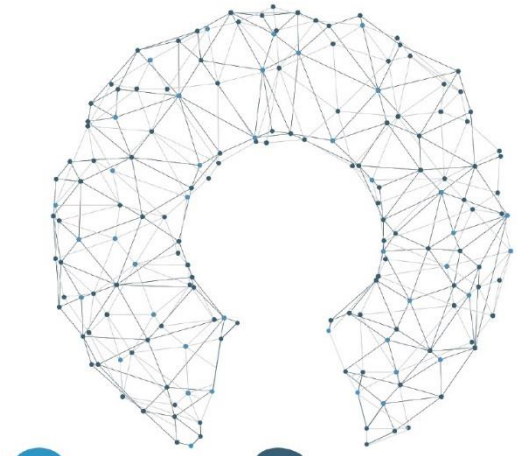

www.openswarm.eu

Call: HORIZON-CL4-2022-DATA-01

Type of action: RIA

Grant agreement: 101093046

Deliverable n°3.1 : Study on the Trade-offs Involved with Partial CNN Over-the-Air Update in Constrained Environments

Work Package n°3: Collaborative energy-aware AI

ADI

WP Lead: imec

This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046.

Funded by the European Union

Document information	
Author(s)	Sergiu Cuciurean
Reviewer	Mohamed S Talamali
Submission date	31-Oct-2024
Due date	31-Oct-2024
Type	Report
Dissemination level	PU

Document history			
Date	Version	Author(s)	Comments
28-Jun-2024	01	Sergiu Cuciurean	Preview
31-Oct-2024	02	Sergiu Cuciurean	Deliverable

Table of contents

Executive Summary

In this deliverable, we present a framework for deploying Convolutional Neural Networks (CNNs) and enabling Over-The-Air (OTA) updates on the resource-constrained MAX78000 microcontroller. This framework includes several key components: a CNN engine driver that manages the hardware accelerator, a model parser that optimizes how the network runs on the device, a firmware management system that allows partial or complete CNN updates and an OTA layer that supports models to be deployed remotely. The objective of this task is to develop a system based on MAX78000 with the capability of running an updating CNNs over the air.

We also implemented a weighted cosine k-means clustering algorithm to compress CNN models, which significantly reduces memory usage while maintaining model accuracy. In testing, this approach reduced the size of the 2d convolutional layers by up to 95% while maintaining a loss of under 1% of the original model's accuracy. The integrated solution not only reduces power consumption but also ensures reliable OTA updates.

.

1. Introduction

This document is part of OpenSwarm's WP3, which aims to enable collaborative AI training and inference on low-power swarm devices in the field. Here, we present the work completed within task T3.1 addressing the design and implementation of a system, capable to deliver over-the-air (OTA) firmware updates for an embedded device with Convolutional Neural Network (CNN) hardware accelerator. The primary goal is to develop a secure and efficient OTA update mechanism, allowing remote updates without physical intervention, targeting memory-constrained embedded devices. Special emphasis is placed on the integration of optimized CNN models for on-device AI inference, addressing the challenges related to memory usage, power efficiency and resource limitations.

In line with the objectives described the objective for T3.1 is defined as:

"Study over-the-air programming (OTAP) approaches for updating firmware images, include the mechanism used by SmartMesh IP, the SWupdate framework, the IETF SUIT series of standards as vendor implementation such as DFU and MCUBoot used in the nRF series of micro-controllers. Design an approach to partially update a Convolutional Neural Network, for example on a layer-by-layer basis, during retraining of the model. Combine this partial CNN update mechanism with OTAP, demonstrate on a combination of SmartMesh IP and MAX78000 and use that to study the trade-off between performance of re-trained the model and time and energy needed for the update."

Over-the-air updates are a crucial aspect of modern embedded systems, offering the possibility of remote updates without external intervention. This process ensures that at any time, the embedded device can be maintained during its lifecycle. In the context of AI-enabled devices, OTA updates are essential for the continuous deployment of CNN models, making it possible for the device to adapt to environmental changes.

With the rapid advancement in machine learning, particularly in Edge AI, Convolutional Neural Networks have become a pillar in computer vision and image processing.

However, the constant stream of data captured by sensors requires continuous updates to the CNN, ensuring the model is aware of new environments and to maintain the overall accuracy. Updating these networks in real-time, especially in remote or distributed environments, poses a significant challenge. Over-the-Air methodology provides a solution to this problem by updating the networks without the need for physical interactions. The goal is to ensure continuous learning and updating of the CNNs with little downtime and manual intervention.

In edge computing environments, especially with swarms of interconnected devices operating in remote areas, OTA updates for CNN models become essential. Each device in the swarm is equipped with limited processing power, memory and energy resources, making the task of maintaining the nodes and updating the CNN challenging. Efficient management of these updates, with minimal energy consumption and downtime, is essential for the swarm to deliver reliable operations. The ability to deploy updates across a distributed network of edge devices ensures consistent learning across the system allowing the devices to adapt to new environments and share insights form the newly acquired data during the updates.

In many cases, the networks or the nodes within a network are constrained, having limited capabilities in terms of processing power, storage and uptime. Energy consumption is a critical concern for battery-powered nodes and to address this, is essential to compress the data during the update, to reduce as much as possible, the energy use. This problem results in a trade-off between energy consumption and model accuracy.

2. KPI

The KPI for T3.1 in the DoA specifies that the system should be capable of storing and transferring over the air an **OTAP image of at least 1MB.** The current implementation supports the update and compression of a model whose parameters exceed 1MB. A concrete example consists of the compression of a PyramidNet implementation. The

network consists of 13 convolutional layers and one fully connected layer, with a total network size of **1.28 MB**. Trained over 40 epochs the model has an accuracy of **44.96% accuracy** on the CIFAR100 dataset. After the compression, the least optimal solution offered a compressed format of **131.76 KB** and **42% accuracy loss,** and an optimal solution of **278.42 KB** and **under 1% accuracy loss**.

3.TRL

This project was developed from first principles, starting at TRL0, without reliance on pre-existing frameworks or prior implementations and now the TRL level is **TRL4**: Technology validated in lab. The device can perform complete or partial updates of the programmed Convolutional Neural Network using over the air methodologies, with integrated compression algorithms to reduce the energy consumption during updates. The next step is to validate the system in relevant environments.

4. Publications

Authors: Sergiu Cuciurean

Title: Hybrid Weighted Cosine K-Means and 2D DCT Approach for Efficient CNN Model Compression in Edge AI: Applications to be submitted to IEEE Transactions on Artificial Intelligence

5. Background

5.1. Bootloaders

The bootloader is an essential component in modern embedded systems, serving as the initial software to be executed when a device is powered on or reset. The main functionality of the bootloader is to moderate the startup process of the device by initializing basic hardware components and preparing the system for loading and running the main software. Typically found in the device's flash memory, its primary function is to make the transfer from hardware to software, ensuring that the device started correctly.

The main role of a bootloader is to load a piece of software in the device's RAM from a predetermined location. This firmware can be any type of software such as operating systems, bare-metal applications, small test setups, or even another bootloader. During this loading process, the bootloader verifies the integrity and authenticity of the firmware. This step is required to avoid executing corrupted software that can lead to system failures. This can be done using checksums, hashing functions or cryptographic signatures, ensuring the firmware comes from a trusted source.

In addition to loading the firmware, the bootloader also includes an error-handling mechanism. In case of failures during the boot process, the bootloader can switch back to a backup version of the software that was previously functional. This rollback capability gives the embedded system an increased level of reliability by eliminating the need to program the software using conventional methods. Bootloaders can be classified into multiple categories based on their functionality.

5.1.1. Primary Bootloaders

Primary bootloaders (PBL), also known as "First-stage bootloaders" are commonly used in applications where the embedded device faces memory-related constraints. Stored

in a non-volatile memory region, the primary bootloader contains basic software components that can manage external memories, accessible to the device. The bootloader code starting point is stored at the reset vector, ensuring that every time the device is started, the bootloader is the first piece of code that is executed. The only task of the PBL is to minimally initialize the hardware and jump to a predetermined area where more complex software is stored. The PBL is part of a two-stage process where it facilitates the execution of the other software components.

5.1.2. Secondary Bootloaders

The role of a secondary bootloader (SBL) is to execute or update a memory region that contains the application code. It has low-level access to the flash memory and depending on the state of the boot process, the SBL initializes the flash driver and starts moving the target software into the designated area. After successfully writing and verifying the new software piece, the bootloader sets the program counter to the previously updated area, executing the firmware.

5.1.3. Universal Bootloaders

Universal bootloaders are designed to function across a wide range of platforms. They provide basic software packages including low-level drivers to support diverse storage media such as NOR flash, NAND or SD cards. This type of bootloader features a modular architecture, giving developers more flexibility, allowing them to extend and develop custom boot functionalities required in specific applications. These extensions can include support for new file systems, networking protocols, new hardware interfaces or diagnostic software. In addition to the supported hardware abstraction layers, the bootloaders incorporate secure boot mechanisms. These mechanisms ensure that the received firmware is authentic and comes from a verified source, preventing unauthorized access. As the last feature, the bootloaders often include command-line interfaces or graphical user interfaces. With those functionalities, the developers can

manipulate the boot process, configure and diagnose the device, allowing real-time adjustments and on-device debugging.

In the case of universal bootloaders, the boot process involves several steps:

1. Power-On: When the device is powered on or reset, the CPU runs instructions from a predetermined area. This code is usually a jump from the reset handler to the bootloader definition.

2. Hardware initialization: The CPU is initialized in its default state, the clock tree, power domain and IO's are configured for the boot process and additional peripherals are enabled depending on the supported features (e.g. UART for console output or network interfaces for network boot).

3. Boot configuration: The bootloader can be configured using hard-coded states in the bootloader code, using a command line interface, or through a boot script. By using the boot script, the boot process can be adjusted dynamically, without re-programming the bootloader or using additional software components.

4. Validation: After the configuration step, the integrity of the firmware is verified. This can be done under the form of checksums, hash functions or signed files.

5. Firmware load: The next step is to load the selected software into the device's RAM. This process involves the simple operation of reading the binary data from the source and writing it to the destination.

6. Jumping to execution: Once the software is loaded in the correct memory area, the bootloader jumps at the new entry point, handing over the control to the new software application.

5.2. Over-The-Air Programming

Over-the-air firmware updates have become an integral part of managing embedded systems, particularly in IoT devices and edge computing. Initially, the process of firmware updates required manual intervention, involving physically connecting to the device to program the new firmware. This process is often labour-intensive, making it inefficient and time-consuming for large-scale developments of swarms.

