operation. It is used to control the spatial dimensions of the output tensor and preserve important information at

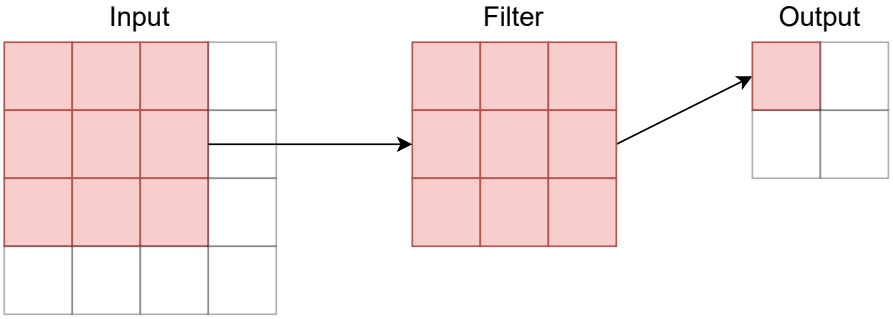


Figure 4.11: Example of Conv2D operator applied to an input tensor with valid padding. The resulting output will not preserve the spatial information of the input.

the borders of the input. There are two commonly used types of padding in Conv2D:

- Valid Padding: In valid padding, no padding is added to the input tensor. The convolution operation is applied only to the valid positions where the filter completely overlaps with the input.
- Same Padding: Same padding is the technique of adding padding to the input tensor so that the output tensor has the same spatial dimensions as the input. This is achieved by adding an equal number of padding elements or values on all sides of the input. Same padding ensures that the output size matches the input size, which can be beneficial for preserving the spatial information and allowing better alignment between the input and output.

An example of Conv2D operations done on an input tensor with valid padding can be seen in Figure 4.11. In the example, the input consists of a 3×3 matrix that gets convoluted with a 3×3 filter. By using the valid padding and assuming the strides to be equal to one, the filter will slide only one position for each dimension, otherwise the view would fall outside of the input matrix, which is not allowed by the valid padding. As a result, the output of the convolution does not preserve the input dimensions. Instead, the resulting dimensions are 2×2. On the other hand, if the same padding had been used for the example, the view would have been allowed to include values outside of the input matrix (i.e. zero values). Therefore, the resulting output of the convolutional operation would have retained the same input dimensions.

It is important to note that, in practical terms, the Conv2D operator uses 4-dimensional tensors. These tensors are composed by a set of matrices (called *batches*), containing multiple values (called *channels*) for each position. The

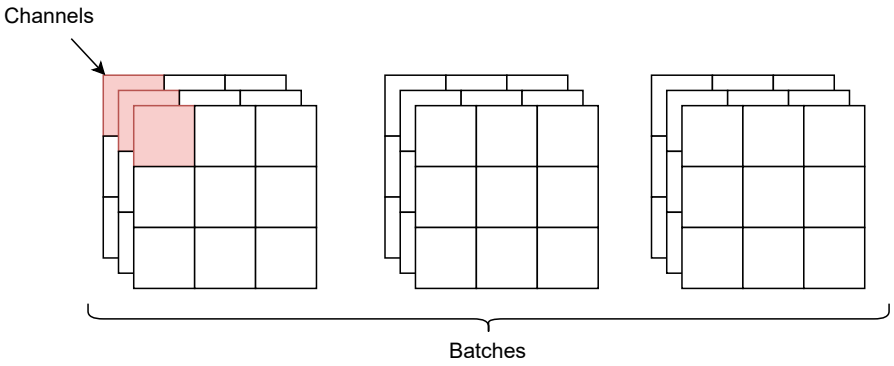


Figure 4.12: Visualization of a 4D tensor. The red cells represent the channels, while the matrix groups represent the batches.

convolutional filters are represented by the batches, while the channels are merged together with a dot product during convolution. The output tensor contains only one batch containing a matrix with, for each channel, the result of the convolution applied at that position in the input matrix. A visualization example of a 4D tensor composed by three batches of 3×3 matrices with 3 channels each can be observed in Figure 4.12.

Quantization

The quantization process of the Conv2D operator focuses on a single convolutional operation rather than the entire operator's execution. In this way, the quantization of the entire operator is obtained by applying the single quantized convolutional operation to all the input regions and kernels. The mathematical properties of each execution of the convolutional operation remain the same. The quantization process is carried out as follows.

Given $X \in \mathbb{R}^{m \times n \times c}$, $F \in \mathbb{R}^{m \times n \times c}$, and $b \in \mathbb{R}$, representing respectively an input region, a filter, and the bias, the output value $y \in \mathbb{R}$ for a given channel is obtained as:

$$y = b + \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{c} X_{i,j,k} F_{i,j,k} \tag{4.4}$$

By applying the dequantization formula, the following result is obtained:

$$
\begin{aligned}
& y \\
& = s_b(b_q - z_b) + \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} s_X(X_{q,i,j,k} - z_X)s_F(F_{q,i,j,k} - z_F) \\
& = s_b(b_q - z_b) + s_X s_F \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c}(X_{q,i,j,k} - z_X)(F_{q,i,j,k} - z_F) \\
& = s_b(b_q - z_b) + s_X s_F \left[\left(\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} X_{q,i,j,k}F_{q,i,j,k} \right) - \left(z_F \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} X_{q,i,j,k} \right) \right. \\
& \qquad \left. - \left(z_X \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} F_{q,i,j,k} \right) + mncz_X z_F \right] \\
& = s_y(y_q - z_y)
\end{aligned}
\tag{4.5}
$$

Where X_q, F_q, b_q, and y_q are the quantized versions of X, F, b, and y, respectively, s_X, s_F, s_b, and s_y are the scales for X, F, b, and y, respectively, and z_X, z_F, z_b, and z_y are the zero points for X, F, b, and y, respectively.

Therefore:

$$
\begin{aligned}
y_q = z_y + \frac{s_b}{s_Y}(b_q - z_b) + \frac{s_X s_F}{s_y} & \left[\left(\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} X_{q,i,j,k}F_{q,i,j,k} \right) \right. \\
& \left. - \left(z_F \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} X_{q,i,j,k} \right) - \left(z_X \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} F_{q,i,j,k} \right) + mncz_X z_F \right]
\end{aligned}
\tag{4.6}
$$

In conclusion, the result in Equation (4.6) is the quantized version of a single convolutional operation for a given filter and input region.

Pre-processing

As for the FullyConnected operator, it is possible to identify the constant terms for this operator too. In fact, by observing Equation (4.6), the following terms can be pre-processed by the compiler:

- $z_y + \frac{s_b}{s_y}(b_q - z_b)$

- $\frac{s_X s_F}{s_y}$
- $z_X \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{c} F_{q,i,j,k}$
- $mncz_X z_F$

Implementation

The development of the Conv2D operator required the implementation of an algorithm within the kernel to extract the input region. This algorithm is responsible for selecting the appropriate input elements to be used in each convolutional operation. The algorithm has to take into account the padding and strides of the convolution. The view extractor algorithm has been implemented as shown in Algorithm 1.

The algorithm takes as arguments the input tensor, the view dimensions, the padding, and the strides. For each position in the input, the algorithm calculates the neighboring components to include in the view. If the components fall out of the input matrix and the padding is same, the algorithm returns 0. The full implementation of the algorithm is reported in Appendix B.

4.4.3 DepthwiseConv2D

The DepthwiseConv2D operator is a specific type of Conv2D operator. Unlike the standard Conv2D operator, which applies a set of filters to the entire input tensor, the DepthwiseConv2D operator applies a separate filter to each input channel. This means that the operator performs depthwise convolutions, where each channel is convolved independently. The DepthwiseConv2D operator shares a lot of properties with the conventional Conv2D operator, including strides, paddings, and data structures.

However, the quantization process differs between the two since the convolutional operation is applied differently. In particular, there is a key distinction: instead of merging the channels together with a dot product, the channels are kept separate and convolved individually with the corresponding channels of the filter. This ensures that channel-wise interactions are maintained, capturing the spatial features within each channel independently. The result is a set of output channels that retain the channel-wise information from the input.

Algorithm 1 View extraction algorithm for the Conv2D operator.

Require: $X \in \mathbb{R}^{m \times n}$
Require: $V \in \mathbb{R}^{p \times q}$
Require: padding $\in \{\text{Same}, \text{Valid}\}$
Require: $\text{stride}_h, \text{stride}_w \in \mathbb{N}$
  $\text{shift}_h \leftarrow \lfloor \frac{p-1}{2} \rfloor$
  $\text{shift}_w \leftarrow \lfloor \frac{q-1}{2} \rfloor$
  for $i \in [0, m)$, $j \in [0, n)$, $k \in [0, p)$, $l \in [0, q)$ **do**
    $\text{index}_h \leftarrow \text{stride}_h * i + k$
    $\text{index}_w \leftarrow \text{stride}_w * j + l$
    if padding = Same **then**
      $\text{index}_h \leftarrow \text{index}_h - \text{shift}_h$
      $\text{index}_w \leftarrow \text{index}_w - \text{shift}_w$
      if $\text{index}_h \in [0, m)$ **and** $\text{index}_w \in [0, n)$ **then**
        $V_{k,l} \leftarrow X_{\text{index}_h, \text{index}_w}$
      else
        $V_{k,l} \leftarrow 0$
      end if
    else if padding = Valid **then**
      $V_{k,l} \leftarrow X_{\text{index}_h, \text{index}_w}$
    end if
  end for
  return V

Quantization

The quantization process for the DepthwiseConv2D operator closely resembles that of the Conv2D operator. However, unlike having batches of 2D filters, the DepthwiseConv2D operator operates on a single batch consisting of a 3D weight matrix. In this 3D weight matrix, the third dimension represents the weights associated with each channel. Therefore, the process of quantizing the DepthwiseConv2D operator can be approached in the following manner.

Given $X \in \mathbb{R}^{m \times n}$, $W \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}$, representing respectively an input region and a weights matrix for a given channel, and the bias, the output value $y \in \mathbb{R}$ is obtained as:

$$y = b + \sum_{i=1}^{m} \sum_{j=1}^{n} X_{i,j} W_{i,j} \tag{4.7}$$

By applying the dequantization formula:

$$\begin{aligned}
& y \\
& = s_b(b_q - z_b) + \sum_{i=1}^{m} \sum_{j=1}^{n} s_X(X_{q,i,j} - z_X) s_W(W_{q,i,j} - z_W) \\
& = s_b(b_q - z_b) + s_X s_W \sum_{i=1}^{m} \sum_{j=1}^{n} (X_{q,i,j} - z_X)(W_{q,i,j} - z_W) \\
& = s_b(b_q - z_b) + s_X s_W \left[\left(\sum_{i=1}^{m} \sum_{j=1}^{n} X_{q,i,j} W_{q,i,j} \right) - \left(z_W \sum_{i=1}^{m} \sum_{j=1}^{n} X_{q,i,j} \right) \right. \\
& \qquad \left. - \left(z_X \sum_{i=1}^{m} \sum_{j=1}^{n} W_{q,i,j} \right) + mn z_X z_W \right] \\
& = s_y(y_q - z_y)
\end{aligned} \tag{4.8}$$

Where X_q, W_q, b_q, and y_q are the quantized versions of X, W, b, and y, respectively, s_X, s_W, s_b, and s_y are the scales for X, W, b, and y, respectively, and z_X, z_W, z_b, and z_y are the zero points for X, W, b, and y, respectively.

Therefore:

$$y_q = z_y + \frac{s_b}{s_y}(b_q - z_b) + \frac{s_X s_W}{s_y}\left[\left(\sum_{i=1}^{m}\sum_{j=1}^{n} X_{q,i,j}W_{q,i,j}\right) - \left(z_W \sum_{i=1}^{m}\sum_{j=1}^{n} X_{q,i,j}\right) - \left(z_X \sum_{i=1}^{m}\sum_{j=1}^{n} W_{q,i,j}\right) + mnz_X z_W\right] \tag{4.9}$$

Equation (4.9) shows the final result for the quantization process of the DepthwiseConv2D operator.

Pre-processing

Similarly to Conv2D, the following terms of Equation (4.9) are constant during inference:

- $z_y + \frac{s_b}{s_y}(b_q - z_b)$
- $\frac{s_X s_W}{s_y}$
- $z_X \sum_{i=1}^{m}\sum_{j=1}^{n} W_{q,i,j}$
- $mnz_X z_W$

Implementation

The view extraction routine, as described in Algorithm 1, is utilized for both the Conv2D and DepthwiseConv2D operators. By sharing the same view extraction task, the implementation remains consistent between the two operators, streamlining the development process.

4.4.4 AveragePool2D

The AveragePool2D operator is a fundamental component in NN architectures that involves spatial pooling. It is typically used to downsample the input data by partitioning it into non-overlapping regions and computing the average value within each region. This operator helps in reducing the spatial dimensions of the input while preserving important features.

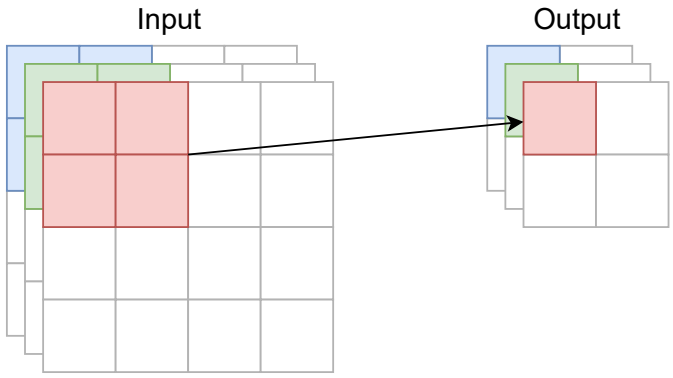


Figure 4.13: Example of the AveragePool2D applied to an input tensor. The operator is executed on a per-channel basis, preserving the input channel dimensions.

During the operation of the AveragePool2D operator, a window or kernel slides over the input data, and at each position, it calculates the average of the values within the window. The size of the window and the stride determine the extent of pooling and the resulting output size. By specifying different window sizes and strides, one can control the amount of downsampling applied to the input.

As for other operators, AveragePool2D works on 4D tensors with a single batch of matrices. The AveragePool2D operator performs average pooling on a per-channel basis, which means that the input channels are preserved throughout the pooling operation until the output is generated. An example of average pooling can be observed in Figure 4.13.

Quantization

The operator quantization has been carried out as follows.

Given $X \in \mathbb{R}^{m \times n}$, representing an input region for a given channel, the output value $y \in \mathbb{R}$ for a given channel is computed as:

$$y = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} X_{i,j} \tag{4.10}$$

By applying the dequantization formula in Equation (2.1):

$$
\begin{aligned}
& y \\
&= \frac{s_X}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (X_{q,i,j} - z_X) \\
&= \frac{s_X}{mn} \left[\left(\sum_{i=1}^{m} \sum_{j=1}^{n} X_{q,i,j} \right) - mnz_X \right] \\
&= s_X \left[\left(\frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} X_{q,i,j} \right) - z_X \right] \\
&= s_y (y_q - z_y)
\end{aligned}
\tag{4.11}
$$

Where X_q and y_q are the quantized versions of X and y, respectively, s_X and s_y are the scales for X and y, respectively, and z_X and z_y are the zero points for X and y, respectively.

Therefore:

$$
y_q = z_y + \frac{s_X}{s_y} \left[\left(\frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} X_{q,i,j} \right) - z_X \right] \tag{4.12}
$$

In summary, Equation (4.12) shows the quantized version of the AveragePool2D operator.

Pre-processing

The constant terms of Equation (4.12) that can be pre-computed by the compiler are:

- $\frac{s_X}{s_y}$
- $\frac{1}{mn}$

Implementation

The implementation of the AveragePool2D operator also utilizes the view extraction algorithm described in Algorithm 1, as the pooling operation is performed on an input region. The main distinction is that, similar to the Depth-

wiseConv2D operator, the channels of the input region in AveragePool2D are not merged together. Instead, the channel dimension is preserved.

4.4.5 Activation Functions

Activation functions are the last component of the processing flow. They are applied to the outputs of individual neurons, transforming them to enable complex and expressive mappings between inputs and outputs. Activation functions can be either applied as a separate operation after a specific layer or combined with an operator, taking the name of *fused* activation functions. All the operators described so far support the addition of a fused activation function at the end of each iteration. Alternatively, MicroFlow supports operators that only perform the activation function, such as the *Softmax* operator.

In terms of implementation, the kernel of an activation function remains the same regardless of its application. What can differ is how the activation function is applied in the context of the NN. However, as for operators, activation functions also have to be quantized, translating the functions from floating-point operations to integer-only operations. Moreover, the activation functions presented in this section are not subject to pre-processing as they contain little to no constant terms.

This section will go through the supported activation functions, describing how the formula has been modified to support quantization. The quantized activation functions currently supported by MicroFlow are:

- ReLU
- ReLU6
- Softmax

ReLU

The ReLU function is a very commonly used activation function. Given an input $x \in \mathbb{R}$ and an output $y \in \mathbb{R}$ its function is defined as follows:

$$y = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{4.13}$$

In other words, the ReLU function returns the input value if it is positive or zero, and it returns zero for any negative input. The key characteristic of the ReLU function is its ability to introduce sparsity and promote sparse activations in a NN.

By applying the dequantization formula, the following is obtained:

$$\begin{aligned} & y \\ & = \begin{cases} 0 & \text{if } s_x(x_q - z_x) < 0 \\ s_x(x_q - z_x) & \text{if } s_x(x_q - z_x) \geq 0 \end{cases} \\ & = \begin{cases} 0 & \text{if } x_q < z_x \\ s_x(x_q - z_x) & \text{if } x_q \geq z_x \end{cases} \\ & = s_y(y_q - z_y) \end{aligned} \tag{4.14}$$

Where x_q and y_q are the quantized versions of x and y, respectively, s_x and s_y are the scales for x and y, respectively, and z_x and z_y are the zero points for x and y, respectively.

Therefore:

$$y_q = \begin{cases} z_y & \text{if } x_q < z_x \\ z_y + \frac{s_x}{s_y}(x_q - z_x) & \text{if } x_q \geq z_x \end{cases} \tag{4.15}$$

It is important to note that, if ReLU is used as a fused activation function, and therefore $s_x = s_y = s$ and $z_x = z_y = z$, the formula in Equation (4.15) simply results in:

$$y_q = \max(x_q, z) \tag{4.16}$$

Therefore, with this procedure, it has been possible to derive the quantized ReLU.

ReLU6

The ReLU6 activation function is a variant of the standard ReLU function that adds an upper bound constraint. It is commonly used in applications where output values need to be limited to a specific range.

Due to its similarity to the ReLU function, the quantized version of the ReLU6 activation function can be immediately derived as follows:

$$y_q = \begin{cases} \text{ReLU}(x_q, s_x, s_y, z_x, z_y) & \text{if } x_q < z_x + \frac{6}{s_x} \\ z_y + \frac{6}{s_y} & \text{if } x_q \geq z_x + \frac{6}{s_x} \end{cases} \tag{4.17}$$

Where x_q and y_q are the quantized versions of the input $x \in \mathbb{R}$ and the output $y \in \mathbb{R}$, respectively, s_x and s_y are the scales for x and y, respectively, and z_x and z_y are the zero points for x and y, respectively.

Similarly to the ReLU function, if ReLU6 is used as a fused activation function, and therefore $s_x = s_y = s$ and $z_x = z_y = z$, the Equation (4.17) results in:

$$y_q = \min(\max(x_q, z), z + \frac{6}{s}) \tag{4.18}$$

Softmax

The last activation function is Softmax. Softmax is commonly used in deep learning for multi-class classification problems. It takes a vector of real-valued scores as input and transforms them into a probability distribution over multiple classes. The resulting probabilities represent the model's confidence or likelihood for each class.

Mathematically, the Softmax function takes an input vector $x \in \mathbb{R}^n$ and produces an output vector $y \in \mathbb{R}^n$ as follows:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{4.19}$$

By combining it with the dequantization formula, it is possible to obtain the quantized Softmax as follows:

$$\begin{aligned} & y \\ &= \frac{e^{s_x(x_{q,i} - z_x)}}{\sum_{j=1}^{n} e^{s_x(x_{q,j} - z_x)}} \\ &= \frac{e^{s_x x_{q,i}}}{\sum_{j=1}^{n} e^{s_x x_{q,j}}} \\ &= s_y(y_{q,i} - z_y) \end{aligned} \tag{4.20}$$

Where x_q and y_q are the quantized versions of x and y, respectively, s_x and s_y are the scales for x and y, respectively, and z_x and z_y are the zero points for x and y, respectively.

Therefore:

$$y_{q,i} = z_y + \frac{e^{s_x x_{q,i}}}{s_y \sum_{j=1}^{n} e^{s_x x_{q,j}}} \tag{4.21}$$

In conclusion, Equation (4.21) shows the quantized version of the Softmax activation function.

5

Evaluation

This chapter provides an in-depth analysis and evaluation of the implemented system in terms of its performance, efficiency, and overall effectiveness. This chapter aims to assess the system's capabilities and measure its performance against specific criteria and benchmarks. By conducting rigorous evaluations and tests, this chapter offers insights into the system's strengths, weaknesses, and potential areas for improvement. The findings presented in this chapter serve as valuable insights for assessing the system's overall success and identifying potential future enhancements.

This evaluation phase aims to demonstrate the project's applicability and effectiveness in such resource-constrained environments. By conducting targeted evaluations and tests, this chapter aims to highlight the project's ability to operate efficiently and deliver accurate results under severe resource limitations.

The chapter is structured as follows: it begins by discussing the experimental setup, providing an in-depth explanation of the tools, methodologies, and resources employed during the evaluation process. This section outlines the specific configurations, hardware platforms, and models utilized to conduct the experiments effectively.

Subsequently, the chapter presents and analyzes the results obtained from the evaluation phase. The findings are presented in a clear and concise manner, accompanied by relevant metrics, graphs, and statistical analyses. Each result is carefully examined and discussed, highlighting the project's performance, efficiency, and effectiveness in resource-constrained embedded systems.