Steps in manual firmware flashing:

1. Physical Access to the Device: The first step in this process consists of physical access to the embedded device. This involves access to the device and its programming ports.
2. Connecting the Programmer: Once the device is physically accessible, the next step is connecting it to an external programmer or a computer. Common interfaces used for programming/flashing the device include:
    - UART: A serial communication protocol often used for debugging.
    - SPI: A serial communication protocol used to program flash memory chips or microcontrollers in a direct way.
    - JTAG: A more advanced interface that provides access to the internal registers of the microcontroller and other areas inside the device.
    - SWD: A two-pin alternative to JTAG used for programming ARM-based devices.

3. Preparing the Firmware: The firmware update is typically provided as a binary file (with .hex, .bin, or .elf format). The file contains the code that the device will execute once programmed. This step involves taking the source code and compiling it to target the device specific architecture.

4. Flashing the Firmware: With the help of specialized software such as OpenOCD, Jlink or ST-Link Utility, the binary is uploaded to the device's memory. This process erases the old firmware and write the new one instead. Depending on the tool or the flashing configuration, the software performs a readback of the

firmware to ensure that it matches the original binary file. This verification step is essential to ensure that the firmware was written successfully, and the software is not corrupted.

The need to physically access every device makes manual flashing a lengthy process and in the case of power loss or a broken connection, the device risks to enter an unspecified state making it unusable. This action is also known as bricking and often appears when the user is editing restricted areas or fuses. The process becomes even more problematic when dealing with a swarm of devices that may be physically distant from one another. In this case, keeping track of which devices have been successfully programmed and which are not becomes a logistical challenge. As the complexity and the scale of embedded systems grew, the limitations of manual firmware flashing became apparent. The demand for a more efficient and scalable method led to the development of OTA updates. With OTA programming, all devices can be updated quickly and simultaneously with a single click, eliminating the need for physical access and ensuring consistency across a large fleet of devices.

The OTA updates refer to the process of remotely delivering a firmware, software or configuration data to an embedded device using wireless communication technologies. This process is widely adopted in various industries, where devices often operate in locations that are hard-to-reach. These updates can fix bugs, introduce new features or even replace the whole software running on the device, all being done with zero physical interaction.

In AI-enabled embedded systems, OTA updates have an added significance. By allowing remote updates to the model running on the system, the device performance is continuously improved and enables it to adapt to changes in the environment.

An OTA update system typically consists of the following components:

1. Update Server: The server is responsible for generating, managing and distributing the firmware updates. In addition, it stores the update file and handles the authentication to ensure that the package is delivered to the correct target.

2. Target Device: The embedded device is required to contain a bootloader with the functionalities of receiving, parsing and installing OTA updates. The bootloader is a key component when comes to receiving the update securely and offering a backup mechanism in the case of update failures.

3. Communication Protocol: The server communicates with the target device using a wireless communication protocol such as Bluetooth, Wi-Fi, LoRa, cellular networks or SmartMesh. The choice of the communication protocol is dependent on the use case and the update size.

4. Update Package: The firmware or software update is provided as a package that can be presented in binary format. This package can include configuration files, software features and even parameters (e.g. to partially, or completely update a CNN model).

5.3. Delta Updates

Delta updates have become a beneficial method in the field of Over-the-Air (OTA) programming for effectively sending software or firmware modifications to embedded devices. These revisions simply send the changes between the previous and current software versions, thus greatly decreasing the data transfer requirement.

When it comes to updating machine learning models, such as Convolutional Neural Networks, delta updates can offer significant advantages for partial updates. CNNs, which are widely used for tasks such as image recognition, object detection, and video analysis, often require updates to improve accuracy or adapt to new environments. Transmitting the entire CNN model, repeatedly, can be resource-intensive in scenarios where the model's size is significant. Delta updates, when applied to CNNs, allow for the partial updating of model parameters, reducing the overhead associated with traditional full-model updates.

CNNs are composed of multiple layers, including convolutional layers, pooling layers, and fully connected layers, each of which plays a distinct role in the model's ability to learn and recognize features. Given the hierarchical structure of CNNs, different parts of the model may require updates at different times. Delta updates can focus on the portions of the model that have changed, rather than replacing the entire network,

5.3.1 Layer-Wise Updates

One efficient method for applying delta updates to CNNs is by targeting specific layers of the model. Typically, the early layers of a CNN are responsible for learning general features that are not specific to any task, making them less likely to require frequent updates. These layers are used to perform generic functions such as pattern detection and filtering, being applicable across various domains. For example, in image processing, the early layers capture features such as edges and textures. In contrast, the later layers of a CNN are responsible for making more task-specific decisions and may require more frequent updates. These layers fine-tune the network for specific outputs, such as classification or decision-making based on the learned features. For example, in an object recognition system deployed on a smart camera, the early layers might not change as new object categories are introduced, while the final fully connected layers may need updating. By focusing delta updates on the final layers of the model, the size of the update can be minimized.

Layer-wise updates are particularly advantageous when fine-tuning a CNN on new data. Instead of updating the entire network, only the task-specific layers are adjusted, reducing the amount of data that must be transmitted during the OTA process. This approach not only reduces the size of the update but also minimizes the computational burden on the embedded device, as fewer parameters need to be recalibrated during the update.

5.3.2 Sparse Updates

Sparse updates take advantage of the fact that not all parameters in a CNN need to be modified during an update. In many cases, only a small subset of weights or biases is adjusted when fine-tuning the model on new data. Delta updates can focus on these non-zero changes, transmitting only the modified parameters. This technique significantly reduces the size of the update file, as only a fraction of the model's parameters is transmitted.

Sparse updates are particularly useful in embedded systems with strict memory and bandwidth limitations. By reducing the number of parameters that need to be updated, sparse updates help maintain the efficiency of the OTA process while preserving the model's performance.

5.3.3 Quantization in Delta Updates

Quantization is a technique used to reduce the size and complexity of CNN models by reducing the precision of weights from 32-bit floating-point numbers to lower-precision formats, such as 8-bit integers. This approach is commonly used in embedded systems to reduce memory and computational requirements without significantly affecting the model's accuracy.

When applying delta updates to quantized CNN models, the resulting data can also be quantized, further reducing the size of the update. This method allows delta updates to remain compatible with quantized inference engines, which are often used in embedded systems to perform low-latency, energy-efficient inference.

Quantized delta updates can be particularly beneficial in devices such as smart cameras and wearables, where memory and computational power are limited. By reducing both the precision and the number of updated parameters, this technique ensures that updates can be applied without overwhelming the system's resources.

5.3.4 Transfer Learning in Delta Updates

Transfer learning allows a CNN that has been pre-trained on one task to be adapted to a new task by fine-tuning only the later layers of the network. This method is commonly used in scenarios where a large pre-trained model is adapted to specific tasks with smaller datasets.

Delta updates can be effectively combined with transfer learning to minimize the size of updates in CNN models. When adapting a pre-trained CNN for a new task, only the task-specific layers (typically the final fully connected layers) need to be updated. By focusing delta updates on these layers, the update process can be made highly efficient, reducing the bandwidth and memory required for the OTA update.

Despite the advantages of delta updates, several challenges arise when applying them to CNN models in embedded systems. These challenges include ensuring model consistency, managing computational complexity, and minimizing the size of the update file. One of the primary challenges in applying delta updates to CNNs is ensuring that the updated model maintains its performance and consistency. CNNs rely on a highly interconnected structure, where changes in one layer can affect the performance of other layers. If delta updates are applied to only certain parts of the network, it is critical to ensure that the updated model remains aligned with the rest of the architecture.

To address this issue, rigorous validation must be performed after the delta update is applied. This may involve running the updated model through a set of test cases to ensure that the changes have not introduced any errors or degraded performance. For embedded systems with limited computational resources, validation may need to be offloaded to the cloud or performed using lightweight test cases. Generating efficient delta updates for CNN models can be computationally intensive. CNNs often contain millions of parameters, and calculating the differences between two versions of a model may require significant processing power. In embedded systems, which typically have limited computational capabilities, this process can be challenging.

Techniques such as sparsity and quantization help reduce the size of delta updates, but they also introduce complexity in the update generation process. Sparse updates, for example, require careful tracking of which parameters have changed, while quantized updates must ensure that the diff file remains compatible with the quantized model. Balancing the need for small delta files with the requirement for accurate model updates is a non-trivial problem that requires careful optimization.

5.4. CNN Model Compression

Convolutional Neural Networks (CNNs) have become a cornerstone of modern deep learning, especially in the domain of image processing. They are lauded for their capability to automatically extract hierarchical features from data, significantly enhancing performance in various tasks such as classification, detection, and segmentation. However, the substantial increase in model complexity and the sheer number of parameters in deep networks lead to challenges, including overfitting, increased computational costs, and difficulties in deployment, particularly in resource-constrained environments such as mobile devices and embedded systems.

Model compression techniques aim to reduce the size and complexity of CNNs while preserving their performance. These methods include pruning, quantization, Huffman coding, low-rank factorization, and weight sharing. Each technique is designed to maintain accuracy while minimizing the memory footprint and computational demands of the models.

5.4.1. Pruning

Pruning is an effective strategy for reducing the size of CNNs by eliminating weights that contribute minimally to the overall model performance. The objective is to retain the most crucial weights while removing those that have little impact, thereby achieving a sparser model.

The pruned weight matrix can be represented mathematically as:

$$W' = \{ w_i \in W \mid |w_i| > \varepsilon \}$$

where:

- W' is the pruned weight matrix
- W is the original weight matrix
- w_i is an individual weight
- ε is a predefined threshold

Weights with absolute values below this threshold are discarded.

Pruning can be further enhanced by considering the sensitivity of each weight, which can be defined as:

$$S(w_i) = \left| \frac{\partial L}{\partial w_i} \right|$$

where:

- $S(w_i)$ is the sensitivity of weight w_i
- L is the loss function of the model

This sensitivity metric helps identify which weights are less critical, allowing for more aggressive pruning without significantly affecting model accuracy.

By utilizing these principles, researchers have demonstrated the potential for CNNs to perform effectively with fewer parameters, resulting in models that are faster and consume less memory, particularly beneficial for deployment on edge devices.

5.4.2 Quantization

Quantization is a technique used to reduce the precision of the weights and activations in a CNN, ultimately minimizing the memory required to store the model. This process typically involves converting 32 or 64-bit floating-point representations of weights into lower precision formats, such as 16, 8, or even sub-8-bit integers.