5.1 Experimental Setup

In this section, the specific hardware platforms, NN models, and configurations employed in the experiments will be presented. Furthermore, this section outlines the experimental procedures followed to ensure accurate and consistent results. It covers aspects such as data collection, parameter settings, benchmarking, and any other relevant considerations.

First, this section focuses on the models selected for the evaluation, providing an exploration of their structures and sizes. Subsequently, this section presents the hardware platforms used for conducting the tests, providing detailed information about their technical specifications and notable features.

5.1.1 Models

The chosen NN models play a crucial role in assessing the performance and capabilities of the project. Ideally, the selected models should encompass various aspects that are important for evaluation, allowing for a comprehensive analysis of different performance and effectiveness metrics.

For these reasons, it has been decided to use three distinct models, each varying in size and complexity. All the models have been quantized to 8-bit signed integers. The chosen models for this experimentation are as follows:

- A sine predictor
- A speech command recognizer
- A person detector

The subsequent sections will provide a detailed description of each model, accompanied by their respective specifications. Finally, an overview of the models is presented in Table 5.1.

Sine Predictor

The sine predictor represents the simplest and smallest model in the evaluation. Its simplicity allows for a focused analysis of fundamental aspects and provides insights into the performance of the systems under minimal computational and memory requirements.

As its name suggests, the sine predictor model is designed to predict the values of the sine function. In other words, given an input $x \in \mathbb{R}$, the sine

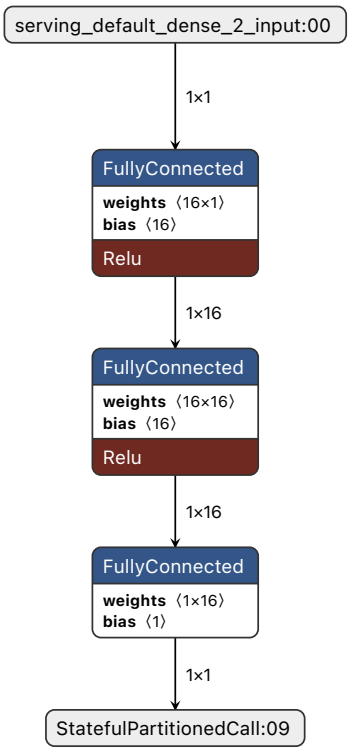


Figure 5.1: Visualization of the sine predictor model.

model returns an output $y = \sin(x)$. The model has been trained on a dataset generated using the sin() function combined with random noise. The model is composed of three FullyConnected layers of 16 neurons each, with the first two having ReLU as the fused activation function. The entire size of the model is approximately 3 kB. A visualization of the model can be seen in Figure 5.1.

Training the sine predictor model on a deterministic function, such as the sin() function, provides a valuable advantage for evaluating the precision of the inference engine. By having access to ground-truth values, which are known in this case due to the deterministic nature of the sine function, it becomes possible to compare the predicted values with the expected values. In summary, this model is mainly used as a benchmark for evaluating accuracy and deployment on resource-constrained MCUs.

Speech Command Recognizer

The speech command recognizer is the intermediate-size model used for evaluation. The goal of the model is to recognize two spoken words: *yes* and *no*.

To do so, the model introduces the concept of convolutional operations ap-

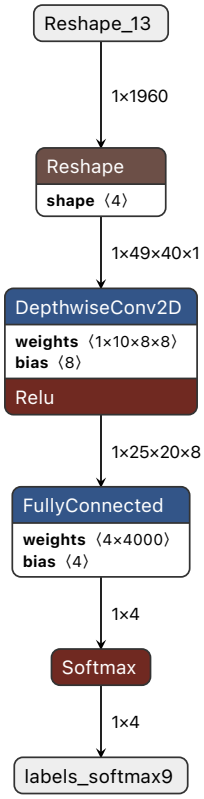


Figure 5.2: Visualization of the speech command recognizer model.

plied to an input signal, specifically the Fast Fourier Transform (FFT) representation of an input audio sample. As output, the model returns the likelihood for the input sample to be in one of these categories: the *yes* word, the *no* word, silence, or unknown speech.

The model follows the *TinyConv* architecture, employing one DepthwiseConv2D layer followed by a FullyConnected layer. The output scores are then converted to probabilities by a Softmax activation layer. Since the speech command recognizer works with convolutional layers, the tensors in this model are 4-dimensional. Therefore, the overall size of the model increases significantly to approximately 19 kB. The structure of the model can be observed in Figure 5.2. The training of the model has been performed on the *Speech Commands Dataset* version 2 [23].

In summary, this model is a trade-off between complexity, size, and real-world applications. In fact, while the sine predictor model does not make sense to be used in a real-world scenario due to its deterministic nature, the speech command recognizer model resembles the wake-word detection task, which is

a significant use case in many applications. Moreover, this model incorporates both convolutional operations and dense layers, providing a good balance for evaluation purposes.

Person Detector

The person detector is the most complex and largest model of the evaluation. The goal of the model is to detect the presence of a person, given an input frame. In particular, the model takes as input a grayscale image of dimensions 96 px × 96 px and provides as output the probabilities of the two classes *person* and *not-person*.

The model follows the *MobileNet* architecture version 1 [24], employing a series of DepthwiseConv2D operators followed by Conv2D operators. The end of the chain consists of an AveragePool2D layer, followed by a Conv2D operator, and a final Softmax activation function layer. A simplified visualization of the model is represented in Figure 5.3. The model has been trained on the *Visual Wake Words Dataset* [25].

Due to the type and quantity of the operators employed, which require the usage of 4D tensors, the size of the model reaches 301 kB. For this reason, the person detector model has been specifically deployed on MCUs that have sufficient resources, including an adequate flash size, to accommodate its requirements.

In summary, the evaluation of the person detector model played an important role in assessing the real-world applicability of the inference engine, as it represents one of the most relevant and useful applications in the field of TinyML.

5.1.2 Hardware

Although MicroFlow can run on a variety of different systems, the analysis has been focused on bare-metal embedded devices. In this way, it is possible to best outline the performance of the inference engine in real-world scenarios. The evaluation has been done using the following MCUs and test boards, arranged in descending order based on their performance and resource constraints:

- ESP32 (Adafruit HUZZAH32)
- ATSAMV71 (SAM V71 Xplained Ultra)
- nRF52840 (Arduino Nano 33 BLE Sense

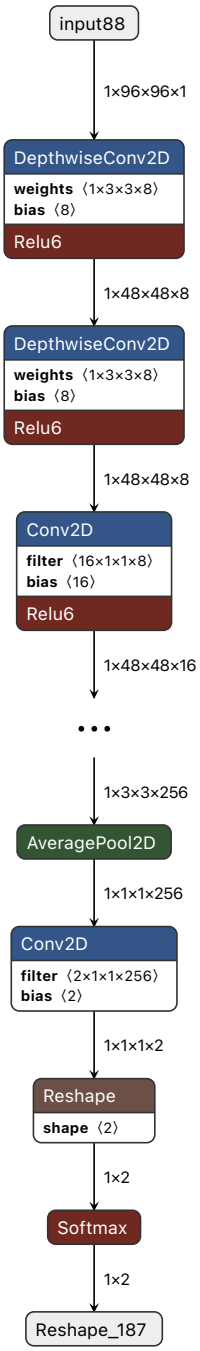


Figure 5.3: Visualization of the person detector model. The central repeated pattern of layers has been hidden due to its size.

Model	Architecture	Operators	Layers	Size
Sine predictor	Custom	FullyConnected ReLU	3	3 kB
Speech command recognizer	TinyConv	DepthwiseConv2D FullyConnected Softmax	4	19 kB
Person detector	MobileNet	DepthwiseConv2D Conv2d AveragePool2D Softmax	30	301 kB

Table 5.1: Summary of the models employed for the system evaluation.

MCU	CPU Architecture	Flash Size	RAM Size	CPU Clock
ESP32	32-bit Xtensa	4 MB	328 kB	240 MHz
ATSAMV71	32-bit Cortex-M7F	2 MB	384 kB	300 MHz
nRF52840	32-bit Cortex-M4F	1 MB	256 kB	64 MHz
LM3S6965	32-bit Cortex-M3	256 kB	64 kB	50 MHz
ATmega328	8-bit AVR	32 kB	2 kB	20 MHz

Table 5.2: Summary of the MCUs used for the experiments.

- LM3S6965 (emulated using QEMU)
- ATmega328 (Arduino Uno)

The chosen MCUs cover a wide range of possible memory sizes, architectures, and peripherals. From the high-performance 32-bit ESP32 with 4 MB of Flash and 328 kB of RAM, to the 8-bit ATmega328, with only 32 kB of Flash and 2 kB of RAM. The details and settings of each MCU are reported in Table 5.2.

5.1.3 Baseline

The MicroFlow project shares some similarities with the TFLM framework, as it originated as an enhanced version of it. Therefore, to evaluate the effectiveness of MicroFlow, TFLM has been selected as the baseline for comparison. As concluded after the survey described in Chapter 2, TFLM can be considered

the current state-of-the-art work, being also the most used inference engine for TinyML applications.

All experiments have been conducted on both platforms, allowing for a comprehensive analysis and comparison of the results. However, to ensure fair and consistent experiments, both the MicroFlow and TFLM engines have been configured under the same test conditions. This approach aimed to eliminate any potential bias coming from the different underlying structures and implementation details of the projects.

5.1.4 Experiments

The experiments conducted for the system evaluation have been carefully selected to cover the most significant metrics relevant to TinyML applications. The objective was to obtain a comprehensive understanding of the following metrics under various test scenarios:

- Accuracy
- Memory usage
- Runtime Performance
- Energy Consumption

The accuracy test serves as a critical evaluation to determine whether the inference engine correctly performs inference on the model. The experiment has been carried out using test sets to calculate the relevant metrics for the compared inference engines. These metrics include the precision, recall, and F_1 score for the classifier models and the MSE for the sine model. For the speech command recognizer model with four output classes, the metrics have been averaged to provide an overall measure of accuracy across all classes. The resulting metrics are then compared between inference engines to assess their performance.

The second experiment focuses on the usage of the MCU memory, specifically the Flash and RAM. This test is crucial to assessing the portability of the inference engine across different MCUs and architectures. The experiment has been carried out by loading a minimal firmware that executes inference on a given model and analyzing the compiled binary. The use of a minimal firmware was necessary to remove other platform-dependent factors that might compromise the test results. For example, the test firmware does not include

any printing statements, because the implementation of formatting and printing varies between platforms and architectures and therefore might bias the results.

Subsequently, the runtime performance experiment has been employed to compare TFLM and MicroFlow in terms of execution times of the experimental models. The experiment consists of performing inference of the test models on different MCUs using both inference engines. The execution is repeated a statistically significant number of times, and the execution times are measured using the MCU timers. Finally, the median point is calculated to have a reliable measurement of the execution time for an inference operation.

The last test aims to analyze an important metric for embedded systems, which is energy consumption. By having accurate execution times and measuring the average power usage of the MCU it has been possible to obtain the average energy consumption for the inference operation of the engine. In the following sections, the analysis will go through the results of these experiments.

5.2 Results

This section presents the outcomes and findings obtained from the experimental evaluation of the MicroFlow and TFLM inference engines. In this section, the focus is on analyzing and interpreting the experiments and their results. The following sections contain the results of the experiments previously described.

5.2.1 Accuracy

The accuracy test has been conducted as described in the previous sections and produced the results shown in Table 5.3. For the sine model, the test set employed for the experiment has been randomly generated, and the MSE has been calculated against the actual values of the sine function. In Figure 5.4 is possible to see a comparison between the TFLM and MicroFlow predictions on the test set. For the other models, the test sets used were the ones provided by the training datasets, namely the Speech Commands and Visual Wake Words datasets.

The experiment shows that the two inference engines resulted in very similar characteristics, with all the metrics almost equal between the two. This means that MicroFlow can accurately predict as well as TFLM, meaning that the operators have been correctly implemented and the mathematical theory behind

Sine Predictor

	TFLM	**MicroFlow**
MSE	0.0157	0.0154
Test samples	1000	1000

Speech Command Recognizer

	TFLM	**MicroFlow**
Precision	91.737%	91.638%
Recall	88.611%	88.972%
F_1 Score	90.147%	90.285%
Test samples	1236	1236

Person Detector

	TFLM	**MicroFlow**
Precision	71.843%	72.003%
Recall	85.382%	85.401%
F_1 Score	78.030%	78.132%
Test samples	406	406

Table 5.3: Results of the accuracy experiment performed on the sample models.

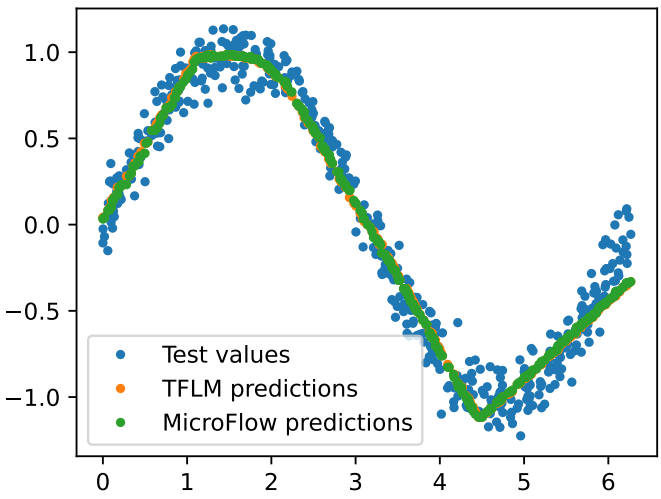


Figure 5.4: Comparison of TFLM and MicroFlow predictions on the test set.

them has been correctly applied.

However, although similar, the results are not identical, meaning that at some point in the computation, some operators gave different results. By analyzing the intermediate quantized outputs of the NNs, it has been possible to notice that in some cases the output of the operators was different. In particular, it has been noticed that some components of the output tensors were different by one integer unit. Sometimes TFLM was one unit above, sometimes one unit below. However, the difference in output has always been one unit.

Hence, it is highly probable that these variations in behavior can be attributed to internal differences in the implementation or representation of floating-point operations. As a consequence, these slight differences lead to rounding discrepancies, resulting in two distinct integer values that differ at maximum by only one unit. Additionally, TFLM and MicroFlow are based on two different programming languages that leverage two different compilers. Therefore, a discrepancy in the implementation or representation of the compiler's built-in operations is plausible.

5.2.2 Memory Usage

This second set of experiments plays a fundamental role in assessing the memory footprint of the code. One of the main goals of the project was to build a highly memory-efficient engine. Therefore, this test is crucial to understanding if this goal has been reached.

For the sine model, the results of the experiment can be observed in Figure 5.5. The chart shows the memory usage, both in terms of Flash and RAM of the compiled binary, for the test MCUs. The RAM usage values refer to the maximum amount of RAM used throughout the program's execution.

The results show a big difference in terms of memory usage between the inference engines. For example, on the ESP32, MicroFlow uses more than 60% less Flash memory than TFLM, and on the nRF52840, MicroFlow uses only 5.296 kB of RAM, against 45.728 kB required by TFLM.

The very low memory usage of MicroFlow made it possible to successfully build and run the sine model on all the test MCUs. In particular, it has been possible to successfully perform inference even on the 8-bit AVR ATmega328, with a Flash usage of 13.619 kB and a RAM usage of 1.706 kB. On the other hand, it has not been possible to build TFLM for the majority of the MCUs. In fact, it

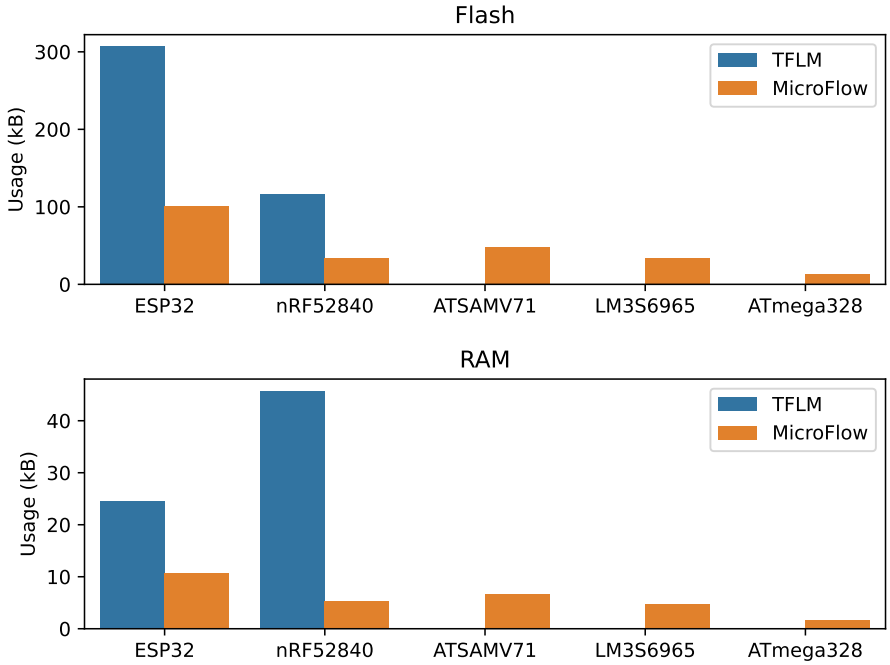


Figure 5.5: Results of the memory usage experiment for the sine predictor model.

has only been possible to run TFLM on the ESP32 and the nRF52840, requiring a significant amount of memory.

The reason for these results can be attributed to the fact that TFLM is an interpreter-based engine, and therefore the interpreter needs to be loaded on the MCU regardless of the size of the model since it is not known at compile time. Moreover, since the operators in the model are not known at compile time, the interpreter with all the operator kernels must be loaded, taking up memory space. These factors, combined with the utilization of a user-defined tensor arena, contribute to the suboptimal memory allocation of TFLM.

On the other hand, with MicroFlow, only the necessary weights and operator kernels are stored in memory, leaving out the parts of the model that are not needed at runtime (e.g. operator versions, tensor names, sizes, options, etc.). For example, instead of having to store all the kernel versions of an operator and decide at runtime which one to use based on the model, MicroFlow reads the operator version at compile time and stores in the target memory only the correct version of it.

Additionally, everything in MicroFlow is allocated on the stack. By doing so, due to the nature of the stack, the memory used by an operator will be freed automatically after the operator has been executed. Therefore, the peak

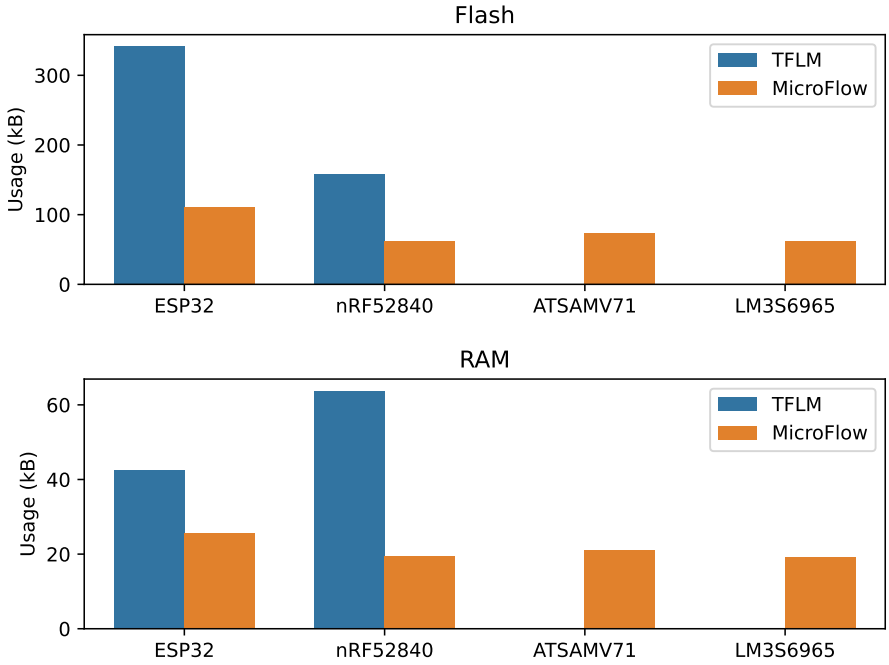


Figure 5.6: Results of the memory usage experiment for the speech command recognizer model.

of memory usage will only occur when the most memory-intensive operator is being executed. After that, its memory is freed, and the next operator allocates it based on its requirements. Consequently, once the entire inference operation has been concluded, the memory used by the last operator is freed, and the memory allocated by the engine is therefore null.

In contrast, TFLM allocates the tensor arena throughout the entire execution of inference. The memory arena must be sized to accommodate the memory requirements of the most memory-intensive operator. Therefore, the memory used is constant during execution, and it is not freed after each operator. Moreover, in TFLM, the user is responsible for manually allocating and deallocating memory, as well as determining the appropriate amount of memory to be used. This can result in suboptimal memory allocations and potential runtime errors if the user allocates either too little or too much memory.

The experiment proceeded to evaluate the other models, and the results obtained were consistent with the previous findings. In particular, the results for the speech command recognizer and person detector models can be seen in Figure 5.6 and Figure 5.7, respectively.

For these models, only the most performing MCUs have been used for evaluation. This is due to the fact that the smaller ones could not physically fit the

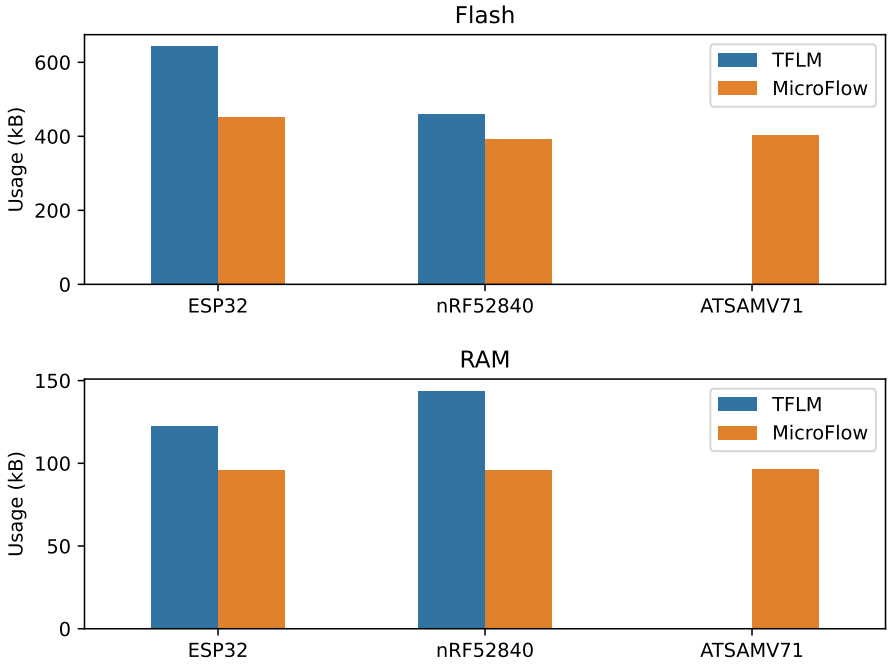


Figure 5.7: Results of the memory usage experiment for the person detector model.

model's weights in memory, making it impossible to perform inference.