The quantization of weights can be expressed as:

$$W_q = Q(W)$$

where:

- W_q represents the quantized weights
- Q is the quantization function

An essential aspect of quantization is minimizing the accuracy loss associated with the reduction in precision. This can be framed as a quantization-aware training loss function:

$$L_q = L\left(y, \hat{y}\left(Q(W)\right)\right)$$

where:

- y denotes the true labels
- y represents the predicted outputs of the quantized model

To achieve efficient model compression, adaptive quantization strategies are employed, which dynamically adjust the precision of different weights based on their significance. The goal can be framed as:

$$\min \sum_{g \in G} E_q(g) \quad \text{subject to} \quad g = Q\left(W_g\right)\text{s}$$

where:

- G is the set of groups of weights
- $E_q(g)$ is the quantization error for group g.

By applying these adaptive techniques, it is possible to balance model efficiency and performance effectively.

5.4.3 Huffman Coding

Huffman coding is a widely adopted lossless compression technique that can significantly reduce the memory footprint of CNN weights. By assigning variable-length

codes to different weights based on their frequency of occurrence, Huffman coding compresses the weight representation without sacrificing any data.

The average length of a Huffman code can be computed as:

$$L = \sum_{i=1}^{n} p_i \cdot l_i$$

where:

- p_i is the probability of the i-th weight
- l_i is the length of its Huffman code.

This technique is particularly advantageous as it reduces the overall model size, facilitating the storage and transmission of CNNs on resource-constrained platforms.

5.4.4 Low-Rank Factorization

Low-rank factorization is a powerful approach to compressing CNNs by decomposing weight tensors into lower-dimensional representations. Techniques such as Singular Value Decomposition (SVD) can be employed to achieve this.

The weight matrix decomposition can be expressed as:

$$W \approx U\Sigma V^T$$

where:

- U and V are orthogonal matrices
- Σ is a diagonal matrix containing the singular values of W

This approximation captures the most significant components of the weight information while reducing the number of parameters.

The rank of the approximated weight matrix is denoted as k. The goal is to minimize the rank while retaining the essential characteristics of the original matrix:

$$\min |W - U_k \Sigma_k V_k^T|$$

where:

- U_k and V_k are the truncated matrices corresponding to the top k singular values

By reducing the rank k, we achieve a more compact representation of W, thus reducing memory usage and improving inference speed. However, caution must be exercised, as overly aggressive reduction in rank can lead to accuracy loss.

5.4.5. Weight Sharing

Weight sharing is a compression strategy that focuses on reducing redundancy among weights in CNNs. Instead of storing unique weights for each connection, similar weights are grouped together and represented by a shared value, resulting in significant memory savings with minimal impact on performance.

In the article "CNN Weight Sharing based on a Fast Accuracy Estimation Metric"[1] by Etienne Dupuis, David Novo, Ian O'Connor, and Alberto Bosio, the authors present a novel method for weight sharing in CNNs. Their approach focuses on compressing CNN models by clustering similar weights based on a metric that estimates the accuracy of the compressed model. The paper introduces a simple yet effective method to compress convolutional layers using k-means, by storing only the cluster centres and the corresponding weight index. The proposed approach uses parameter sharing via row-wise k-Means clustering. They apply k-Means across rows of weight matrices across all the fully connected and convolutional layers. Each row is treated as a feature and similar rows will be grouped in a cluster. For each cluster, the centroid will be assigned as the shared value between all the weights in that row. Once the clustering is complete, all the weights in each row are replaced by the corresponding shared value, reducing the number of unique weights.

The objective function for k-means clustering, often used in weight sharing, can be defined as:

$$J = \sum_{j=1}^{k} \sum_{x_i \in C_j} ||\, x_i - \mu_j ||^2$$

where:

- J is the total cost function
- k is the number of clusters
- C_j is the set of points in j
- μ_j is the centroid of cluster j

By applying k-means clustering to the weights, each weight can be replaced with the centroid of its respective cluster, thereby reducing the number of unique weights stored.

In the paper "Deep Learning Model Compression using Network Sensitivity and Gradients"[2], the authors introduce a compression methodology that leverages the sensitivity of weights and gradients. Their approach focuses on the importance of each weight in relation to the model's performance by analysing the gradient magnitude and model sensitivity during training. By evaluating the gradients of the weights, the method can determine which weighs are critical and should be less affected by the compression method to maintain the accuracy. The network sensitivity is determined by how the model reacts to changes in the weight matrices, and based on the resulting values, the less sensitive weights are targeted for more aggressive compression. The paper presents experimental results that demonstrates how gradient-based compression can achieve significant compression while maintaining accuracy: for VGG-16 a 4.56x compression was achieved with only 0.5\% top-5 accuracy drop. By assessing the gradients of the weights, less critical weights are targeted for aggressive compression. The resulting sensitivity can be expressed as:

$$S(w_i) = \left|\frac{\partial L}{\partial w_i}\right|$$

where:

- $S(w_i)$ is the sensitivity of weight w_i
- L is the loss function of the model

This allows the method to evaluate how model performance changes with respect to alterations in weight values, ensuring that the most critical weights remain intact during the compression process.

6. The MAX78000 Microcontroller

This chapter provides a detailed look at the microcontroller used for this project and the corresponding evaluation board.

The microcontroller (MCU) is the primary processing unit of the embedded device. It handles real-time operations, including sensor input, data processing, communication and decision-making. For OTA updates, the MCU manages the downloading of the firmware with its specific protocol, the receiving side validation, and the installation of the new firmware using the bootloader. When choosing the MCU, a few key aspects will be taken into consideration.

The first aspect refers to the MCU processing power. It must have sufficient resources to handle fast communication protocols, encrypt/decrypt files and decompress packets in a timely manner. In terms of AI acceleration, the MCU requires sufficient computational power to handle CNN models efficiently. For example, the ARM Cortex-M family offers high-performance MCUs with enough processing power to support CNN inference while still maintaining relatively low power consumption. The Multiply-and-Accumulate (MAC) units are an essential component for matrix multiplications, providing a more efficient execution of the convolutional layers present in the model. Furthermore, the Cortex-M7 includes both single and double precision FPUs which are helpful when running neural networks that use floating-point operations.

To fully leverage the Cortex-M hardware for CNN inference, ARM provides the CMSIS-NN library, which optimizes the operators defined in the neural network to be run on the Cortex-M processors. This library includes optimized functions used in deep learning

models such as: quantization support for 8-bit and 16-bit fixed-point operations reducing the size of the model and improving the inference times, optimized convolution functions adapted to reduce both memory usage and execution time, and also implementations of pooling operators and activation functions that are commonly used in CNN architectures. Using the CMSIS-NN library with a Cortex-M enabled processor, the task of developing CNNs for tasks such as image classification, object detection or audio processing at the edge becomes possible with smaller architectures like MobileNetV1 running efficiently on Cortex-M platforms.

The second aspect is related to the device flash memory. The flash memory stores both the firmware and CNN models and often includes a partitioning scheme to support OTA updates and rollback to a trusted and functional version of the software in the case of failed updates. Many embedded systems implement a dual-partition approach where one partition stores the current running software, and the other one is reserved for new updates. When receiving an update, it is stored on the second partition and after validating the newly downloaded update, the system switches to use the new partition.

The last and most important aspect is the integrated CNN accelerators. For applications that require more computational power for real-time CNN execution, some MCUs are equipped with specialized hardware, dedicated for neural network acceleration. These hardware cores are designed to parallelize and optimize AI workloads, offloading a part of the neural network operations from the CPU, resulting in a bigger performance and lower energy consumption.

The MAX78000 is the optimal solution for edge AI applications that require low-power and efficient execution of CNN models. A simplified diagram of the MAX78000 architecture is provided in Figure 1.

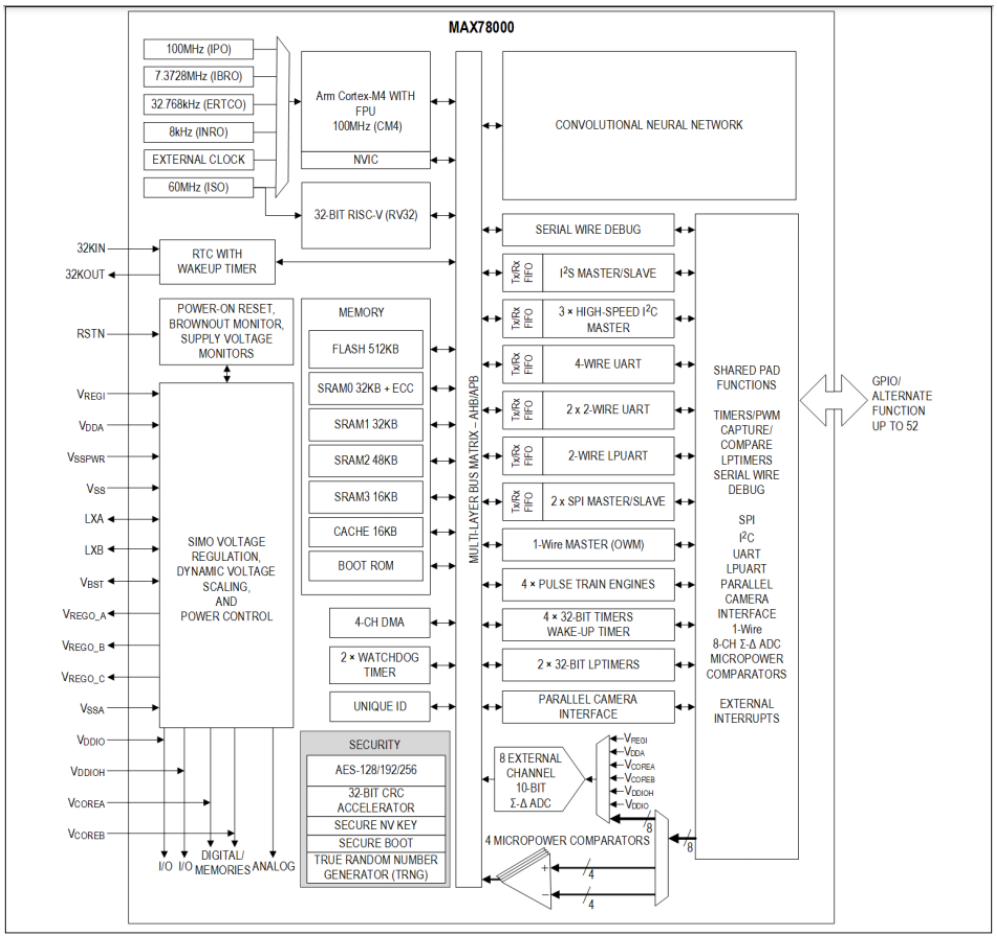

Figure 1 – Simplified block diagram of MAX78000

The MAX78000 is an advanced ultra-low-power microcontroller featuring an ARM Cortex-M4 with FPU, a 32-bit RISC-V coprocessor and an ultra-low-power neural network accelerator:

- Dual-Core Ultra-Low-Power Microcontroller
    - Arm Cortex-M4 Processor with FPU up to 100MHz
    - 512KB Flash and 128KB SRAM
    - Optimized Performance with 16KB Instruction Cache
    - Optional Error Correction Code (ECC-SEC-DED) for SRAM
    - 32-Bit RISC-V Coprocessor up to 60MHz

- Up to 52 General-Purpose I/O Pins
- 12-Bit Parallel Camera Interface
- One I2S Master/Slave for Digital Audio Interface

- Neural Network Accelerator
  - Highly Optimized for Deep Convolutional Neural Networks
  - 442k 8-Bit Weight Capacity with 1,2,4,8-Bit Weights
  - Programmable Input Image Size up to 1024 x 1024 pixels
  - Programmable Network Depth up to 64 Layers
  - Programmable per Layer Network Channel Widths up to 1024 Channels
  - 1- and 2-Dimensional Convolution Processing
  - Streaming Mode
  - Flexibility to Support Other Network Types, Including MLP and Recurrent Neural Networks

- Power Management Maximizes Operating Time for Battery Applications
  - Integrated Single-Inductor Multiple-Output (SIMO) Switch-Mode Power Supply (SMPS)
  - 2.0V to 3.6V SIMO Supply Voltage Range
  - Dynamic Voltage Scaling Minimizes Active Core Power Consumption
  - 22.2µA/MHz While Loop Execution at 3.0V from Cache (CM4 Only)
  - Selectable SRAM Retention in Low-Power Modes with Real-Time Clock (RTC) Enabled

The MAX78000FTHR board is an ultra-compact development platform designed for rapid prototyping of AI and machine learning applications. It features the MAX78000

MCU with its integrated CNN accelerator, providing a powerful, energy-efficient solution for edge AI tasks. The board is equipped with a wide range of peripherals, including a QSPI Flash, USB interface, and debugging support, making it easy to develop, test, and deploy AI models.

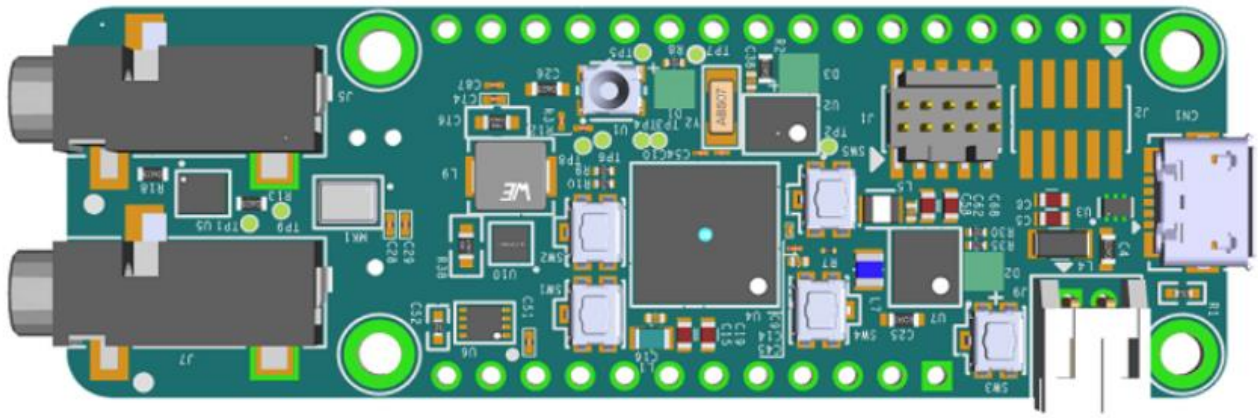

Figure 2 - MAX78000FTHR development board

The MAX78000FTHR is packed with features that make it versatile for various edge AI applications. The microSD card port, which increases the board's storage capacity, is one of its most notable features. This capability allows the board to hold numerous firmware versions, multiple AI models, large datasets, and different versions of the operating software. This is particularly useful for applications that require big collections of data that are periodically updated or in scenarios when multiple models are swapped during operation to mimic the operation of a bigger and more complex model.

The USB-to-UART interface of the MAX78000FTHR makes debugging and serial communication straightforward, with a convenient method for logging and interacting with the device. The onboard USB interface also supports USB power, making it easy to use during early development.

7. Implementation

In this chapter, we will detail the implementation of the framework designed to deploy Convolutional Neural Networks (CNNs) and enable Over-The-Air (OTA) updates on the MAX78000 microcontroller.

A pivotal element of this project is the sophisticated software ecosystem designed to maximize the performance and flexibility of the MAX78000. This software integrates seamlessly with the microcontroller's specialized CNN hardware, managing essential functions such as hardware initialization, CNN layer configuration, model weight and bias loading, and the efficient feeding of input data for inference. By optimizing memory management and computational processes, the software ensures that CNN models operate effectively within the limited resources of the MAX78000, achieving high inference speeds and energy efficiency. Furthermore, the software framework incorporates robust mechanisms for OTA updates, enabling the remote deployment and secure installation of new CNN models without requiring physical access to the device. This capability is crucial for maintaining and enhancing AI applications in dynamic environments, allowing for continuous improvement and adaptation in response to evolving requirements. Additionally, the framework supports advanced model compression techniques, which significantly reduce the memory footprint of CNN models while preserving their accuracy, thereby facilitating the deployment of sophisticated AI tasks on the MAX78000. This comprehensive software solution not only enhances the operational efficiency and scalability of AI applications on the MAX78000 but also ensures their adaptability and longevity in diverse and evolving embedded environments.

The diagram below (Figure 3) illustrates the architecture of this software ecosystem, showing how various components interact to achieve optimal performance on the MAX78000 microcontroller. The CNN engine, responsible for executing neural network computations, works in conjunction with the model parser, which processes model architecture, weights, and biases to prepare them for inference. The firmware management module oversees OTA updates, ensuring secure deployment of new

models, while the communication module supports multiple interfaces such as Bluetooth, Wi-Fi, SmartMesh, and UART, enabling seamless connectivity with external devices. Additionally, a Command Line Interface (CLI) facilitates user interaction for managing firmware updates, model loading, and initiating inference tasks. Together, these components form a cohesive framework that drives efficient AI processing and remote updates on resource-constrained platforms like the MAX78000

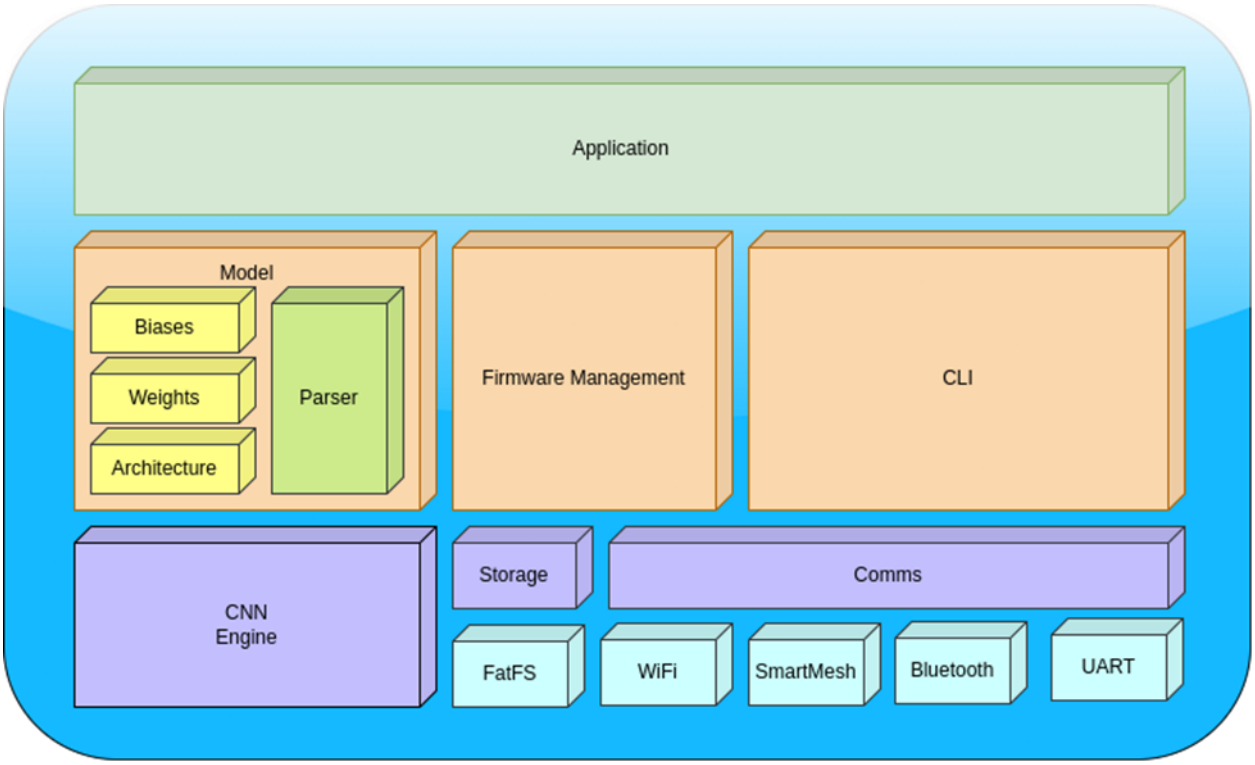

Figure 3 – The proposed software architecture

This architecture is designed not only to ensure efficient operation but also to facilitate adaptability and scalability in real-world applications. Given the resource constraints of embedded devices like the MAX78000, it is critical to evaluate how well the framework supports key functionalities such as model compression and OTA updates. To ensure the system operates as intended and meets performance expectations, we must assess its effectiveness using specific metrics. These metrics will help quantify the impact of model compression on inference speed and accuracy, as well as evaluate the reliability of OTA updates in dynamic environments. To measure the effectiveness of

compressing and updating over-the-air of a CNN model, the following metrics will be used:

- OTA update
    - Update Success Rate: The percentage of OTA updates that are successfully applied without errors or failures.
    - Update Latency: The time taken from the initiation of the OTA update process to its successful completion on the device.
    - Downtime: The amount of time the system is unavailable or offline during the OTA update process.
    - Update Frequency: The number of OTA updates that can be applied within a given time frame.