Overall, the considerations made for the results of the sine model can be applied to these cases too. However, as the size of the model increases, the memory gap between MicroFlow and TFLM becomes smaller. This can be attributed to the fact that the model's weights, which cannot be optimized, occupy the majority of the allocated memory. As a result, the impact of the interpreter overhead becomes relatively less significant in comparison. However, MicroFlow still has 15% less memory usage with respect to TFLM on the nRF52840 with the person detector model. In conclusion, the experiment highlights the efficiency and resource optimization achieved by MicroFlow in memory-constrained environments. Therefore, the initial design goal regarding this aspect has been successfully reached.

5.2.3 Runtime Performance

This last series of experiments aims to assess the performance of the compared inference engines in terms of execution times. The experiment has been carried out as described in the previous sections. In particular, to provide a meaningful comparison, the test has been performed only on the MCUs sup-

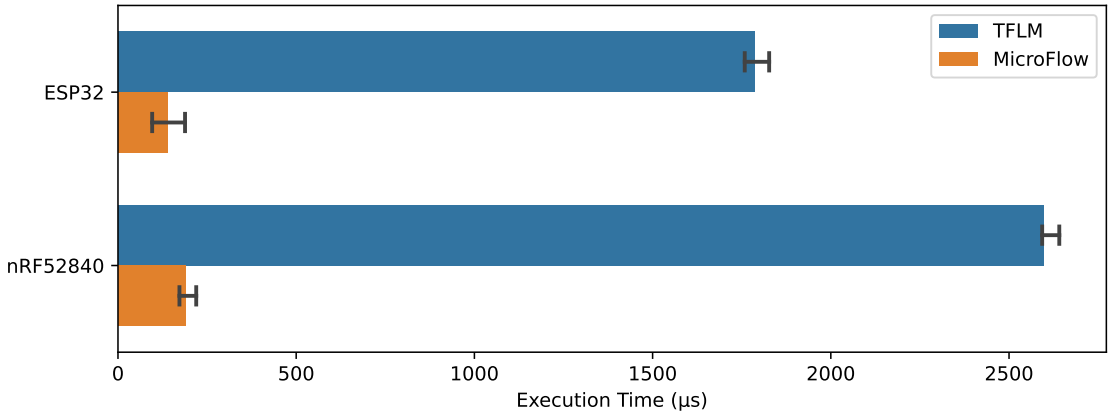


Figure 5.8: Comparison of the runtime performance test results for the sine predictor model.

ported by TFLM, namely the ESP32 and the nRF52840.

The firmware used for the experiments consists of a minimal program that cyclically executes inference on the given model for a statistically significant number of times, measuring the execution time for each cycle using the MCU timers. The measurements are then exported and used to plot a histogram, highlighting the median value of the measurements, to obtain an accurate estimate of the true execution time. The full set of resulting histograms for each experiment can be seen in Appendix C. The following sections will present a comprehensive overview of the findings. The measurement estimates are compared between inference engines by using bar plots. These plots include the distribution's 95% percentile interval, effectively visualizing the data point dispersion.

Sine Predictor

The results for the sine predictor model on the ESP32 and the nRF52840 can be observed in Figure 5.8. As it is possible to note from the plots, MicroFlow has a significant performance gain over TFLM. In the case of the sine model, the execution time of MicroFlow on both the ESP32 and nRF52840 is over 10 times lower compared to TFLM. This significant reduction in execution time highlights the efficiency and optimization achieved by the MicroFlow inference engine, especially for small models.

The observed outcome can be primarily attributed to two factors: the interpreter-based nature of TFLM and the optimizations implemented in MicroFlow using the Rust programming language. The interpreter-based approach of TFLM in-

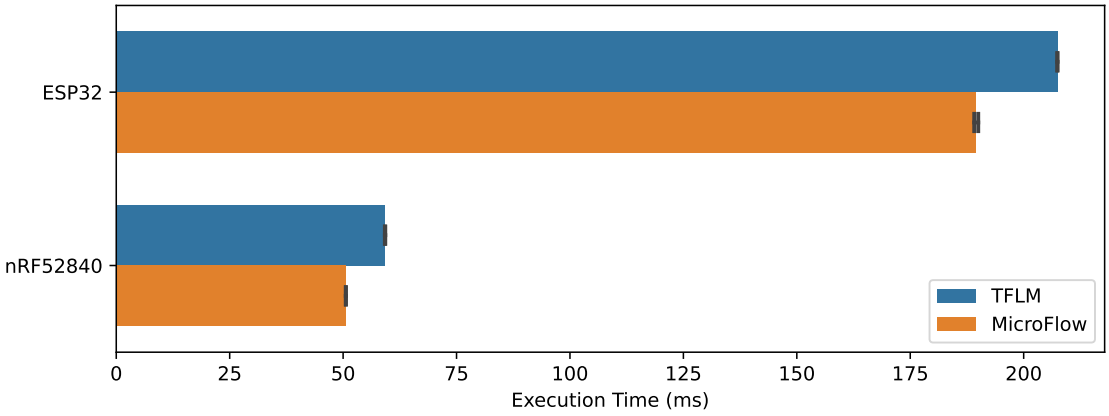


Figure 5.9: Comparison of the runtime performance test results for the speech command recognizer model.

troduces additional overhead during the execution of the inference process. This overhead includes interpretation of the model's operations, dynamic memory management, and other interpreter-related tasks. As a result, the total execution time increases, affecting the overall performance of TFLM. This is even more pronounced when the model is small, as the inference execution time is comparable to the interpreter overhead. On the other hand, MicroFlow leverages the Rust programming language and a heavy pre-processing phase. This leads to effective static analysis and optimizations from the compiler, which result in reduced CPU operations and consequently reduced execution times.

However, it is expected that the performance gap between MicroFlow and TFLM will narrow as the size and complexity of the models increase. This is because the actual inference operations become more dominant in determining the overall performance, while the relative impact of the interpreter overhead diminishes.

Speech Command Recognizer

The results of the experiment for the speech command recognizer model can be observed in Figure 5.9. As expected, the performance gap between the two inference engines narrows significantly. However, MicroFlow still records a performance gain over TFLM of approximately 9% on the ESP32 and 15% on the nRF52840. As introduced earlier, the gap in performance narrows due to the fact that the overhead introduced by the TFLM interpreter becomes less significant compared to the actual inference operation. However, MicroFlow optimizations

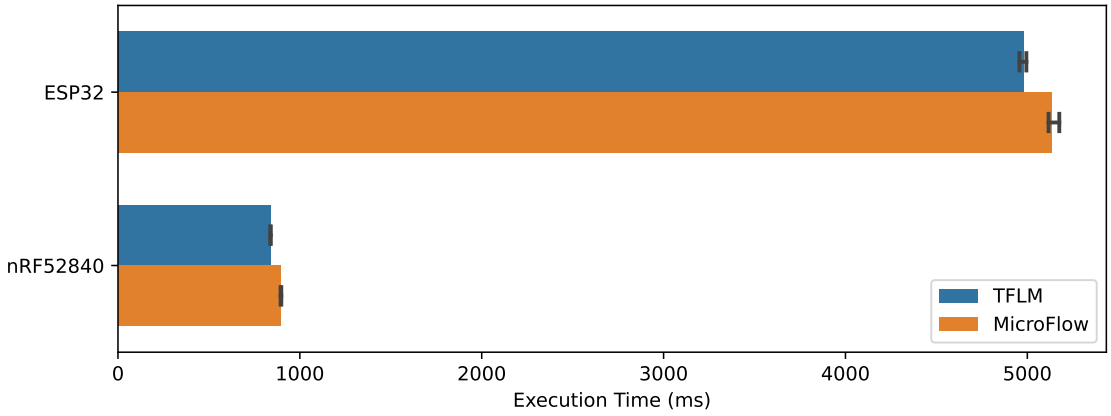


Figure 5.10: Comparison of the runtime performance test results for the person detector model.

and efficient operators result in faster inference.

One more thing to note is that the performances between the ESP32 and the nRF52840 are starting to diverge significantly, with the nRF52840 being more than three times faster than the ESP32. This outcome is counterintuitive since the ESP32 has a CPU clock significantly higher than the nRF52840, and therefore it should theoretically be faster. However, the ESP32 is known for its inefficient Floating-Point Unit (FPU), which can impact the performance of the inference engine since it heavily relies on it. Therefore, this inefficiency is most likely the cause of these poor performances, and they are expected to degrade even more for bigger and more complex models.

Person Detector

The results of the final performance experiment on the most complex model on the ESP32 and nRF52840 can be seen in Figure 5.10. The outcome for the person detector model is different from the results of the other models. In this configuration, TFLM performed slightly better than MicroFlow. However, the gap is minimal and approximately equal to 6.5%.

The observed result can be attributed to the nature of the MobileNet model, which primarily consists of convolutional operations. As the model size increases, the execution time becomes dominated by these computationally intensive operations. Consequently, the overall execution time reaches a saturation point that is primarily determined by the hardware constraints of the MCU, such as the CPU clock speed and the number of cores. Given these limitations, there

are limited software optimizations available to improve performance. Instead, enhancing the hardware capabilities of the MCU can lead to performance improvements in such scenarios. For example, these hardware enhancements can include increasing the clock speed, parallelizing computation by using multiple cores, or employing specialized hardware to speed up tensor operations, such as a GPUs or TPUs.

However, the slightly faster performance of TFLM can be attributed to the fact that TFLM uses specialized kernels for specific MCUs instead of the reference ones. This is the case for the nRF52840, where TFLM uses a specific set of kernels provided by the MCU manufacturer. These kernels are contained in the *CMSIS-NN* software library, by ARM. On the other hand, MicroFlow does not do any MCU-specific optimization for the sake of this research. However, this can be a future improvement for the project. Another thing to note is that, as in the previous experiment, the ESP32 performed significantly worse than the nRF52840. Once again, this can be attributed to the poor performance of the ESP32's FPU.

In conclusion, MicroFlow demonstrated that it performed better or as well as TFLM for the majority of the experiments. In particular, MicroFlow proved to be faster than TFLM for small models. However, as the size and complexity increase, both MicroFlow and TFLM approach the limitations imposed by the underlying hardware.

5.2.4 Energy Consumption

The energy consumption experiment has been conducted as the final step, utilizing the results from the performance experiments to provide precise measurements. The results of the experiment are summarized in Table 5.4. As can be seen, the consumption values are proportional to the execution times. In fact, from the experiment, it has been determined that the average power usage of TFLM and MicroFlow was substantially equal. Therefore, the overall energy consumption depended mainly on the execution time.

This behavior can be attributed to the fact that the types of operations that the MCU has to perform are essentially equal between the two compared inference engines. Therefore, the power usage is similar. Moreover, the peripherals of the MCU used by the inference engines are identical, resulting in limited possibilities for power usage optimizations. In conclusion, the energy consumption of the

Sine Predictor

	TFLM	**MicroFlow**
ESP32	149 nW h	11 nW h
nRF52840	216 nW h	16 nW h

Speech Command Recognizer

	TFLM	**MicroFlow**
ESP32	23.05 mW h	21.04 mW h
nRF52840	6.58 mW h	5.62 mW h

Person Detector

	TFLM	**MicroFlow**
ESP32	691.11 mW h	694.44 mW h
nRF52840	116.58 mW h	124.44 mW h

Table 5.4: Results of the energy consumption experiment conducted on the sample models.

inference engines is directly proportional to their execution times.

6

Conclusion

In this thesis, MicroFlow, a lightweight ML inference engine designed specifically for resource-constrained MCUs, has been presented. Throughout the research and evaluation process, MicroFlow has demonstrated promising results and showcased its effectiveness in terms of performance, memory usage, and energy consumption compared to the TFLM framework.

Overall, the goals set at the beginning of the project have been successfully achieved. The project has demonstrated efficiency in terms of both performance and memory usage. The decision to utilize the Rust programming language has proven to be beneficial, providing the project with memory guarantees and powerful tools for developing a robust inference engine. Additionally, the decision to follow a compiler-based approach turned out to be correct. The evaluation and experimentation conducted throughout the project have validated the effectiveness of this strategy, especially for small models.

However, it is important to note that the performance of MicroFlow degrades when handling large models, eventually reaching a saturation point. This unexpected result highlights an area for further research and potential improvements in future iterations of the project. In summary, the project has reached a stage where it can be tested in real-world applications for further evaluation and provides a solid foundation for future enhancements and feature additions.

6.1 Future Work

The modular nature of the project makes it highly adaptable and open to future additions and optimizations. One of the key areas for potential improvements is the addition of new operators and the optimization of existing ones for specific MCU families. By expanding the library of operators, MicroFlow can support a wider range of operations and NN architectures.

Furthermore, MicroFlow can be used for developing applications in collaboration with the company that has been involved in this project. These applications include the optimization of the charging process for electric vehicles. In fact, by leveraging MicroFlow, it is possible to safely and efficiently include ML models and techniques in the existing embedded solutions to provide a more efficient charging process. Finally, the open-source nature of the project makes it possible to expand its functionalities according to the needs of the community.

A

Operator Kernels

This appendix chapter presents the code developed for the major operator kernels used in the research. Each kernel is represented by a Rust function that takes one or more tensors as input and produces a tensor as its output.

A.1 FullyConnected

```rust
/// Performs the FullyConnected operation.
/// Returns a 2-dimensional output tensor containing the result of the operation.
///
/// # Arguments
/// * `input` - The 2-dimensional input tensor
/// * `weights` - The 2-dimensional tensor representing the weights of the operator
/// * `output_scale` - The scale of the resulting output tensor
/// * `output_zero_point` - The zero point of the resulting output tensor
/// * `options` - Operator's options as an [`FullyConnectedOptions`] struct
/// * `constants` - Constant values coming from the pre-processing phase
///
pub fn fully_connected<
  T: Quantized,
  const INPUT_ROWS: usize,
  const INPUT_COLS: usize,
```

```
  const WEIGHTS_COLS: usize,
>(
  input: Tensor2D<T, INPUT_ROWS, INPUT_COLS, 1>,
  weights: &Tensor2D<T, INPUT_COLS, WEIGHTS_COLS, 1>,
  output_scale: [f32; 1],
  output_zero_point: [T; 1],
  options: FullyConnectedOptions,
  constants: (
    Buffer2D<f32, WEIGHTS_COLS, 1>,
    f32,
    Buffer2D<i32, 1, WEIGHTS_COLS>,
    i32,
  ),
) -> Tensor2D<T, INPUT_ROWS, WEIGHTS_COLS, 1> {
  let x: (
    Buffer2D<i32, INPUT_ROWS, WEIGHTS_COLS>,
    Buffer2D<i32, INPUT_ROWS, 1>,
  ) = (
    // Perform the dot product between the input and the weights
    Buffer2D::from_fn(|i, j| {
      input
        .buffer
        .row(i)
        .iter()
        .zip(weights.buffer.column(j).iter())
        .fold(0i32, |acc, (i, w)| {
          acc + i32::from_subset(i) * i32::from_subset(w)
        })
    }),
    // Perform the row-sum of the weights
    Buffer2D::from_fn(|i, _| {
      input
        .buffer
        .row(i)
        .fold(0i32, |acc, e| acc + i32::from_subset(&e))
        * i32::from_subset(&weights.zero_point[0])
    }),
  );
  // Combine the constant values and the variants to obtain the
   output
  let output = Buffer2D::from_fn(|i, j| {
    let y = T::from_superset_unchecked(&roundf(
      f32::from_subset(&output_zero_point[0])
```

```
        + constants.0[j]
        + constants.1
          * f32::from_subset(&(x.0[(i, j)] - x.1[i] - constants.2[j]
    + constants.3)),
    ));
    // Apply the fused activation function (if any)
    match options.fused_activation {
      FusedActivation::None => y,
      FusedActivation::Relu => relu(y, output_zero_point[0]),
      FusedActivation::Relu6 => relu6(y, output_scale[0],
    output_zero_point[0]),
    }
  });
  Tensor2D::new(output, output_scale, output_zero_point)
}
```

A.2 Conv2D

```
/// Performs the Conv2D operation.
/// Returns a 4-dimensional output tensor containing the result of
    the operation.
///
/// # Arguments
/// * `input` - The 4-dimensional input tensor
/// * `filters` - The 4-dimensional tensor representing the filters
    of the operator
/// * `output_scale` - The scale of the resulting output tensor
/// * `output_zero_point` - The zero point of the resulting output
    tensor
/// * `options` - Operator's options as an [`Conv2DOptions`] struct
/// * `constants` - Constant values coming from the pre-processing
    phase
///
pub fn conv_2d<
  T: Quantized,
  const INPUT_ROWS: usize,
  const INPUT_COLS: usize,
  const INPUT_CHANS: usize,
  const FILTERS_BATCHES: usize,
  const FILTERS_ROWS: usize,
  const FILTERS_COLS: usize,
```

```
  const FILTERS_QUANTS: usize,
  const OUTPUT_ROWS: usize,
  const OUTPUT_COLS: usize,
>(
  input: Tensor4D<T, 1, INPUT_ROWS, INPUT_COLS, INPUT_CHANS, 1>,
  filters: &Tensor4D<T, FILTERS_BATCHES, FILTERS_ROWS, FILTERS_COLS,
    INPUT_CHANS, FILTERS_QUANTS>,
  output_scale: [f32; 1],
  output_zero_point: [T; 1],
  options: Conv2DOptions,
  constants: (
    Buffer2D<f32, FILTERS_BATCHES, 1>,
    Buffer2D<f32, FILTERS_QUANTS, 1>,
  ),
) -> Tensor4D<T, 1, OUTPUT_ROWS, OUTPUT_COLS, FILTERS_BATCHES, 1> {
  let output = [Buffer2D::from_fn(|i, j| {
    // Extract the view using the view extraction algorithm
    let view: View<T, FILTERS_ROWS, FILTERS_COLS, INPUT_CHANS> =
      input.view((i, j), 0, options.view_padding, options.strides);
    // Perform the convolution for each filter batch
    array::from_fn(|b| {
      let input_zero_point = i32::from_subset(&input.zero_point[0]);
      let filters_zero_point = i32::from_subset(
        &filters
          .zero_point
          .get(b)
          .copied()
          .unwrap_or(filters.zero_point[0]),
      );
      let x = (
        // Perform the dot product between the input region and the
    filter
        view.buffer.zip_fold(&filters.buffer[b], 0i32, |acc, v, f| {
          acc + v
            .iter()
            .zip(f.iter())
            .map(|(e1, e2)| i32::from_subset(e1) * i32::from_subset(
    e2))
            .sum::<i32>()
        }),
        // Perform the 3-dimensional component-sum of the view
        view.buffer.fold(0i32, |acc, a| {
          acc + a.iter().fold(0i32, |acc, e| acc + i32::from_subset(e
```

```
    ))
60          }) * filters_zero_point,
61        );
62        // Elaborate the constants
63        let constants = (
64          constants.0,
65          constants.1,
66          input_zero_point
67            * filters.buffer[b].zip_fold(&view.mask, 0i32, |acc, f, m|
    {
68              if m {
69                acc + f.iter().fold(0i32, |acc, e| acc + i32::
    from_subset(e))
70              } else {
71                acc
72              }
73            }),
74          view.len as i32 * INPUT_CHANS as i32 * input_zero_point *
    filters_zero_point,
75        );
76        // Combine the constant values and the variants to obtain the
    output
77        let y = T::from_superset_unchecked(&roundf(
78          f32::from_subset(&output_zero_point[0])
79            + constants.0[b]
80            + constants.1.get(b).copied().unwrap_or(constants.1[0])
81              * f32::from_subset(&(x.0 - x.1 - constants.2 + constants
    .3)),
82        ));
83        // Apply the fused activation function (if any)
84        match options.fused_activation {
85          FusedActivation::None => y,
86          FusedActivation::Relu => relu(y, output_zero_point[0]),
87          FusedActivation::Relu6 => relu6(y, output_scale[0],
    output_zero_point[0]),
88        }
89      })
90    })];
91    Tensor4D::new(output, output_scale, output_zero_point)
92  }
```