- Compression
    - Compression Ratio: The ratio of the original data size to the compressed data size, indicating the effectiveness of the compression algorithm.
    - Decompression Time: The time taken to decompress the data back to its usable form on the device.
    - Energy Consumption: The amount of energy consumed during the compression and decompression processes.
    - Impact on Model Accuracy: The change in model accuracy before and after applying the compressed update, measuring any loss in performance.
    - Bandwidth Utilization: The amount of network bandwidth used during the transmission of compressed updates, compared to uncompressed updates.
    - Compression Overhead: The additional computational resources and time required to compress.

7.1. Inference on the MAX78000

The MAX78000 is specifically made to execute CNNs at the edge and is optimized for low-power AI task execution. This device provides balanced and energy-efficient

computational performance by fusing general-purpose processor units with specialized CNN hardware.

In AI-enabled embedded systems, the efficient execution of inference is a crucial task. However, using the MAX78000's CNN accelerator capabilities for faster inference requires more than just hardware integration. It demands a comprehensive software stack that manages the low-level operations of the hardware, the configuration, data flow, memory management and the communication between the hardware components. The development of low-level drivers, model parsers, operator wrappers and inference pipelines are necessary to manage the CNN execution and provide a flexible workflow.

This chapter provides a detailed breakdown of the system developed to enable CNN inference on the MAX78000 with OTA update capabilities.

7.1.1. Low-level CNN Engine Driver

The low-level CNN engine driver is the foundational software layer that directly interacts with the CNN hardware accelerator of the MAX78000. The driver is responsible for configuring, initializing, and switching the execution states of the hardware, enabling the basic functionalities for the other software blocks used in this configuration. It also integrates the execution of CNNs in an optimized manner, serving as the key interface between the higher-level software components such as model parser and the hardware.

The CNN accelerator consists of 64 parallel processors with 512KB of SRAM-based storage. Each processor includes a pooling unit and a convolutional engine with dedicated weight memory. Four processors share one data memory. These are further organized into groups of 16 processors that share common controls. A group of 16 processors operates as a slave to another group or independently. Data is read from SRAM associated with each processor and written to any data memory located within the accelerator. Any given processor has visibility of its dedicated weight memory and to the data memory instance it shares with the three others. The CNN is organized in

four CNNx16 processor quadrants, as shown in Figure 4, generically referred to as CNNx16_n. Each CNNx16_n processor quadrant contains sixteen processors grouped in four groups of four. These groups are labeled group 0 to group 3 in the CNNx16_ processor quadrant block diagram.

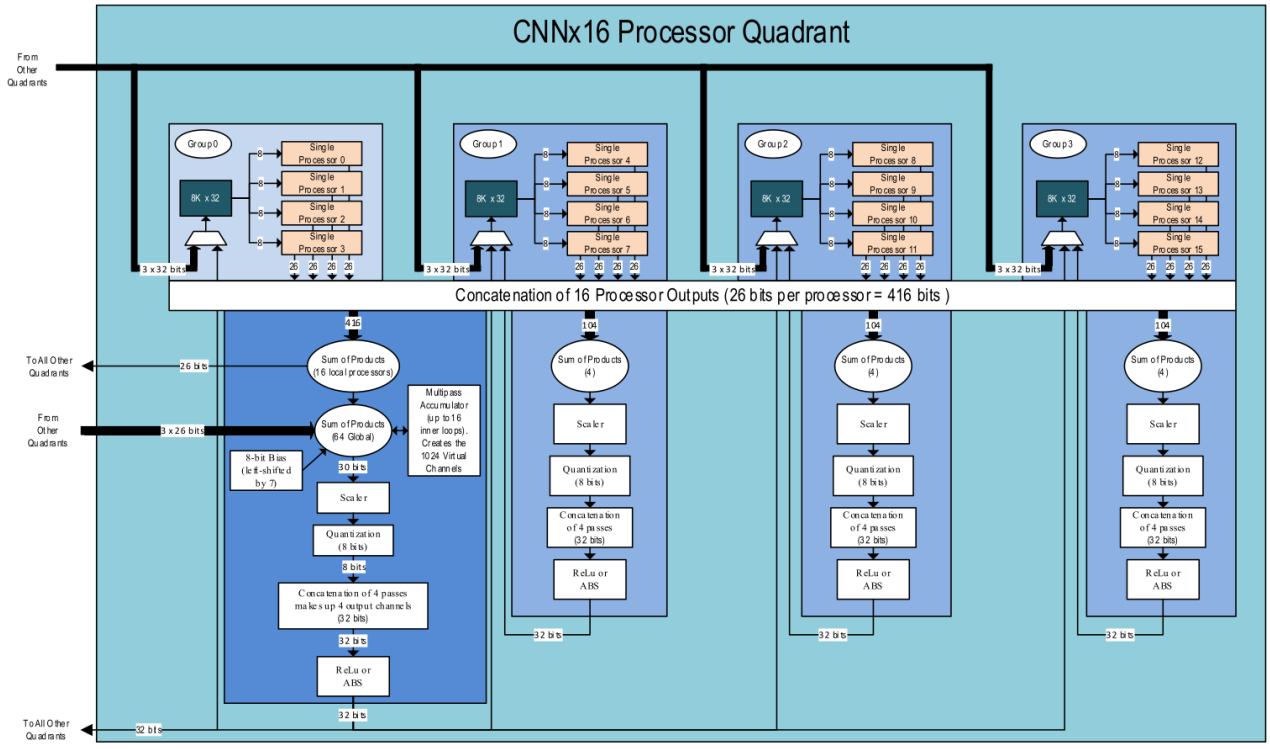

Figure 4 - CNNx16_n Processor Quadrant Block Diagram

Each processor array contains 128KB of SRAM, 110KB of mask RAM (MRAM), 512B of bias RAM (BRAM), and 98KB of tornado RAM (TRAM).

Before executing a CNN model, the CNN engine must be properly configured. The low-level CNN engine driver provides functionality to:

- Initialize the accelerator hardware: This includes setting up the hardware registers, configuring the initial memory, clocks, and interrupts.
- Configure CNN layers: The driver writes the configuration for each layer in the neural network, ensuring that the parameters are mapped correctly to the hardware. For each layer, the driver translates the high-level CNN architecture into hardware-specific instructions.

- Load the model weights and biases: Once the CNN architecture is defined, the weights and biases are transferred from the initial storage to the accelerators specialized RAM areas, layer-by-layer, with respect to each layer configuration.
- Feed input data to the accelerator: Input data, such as images or sensor readings, are loaded into the accelerator RAM and prepared for execution.

The driver's modular design and integration with the broader software ecosystem ensures flexibility and scalability. It partially translates high-level neural network architectures into hardware-specific instructions making it possible to take full advantage of the provided CNN hardware accelerator.

7.1.2. CNN Model Parser

The model parser plays a vital role in the execution of CNNs on the MAX78000, acting as the intermediary between high-level model representations and low-level hardware execution. Its primary functions include determining layer configurations, optimizing processing resource allocation, parsing layer operators, and managing the efficient organization of weights and biases.

The first task of the model parser is to analyze the architecture of the CNN. Understanding the architecture is a crucial step because it lays the groundwork for the next optimization steps. After establishing the number of layers and their configuration, the processor mapping begins. This step involves determining the optimal number of parallel processors to be allocated for each layer, balancing efficiency, and compatibility with the next layer's processing configuration. In this way the parser searches for an optimal solution that maximizes throughput while also considering processing unit availability. This not only accelerates inference times but also enables better power management, making it especially important for battery-powered edge devices.

The next step involves parsing each layer's operators and configurations. During this process, the parsers evaluate how each operation can be optimized for the driver and see if parts of the layer can be fused together to reduce the execution steps.

An essential aspect of the parser's functionality is the memory management of the weights and biases. Each operator in the network may require different modes of indexing for the weights and biases, complicating memory access. The parsers address this challenge by organizing the model parameters in a manner that aligns with the processor mapping and layer configurations. This memory optimization is crucial for minimizing the latency during a model update.

7.1.3. Firmware Management

Firmware management is a crucial component of the whole OTA update system, being responsible for handling multiple software components stored under the format of firmware blobs. It plays an essential role in ensuring that the right CNN model is available and ready to be switched.

It retrieves the binary blob containing the model's architecture, weights, biases, and configuration data from a pre-designated memory address. In addition to internal memory management, the firmware block also supports loading models from external storage such as SD cards. Using the FAT file system, the user can navigate through directories and select the appropriate firmware to be loaded. This functionality is particularly important when the device's internal memory is not sufficient to store multiple models or different layer configurations. Running on top of a storage software block, the firmware management system can mount partitions, navigate through directories, and load binary blobs that are passed to the model parser.

7.1.4. Communication Layer

The communication layer is also a vital component of the system. It enables the interaction of the MAX78000 with external devices offering functionalities such as sending and receiving data, updating the firmware content, and instructing commands to the embedded device. It was designed to be modular and flexible, covering multiple protocols including UART, Bluetooth, Wi-Fi and SmartMesh.

The modular nature of the communication layer allows the system to interface several communication protocols. Each protocol is implemented with its specific handlers but shares a common transfer flow, giving the MAX78000 the flexibility to switch between interfaces. The supported protocols are:

- UART: a quite simple and reliable communication method that is usually used for debugging in the initial stages of development.

- Bluetooth: allows lower speed short-range wireless communication, enabling the embedded device to connect to computers or mobile devices.

- Wi-Fi: more efficient data transfer with network-based connectivity. It allows the device to transfer large files with a higher bandwidth but also gives the possibility to run webservers that can provide more detailed information and controls of the node.

- SmartMesh: SmartMesh is the targeted communication protocol, designed for mesh networking. It enables communication with the nodes within a mesh network through neighbor nodes.

At its core, the communication layer provides handles to send, receive and acknowledge data packets. Due to the constraints of the MAX78000 embedded memory, the software is required to support a more complex file transfer protocol to handle a large file to be sent in multiple chunks, matching the available memory of the device. The protocol is illustrated in Figure 5, starting with a "READY" signal. The file name and size are passed by the server and the chunk size is set to fit into the allocated RAM buffer. The protocol ensures that every chunk is valid during the transmission of the file through a CRC step that is computed by both the server and the device. After the last chunk, the device acknowledges the successful transfer and closes the connection.