A.3 DepthwiseConv2D

```rust
/// Performs the DepthwiseConv2D operation.
/// Returns a 4-dimensional output tensor containing the result of the operation.
///
/// # Arguments
/// * `input` - The 4-dimensional input tensor
/// * `weights` - The 4-dimensional tensor representing the weights of the operator
/// * `output_scale` - The scale of the resulting output tensor
/// * `output_zero_point` - The zero point of the resulting output tensor
/// * `options` - Operator's options as an [`DepthwiseConv2DOptions`] struct
/// * `constants` - Constant values coming from the pre-processing phase
///
pub fn depthwise_conv_2d<
  T: Quantized,
  const INPUT_ROWS: usize,
  const INPUT_COLS: usize,
  const INPUT_CHANS: usize,
  const WEIGHTS_ROWS: usize,
  const WEIGHTS_COLS: usize,
  const WEIGHTS_CHANS: usize,
  const WEIGHTS_QUANTS: usize,
  const OUTPUT_ROWS: usize,
  const OUTPUT_COLS: usize,
>(
  input: Tensor4D<T, 1, INPUT_ROWS, INPUT_COLS, INPUT_CHANS, 1>,
  weights: &Tensor4D<T, 1, WEIGHTS_ROWS, WEIGHTS_COLS, WEIGHTS_CHANS, WEIGHTS_QUANTS>,
  output_scale: [f32; 1],
  output_zero_point: [T; 1],
  options: DepthwiseConv2DOptions,
  constants: (
    Buffer2D<f32, WEIGHTS_CHANS, 1>,
    Buffer2D<f32, WEIGHTS_QUANTS, 1>,
  ),
) -> Tensor4D<T, 1, OUTPUT_ROWS, OUTPUT_COLS, WEIGHTS_CHANS, 1> {
  let output = [Buffer2D::from_fn(|i, j| {
```

```
35      // Extract the view using the view extraction algorithm
36      let view: View<T, WEIGHTS_ROWS, WEIGHTS_COLS, INPUT_CHANS> =
37        input.view((i, j), 0, options.view_padding, options.strides);
38      // Perform the convolution for each input channel
39      array::from_fn(|c| {
40        let input_zero_point = i32::from_subset(&input.zero_point[0]);
41        let weights_zero_point = i32::from_subset(
42          &weights
43            .zero_point
44            .get(c)
45            .copied()
46            .unwrap_or(weights.zero_point[0]),
47        );
48        let x = (
49          // Perform the dot product between the input region and the
   weights
50          view.buffer.zip_fold(&weights.buffer[0], 0i32, |acc, v, w| {
51            acc + i32::from_subset(&v.get(c).copied().unwrap_or(v[0]))
52              * i32::from_subset(&w[c])
53          }),
54          // Perform the 2-dimensional component-sum of the view for
   the given channel
55          view.buffer.fold(0i32, |acc, a| {
56            acc + i32::from_subset(&a.get(c).copied().unwrap_or(a[0]))
57          }) * weights_zero_point,
58        );
59        // Elaborate the constants
60        let constants = (
61          constants.0,
62          constants.1,
63          input_zero_point
64            * weights.buffer[0].zip_fold(&view.mask, 0i32, |acc, w, m|
   {
65              if m {
66                acc + i32::from_subset(&w[c])
67              } else {
68                acc
69              }
70            }),
71          view.len as i32 * input_zero_point * weights_zero_point,
72        );
73        // Combine the constant values and the variants to obtain the
   output
```

```
74        let y = T::from_superset_unchecked(&roundf(
75          f32::from_subset(&output_zero_point[0])
76            + constants.0[c]
77            + constants.1.get(c).copied().unwrap_or(constants.1[0])
78              * f32::from_subset(&(x.0 - x.1 - constants.2 + constants
    .3)),
79        ));
80        // Apply the fused activation function (if any)
81        match options.fused_activation {
82          FusedActivation::None => y,
83          FusedActivation::Relu => relu(y, output_zero_point[0]),
84          FusedActivation::Relu6 => relu6(y, output_scale[0],
    output_zero_point[0]),
85        }
86      })
87    })];
88    Tensor4D::new(output, output_scale, output_zero_point)
89  }
```

A.4 AveragePool2D

```
1  /// Performs the AveragePool2D operation.
2  /// Returns a 4-dimensional output tensor containing the result of
       the operation.
3  ///
4  /// # Arguments
5  /// * `input` - The 4-dimensional input tensor
6  /// * `_filter_shape` - The phantom shape of the filter
7  /// * `output_scale` - The scale of the resulting output tensor
8  /// * `output_zero_point` - The zero point of the resulting output
       tensor
9  /// * `options` - Operator's options as an [`AveragePool2DOptions`]
       struct
10 /// * `constants` - Constant values coming from the pre-processing
       phase
11 ///
12 pub fn average_pool_2d<
13   T: Quantized,
14   const INPUT_ROWS: usize,
15   const INPUT_COLS: usize,
16   const INPUT_CHANS: usize,
```

```
  const FILTER_ROWS: usize,
  const FILTER_COLS: usize,
  const OUTPUT_ROWS: usize,
  const OUTPUT_COLS: usize,
>(
  input: Tensor4D<T, 1, INPUT_ROWS, INPUT_COLS, INPUT_CHANS, 1>,
  _filter_shape: (Const<FILTER_ROWS>, Const<FILTER_COLS>),
  output_scale: [f32; 1],
  output_zero_point: [T; 1],
  options: AveragePool2DOptions,
  constants: (f32, f32),
) -> Tensor4D<T, 1, OUTPUT_ROWS, OUTPUT_COLS, INPUT_CHANS, 1> {
  let output = [Buffer2D::from_fn(|i, j| {
    // Extract the view using the view extraction algorithm
    let view: View<T, FILTER_ROWS, FILTER_COLS, INPUT_CHANS> =
      input.view((i, j), 0, options.view_padding, options.strides);
    // Compute the average pooling for each channel
    array::from_fn(|c| {
      let x = 1. / view.len as f32
        * view
          .buffer
          .fold(0i32, |acc, a| acc + i32::from_subset(&a[c])) as f32;
      let y = T::from_superset_unchecked(&roundf(constants.0 * x +
    constants.1));
      // Apply the fused activation function (if any)
      match options.fused_activation {
        FusedActivation::None => y,
        FusedActivation::Relu => relu(y, output_zero_point[0]),
        FusedActivation::Relu6 => relu6(y, output_scale[0],
    output_zero_point[0]),
      }
    })
  })];
  Tensor4D::new(output, output_scale, output_zero_point)
}
```

B

View Extraction Algorithm

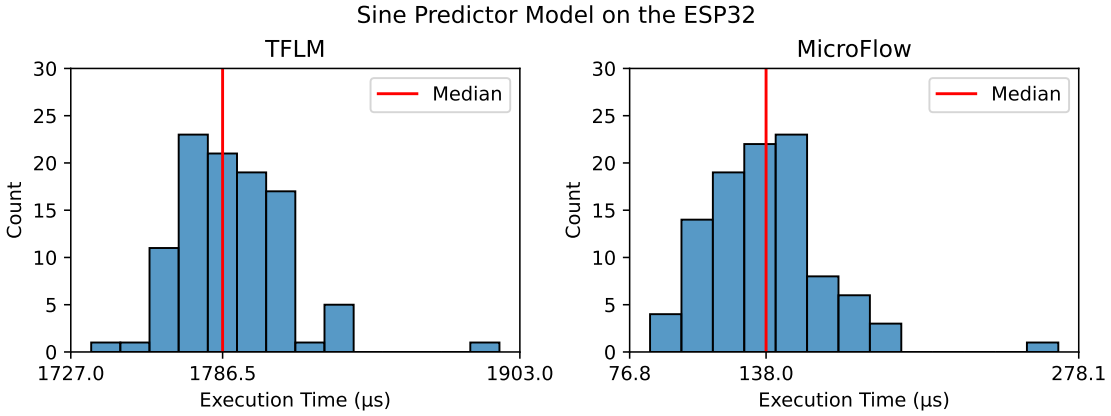
This appendix chapter presents the implementation of the view extraction algorithm presented in Algorithm 1. The algorithm has been implemented as a method of the `Tensor4D` struct.

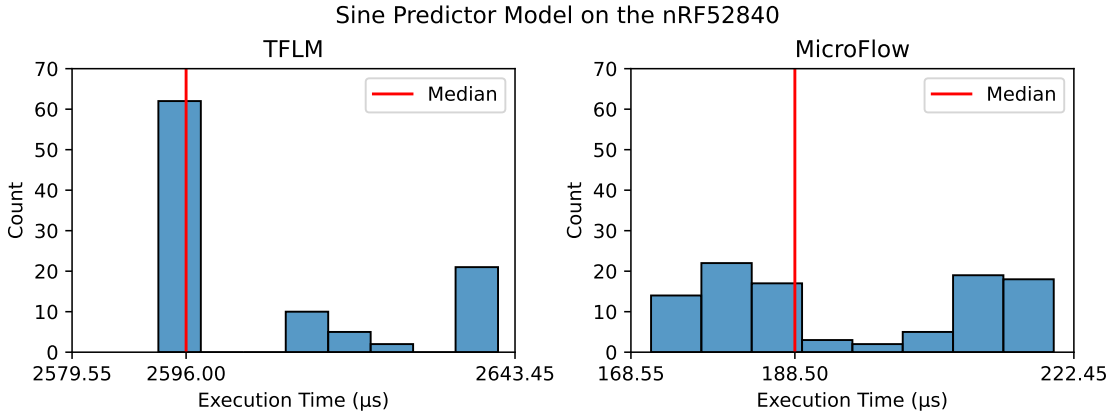
```rust
/// Extracts a view from the tensor.
/// Returns the 4-dimensional tensor view as a [`TensorView`] struct.
///
/// # Arguments
/// * `focus` - The focus point of the view, i.e., the pseudo-center of the view
/// * `batch` - The tensor batch from which to extract the view
/// * `padding` - The view padding as a [`TensorViewPadding`] enum
/// * `strides` - The view strides on the width and height of the tensor, repectively
///
pub fn view<const VIEW_ROWS: usize, const VIEW_COLS: usize>(
  &self,
  focus: (usize, usize),
  batch: usize,
  padding: TensorViewPadding,
  strides: (usize, usize),
) -> TensorView<T, VIEW_ROWS, VIEW_COLS, CHANS> {
  let mut len = VIEW_ROWS * VIEW_COLS;
  let mut mask = Buffer2D::from_element(true);
  TensorView {
    buffer: Buffer2D::from_fn(|m, n| match padding {
      TensorViewPadding::Same => {
        // Compute the index shift based on the view dimensions
```

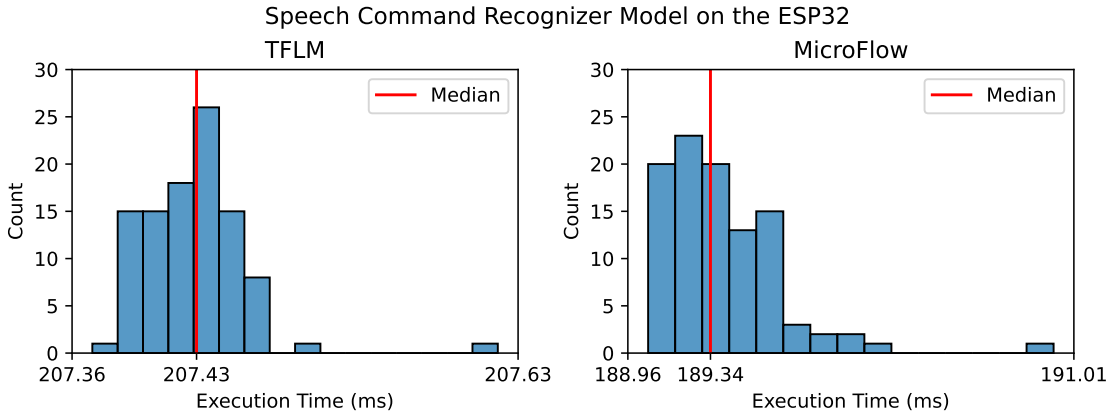
```
        let shift = ((VIEW_ROWS - 1) / 2, (VIEW_COLS - 1) / 2);
        let index = (
          // If the calculated index falls within the tensor bounds, keep it
          if let Some(x) = (strides.0 * focus.0 + m).checked_sub(shift.0) {
            x
          // Otherwise, return zero (as per "same" padding)
          } else {
            len -= 1;
            mask[(m, n)] = false;
            return [T::from_superset_unchecked(&0); CHANS];
          },
          // Same for the other index value
          if let Some(x) = (strides.1 * focus.1 + n).checked_sub(shift.1) {
            x
          } else {
            len -= 1;
            mask[(m, n)] = false;
            return [T::from_superset_unchecked(&0); CHANS];
          },
        );
        // Extract the view for the computed index
        self.buffer[batch].get(index).copied().unwrap_or_else(|| {
          len -= 1;
          mask[(m, n)] = false;
          [T::from_superset_unchecked(&0); CHANS]
        })
      }
      TensorViewPadding::Valid => {
        // For "valid" paddings, directly extract the view for valid indexes only
        self.buffer[batch][(strides.0 * focus.0 + m, strides.1 * focus.1 + n)]
      }
    }),
    mask,
    len,
  }
}
```

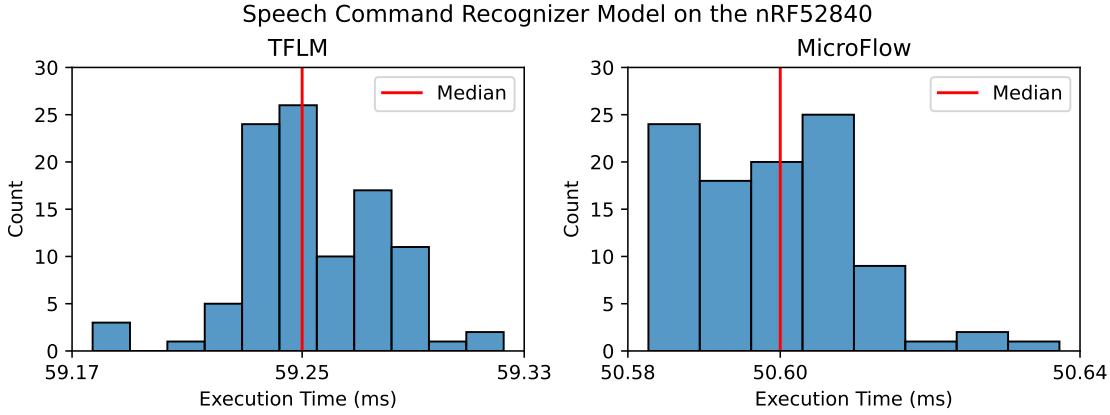
Runtime Performance Histograms

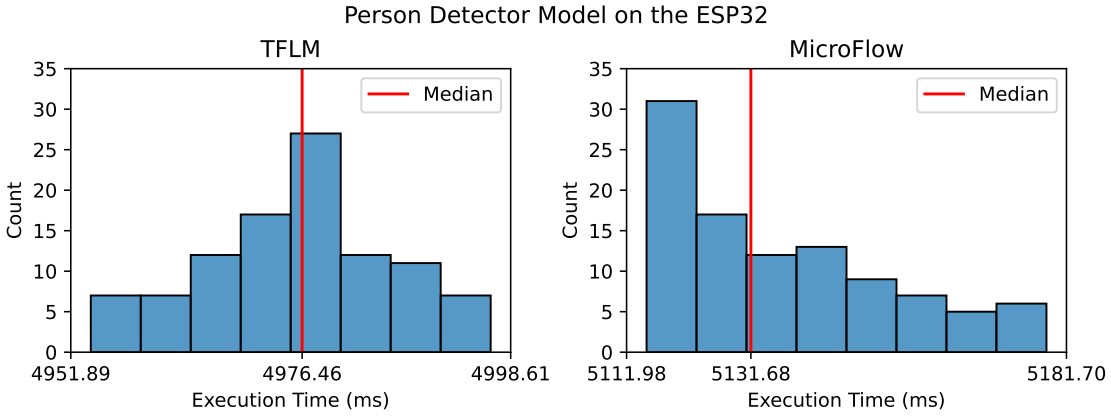
This appendix chapter presents the full set of histograms resulting from the runtime performance experiment. The plots have the minimum, median, and maximum values of the measured execution times on the x-axis and the number of executions falling within each histogram bin on the y-axis.

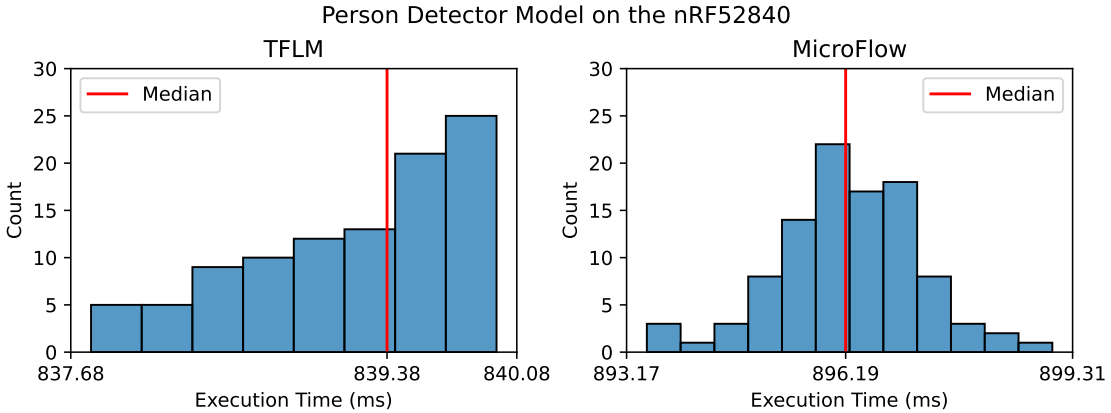



Sine Predictor Model on the nRF52840
TFLM
MicroFlow
Median
Count
Execution Time (µs)
2579.55
2596.00
2643.45
168.55
188.50
222.45


Speech Command Recognizer Model on the ESP32
TFLM
MicroFlow
Median
Count
Execution Time (ms)
207.36
207.43
207.63
188.96
189.34
191.01


Speech Command Recognizer Model on the nRF52840
TFLM
MicroFlow
Median
Count
Execution Time (ms)
59.17
59.25
59.33
50.58
50.60
50.64


Person Detector Model on the ESP32
TFLM
MicroFlow
Median
Count
Execution Time (ms)
4951.89
4976.46
4998.61
5111.98
5131.68
5181.70


Person Detector Model on the nRF52840
TFLM
MicroFlow
Median
Count
Execution Time (ms)
837.68
839.38
840.08
893.17
896.19
899.31

References

[1] Ramon Sanchez-Iborra and Antonio F. Skarmeta. "TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities". In: *IEEE Circuits and Systems Magazine* 20.3 (2020), pp. 4–18. DOI: 10.1109/MCAS.2020.3005467.

[2] Lachit Dutta and Swapna Bharali. "TinyML Meets IoT: A Comprehensive Survey". In: *Internet of Things* 16 (2021), p. 100461. DOI: https://doi.org/10.1016/j.iot.2021.100461.

[3] Guoguo Chen, Carolina Parada, and Georg Heigold. "Small-footprint keyword spotting using deep neural networks". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2014, pp. 4087–4091. DOI: 10.1109/ICASSP.2014.6854370.

[4] Miguel A Labrador and Oscar D Lara Yejas. *Human activity recognition: Using wearable sensors and smartphones*. CRC Press, 2013.

[5] Alexander Wong et al. "YOLO Nano: a Highly Compact You Only Look Once Convolutional Neural Network for Object Detection". In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. 2019, pp. 22–25. DOI: 10.1109/EMC2-NIPS53020.2019.00013.

[6] Anargyros Gkogkidis et al. "A TinyML-based system for gas leakage detection". In: *2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 2022, pp. 1–5. DOI: 10.1109/MOCAST54814.2022.9837510.

[7] Maria Francesca Alati et al. "Time series analysis for temperature forecasting using TinyML". In: *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. 2022, pp. 691–694. DOI: 10.1109/CCNC49033.2022.9700573.

[8] *Microcontroller Market Size, Share & Trends Analysis Report By Product (8-bit, 16-bit, 32-bit), By Application (Consumer Electronics & Telecom, Automotive, Industrial, Medical Devices, Aerospace & Defense), By Region, And Segment Forecasts, 2023 - 2030*. Tech. rep. 978-1-68038-141-2. Grand View Research, 2023. URL: https://www.grandviewresearch.com/industry-analysis/microcontroller-market.

[9] Daniel Zhang et al. *The AI Index 2022 Annual Report*. Tech. rep. AI Index Steering Committee, Stanford Institute for Human-Centered AI, Stanford University, Mar. 2022. URL: https://aiindex.stanford.edu/report.

[10] Sally Ward-Foxton. *TinyML Comes to Embedded World 2023*. 2023. URL: https://www.eetimes.com/tinyml-comes-to-embedded-world-2023.

[11] Samson Otieno Ooko et al. "TinyML in Africa: Opportunities and Challenges". In: *2021 IEEE Globecom Workshops (GC Wkshps)*. 2021, pp. 1–6. DOI: 10.1109/GCWkshps52748.2021.9682107.

[12] Robert David et al. "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems". In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 800–811. URL: https://proceedings.mlsys.org/paper_files/paper/2021/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[13] *MLPerf v1.0 Results Inference Tiny*. ML Commons Association, Nov. 2022. URL: https://mlcommons.org/en/inference-tiny-10.

[14] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[15] Matt Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. Microsoft Security Response Center (MSRC), Feb. 2019. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[16] *Memory Safety*. The Chromium Projects, 2020. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[17] Jeff Vander Stoep and Stephen Hines. *Rust in the Android platform*. Android Security, Apr. 2021. URL: https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[18] Linus Torvalds. *Linux 6.1*. Dec. 2022. URL: https://lkml.org/lkml/2022/12/11/206.

[19] Colby Banbury et al. "MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers". In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 517–532. URL: https://proceedings.mlsys.org/paper_files/paper/2021/file/c4d41d9619462c534b7b61d1f772385e-Paper.pdf.

[20] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].

[21] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. URL: https://doc.rust-lang.org/book/.

[22] Pete Warden and Daniel Situnayake. *TinyML*. O'Reilly Media, Inc., Dec. 2019. ISBN: 9781492051992. URL: https://tinymlbook.com.

[23] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. arXiv: 1804.03209 [cs.CL].

[24] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[25] Aakanksha Chowdhery et al. *Visual Wake Words Dataset*. 2019. arXiv: 1906.05721 [cs.CV].

Acknowledgments

This project would not have been possible without the support and help of all the people who followed this journey. First of all, I would like to express my gratitude to my family and all the people who have been close to me for their support and understanding throughout this project. Their encouragement and belief in me have been a constant source of motivation. I would also like to extend my appreciation to my supervisor and mentor for their guidance and expertise. Their valuable insights and constructive feedback have shaped the development of this project and enriched my learning experience.

Lastly, I want to express my gratitude to the Grepit company and its employees for believing in and supporting the project from its inception. The tools, knowledge, and spaces provided by the company have been essential to the successful development and execution of this thesis. Thank you all for being an integral part of this journey and for making it a truly remarkable experience.