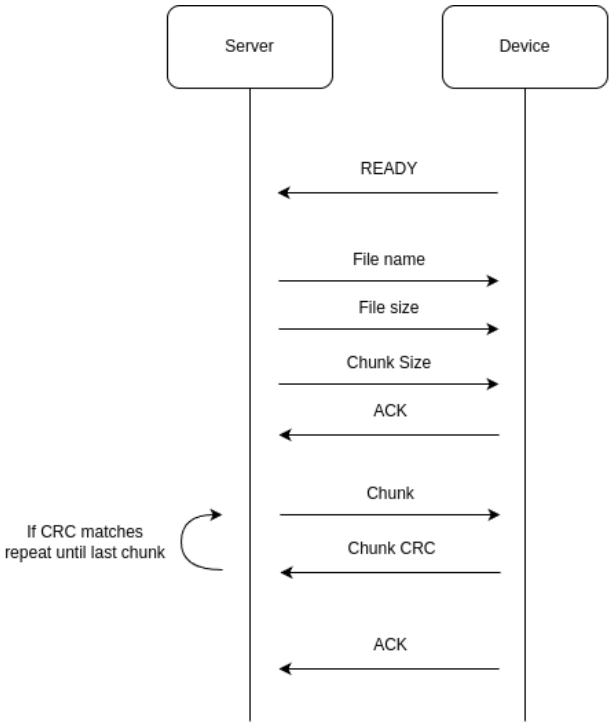

Figure 5 – Protocol for large file transfers using MAX78000

7.1.5. Command Line Interface

The Command Line Interface (CLI) is a useful component in the system, providing the user a straightforward way to interact with the device. The CLI offers the possibility to browse the file system, manage files, load firmware files, initiate file transfers with the server and execute inference tasks. It was designed to be simple and user-friendly, mirroring common UNIX-like commands.

The CLI provides a set of basic file management commands used to interact with the FAT file system on the device:

- pwd (Print Working Directory): This command prints the current directory that that the user moved to, inside the file system.

- cd (Change Directory): The "cd" command is used to navigate between directories, entering the paths provided by the user.

- ls (List Directory Contents): This command displays all the files and directories contained in the current working directory.

- rm (Remove File): The “rm” command allows the user to delete unused files or folders to free up storage space.

In addition to the file system functionalities, the CLI offers handles to initiate firmware updates, transferring new CNN updates from the server to the device. Using the communication layer, the device initiates the file transfer, requesting a file from the server:

- update: The “update” function called without parameters, saves the file to a path defined by the server. If the server does not provide a path, the file will be saved in the root directory of the device file system.
- update [path]: The function also can be called with a parameter containing the file name and the path where the user wishes to store the file.

For the CNN engine, the CLI offers two separate commands to interact with:

- load [path]: To load a certain CNN model to the engine

- predict: To initialize the CNN engine and start the inference

The “predict” command is a more special one since it also requires input data to be fed to the network to label it. The process used is like the file transfer, where the device signals the fact that it is ready to receive the input data, the server sends a header describing the data and it is acknowledged by the device, and after the data sample is completely transferred into the device memory, the inference starts. With the model loaded into the engine, the driver reads the engine response, parses it, and returns the corresponding label to the server flagging a complete prediction cycle.

7.2. Implementing OTA Updates

In this section, we present the implementation of the OTA update system designed for the MAX78000 microcontroller. The architecture of the OTA update system is meticulously designed to ensure seamless, secure, and efficient delivery of updates to the MAX78000. It comprises several interrelated components that collectively manage the entire update lifecycle from initiation to deployment.

The Update Server acts as the central repository hosting the latest versions of CNN models and firmware. It manages version control, ensuring that only compatible and validated updates are available for deployment. The server handles incoming update requests from edge devices and orchestrates the distribution of firmware blobs to the appropriate recipients.

Leveraging the versatile Communication Layer described in section 7.1.4, the OTA update system utilizes multiple protocols (in this case SmartMesh) to establish reliable connections between the update server and the MAX78000. This integration ensures that updates can be delivered over various network conditions and infrastructure setups, enhancing the system's flexibility and reach.

The firmware module is responsible for handling the reception, validation, and storage of firmware blobs received via OTA updates. It interfaces with the Firmware Management component detailed in section 7.1.3, ensuring that incoming updates are correctly parsed, validated for integrity, and securely stored in designated memory regions.

The OTA update process is orchestrated through a series of well-defined steps to ensure that updates are delivered reliably and securely. The workflow encompasses initiation, transmission, validation, deployment, and verification phases.

The update process can be initiated either automatically, based on predefined criteria such as periodic checks or version discrepancies, or manually through CLI commands. Upon initiation, the Update Controller queries the Update Server for available updates.

Once an update is identified, the Communication Layer establishes a connection with the Update Server using SmartMesh. The firmware blob is then transmitted in segmented packets to accommodate the MAX78000's memory constraints.

The Firmware Management Module receives the firmware packets, reassembles them into the complete firmware blob, and stores them in a temporary memory region designated for incoming updates. This ensures that the existing operational firmware remains unaffected during the update process.

Before deployment, the firmware blob undergoes validation checks. The integrity check is done using sha256 hashing function and the server verifies if the hash of the update matches with the hash received from the device. Upon successful validation, the Firmware Management Module transitions the device to the new firmware. This process involves overwriting the existing firmware or updating specific components, depending on the nature of the update. After deployment, the system conducts post-update verification to ensure that the new firmware is functioning as intended. This includes running diagnostic tests, verifying the operational status of the CNN models, and confirming that all services are operational. If any anomalies are detected, the system can automatically initiate rollback procedures.

In cases where the update fails validation or introduces critical issues, the system reverts to the previous firmware version. The Firmware Management Module maintains a backup of the last stable firmware to facilitate this rollback, ensuring that the device remains operational even in the event of update failures.

7.3. CNN compression based on weight sharing

Convolutional Neural Network has revolutionized the field of deep learning, particularly image processing tasks, with their ability to learn hierarchical feature representations from images. However, the significant increase in performance is also associated with many parameters, leading to problems such as overfitting or increased training times. To address these issues, various techniques have emerged to reduce model complexity with minimal impact on the overall accuracy. These methods include compression techniques, parameter reduction strategies and architecture simplification schemes that can maintain model performance and reduce the memory footprint.

This section delineates the compression algorithm implemented for CNN models, with a particular focus on Convolutional 2D (Conv2D) filters. The algorithm encompasses filter extraction, a k-means clustering approach leveraging cosine similarity and weighted centroids, and an additional compression layer using the Two-Dimensional Discrete Cosine Transform (2D DCT).

The proposed compression algorithm operates in three primary stages:

1. Conv2D Filter Extraction and Stacking: Extracting individual Conv2D filters from the CNN model and organizing them into a suitable format for clustering.

2. Weighted Cosine K-Means Clustering: Applying an enhanced k-means clustering algorithm that utilizes cosine similarity and weighted centroids based on filter importance.

3. 2D Discrete Cosine Transform (DCT) on Centroids: Further compressing the clustered centroids by applying 2D DCT and selectively retaining low-frequency coefficients.

By isolating and stacking individual filters, the algorithm prepares the data for effective clustering, ensuring that each filter is treated as an independent entity. This method contrasts with row-wise k-means clustering, which treats each row of the tensor as a separate data point, potentially overlooking inter-filter similarities and redundancies.

7.3.1. Weighted Cosine K-Means Clustering

Clustering is a pivotal step in the compression algorithm, aimed at identifying and consolidating similar filters to reduce redundancy. The proposed approach enhances traditional k-means clustering through two significant modifications: the use of cosine similarity and the incorporation of weighted centroids based on filter importance.

Unlike the Euclidean distance, cosine similarity measures the cosine of the angle between two vectors, effectively capturing their directional alignment. Given that Conv2D filters are square-shaped and can be represented as vectors, cosine similarity is a more appropriate metric for assessing their similarity.

The cosine similarity between two *fi* and *fj* is defined as:

$$\text{cosine\_similarity}(f_i, f_j) = \frac{f_i \cdot f_j}{|f_i||f_j|}$$

Cosine similarity is insensitive to the magnitude of the vectors, focusing solely on their orientation. This property is beneficial for identifying filters that perform similar feature

extraction tasks regardless of their scale. By focusing on directional similarity, the clustering process becomes more robust to variations in filter magnitudes, leading to more meaningful centroid representations.

The importance of each filter is quantified by measuring the variance in the model's confidence when the specific filter's weights are eliminated. This metric assesses the impact of each filter on the model's performance, thereby identifying filters that contribute significantly to the inference accuracy.

Traditional k-means clustering computes the centroid as the arithmetic mean of the assigned data points. In contrast, the proposed method calculates the centroid as a weighted sum, where each filter's contribution is scaled by its importance weight.

The centroid *C* is computed as:

$$c = \frac{\sum_{i=1}^{N} w_i f_i}{\sum_{i=1}^{N} w_i}$$

where:

- N is the number of filters in the cluster.
- $\boldsymbol{f_i}$ is the i-th filter

By weighting the centroid calculation with filter importance, the algorithm ensures that more influential filters have a greater impact on the centroid's representation. This leads to a centroid that better reflects the essential characteristics of the cluster. Weighted centroids help maintain the model's inference confidence by prioritizing the preservation of impactful filters, thereby mitigating the loss in accuracy typically associated with model compression.

The combination of cosine similarity and weighted centroids results in more meaningful and performance-preserving clusters. This approach addresses the limitations of traditional k-means clustering, which may produce suboptimal centroids that do not account for the varying importance of filters within the model.

7.3.2. Two-Dimensional Discrete Cosine Transform on Centroids

To further enhance the compression of the CNN models, the algorithm employs the Two-Dimensional Discrete Cosine Transform (2D DCT) on a consolidated matrix of all clustered centroids. Unlike applying 2D DCT to each centroid individually, this approach leverages the spatial correlations and redundancies across multiple centroids, enabling more effective compression through collective frequency domain analysis.

After performing weighted cosine k-means clustering, all resulting centroids are organized into a single two-dimensional matrix. Suppose there are *K* clusters, each centroid being a *C*×*D* matrix. These centroids are stacked row-wise (or column-wise, depending on implementation) to form a comprehensive matrix *C* of size K×(C×D). The 2D DCT is applied to the entire centroid matrix *C*, transforming it from the spatial domain to the frequency domain. The transformation is defined as:

$$F(u,v) = \frac{1}{\sqrt{K}}\alpha(u)\alpha(v)\sum_{x=0}^{K-1}\sum_{y=0}^{C\times D-1} C(x,y)\cos\left[\frac{(2x+1)u\pi}{2K}\right]\cos\left[\frac{(2y+1)v\pi}{2(C\times D)}\right]$$

where:

- $\alpha(u) = \alpha(v) = \frac{1}{2}$ for $u = 0$ or $v = 0$, and $\alpha(u) = \alpha(v) = 1$ otherwise
- K is the number of columns
- $\alpha(u), \alpha(v)$ are scaling factors
- $C(x, y)$ is the input matrix
- x is the horizontal index
- y is the vertical index
- $C \times D$ is the total number of vertical samples
- u is the horizontal frequency index
- v is the vertical frequency index

In the frequency domain, the 2D DCT concentrates the signal's energy in the lower frequencies, while higher frequencies typically represent finer details and noise. By analyzing the transformed centroid matrix, the algorithm identifies high-frequency coefficients that contribute minimally to the overall representation. A predefined threshold determines which high-frequency coefficients are deemed insignificant and removed. This truncation is performed across the entire frequency matrix F(u,v), In the frequency domain, the 2D DCT concentrates the signal's energy in the lower

frequencies, while higher frequencies typically represent finer details and noise. By analyzing the transformed centroid matrix, the algorithm identifies high-frequency coefficients that contribute minimally to the overall representation. A predefined threshold determines which high-frequency coefficients are deemed insignificant and removed. This truncation is performed across the entire frequency matrix F(u,v), effectively reducing the amount of data required to represent the centroids without substantially affecting their essential characteristics. On the reconstruction side, the inverse 2D DCT is applied to the modified frequency matrix to reconstruct the original version of the centroid matrix C' in the spatial domain. This reconstructed matrix retains the most significant low-frequency components, ensuring that the primary structural and feature information of the centroids is preserved. The compressed centroid matrix C'C' requires significantly less memory, facilitating efficient storage on resource-constrained edge devices like the MAX78000. Additionally, the reduced data size accelerates transmission during model deployment and updates.

8. Experiments

In this section, we evaluate the compression techniques proposed earlier, specifically focusing on how they are applied to a CNN model running on the MAX78000 microcontroller. Through a combination of pruning, quantization, and weight sharing, we aim to reduce the model's memory footprint while preserving its performance. For clarity, a simplified digit recognition task using the MNIST dataset is chosen as the test case. We will walk through each step of the compression process, from the initial model architecture and training to the final compressed model, demonstrating how these techniques result in significant memory savings with minimal accuracy loss.

8.1. Compressing the CNN

This section will provide a detailed walkthrough of the CNN compression approach used in combination with the MAX78000. For the simplicity of the explanations, a more basic model will be used.

The chosen application for this experiment consists of a digit recognition system based on a convolutional neural network trained on the MNIST dataset. The MNIST dataset consists of approximately 70.000 images of handwritten digits, each image having a size of 28 by 28 pixels.

The CNN architecture used will be a variation of the LeNet model (Figure 6), which was originally designed for this type of application but with slight modifications to be compatible with the MAX78000 CNN accelerator. The model is structured as follows:

- Input Layer: 1x1x28x28 (grayscale input image)
- Conv2D: 32 filters, 3x3 kernels
- ReLu
- Conv2D: 32->64 (2048) filters, 3x3 kernels
- MaxPooling: 2x2
- Fully Connected: 9216 -> 128 neurons
- Fully Connected: 128 -> 10 neurons

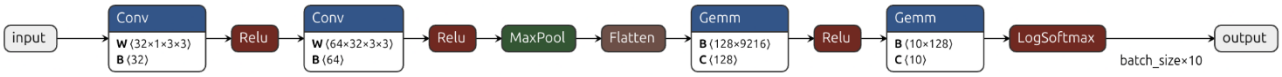

Figure 6 – Proposed CNN architecture

After training the model using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.01 over 40 epochs, the model achieved 98.43% accuracy.

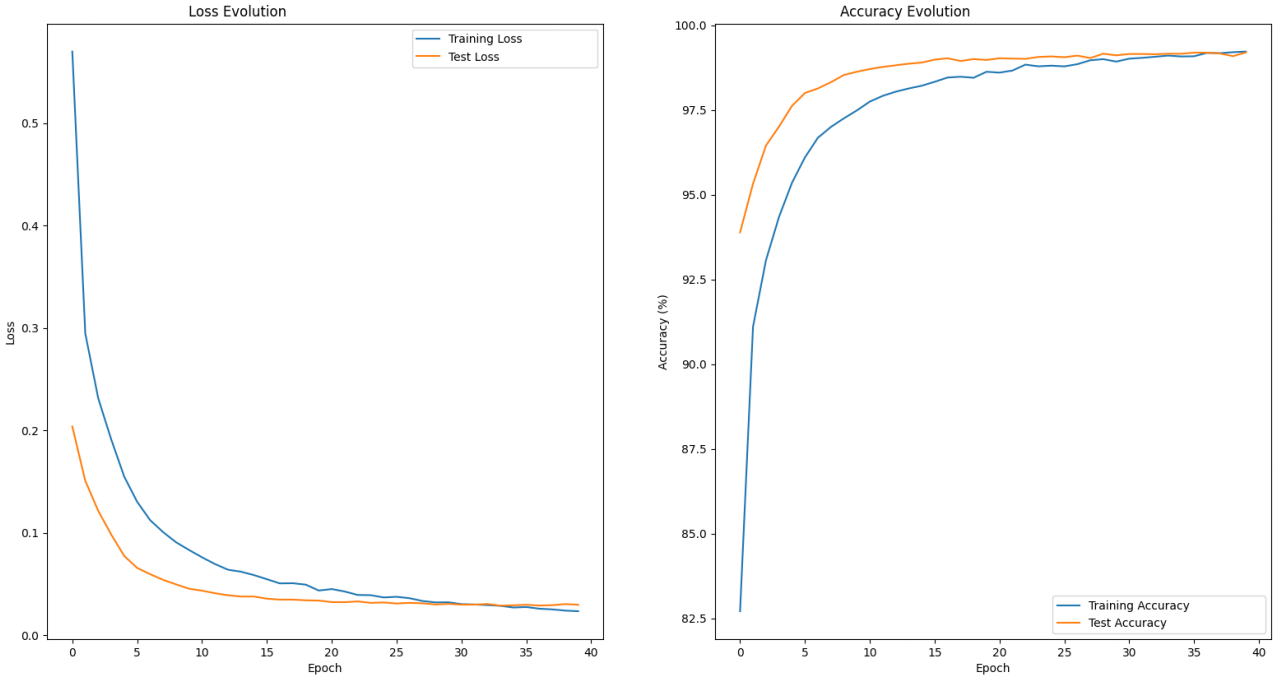

Figure 7 – Loss and accuracy evolution of the CNN over 40 epochs

8.1.1. Pruning and Quantization

As previously discussed, in this section we will focus only on the Conv2D operators of the proposed network. There is a total of 18.720 individual weights in the Conv2D layers, respectively 2080 3x3 filters. With a float32 representation, the total amount of memory occupied by those weights is 74880 bytes. From the histogram of those weights, we can observe the fact that they resemble a Gaussian distribution, with a big concentration of values between -0.1 and 0.1 and zero elsewhere.

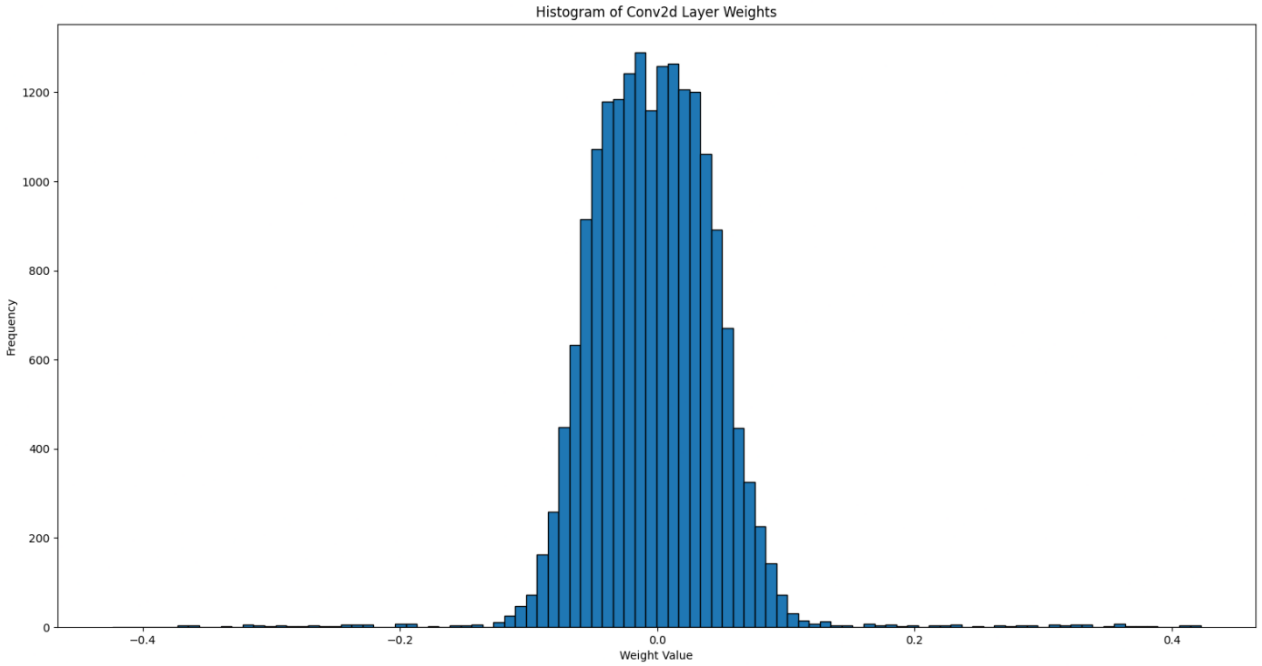

Figure 8 – The histogram of the Conv2D weights

At this point we start reducing the number of weights stored by this model. We can achieve that by performing structured pruning. Using the L1 norm we can evaluate the importance of each filter and based on that value we can remove entire filters from the weight tensor.

For each weight *wi* in the layer, the L1 norm is computed, and each result is sorted ascending, allowing us to drop the lowest p% of the filters.

By applying this method for 50% of the weights we measure a 0.02% drop in accuracy but using only half of the initial memory (37440 bytes).

The next step aims to further reduce the memory used by quantizing the weights. This technique is used to reduce the precision of the weights by using fewer bits to represent each value. In this case we will reduce the representation from float32 to int8 by following these steps:

First, we find the min and max value of the weights. Then the scale is determined by:

$$s = \frac{\max - \min}{256 - 1}$$

where 256 is the number of values that can be stored in an int8 variable. The zero point is determined by:

$$z = \text{round}\left(-\frac{\min}{s}\right)$$

The second step is to convert the floating-point weights w to quantized integers q using the scale and zero point:

$$q = \text{round}\left(\frac{w}{s}\right) + z$$

At this point we have reduced the memory usage by 75% with no accuracy loss, totaling 9360 bytes and 1/8 of the original memory footprint.

8.1.2. Weight sharing

Compression works by identifying and removing redundant information from the weights. In addition to pruning and quantization, we can index and reference duplicate values using a lookup table. Applied to the model weights, we will index similar filters using a more complex technique. Instead of treating each weight individually, we aim to represent each filter by a single feature (value). This feature extraction process allows us to reduce the complexity of the indexing method. We will name it as a "metric" for the weights, capturing a more high-level feature of the filter. The preferred metric in this case will be the Cosine Similarity. It measures the angle between two weight vectors determining their relative orientation:

$$\text{metric} = \frac{A \cdot B}{\text{norm}(A) \times \text{norm}(B)}$$

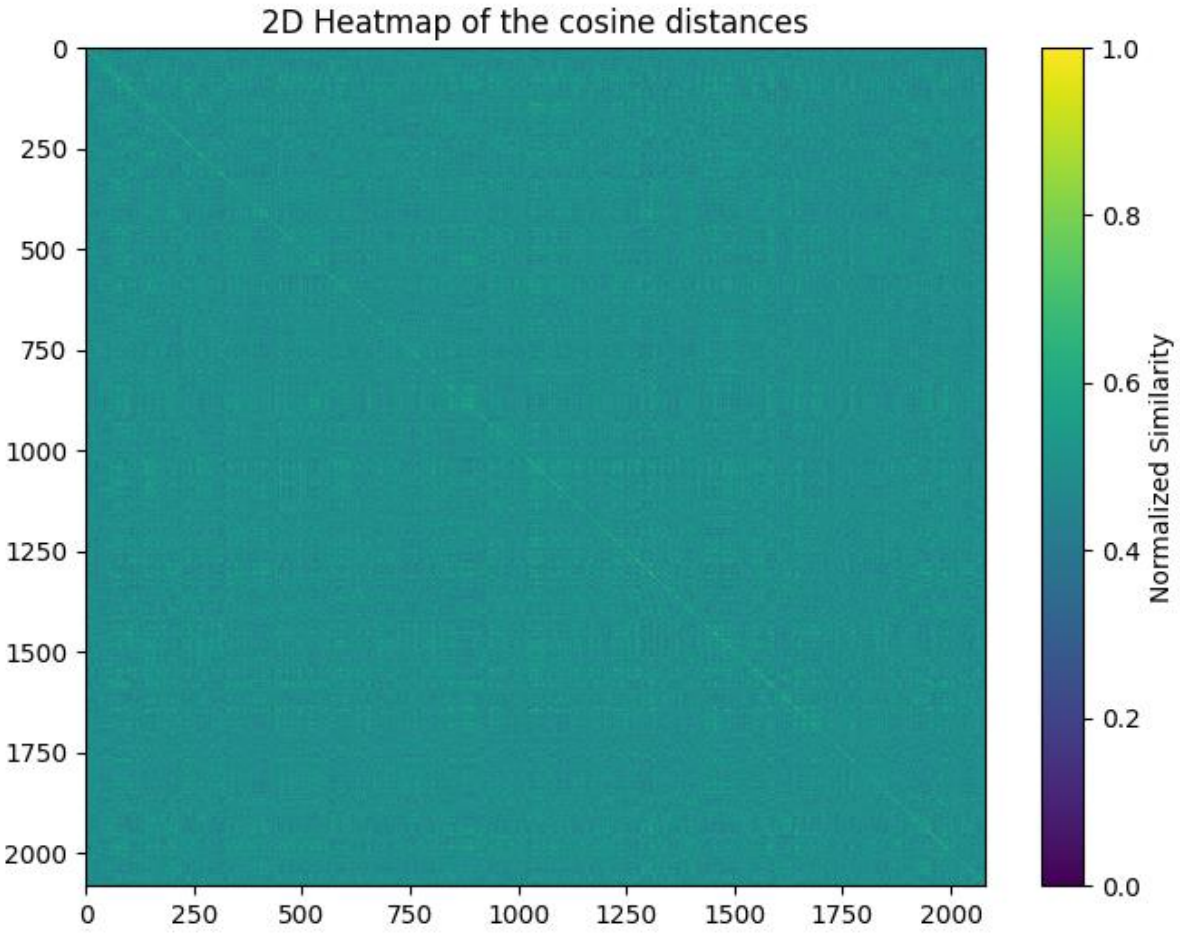

Figure 9 – Similarity map of the Conv2D filters

With the similarity matrix ready, the next step is to cluster similar weights together. The goal is to group weights that are like a degree, based on the cosine similarity, and to index each group as a single filter. To achieve an optimal grouping, we rely on a clustering algorithm. K-Means is a popular clustering algorithm used to partition a dataset into a given number of distinct groups. Its objective is to minimize the variance within each cluster while maximizing the variance between clusters. By using K-Means to group the weights, we obtain K clusters, each cluster containing a part of the filters. In the context of K-Means clustering, a centroid is the central point of a cluster, serving as a representative for that group of data points. The centroid of the clusters is computed using the mean of the vectors inside of the cluster. This method of

determining the centroid is improved by using instead a weighted sum of the elements, where each proportion is equivalent with the filter importance in the model. This importance is equivalent with the model accuracy variance when that filter is absent from the model. In this way the centroid is being shifted to the most influential filter instead.

Continuing the compression process, we start clustering the weights into K=44 groups. In the previous steps, while pruning we have reduced the number of filters from 2080 to 1040. Now, after clustering, the filters are reduced to 44 distinct values with their corresponding look-up table used for indexing. The result is a matrix of 44 filters of kernel size 3x3 and with this compressed format, the model achieves 97.44 accuracy (a 0.99% loss).

To compress further, we apply the Discrete Cosine Transform (DCT) to break the filter matrix into frequency components. High-frequency components, which contribute less to the overall model, are removed to reduce the number of parameters. By eliminating one column of the 44x9 coefficients matrix, we cut another 12% of the data without affecting the model's accuracy, resulting in a 1432-byte representation of the weights, including the lookup table.

In the end, this approach reduces the model's memory size by 26 times with just a 1% loss in accuracy.

9. Conclusion

The implementation of OTA update capabilities for CMM models on the MAX78000 microcontroller significantly enhances the system's flexibility, maintainability, and security. By enabling remote updates, the system ensures that deployed AI models remain current, optimized, and secure, thereby extending the device's operational lifespan and adaptability. The comprehensive architecture, robust security measures, and performance optimizations collectively contribute to a reliable and efficient OTA update mechanism, positioning the MAX78000 as a formidable solution for edge AI applications.

The compression algorithm presented in this document offers a significant advancement in optimizing Convolutional Neural Network (CNN) models for deployment on resource-constrained edge devices like the MAX78000. By focusing on Conv2D filters, the algorithm effectively reduces the memory footprint of CNN models through a meticulously designed process that integrates weighted cosine k-means clustering with the Two-Dimensional Discrete Cosine Transform (2D DCT). The initial extraction and stacking of Conv2D filters facilitate efficient clustering, while the adoption of cosine similarity and weighted centroids ensures that the most impactful filters are preserved, maintaining the model's accuracy despite substantial compression. Moreover, the integration of this compression algorithm within the MAX78000's software ecosystem underscores its practical applicability and effectiveness in real-world scenarios. The reduced model size facilitates faster inference times and lower power consumption, aligning perfectly with the operational constraints of edge devices. Additionally, the algorithm's minimal impact on inference accuracy ensures that the deployed AI models remain reliable and effective, even after substantial compression.

References

[1] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, Yingyan Lin: CNN weight sharing based on a fast accuracy estimation metric

[2] Madhumitha Sakthi, Niranjan Yadla, Raj Pawate.: Deep learning model compression using network sensitivity and gradients

Glossary

Term	Definition	Page Number
AI	Artificial Intelligence: The simulation of human intelligence in machines that are programmed to think and learn.	5
BRAM	Block Random Access Memory: A type of memory used in FPGAs and ASICs for fast data storage and retrieval.	33
DED	Double Error Detection: A capability to identify when two bits in a data block have errors, though it may not correct them.	29
CLI	Command Line Interface: A text-based user interface used to interact with software and operating systems through commands.	6
CNN	Convolutional Neural Network: A deep learning algorithm primarily used for processing structured grid data, such as images.	5

DCT	Discrete Cosine Transform: A mathematical transformation used in signal processing and image compression (e.g., JPEG).	42
ECC	Error-Correcting Code: A method for detecting and correcting errors in data storage or transmission.	29
MAC	Multiply and Accumulate: An arithmetic operation that multiplies two numbers and adds the result to an accumulator.	28
MCU	Microcontroller Unit: A compact integrated circuit designed to control specific functions in embedded systems.	27
MRAM	Mask Random Access Memory: A type of non-volatile memory that uses magnetic states to store data.	33
OTA	Over the Air: A method of delivering software updates wirelessly to devices.	5
OTAP	Over the Air Programming: A process for updating firmware or software on devices remotely via wireless communication.	7
PBL	Primary Bootloader: The first software that runs on a device, initializing hardware and loading the main operating system.	10
RAM	Random Access Memory: A type of volatile memory that temporarily stores data for quick access by the CPU.	9
RTC	Real Time Clock: A clock that keeps track of the current time and date, even when the device is powered off.	30
SBL	Secondary Bootloader: A secondary piece of software that may manage the loading of the main application after the primary bootloader.	10
SEC	Single Error Correction: A capability to detect and correct a single-bit error in a data block.	29

SIMO	Single-Inductor Multiple-Output: A power supply architecture that uses one inductor to provide multiple output voltages.	30
SMPS	Switch-Mode Power Supply: A type of power supply that uses switching regulators to efficiently convert electrical power.	30
SRAM	Static Random Access Memory: A fast, volatile memory that retains data if power is supplied, without needing refresh cycles.	29
SVD	Singular Value Decomposition: A mathematical technique used in linear algebra for matrix factorization and dimensionality reduction.	22
TRAM	Tornado Random Access Memory: A type of memory architecture designed for high-speed data access and reliability.	34
UART	Universal Asynchronous Receiver-Transmitter: A hardware communication protocol used for asynchronous serial communication between devices.	11